Slide Gallery

Redirection Notice
This page should redirect to http://www.10gen.com/presentations.

.bookmarks

Recent bookmarks in MongoDB

This page is a container for all the bookmarks in this space. Do not delete or move it or you will lose all your bookmarks.
Bookmarks in MongoDB | Links for MongoDB

The 15 most recent bookmarks in MongoDB

There are no bookmarks to display.

1.1 Development Cycle

Redirection Notice
This page should redirect to [1.2.0 Release Notes].

Creating and Deleting Indexes

Redirection Notice
This page should redirect to Indexes.

C Sharp Language Center

Redirection Notice
This page should redirect to CSharp Language Center.

Diagnostic Tools

Redirection Notice
This page should redirect to Monitoring and Diagnostics.

Django and MongoDB

Redirection Notice
This page should redirect to Python Language Center.

Getting Started

Redirection Notice
This page should redirect to Quickstart.

International Documentation

Redirection Notice
This page should redirect to International Docs.

Monitoring

Redirection Notice
This page should redirect to Monitoring and Diagnostics.

Older Downloads

Redirection Notice
This page should redirect to Downloads.

PyMongo and mod_wsgi

Redirection Notice
This page should redirect to Python Language Center.

Python Tutorial

Redirection Notice
This page should redirect to Python Language Center.

Recommended Production Architectures

Redirection Notice
This page should redirect to Production Notes.

v0.8 Details

Existing Core Functionality

- Basic Mongo database functionality: inserts, deletes, queries, indexing.
- Master / Slave Replication
- Replica Pairs
- Server-side javascript code execution

New to v0.8

- Drivers for Java, C++, Python, Ruby.
- db shell utility
- (Very) basic security
- $or
- Clean up logging
- Performance test baseline
- getlasterror
- Large capped collections
- Bug fixes (compound index keys, etc.)
- Import/Export utility
- Allow any _id that is unique, and verify uniqueness

Wanted, but may not make it

- AMI's
- Unlock eval()?
- Better disk full handling
- better replica pair negotiation logic (for robustness)

Building SpiderMonkey

Redirection Notice
This page should redirect to Building Spider Monkey.

Documentation

Redirection Notice
This page should redirect to Home.

Dot Notation

Redirection Notice
This page should redirect to Dot Notation (Reaching into Objects).

Dot Notation

Redirection Notice
This page should redirect to Dot Notation (Reaching into Objects).

Getting the Software

Placeholder - $$$ TODO

Language Support

Redirection Notice
This page should redirect to Drivers.

Mongo Administration Guide

Redirection Notice
This page should redirect to Admin Zone.

Working with Mongo Objects and Classes in Ruby

> **Redirection Notice**
> This page should redirect to Ruby Language Center.

MongoDB Language Support

> **Redirection Notice**
> This page should redirect to Language Support.

Community Info

> **Redirection Notice**
> This page should redirect to Community.

Internals

Cursors

Tailable Cursors

See *p/db/dbclient.h* for example of how, on the client side, to support tailable cursors.

Set

```
Option_CursorTailable = 2
```

in the `queryOptions int` field to indicate you want a tailable cursor.

If you get back no results when you query the cursor, keep the cursor live if cursorid is still nonzero. Then, you can issue future `getMore` requests for the cursor.

If a `getMore` request has the `resultFlag ResultFlag_CursorNotFound` set, the cursor is not longer valid. It should be marked as "dead" on the client side.

```
ResultFlag_CursorNotFound = 1
```

See the Queries and Cursors section of the Mongo Developers' Guide for more information about cursors.

See Also

- The Queries and Cursors section of the Mongo Developers' Guide for more information about cursors

TreeNavigation

Follow @mongodb

MongoDB Pittsburgh - May 15
MongoNYC - May 23
MongoDB Paris - Jun 14
MongoDB UK - Jun 20
MongoDC - June 26

Old Pages

Storing Data

> **Redirection Notice**
> This page should redirect to Inserting.

Indexes in Mongo

> **Redirection Notice**
> This page should redirect to Indexes.

HowTo

> **Redirection Notice**
> This page should redirect to Developer FAQ.

Searching and Retrieving

> **Redirection Notice**
> This page should redirect to Querying.

Locking

Redirection Notice
This page should redirect to Atomic Operations.

Mongo Developers' Guide

Redirection Notice
This page should redirect to Manual.

Locking in Mongo

Redirection Notice
This page should redirect to Developer FAQ.

Mongo Database Administration

Redirection Notice
This page should redirect to Admin Zone.

Mongo Concepts and Terminology

See the Manual page for information on the following:

- BSON
- Collections
- Cursors
- Databases
- Documents
- GridFS (for files and very large objects)
- Indexes
- Transactions / Atomic Operations

Other Concepts

- Config server. In a sharded environment, config servers store the metadata of the cluster. Each config server has a `mongod` process which stores metadata. Typically there are three config servers which have replicas of the same metadata (for data safety and HA). The config server mongod process is fairly lightweight and can be ran on machines performing other work.
- Durability / [Journaling. Write-ahead logging for crash safety.
- Member. A member (server) in a replica set.
- mongod, mongos, mongo. MongoDB processes.
- Object IDs. Mongo documents include an `_id` field in each document.
- Oplog. High level log of operations used by replication.
- Replication. Duplicating data on multiple servers for HA, safety, disaster recovery, and a bit of scaling. Sharding and replication are used together.
- Sharding. The partioning / distribution of data among machines in a cluster. Each shard has different data. Sharding is the mechanism in MongoDB for building very large clusters. Note: we recommend you begin using MongoDB without sharding. It is easy to transition over to sharding later.
- Shard, Chunk, Shard Key, Config Server. See the sharding intro page.

See Also

- Architecture and Components
- SQL to Mongo Mapping Chart

MongoDB - A Developer's Tour

Redirection Notice
This page should redirect to Manual.

Updates

Redirection Notice
This page should redirect to Updating.

Structuring Data for Mongo

Redirection Notice
This page should redirect to Inserting.

Design Overview

Redirection Notice
This page should redirect to Developer Zone.

Document-Oriented Datastore

Redirection Notice
This page should redirect to Databases.

Why so many "Connection Accepted" messages logged?

Redirection Notice
This page should redirect to Developer FAQ.

Why are my datafiles so large?

Redirection Notice
This page should redirect to Developer FAQ.

Storing Files

Redirection Notice
This page should redirect to GridFS.

Introduction - How Mongo Works

Redirection Notice
This page should redirect to Developer Zone.

Optimizing Mongo Performance

Redirection Notice
This page should redirect to Optimization.

Mongo Usage Basics

Redirection Notice
This page should redirect to Tutorial.

Server-Side Processing

Redirection Notice
This page should redirect to Server-side Code Execution.

Home

Getting Started

- Quickstart - how to install
- Tutorial - quick tutorial showing examples of manipulation of a database from the mongo shell
- SQL to Mongo Mapping Chart

Development

- Developer Manual

See the documentation below for the driver for your application's programming language of choice:

- C | C# | C++ | C# & .NET | ColdFusion | Erlang | Haskell | Factor
- Java | Javascript | PHP | Python | Ruby | Perl | More...

The MongoDB Manual is a complete rewrite of the MongoDB documentation that will eventually replace this site. You can view the latest build at docs.mongodb.org, and contribute on github and Jira.

Production / Ops

- Production Notes
- Replication | Sharding
- Security | Backups
- Hosting
- MongoDB Monitoring Service (MMS) is a free way to monitor your mongodb servers

Support

- Free support forum: groups.google.com/group/mongodb-user
- IRC: irc.freenode.net/#mongodb
- Bug DB: jira.mongodb.org
- Commercial support | Training | Consulting

MongoDB Conferences

NYC - May 23
Paris - June 14
London - June 20
Washington DC - June 26

Community

- Slides and Video
- Events | Blog | Articles | Twitter | Forum | MongoDB Masters | Facebook | LinkedIn | Job Board
- User groups: NY, SF, DC, Chicago, London, Stockholm and more.
- Your Go-To Guide to Running a MongoDB User Group

Meta

- Use Cases | Philosophy | License | Events

Translations

New Docs (in Progress)

- |

Existing Docs

- Español | | Português | | |

Please help translate the documentation! Email docs@10gen.com and tell us which language you can translate.

MongoDB Masters

- **Rick Copeland**
- **Justin Dearing**
- **Mike Dirolf**

- **Kenny Gorman**
- **Jonas Haag**
- **Nathen Harvey**
- **Aaron Heckmann**
- **Takahiro Inoue**
- **Lennart Koopmann**
- **Christian Kvalheim**
- **Ross Lawley**
- **Nat Luengnaruemitchai**
- **David Makogon**
- **Harry Marr**
- **David Mytton**
- **Gustavo Niemeyer**
- **John Nunemaker**
- **Niall O'Higgins**
- **Flavio Percoco**
- **Mitch Pirtle**
- **Karl Seguin**
- **Mark Smalley**
- **Tony Tam**
- **Rose Toomey**
- **Jonathan Wage**
- **Ian White**
- **Craig Wilson**
- **Aristarkh Zagorodnikov**

(sorted alphabetically by last name)

Rick Copeland

Rick Copeland is a Lead Software Engineer at SourceForge where he joined the team that introduced MongoDB to the SourceForge technology stack with the migration of the consumer-facing pages of SourceForge from a PHP/relational database platform to Python/MongoDB. Out of that experience came Ming, an MongoDB object/document mapper for Python that he maintains and continues to develop. Rick also helped lead the effort to rewrite the developer tools (wiki, tickets, forums, repos, etc.) portion of the SourceForge site on the Python/MongoDB platform and released that platform as Allura under the Apache License. He also created the Zarkov realtime analytics framework (also released under the Apache license) used at SourceForge to calculate project statistics. He is a frequent speaker at MongoDB events and an avid MongoDB enthusiast.

GitHub
@rick446 on Twitter
Blog
Presentations

MongoDB contributions

Ming, an object/document mapper for MongoDB in Python
Zarkov, a realtime analytics framework using MongoDB
Allura, the new MongoDB-powered platform for SourceForge

Justin Dearing

Justin Dearing has been working in IT in 2002. He started his career as a night shift AS/400 operator and rose through the ranks at a series of companies.

Justin has worked in both the development and production side of the house on Windows, Unix and Midrange Platforms. Besides MongoDB his database experience includes MySQL, Postgres and Microsoft SQL server. These days he mainly programs in C#, Powershell and PHP.

Justin's life was forever changed on 2009-10-27 when Kristinia Chodorow presented a talk on mongodb at NYPHP and destroyed everything he knew to be right, holy and true about databases. A few months later he push a small app using MongoDB to production. In addition to afflicting the world with apps that use MongoDB, he has contributed to the core server and the official .NET driver.

Justin lives in Jersey City with his wife and 3 laptops.

@Zippy1981 on Twitter
GitHub
Blog

Mike Dirolf

Mike was the original author of the PyMongo project and a maintainer of the mongo-ruby-driver. He co-authored O'Reilly's MongoDB: The

Definitive Guide. He maintains several MongoDB-related open source projects, and runs a web service, Fiesta (https://fiesta.cc), that uses MongoDB as its primary data store.

@mdirolf on Twitter
GitHub
Blog

MongoDB Contributions

MongoDB: The Definitive Guide
Wrote PyMongo
Mongo-ruby-driver
nginx-gridfs

Kenny Gorman

Kenny Gorman has over a decade of experience with various database platforms behind some of the busiest websites in the world. He has had roles as Developer, DBA, Architect, Manager and Director. He was an early adopter of MongoDB in 2009 using it for various projects at Shutterfly. He wrote an early python version of the Mongostat tool that is distributed with MongoDB today. He enjoys performance tuning, large scale systems development, and tricky database problems.

Github
@kennygorman
Blog

Contributions
Wrote the original mongostat in python, since then it's moved to core distribution.

Jonas Haag

Jonas Haag is a passionate Python programmer and free software enthusiast from Stuttgart, Germany. He maintains Django MongoDB Engine, a MongoDB backend for the Python Web framework Django.

GitHub
Blog

MongoDB Contributions

Django MongoDB Engine
PyMongo

Nathen Harvey

Nathen Harvey is the manager of Web Operations for CustomInk.com, a website that allows you to design and purchase custom apparel for your group or special event. Nathen is the co-organizer of the Washington DC MongoDB Users' Group and DevOps DC. Most of Nathen's open source contributions are for the Opscode Chef framework and include cookbooks for managing MongoDB. When not working or hosting meetups, Nathen enjoys going to concerts, drinking craft beer, and over sharing on sites like twitter, untappd, and foursquare.

Github
@NathenHarvey on Twitter
Blog
Untappd

MongoDB contributions

MongoDC User Group

Aaron Heckmann

Aaron is currently an engineer at LearnBoost, an education startup built on node.js. An active member of the node.js community, Aaron is the maintainer of Mongoose, the MongoDB object modeling tool, as well as the author of express-mongoose, gm, gleak, and contributing to other projects such as Express and the node-mongodb-native mongodb driver.

@aheckmann on Twitter
GitHub
Blog

MongoDB Contributions

Mongoose - nodejs ODM for MongoDB

Mongoose on GitHub
Express-Mongoose - adds Mongoose support to Express
MongoDB Driver for Node.js

Takahiro Inoue

Takahiro is a Chief Data Scientist at Treasure-Data Inc where he uses MongoDB for log data analysis. He is a frequent speaker on MongoDB and Data and the organizer of the Tokyo MongoDB User Group

GitHub
@doryokujin on Twitter
Slideshare
Blog

MongoDB Contributions

Organizer of the Tokyo MongoDB User Group

Lennart Koopmann

Lennart Koopmann is a developer from Hamburg, Germany and author of Graylog2 - A free and open source log mangement system that uses MongoDB as database. He also wrote mongo_analyzer, a little web frontend on top of the MongoDB profiler that helps you optimizing your queries.

@_Lennart on Twitter
GitHub
Blog

MongoDB Contributions

Graylog
Mongo Analyzer

Christian Kvalheim

Christian Kvalheim has been coding since the days of the c64 and still enjoys it. He got into the node.js scene 2 years ago and started writing the node.js driver for mongodb as he saw the need for a decent javascript driver to fill the gap and felt that mongo was a natural fit as a database for his node.js projects. He spends his free time dabbling in open source and learning new programming languages.

GitHub
@Christkv on Twitter
Blog
SlideShare

MongoDB Contributions

Node.js MongoDB Driver
Introduction to the Node.js MongoDB Driver

Ross Lawley

Ross Lawley is an pro active and enthusiastic software engineer who loves to get things done. Holding a deep passion for web development, Ross loves to contribute back to open source communities by doing what he can: committing code, documentation fixes or mentoring. Over 10 years experience in web development and leading teams, Ross is joining 10gen in December as a python evangalist and engineer. Ross maintains the popular MongoEngine ODM.

Github
@RossC0 on Twitter
Blog

Nat Luengnaruemitchai

Bio: working in financial industry. Help out on a couple projects such as ikvm, dnanalytics, mongodb
Github

MongoDB Contributions
Bug fixes/small enhancement in mongodb core, c# driver, java driver
Over 2,700 posts on the mongodb-user group (free support forum)

David Makogon

David Makogon has been a software creationist and architect for over 25 years. He's currently a Senior Cloud Architect at Microsoft specializing in Windows Azure.

Since 2010, David has been working with MongoDB, specifically in Windows Azure. He built both standalone and replica set samples, presenting these at MongoDC and MongoSV in 2010. He's also provided architectural guidance to several ISV's as they build Windows Azure solutions coupled with MongoDB.

Outside of computing, David is an avid photographer and family man, with a penchant for puns and an uncanny ability to read backwards.

Twitter
Blog
Presentations

Harry Marr

Harry Marr (@harrymarr) is the author of MongoEngine, a popular Python ODM (Object-Document Mapper). He hails from London, where he spends most of his time working in Python and Ruby. He was previously employed at Conversocial, where he drove a migration from MySQL to MongoDB using MongoEngine. He currently works at GoCardless, an early-stage startup with some exciting ambitions in the payments space. When he's not working on disrupting the payments industry, he can be found hacking on various open source projects (https://github.com/hmarr).

Github
@harrymarr
Blog

MongoDB Contributions

MongoEngine
MongoEngine source

David Mytton

David has been a PHP/Python programmer for 10 years. He is the founder of Server Density a hosted server monitoring service where he built the original code and server infrastructure behind the application which is now processing over 1bn documents (7TB data) each month.
Server Density uses MongoDB extensively as our primary data store since 2009, and it now deployed across 50 servers on the Terremark Enterprise Cloud. He is a regular speaker on MongoDB and runs the London MongoDB User Group.

@Davidmytton on Twitter
GitHub
Blog

MongoDB Contributions

Server Density
MongoDB Monitoring Tool
London MongoDB User Group

Gustavo Niemeyer

Gustavo acts as the technical lead behind projects from Canonical such as the Landscape systems management platform, the juju orchestration framework, and the Storm object-relational mapper for Python. In his free time, among other things Gustavo is a contributor to Google's Go language, is the author of the mgo (mango) MongoDB driver for Go, and also designed the Geohash concept that is used internally by MongoDB.

@Gniemeyer on Twitter
Code Repository
Blog

MongoDB Contributions

mgo
Geohash

John Nunemaker

John Nunemaker develops simple and beautiful software at Ordered List, which has several MongoDB backed applications in production – Gauges, Harmony and Speaker Deck. He is also the creator of MongoMapper, a popular Ruby object mapping library for MongoDB.

@Jnunemaker on Twitter

GitHub
Blog

MongoDB Contributions
MongoMapper

Niall O'Higgins

Niall O'Higgins is the co-founder of a software product & services company specializing in NoSQL, mobile and cloud computing. He is the author of the book "MongoDB and Python" published by O'Reilly. He is the founder and organizer of both the San Francisco Python Web Technology Meet-up, PyWebSF and the Bay Area Tablet Computing Group, We Have Tablets. He has published quite a bit of Open Source software - contributing to OpenBSD and Pyramid among others - and frequently speaks at conferences and events.

@niallohiggins on Twitter
GitHub
Blog

MongoDB Contributions
MongoDB and Python

Flavio Percoco

Flavio works in the Research and Development department at The Net Planet Europe and is an avid MongoDB community contributor. His host of contributions include Pymongo, the Django Database Engine (co-author and maintainer), the MongoDB plugin for eclipse and the python virtual machine for MongoDB. He lives in Milan, Italy and is a frequent speaker at MongoDB and Europe technology conferences.

@flaper87 on Twitter
GitHub
BitBucket
Blog

MongoDB Contributions

Django Database Engine for MongoDB
Python Virtual Machine inside MongoDB
MongoDB Plugin for Eclipse
MongoDB CDR Backend for Asterisk
MongoDB Transport for Kombu

Mitch Pirtle

Mitch is currently CTO at Sounday Music, a social and services platform catering to the music industry. There he maintains the core platform comprised of MongoDB and the Lithium framework for PHP.

He was first corrupted by Dwight Merriman while launching Jetsetter for Gilt Groupe, which went on to be the first e-commerce site powered by MongoDB. He then followed that up by launching Totsy, the first e-commerce site to rely solely on MongoDB for all data storage. He is also an original core member for the Mambo content management system, where he went on to found Joomla! and Open Source Matters. Before that he was contributing to many open source projects, and was an outspoken advocate for PostgreSQL, which still remains his favorite relational database.

He is based in Turin Italy with his wife, kids, and rapidly proliferating devices.

GitHub: http://github.com/spacemonkey
@mitchitized on Twitter

Karl Seguin

Karl Seguin is a developer with experience across various fields and technologies. He's an active contributor to OSS projects, a technical writer and an occasional speaker. With respect to MongoDB, he was a core contributor to the C# MongoDB library NoRM, wrote the interactive tutorial mongly, the Mongo Web Admin and the free Little MongoDB Book. His service for casual game developers, mogade.com, is powered by MongoDB.

@KarlSeguin on Twitter
GitHub
Blog

MongoDB Contributions

Mongoly.com
The Little MongoDB Book

Mark Smalley

Mark Smalley is a Brit on a mission. Currently based out of Kuala Lumpur, Malaysia, he roams around Asia making every effort he can to convert anyone and everyone into avid MongoDB enthusiasts. He is also one of the lead organizers for the monthly Kuala-Lumpur MongoDB User-Group and lead-developer on several MongoDB powered OpenSource initiatives.

Twitter
GitHub
Blog

MongoDB Contributions

MongoBase
Geoply
MongoPress

Tony Tam

@fehguy on Twitter
GitHub
Presentations
Swagger

MongoDB Contributions

mongodb oss admin tools

Tony is a San Francisco Bay Area native. He received his undergraduate degree in Mechanical Engineering from UC Santa Barbara and his MBA from Santa Clara University. He was the founding engineer and SVP of Engineering at Think Passenger, the leading provider of customer collaboration software. Prior to joining Passenger, he was lead engineer at Composite Software of San Mateo, California. At Composite Software he helped developed the company's first- and second-generation query processing engines and led the research and implementation of their patented cost-based federated query optimizer. Prior to that he led software development in the bioinformatics group at Galileo Labs, a drug-discovery company based in the Silicon Valley.

Rose Toomey

Rose Toomey is the creator of Salat, a simple serialization for Scala and MongoDBSalat was developed to make using Scala with Casbah and MongoDB as simple as possible. While Casbah increased the usability of the mongo-java-driver in Scala, there was no correspondingly elegant solution for serializing and deserializing objects. The new horizons opened up by using MongoDB as a document store demanded something better than the complexity and ceremony of the ORMs I'd worked with in the past. I also faced the challenge that my company, Novus Partners, is a financial startup that needs to process massive amounts of data very quickly. What to do? Enter Salat: it not only serializes to and from Mongo documents quickly, but uses hi-fi type information provided by the Scala compiler instead of explicit mappings. No fuss, no muss: my goal is that someone who wants to use Scala and MongoDB can be up and running with Salat in fifteen minutes.

GitHub
@Prasinous on Twitter

MongoDB Contributions

Salat

Jonathan Wage

Software engineer from Nashville, TN currently working for OpenSky.com
@Jwage on Twitter
GitHub
Blog

MongoDB Contributions

Doctrine MongoDB Object Document Mapper for PHP open source project

Ian White

Ian is the co-founder and CTO of Sailthru, a company that automatically tailors email, web and advertising content down to the unique user. He was the first non-10gen employee to use MongoDB in production, and built both Business Insider and Sailthru using MongoDB as the datastore.

@EonWhite on Twitter

GitHub

MongoDB Contributions

SimpleMongoPHP

Craig Wilson

Craig Wilson is a developer out of Dallas, TX where he has a wonderful family complete with 3 kids. He works for RBA Consulting giving guidance and developing solutions for Microsoft-centric clients around mobile development and service oriented architectures. He helped write the original csharp driver for Mongo and currently maintains two mongo related projects: FluentMongo is a linq provider for the 10gen C# driver and Simple.Data.MongoDB is a dynamic data adapter for Simple.Data to connect with MongoDB.

@craiggwilson
GitHub

MongoDB Contributions

Simple.Data.Mongo
FluentMongo

Aristarkh Zagorodnikov

Started using MongoDB about half a year ago, made it the default database (we still have most of our stuff using PostgreSQL, but all new development except billing services is done with MongoDB) for our company Bolotov .

GitHub
BitBucket
Blog

MongoDB Contributions

MongoDB C# Driver

Introduction

> *MongoDB wasn't designed in a lab. We built MongoDB from our own experiences building large scale, high availability, robust systems. We didn't start from scratch, we really tried to figure out what was broken, and tackle that. So the way I think about MongoDB is that if you take MySql, and change the data model from relational to document based, you get a lot of great features: embedded docs for speed, manageability, agile development with schema-less databases, easier horizontal scalability because joins aren't as important. There are lots of things that work great in relational databases: indexes, dynamic queries and updates to name a few, and we haven't changed much there. For example, the way you design your indexes in MongoDB should be exactly the way you do it in MySql or Oracle, you just have the option of indexing an embedded field.*
>
> *– Eliot Horowitz, 10gen CTO and Co-founder*

Why MongoDB?

- **Document-oriented**
  - Documents (objects) map nicely to programming language data types
  - Embedded documents and arrays reduce need for joins
  - Dynamically-typed (schemaless) for easy schema evolution
  - No joins and no multi-document transactions for high performance and easy scalability
- **High performance**
  - No joins and embedding makes reads and writes fast
  - Indexes including indexing of keys from embedded documents and arrays
  - Optional streaming writes (no acknowledgements)
- **High availability**
  - Replicated servers with automatic master failover
- **Easy scalability**
  - Automatic sharding (auto-partitioning of data across servers)
    - Reads and writes are distributed over shards
    - No joins or multi-document transactions make distributed queries easy and fast
  - Eventually-consistent reads can be distributed over replicated servers
- **Rich query language**

Large MongoDB deployment

1. One or more shards, each shard holds a portion of the total data (managed automatically). Reads and writes are automatically routed to the appropriate shard(s). Each shard is backed by a replica set – which just holds the data for that shard.

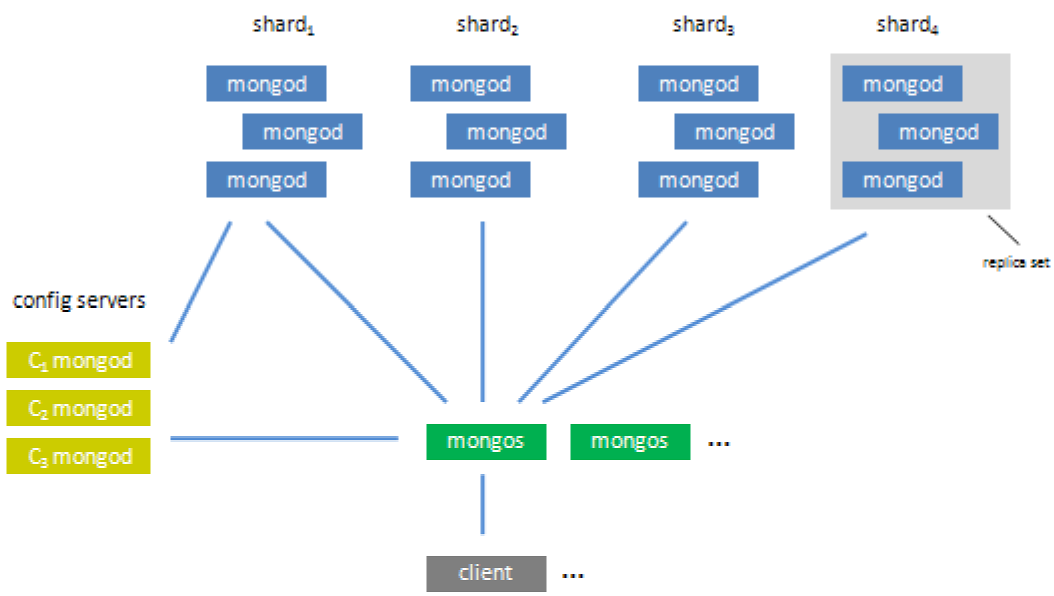
A replica set is one or more servers, each holding copies of the same data. At any given time one is primary and the rest are secondaries. If the primary goes down one of the secondaries takes over automatically as primary. All writes and consistent reads go to the primary, and all eventually consistent reads are distributed amongst all the secondaries.

2. Multiple config servers, each one holds a copy of the meta data indicating which data lives on which shard.

3. One or more routers, each one acts as a server for one or more clients. Clients issue queries/updates to a router and the router routes them to the appropriate shard while consulting the config servers.

4. One or more clients, each one is (part of) the user's application and issues commands to a router via the mongo client library (driver) for its language.

`mongod` is the server program (data or config). **`mongos`** is the router program.



Small deployment (no partitioning)

1. One replica set (automatic failover), or one server with zero or more slaves (no automatic failover).

2. One or more clients issuing commands to the replica set as a whole or the single master (the driver will manage which server in the replica set to send to).

Mongo data model

- A Mongo system (see deployment above) holds a set of databases
- A **database** holds a set of collections
- A **collection** holds a set of documents
- A **document** is a set of fields
- A **field** is a key-value pair
- A **key** is a name (string)
- A **value** is a
    - basic type like string, integer, float, timestamp, binary, etc.,
    - a document, or
    - an array of values

Mongo query language

To retrieve certain documents from a db collection, you supply a query document containing the fields the desired documents should match. For example, {`name: {first: 'John', last: 'Doe'}`} will match all documents in the collection with name of John Doe. Likewise, {`name.last: 'Doe'`} will match all documents with last name of Doe. Also, {`name.last: /^D/`} will match all documents with last name starting with 'D' (regular expression match).

Queries will also match inside embedded arrays. For example, `{keywords: 'storage'}` will match all documents with 'storage' in its keywords array. Likewise, `{keywords: {$in: ['storage', 'DBMS']}}` will match all documents with 'storage' or 'DBMS' in its keywords array.

If you have lots of documents in a collection and you want to make a query fast then build an index for that query. For example, `ensureIndex({name.last: 1})` or `ensureIndex({keywords: 1})`. Note, indexes occupy space and slow down updates a bit, so use them only when the tradeoff is worth it.

See also:

- Philosophy

Quickstart

- Quickstart OS X
- Quickstart Unix
- Quickstart Windows

For an even quicker start go to http://try.mongodb.org/.

Installation Guides

Consider the [installation guides from the MongoDB Manual]:

- Install MongoDB on RedHat Enterprise Linux, CentOS, or Fedora Linux
- Install MongoDB on Debian, Ubuntu or other Linux Systems
- Install MongoDB on other Unix/Linux Systems
- Install MongoDB on OS X

See Also

- SQL to Mongo Mapping Chart
- Tutorial
- Tutorials in the MongoDB Manual
- The MongoDB Manual

Quickstart OS X

Install MongoDB

The easiest way to install MongoDB is to use a package manager or the pre-built binaries, described below and on the MongoDB Manual's OS X Installation Guide. Consider all of the available installation guides.

Package managers

If you use the Homebrew package manager, run:

```
$ brew update
$ brew install mongodb
```

If you use MacPorts you can install with:

```
$ sudo port install mongodb
```

This will take a while to install.

32-bit binaries

Note: 64-bit is recommended (if you have a 64-bit system).

```
$ curl http://downloads.mongodb.org/osx/mongodb-osx-i386-x.y.z.tgz > mongo.tgz
$ tar xzf mongo.tgz
```

Replace *x.y.z* with the current stable version.

64-bit binaries

```
$ curl http://downloads.mongodb.org/osx/mongodb-osx-x86_64-x.y.z.tgz > mongo.tgz
$ tar xzf mongo.tgz
```

Replace *x.y.z* with the current stable version.

Create a data directory

By default MongoDB will store data in `/data/db`, but it won't automatically create that directory. To create it, do:

```
$ sudo mkdir -p /data/db/
$ sudo chown `id -u` /data/db
```

You can also tell MongoDB to use a different data directory, with the `--dbpath` option.

Run and connect to the server

First, start the MongoDB server in one terminal:

```
$ ./mongodb-xxxxxxx/bin/mongod
```

In a separate terminal, start the shell, which will connect to localhost by default:

```
$ ./mongodb-xxxxxxx/bin/mongo
> db.foo.save( { a : 1 } )
> db.foo.find()
```

Congratulations, you've just saved and retrieved your first document with MongoDB!

Learn more

Once you have MongoDB installed and running, head over to the Tutorial.

Quickstart Unix

Consider one of the following installation tutorials from the new Manual for more specific instructions for getting started with MongoDB.

- Install MongoDB on RedHat Enterprise Linux, CentOS, or Fedora Linux
- Install MongoDB on Debian, Ubuntu or other Linux Systems
- Install MongoDB on other Unix/Linux Systems
- Install MongoDB on OS X

Alternately, you may continue reading this document for installation instructions.

Download

If you are running an old version of Linux and the database doesn't start, or gives a floating point exception, try the "legacy static" version on the Downloads page instead of the versions listed below.

Via package manager

Ubuntu and Debian users can now install nightly snapshots via apt. See Ubuntu and Debian packages for details.

CentOS and Fedora users should head to the CentOS and Fedora Packages page.

32-bit Linux binaries

Note: 64 bit is recommended.

```
$ # replace "1.6.4" in the url below with the version you want
$ curl http://downloads.mongodb.org/linux/mongodb-linux-i686-1.6.4.tgz > mongo.tgz
$ tar xzf mongo.tgz
```

64-bit Linux binaries

```
$ # replace "1.6.4" in the url below with the version you want
$ curl http://downloads.mongodb.org/linux/mongodb-linux-x86_64-1.6.4.tgz > mongo.tgz
$ tar xzf mongo.tgz
```

Other Unixes

See the Downloads page for some binaries, and also the Building page for information on building from source.

Create a data directory

By default MongoDB will store data in `/data/db`, but it won't automatically create that directory. To create it, do:

```
$ sudo mkdir -p /data/db/
$ sudo chown `id -u` /data/db
```

You can also tell MongoDB to use a different data directory, with the `--dbpath` option.

Run and connect to the server

First, start the MongoDB server in one terminal:

```
$ ./mongodb-xxxxxxx/bin/mongod
```

In a separate terminal, start the shell, which will connect to localhost by default:

```
$ ./mongodb-xxxxxxx/bin/mongo
> db.foo.save( { a : 1 } )
> db.foo.find()
```

Congratulations, you've just saved and retrieved your first document with MongoDB!

Next : Tutorial

Quickstart Windows

Download

The easiest (and recommended) way to install MongoDB is to use the pre-built binaries. Note: 64-bit is recommended, although you must have a 64-bit version of Windows to run that version.

32-bit binaries

Download and extract the 32-bit .zip. The "Production" build is recommended.

64-bit binaries

Download and extract the 64-bit .zip.

Unzip

Unzip the downloaded binary package to the location of your choice. You may want to rename mongo-xxxxxxx to just "mongo" for convenience.

Create a data directory

By default MongoDB will store data in `\data\db`, but it won't automatically create that folder, so we do so here:

```
C:\> mkdir \data
C:\> mkdir \data\db
```

Or you can do this from the Windows Explorer, of course.

If you prefer to place datafiles elsewhere, use the `--dbpath` command line parameter when starting mongod.exe.

Run and connect to the server

The important binaries for a first run are:

- `mongod.exe` - the database server. Try `mongod --help` to see startup options.
- `mongo.exe` - the administrative shell

To run the database, click `mongod.exe` in Explorer, or run it from a CMD window.

```
C:\> cd \my_mongo_dir\bin
C:\my_mongo_dir\bin> mongod
```

This will start the mongod.exe server running, and you will see log messages displayed in this window as the server runs. Note: It is also possible to run the server as a Windows Service. But we can do that later.

Now, start the administrative shell, either by double-clicking `mongo.exe` in Explorer, or from a new CMD prompt. By default mongo.exe connects to a `mongod` server running on `localhost` and uses the database named `test`. Run `mongo --help` to see other options.

```
C:\> cd \my_mongo_dir\bin
C:\my_mongo_dir\bin> mongo
> // the mongo shell is a javascript shell connected to the db
> // by default it connects to database 'test' at localhost
> 3+3
6
> db
test
> // the first write will create the db:
> db.foo.insert( { a : 1 } )
> db.foo.find()
{ _id : ..., a : 1 }
> show dbs
...
> show collections
...
> help
```

Congratulations, you've just saved and retrieved your first document with MongoDB!

Writing Apps

You can write apps that use MongoDB in virtually any programming language. See the Drivers page for a full list, and also the C# page if writing .NET applications.

Learn more

- The MongoDB Tutorial (not Windows specific)
- Main MongoDB Windows Doc Page
- [More on using the mongo shell]

Downloads

	OS X 32-bit note	OS X 64-bit	Linux 32-bit note	Linux 64-bit	Windows 32-bit note	Windows 64-bit	Solaris i86pc note	Solaris 64	Source
Production Release (Recommended)									
1.8.1 4/6/2011 Changelog Release Notes	download	download	download *legacy-static	download *legacy-static	download	download	download	download	tgz zip
Nightly Changelog	download	download	download *legacy-static	download *legacy-static	download	download	download	download	tgz zip
Previous Release									
1.6.5 12/9/2010 Changelog Release Notes	download	download	download *legacy-static	download *legacy-static	download	download	download	download	tgz zip
Nightly Changelog	download	download	download *legacy-static	download *legacy-static	download	download	download	download	tgz zip
Development Release (Unstable)									
Nightly Changelog	download	download	download *legacy-static	download *legacy-static	download	download	download	download	tgz zip
Archived Releases									
	list	list	list	list	list	list	list	list	list

2.0 Release Notes

Upgrading

Although the major version number has changed, MongoDB 2.0 is a standard, incremental production release and works as a drop-in replacement for MongoDB 1.8. However, there are a few changes you must be aware of before attempting to upgrade:

1. If you create new indexes in 2.0, then downgrading to 1.8 is possible but reindexing the new collections will be required.
2. `mongoimport` and `mongoexport` now correctly adhere to the CSV spec for handling CSV input/output. This may break existing import/export workflows if they relied on the broken behavior. For more information see the related JIRA case.
3. Journaling is **enabled by default** in 2.0 for 64-bit builds. If you still prefer to run without journaling, start `mongod` with the `--nojournal` option. Otherwise, the journal files will be created on startup. The first time you start `mongod` with journaling, you will see a delay while the new files are being created. In addition, you may see reduced write throughput.

- 2.0 processes can talk to 1.8 processes and vice versa, so you can upgrade various parts of a cluster in any order.

- To upgrade a standalone server. Shutdown the old `mongod` and then restart with the new `mongod` binary. You can download the v2.0 binaries from the MongoDB Download Page.

- To upgrade a replica set. Upgrade the secondaries first one at a time, then stepDown the primary and upgrade the primary. Using the stepDown command is better than simply shutting it down since the failover will happen quicker. To avoid losing the last few updates on failover you can temporarily halt your application (failover should take less than 10 seconds) or change your application code to confirm that each update reaches multiple servers. Note, after upgrading to 2.0 you can use the shutdown command to shutdown the primary without losing any *safe* updates.

- To upgrade a sharded cluster. Upgrade config servers one at a time, in any order. Since config servers use two phase commit, shard configuration metadata updates will halt until all are up and running. `mongos` routers can be upgraded in any order.

What's New

Compact Command

A compact command is now available for compacting a single collection and its indexes. Previously, the only way to compact was to repair the entire database.

Concurrency Improvements

When going to disk, the server will yield the write lock if the data being acted upon isn't likely to be in memory. The initial implementation of this feature now exists: (SERVER-2563)

The specific operations yield in 2.0 are:

- Updates by _id
- Removes
- Long cursor iterations

Default Stack Size

The default stack size has been reduced. This can reduce total memory usage when there are many (e.g., 1000+) client connections, as there is a thread per connection. While portions of a thread's stack can be swapped out if unused, some operating systems do this slowly enough that it might be an issue. The stack size will be set to the lesser of the system setting or 1MB.

Index Performance Enhancements

v2.0 includes significant improvements to the index structures. Indexes are often 25% smaller and 25% faster (depends on the use case). When upgrading from previous versions, the benefits of the new index type are realized only if you create a new index or re-index an old one.

Dates are now signed, and the max index key size has increased slightly from 819 to 1024 bytes.

Once you create new indexes, downgrading to 1.8.x will require a re-index of any indexes created using 2.0.

Sharding Authentication

Authentication can now be used with sharded clusters.

Replica Sets

Priorities

Each replica set node can now have a priority value consisting of a floating-point from 0 to 1000, inclusive. Priorities let you control which member of the set you prefer to have as primary: the member with the highest priority that can see a majority of the set will be elected primary.

For example, suppose we have a replica set with three nodes:

Host	Priority
A	2
B	3
C	1

During normal operation, B will always be chosen as primary. If B goes down, A will be elected primary.

See the priorities documentation for more information.

Data-center awareness

You can now "tag" replica set members to indicate their location. You can use these tags to design custom write rules across data centers, racks, specific servers, or any other architecture choice.

For example, a DBA could define rules such as "very important write" or "customerData" or "audit-trail" to be replicated to certain servers, racks, data centers, etc. Then in the application code, the developer would say:

```
> db.foo.insert(doc, {w : "very important write"})
```

which would succeed if it fulfilled the conditions the DBA defined for "very important write".

See the tagging documentation for more information.

Your driver may also support tag-aware reads. Instead of simply specifying `slaveOk`, you specify slaveOk with tags indicating which data-centers you want to read from. See your driver for details.

w : "majority"

You can also set w to `"majority"` to ensure that a write has been propagated to a majority of nodes, effectively committing it. The value for `"majority"` will automatically be adjusted as you add or remove nodes from the set.

See the majority docs for more information.

Reconfiguration with a minority up

If the majority of servers in a set has been permanently lost, you can now force a reconfiguration of the set to bring it back online.

See more information on Reconfiguring a replica set when members are down.

Primary checks for a caught up secondary before stepping down

To minimize time without a primary, the `stepDown` command will now fail if the primary does not see a secondary within 10 seconds of its latest optime. You can force the primary to step down anyway, but by default it will return an error message.

See also Forcing a Member to be Primary.

Extended shutdown on the primary to minimize interruption

When you call the shutdown command the primary will refuse to shut down unless there is a secondary whose optime is within 10 seconds of the primary. If such a secondary isn't available, the primary will step down and wait up to a minute for the secondary to be fully caught up before shutting down.

Note that to get this behavior, you must issue the `shutdown` command explicitly; sending a signal to the process will not trigger this behavior.

You can also force the primary to shut down, even without an up-to-date secondary available.

Maintenance Mode

When `repair` or `compact` is run on a secondary, the secondary will automatically drop into "recovering" mode until the operation is finished. This prevents clients from trying to read from it while it's busy.

Geospatial Features

Multi-location documents

Indexing is now supported on documents which have multiple location objects, embedded either inline or in nested sub-documents. Additional command options are also supported, allowing results to be returned with not only distance but the location used to generate the distance.

See the Geospatial documentation for further information.

Polygon searches

Polygonal `$within` queries are also now supported for simple polygon shapes. Details are here.

Journaling enhancements

- Journaling is now enabled by default for 64-bit platforms. Use the `--nojournal` command line option to disable it.
- The journal is now compressed for faster commits to disk.
- A new `--journalCommitInterval` command line option exists for specifying your own group commit interval. 100ms is the default (same as in 1.8).
- A new getLastError {`j: true`} option is available to wait for the group commit. The group commit will happen sooner when a client is waiting on {`j: true`}. If journaling is disabled, {`j: true`} is a no-op.

New `ContinueOnError` option for bulk insert

When this flag is set (see your driver on how to set it), bulk insert will continue to insert any remaining documents even if an insert fails (due, for example, to a duplicate key). The getLastError command will report whether any doc inserts have failed (not just last one). If multiple errors occur, only the most recent will be reported by getLastError. See OP_INSERT.

Map Reduce

Output to a sharded collection

Using the new flag "sharded" it is possible to send the result of a map/reduce to a sharded collection. Combined with the "reduce" or "merge" flags, it is possible to keep adding data to very large collections from map/reduce jobs. See documentation of output options.

Performance improvements

Map/reduce performance will benefit from the following:

- larger in-memory buffer sizes, reducing the amount of disk I/O needed during a job
- larger javascript heap size, allowing for larger objects and less GC
- supports pure JS execution with the jsMode flag.

New Querying Features

Additional regex options: s

Allows the dot (.) to match all characters including new lines. This is in addition to the currently supported i, m and x. See Using regular expressions in queries.

$and

A special boolean `$and` query operator is now available.

Command output changes

The output of the Validate Command and the documents in the system.profile collection have both been enhanced to return information as BSON objects with keys for each value rather than as free-form strings.

Shell features

Custom prompt

You can define a custom prompt for the `mongo` shell. You can change the prompt at any time by setting the `prompt` variable to a string or a custom JavaScript function returning a string. See the documentation for examples.

Default shell init script

On startup, the shell will check for a `.mongorc.js` file in the user's home directory. The shell will execute this file after connecting to the database and before displaying the prompt.

If you would like the shell **not** to run the `.mongorc.js` file automatically, start the shell with `--norc`.

See .mongorc.js documentation.

See Also

- Download MongoDB
- Full list of changes on JIRA
- All backward incompatible changes

1.8 Release Notes

MongoDB 1.8 is a drop-in replacement for 1.6, except:

- *replica set nodes should be upgraded in a particular order*.

- The options to the Map/Reduce command have changed in 1.8, causing incompatibility with previous releases. If you use MapReduce, please see Map Reduce Output Options page. If using map/reduce, this likely also means you need a recent version of your client driver.

To upgrade, shutdown the old `mongod` and the restart with the new binaries. See Upgrading to 1.8.x for more details.

MongoDB Download Page

What's New

Journaling

MongoDB now supports write-ahead journaling to facilitate fast crash recovery and durability in the storage engine. With journaling enabled, a `mongod` can be quickly restarted following a crash without needing to repair the collections.

- Journaling Documentation

Sparse and Covered Indexes

Sparse Indexes are indexes that only include documents that contain the fields specified in the index. Documents missing the field will not appear in the index at all. This can significantly reduce index size for attributes that are contained in a subset of documents within a collection.

Covered Indexes enable queries to be answered entirely from the index when the query only requests fields that are contained in the index.

- Sparse Index Documentation
- Covered Index Documentation

Incremental Map/Reduce Support

Map/Reduce supports new output options that enable incrementally updating existing collections. Previously, a Map/Reduce job could either output to a temporary collection a named permanent collection; which it would overwrite with new data.

You now have several options for the output of your map/reduce jobs

- You can merge Map/Reduce output into an existing collection. Output from the Reduce phase will replace existing keys in the output collection if it already exists. Other keys will remain in the collection.
- You can now re-reduce your output with the contents of an existing collection. Each key output by the reduce phase will be reduced with the existing document in the output collection.
- You can replace the existing output collection with the new results of the Map/Reduce job (equivalent to setting a permanent output collection in previous releases)
- You can compute map/reduce inline and return results to the caller without persisting the results of the job. This is similar to the temporary collections generated in previous releases, except results are limited to 8MB.

The new map/reduce options are documented here.

Additional Changes and Enhancements

1.8.1

- sharding migrate fix when moving larger chunks
- durability fix with background indexing
- fixed mongos concurrency issue with many incoming connections

1.8.0

- All changes from 1.7.x series.

1.7.6

- bug fixes

1.7.5

- journaling
- extent allocation improvements
- improved replica set connectivity for mongos
- getLastError improvements for sharding

1.7.4

- mongos will route SLAVE_OK queries to secondaries in replica sets
- new map/reduce output options
- sparse indexes

1.7.3

- initial covered index support
- distinct can use data from indexes when possible
- map/reduce can merge or reduce results into an existing collection
- mongod tracks and mongostat displays network usage
- sharding stability improvements

1.7.2

- $rename operator to allow renaming of attributes in a document
- option to `db.eval` not to block
- geo queries with sharding
- mongostat --discover option
- chunk splitting enhancements
- replica sets network enhancements for servers behind a nat

1.7.1

- Many sharding performance enhancements
- better support for $elemMatch on primitives in embedded arrays
- Query optimizer enhancements on range queries
- Window service enhancements
- Replica set setup improvements
- $pull works on primitives in arrays

1.7.0

- sharding performance improvements for heavy insert loads
- slave delay support for replica sets
- getLastErrorDefaults for replica sets
- auto completion in the shell
- spherical distance for geo search
- all fixes from 1.6.1 and 1.6.2

Release Announcement Forum Pages

- 1.8.1 1.8.0

- 1.7.6 1.7.5 1.7.4 1.7.3 1.7.2 1.7.1 1.7.0

See Also

- Download MongoDB v1.8
- Upgrading to 1.8.x
- Full list of changes on jira

Upgrading to 1.8

First, upgrade your shell (`mongo`) to the 1.8.x shell.

Upgrading Replica Sets

1.8.x secondaries **can** replicate from 1.6.x primaries.

1.6.x secondaries **cannot** replicate from 1.8.x primaries.

Thus, the trick is to replace all of your secondaries, then the primary.

For example, suppose we have a typical replica set with 1 primary, 1 secondary, and 1 arbiter. To upgrade this set, do the following:

1. For each arbiter:
    - Shut down the arbiter
    - Start it back up with the 1.8 binary
2. Change your config (optional)
   It is possible that, when you start shutting down members of the set, a new primary will be elected. If you wish to prevent this, you can give all of the slaves a priority of 0 before upgrading, then change them back afterwards.
    - Record your current config. Run rs.conf() and paste the results into a text file.
    - Update your config so that all secondaries have priority 0. For example:

```
> config = rs.conf()
{
     "_id" : "foo",
     "version" : 3,
     "members" : [
             {
                     "_id" : 0,
                     "host" : "ubuntu:27017"
             },
             {
                     "_id" : 1,
                     "host" : "ubuntu:27018"
             },
             {
                     "_id" : 2,
                     "host" : "ubuntu:27019",
                     "arbiterOnly" : true
             }
             {
                     "_id" : 3,
                     "host" : "ubuntu:27020"
             },
             {
                     "_id" : 4,
                     "host" : "ubuntu:27021"
             },
     ]
}
> config.version++
3
> rs.isMaster()
{
     "setName" : "foo",
     "ismaster" : false,
     "secondary" : true,
     "hosts" : [
             "ubuntu:27017",
             "ubuntu:27018"
     ],
     "arbiters" : [
             "ubuntu:27019"
     ],
     "primary" : "ubuntu:27018",
     "ok" : 1
}
> // for each slave
> config.members[0].priority = 0
> config.members[3].priority = 0
> config.members[4].priority = 0
> rs.reconfig(config)
```

3. For each slave:
    - Shut down the slave
    - Start it back up with the 1.8 binary
4. If you changed the config, change it back to its original state

```
> config = rs.conf()
> config.version++
> config.members[0].priority = 1
> config.members[3].priority = 1
> config.members[4].priority = 1
> rs.reconfig(config)
```

5. Shut down the primary (the final 1.6 server) and restart it with the 1.8 binary.

Upgrading Sharded Clusters

1. Turn off the balancer:

```
$ mongo <a_mongos_hostname>
> use config
> db.settings.update({_id:"balancer"},{$set : {stopped:true}}, true)
```

2. For each shard:
    - If the shard is a replica set, follow the directions above for replica sets.
    - If the shard is a single mongod process, shut it down and start it back up with the 1.8 binary.
3. For each `mongos`:
    - Shut down the mongos process
    - Restart with the 1.8 binary
4. For each config server:
    - Shut down the config server process
    - Restart with the 1.8 binary
5. Turn on the balancer

```
> use config
> db.settings.update({_id:"balancer"},{$set : {stopped:false}})
```

Returning to 1.6

If something goes wrong and you wish to move back to 1.6, follow the steps above in reverse. Please be careful that you have not inserted any documents larger than 4MB while running on 1.8 (where the max size has increased to 16MB); if you have you will get errors when the server tries to read those documents.

Journaling

Returning to 1.6 after using 1.8 journaling works fine, as journaling does not change anything about the data file format. Suppose you are running 1.8.0 with journaling enabled and something isn't working for you, so you decide to switch back to 1.6. There are two scenarios:

1. If you shut down cleanly with 1.8.x, just restart with the 1.6 mongod binary.
2. If 1.8.x shut down uncleanly, start 1.8.x up again and let the journal files run to fix any damage (incomplete writes) that may have existed at the crash. Then shut down 1.8.0 cleanly and restart with the 1.6 mongod binary.

See Also

- 1.8 Release Notes page for details on changes in v1.8 to map/reduce.

Download

- Download v1.8

1.6 Release Notes

MongoDB 1.6 is a drop-in replacement for 1.4. To upgrade, simply shutdown `mongod` then restart with the new binaries.*

> ⚠ MongoDB v1.8 is now available.

** Please note that you should upgrade to the latest version of whichever driver you're using. Certain drivers, including the Ruby driver, will require the upgrade, and all the drivers will provide extra features for connecting to replica sets.*

Sharding

Sharding is now production-ready, making MongoDB horizontally scalable, with no single point of failure. A single instance of *mongod* can now be upgraded to a distributed cluster with zero downtime when the need arises.

- Sharding Tutorial
- Sharding Documentation
- Upgrading a Single Server to a Cluster

Replica Sets

Replica sets, which provide automated failover among a cluster of *n* nodes, are also now available.

Plese note that replica *pairs* are now deprecated; we strongly recommend that replica pair users upgrade to replica sets.

- Replica Set Tutorial
- Replica Set Documentation
- Upgrading Existing Setups to Replica Sets

Other Improvements

- The w option (and wtimeout) forces writes to be propagated to *n* servers before returning success (this works especially well with replica sets)
- $or queries
- Improved concurrency
- $slice operator for returning subsets of arrays
- 64 indexes per collection (formerly 40 indexes per collection)
- 64-bit integers can now be represented in the shell using NumberLong
- The findAndModify command now supports upserts. It also allows you to specify fields to return
- $showDiskLoc option to see disk location of a document
- Support for IPv6 and UNIX domain sockets

Installation

- Windows service improvements
- The C++ client is a separate tarball from the binaries

1.6.x Release Notes

- 1.6.5

1.5.x Release Notes

- 1.5.8
- 1.5.7
- 1.5.6
- 1.5.5
- 1.5.4
- 1.5.3
- 1.5.2
- 1.5.1
- 1.5.0

You can see a full list of all changes on Jira.

Thank you everyone for your support and suggestions!

1.4 Release Notes

We're pleased to announce the 1.4 release of MongoDB. 1.4 is a drop in replacement for 1.2. To upgrade you just need to shutdown mongod, then restart with the new binaries. (Users upgrading from release 1.0 should review the 1.2 release notes, in particular the instructions for upgrading the DB format.)

Release 1.4 includes the following improvements over release 1.2:

Core server enhancements

- concurrency improvements
- indexing memory improvements
- background index creation
- better detection of regular expressions so the index can be used in more cases

Replication & Sharding

- better handling for restarting slaves offline for a while
- fast new slaves from snapshots (--fastsync)
- configurable slave delay (--slavedelay)

* replication handles clock skew on master
* $inc replication fixes
* sharding alpha 3 - notably 2 phase commit on config servers

Deployment & production

* configure "slow threshold" for profiling
* ability to do fsync + lock for backing up raw files
* option for separate directory per db (--directoryperdb)
* `http://localhost:28017/_status` to get serverStatus via http
* REST interface is off by default for security (--rest to enable)
* can rotate logs with a db command, logRotate
* enhancements to serverStatus command (db.serverStatus()) - counters and replication lag stats
* new mongostat tool

Query language improvements

* $all with regex
* $not
* partial matching of array elements $elemMatch
* $ operator for updating arrays
* $addToSet
* $unset
* $pull supports object matching
* $set with array indices

Geo

* 2d geospatial search
* geo $center and $box searches

1.2.x Release Notes

New Features

* More indexes per collection
* Faster index creation
* Map/Reduce
* Stored JavaScript functions
* Configurable fsync time
* Several small features and fixes

DB Upgrade Required

There are some changes that will require doing an upgrade if your previous version is <= 1.0.x. If you're already using a version >= 1.1.x then these changes aren't required. There are 2 ways to do it:

* --upgrade
    * stop your mongod process
    * run ./mongod --upgrade
    * start mongod again
* use a slave
    * start a slave on a different port and data directory
    * when its synced, shut down the master, and start the new slave on the regular port.

Ask in the forums or IRC for more help.

Replication Changes

* There have been minor changes in replication. If you are upgrading a master/slave setup from <= 1.1.2 you have to update the slave first.

mongoimport

* mongoimportjson has been removed and is replaced with mongoimport that can do json/csv/tsv

field filter changing

* We've changed the semantics of the field filter a little bit. Previously only objects with those fields would be returned. Now the field filter

only changes the output, not which objects are returned. If you need that behavior, you can use $exists

other notes

http://www.mongodb.org/display/DOCS/1.1+Development+Cycle

1.0 Changelist

Wrote MongoDB. See documentation.

Version Numbers

MongoDB uses the odd-numbered versions for development releases.

There are 3 numbers in a MongoDB version: *A.B.C*

- *A* is the major version. This will rarely change and signify very large changes
- *B* is the release number. This will include many changes including features and things that possible break backwards compatibility. Even Bs will be stable branches, and odd Bs will be development.
- *C* is the revision number and will be used for bugs and security issues.

For example:

- 1.0.0 : first GA release
- 1.0.x : bug fixes to 1.0.x - highly recommended to upgrade, very little risk
- 1.1.x : development release. this will include new features that are not fully finished, and works in progress. Some things may be different than 1.0
- 1.2.x : second GA release. this will be the culmination of the 1.1.x release.

What's New by Version

This is a summary of high level features only. See jira and release notes for full details.

- 1.4
  - Geospatial
  - Background indexing
  - --directoryperdb
  - Log rotate
  - $not
  - $ operator for updating arrays
  - $addToSet
  - $unset
- 1.6
  - Sharding
  - Replica Sets
  - getLastError w param
  - $or
  - $slice
  - 64 indexes per collection
  - IPv6
- 1.8
  - Journaling
  - Sparse and covered indexes
  - $rename
  - mongos (i.e., sharded environment) will route SLAVE_OK queries to secondaries in replica sets

Ubuntu and Debian packages

> Please read the notes on the Downloads page.

10gen publishes apt-gettable packages. Our packages are generally fresher than those in the Debian or Ubuntu repositories. We publish stable releases in our repository, under the name **mongodb-10gen**, corresponding to the latest stable release.   This package conflicts with the **mongodb** packages in Debian/Ubuntu.

The packaging is still a work-in-progress, so we invite Debian and Ubuntu users to try it out and let us know how the packaging might be improved.

Installing

To use our packages, add a line to your /etc/apt/sources.list, then 'aptitude update' and 'aptitude install mongodb-10gen',. Make sure you add the 10gen GPG key, or apt will disable the repository (apt uses encryption keys to verify the repository is trusted and disables untrusted ones). To add the GPG key, run this command:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
```

Then, if you're on a Debianoid with SysV style init scripts (e.g., Debian Lenny or older Ubuntus), add this line verbatim to your /etc/apt/sources.list

```
deb http://downloads-distro.mongodb.org/repo/debian-sysvinit dist 10gen
```

If you're on a Debianoid with Upstart (e.g., recent Ubuntus), use this line in your sources.list:

```
deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen
```

GPG Key

The public gpg key used for signing these packages follows. It should be possible to import the key into apt's public keyring with a command like this:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
```

Configuration

To configure these packages beyond the defaults, have a look at */etc/mongodb.conf*, and/or the initialization script, (*/etc/init.d/mongodb* on older, non-Upstart systems, */etc/init/mongodb.conf* on Upstart systems). Most MongoDB operational settings are in */etc/mongodb.conf*; a few other settings are in the initialization script. Note that if you customize the *userid* in the initialization script or the *dbpath* or *logpath* settings in */etc/mongodb.conf*, you must ensure that the directories and files you use are writable by the userid you run the server as.

Packages for other distros coming soon!

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.10 (Darwin)


mQENBEtsQe8BCACm5G0/ei0IxyjVEp6EEtbEbWK1Q4dKaONtiCODwB8di+L8tlUd
Ra5QYxeyV90C+dqdh34o79enXxT6idHfYYqDdob2/kAPE6vFi4sLmrWIVGCRY++7
RPclZuezPm1sxG1TRAYEsW0VZUE9ofdoQ8xlUZDyn2BSjG8OCT2e4orRg1pHgzw2
n3hnWqJNuJS4jxcRJOxI049THIGUtqBfF8bQoZw8C3Wg/R6pGghUfNjpA6uF9KAH
gnqrC0swZ1/vwIJt9fnvAlzkqLrssYtKH0rMdn5n4g5tJLqY5q/NruHMq2rhoy3r
4MC1w8GTbP7qR83wAyaLJ7xACOKqxB3SrDFJABEBAAG0I1JpY2hhcmQgS3JldXRl
ciA8cmljaGFyZEAxMGdlbi5jb20+iQE4BBMBAgAiBQJLbEHvAhsDBgsJCAcDAgYV
CAIJCgsEFgIDAQIeAQIXgAAKCRCey+xGfwzrEGXbB/4nrmf/2rEnztRelmup3duI
eepzEtw1cv3uHg2oZXGS6S7o5Fsk+amngaWelWKfkSw5La7aH5vL4tKFKUfuaME1
avInDIU/0IEs8jLrdSWq601HowLQcxAhqNPdaGONDtHw56Qhs0Ba8GA6329vLWgZ
ODnXweiNSCDrv3xbIN6IjPyyO5AoUkxmJfD0mVtp3u5Ar7kfIw7ieGGxokaHewNL
Xzqcp9rPiUR6dFw2uRvDdVRrXFUPlgVugaHKytm15JpHmQfyzQiMdYXnIz0oofJO
WM/PY1iw+QJZ2M7PnfbTJeADXIc/EoOAJDRggih533SjhiCaT6FdPMMk6rCZ5cgl
uQENBEtsQe8BCADlNPIJZVSL2i6H9X19YK4CpEqsjIUGISMB1cDT311WFSnhfuMs
GL9xYRb8dlbyeJFFOyHNkIBmH5ekCvGRfS6qJYpcUQZZcWSjEMqBYQV5cw1efd0B
ek64jfvrsLz8+YhKzn+NI8O3nyGvpEEWvOhN4hNjwkDhYbXLvAlsqagbnSMf+Htf
3lgCGYa2gLiNIqNKWCsEVAan/Er6KS39WANGXi6ih0yjReBiU8WR6Qh2ylMi2xKw
yHnTOsbWxP0hqALUa7N4AEGCXS/qn+vUz/hcIbt+eUNy45qoZcTT3dZsWGfJqknh
RFMIuPiej7/WY4Ugzes5NG02ecDkDkpJvrSNABEBAAGJAR8EGAECAAkFAktsQe8C
GwwACgkQnsvsRn8M6xABeggAlNkqbqa12L1bgaCgnGGdCiuXB3F6/VFmSQdUKpts
EuqWH6rSp30r67PupzneX++ouh+9WD5O7gJ0kP3VQJpmXjT/QnN5ANjI4kAtRZUW
qCXlXOxVAeXHL5oiKz0NM23Xc2rNAyfBQY8+SUYrKBalNBq5m68g8oogX8QD5u2F
x+6C+QK9G2EBDD/NWgkKN3GOxpQ5DTdPHI5/fjwYFs1leIaQjjiyJwAifxB/1+w0
VCHe2LDVpRXY5uBTefF2guhVYisKY6n5wNDaQpBmA8w17it5Yp8ge0HMN1A+aZ+6
L6MsuHbG2OYDZgAk8eKhvyd0y/pAhZpNuQ82MMGBmcueSA==
=74Cu
-----END PGP PUBLIC KEY BLOCK-----
```


Install

In order to complete the installation of a package, you need to update the sources and then install the desired package

```
sudo apt-get update
```

```
sudo apt-get install mongodb-10gen
```

CentOS and Fedora Packages

10gen publishes yum-installable RPM packages that for x86 and x86_64 platforms. The packages are named mongo-10gen and mongo-10gen-server for production releases (versions with even middle numbers), and mongo-10gen-unstable and mongo-10gen-unstable-server for development releases (odd middle version numbers, except for some release candidates).

For all 64-bit RPM-based distros with yum, put this at /etc/yum.repos.d/10gen.repo:

```
[10gen]
name=10gen Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64
gpgcheck=0
```

For all 32-bit RPM-based distros with yum, put this at /etc/yum.repos.d/10gen.repo:

```
[10gen]
name=10gen Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/i686
gpgcheck=0
```

Note: for users upgrading from our older (pre-2/2011) packaging scheme, it may be necessary to uninstall your existing "mongo-stable", "mongo-stable-server", "mongo-unstable", "mongo-unstable-server" packages before installing the new mongo-10gen, mongo-10gen-server pacakges.

For the moment, these packages aren't signed. (If anybody knows how to automate signing RPMs, please let us know!)

Drivers

MongoDB currently has client support for the following programming languages:

mongodb.org Supported

- C
- C++
- Erlang
- Haskell
- Java
- Javascript
- .NET (C# F#, PowerShell, etc)
- Node.js
- Perl
- PHP
- Python
- Ruby
- Scala

Community Supported

- ActionScript3
    - http://github.com/argoncloud
- C
    - libmongo-client
- C# and .NET
- Clojure
    - See the Java Language Center
- ColdFusion
    - cfmongodb
    - Blog post: Part 1 | Part 2 | Part 3
    - http://github.com/virtix/cfmongodb/tree/0.9
    - http://mongocfc.riaforge.org/
- D
    - Port of the MongoDB C Driver for D
- Dart
    - https://bitbucket.org/vadimtsushko/mongo-dart
- Delphi
    - mongo-delphi-driver - Full featured Deplhi interface to MongoDB built on top of the mongodb.org supported C driver
    - pebongo - Early stage Delphi driver for MongoDB
    - TMongoWire - Maps all the VarTypes of OleVariant to the BSON types, implements IPersistStream for (de)serialization, and uses TTcpClient for networking
- Entity
    - entity driver for mongodb on Google Code, included within the standard Entity Library
- Erlang
    - emongo - An Erlang MongoDB driver that emphasizes speed and stability. "The most emo of drivers."
    - Erlmongo - an almost complete MongoDB driver implementation in Erlang
- Factor
    - http://github.com/slavapestov/factor/tree/master/extra/mongodb/
- Fantom
    - http://bitbucket.org/liamstask/fantomongo/wiki/Home
- F#
    - http://gist.github.com/218388
- Go
    - gomongo
    - go-mongo
    - mgo
    - mongogo
- Groovy
    - gmongo
    - Also see the Java Language Center
      **Blog Post: Groovy on Grails in the land of MongoDB
- Javascript
- Lisp

    - https://github.com/fons/cl-mongo
- Lua
    - LuaMongo on Google Code
    - LuaMongo fork on Github
- MatLab
    - mongo-matlab-driver
- node.js
- Objective C
    - NuMongoDB
- Opa
    - Opa Standard Library MongoDB Driver
- PHP
    - Asynchronous PHP driver using libevent
- PowerShell
    - mosh Powershell provider for MongoDB
    - mdbc module cmdlets using official 10gen driver
    - Doug Finke's blog post on using the original community C# driver with PowerShell
- Prolog
    - https://github.com/khueue/prolongo
- R
    - rmongodb - Full featured R interface to MongoDB built on top of the mongodb.org supported C driver
    - RMongo - R client to interface with MongoDB
- REST
- Ruby
    - MongoMapper
    - rmongo - An event-machine-based Ruby driver for MongoDB
    - jmongo A thin ruby wrapper around the mongo-java-driver for vastly better jruby performance.
    - em-mongo EventMachine MongoDB Driver (based off of RMongo).
- Scala
    - See the Java Language Center
- Racket (PLT Scheme)
    - http://planet.plt-scheme.org/display.ss?package=mongodb.plt&owner=jaymccarthy
    - docs
- Smalltalk
    - Squeaksource Mongotalk
    - Dolphin Smalltalk

Get Involved, Write a Driver!

- Writing Drivers and Tools

Hadoop

https://github.com/mongodb/mongo-hadoop

http://blog.10gen.com/post/20840407875/mongodb-hadoop-connector-announced

Scala Language Center

Casbah Casbah is the officially supported Scala driver for MongoDB. It provides wrappers and extensions to the Java driver meant to allow a more Scala-friendly interface to MongoDB. It supports serialization/deserialization of common Scala types (including collections and regex), Scala 2.8 collection versions of DBObject and DBList and a fluid query DSL.

- API documentation
- Tutorial
- Mailing List
- Java Driver Doc Page

Community

- Lift-MongoDB - Lift Web Framework supports MongoDB, including object mapping via the Record back-end implementation.
- Rogue: A Type-Safe Scala DSL - Foursquare's DSL for querying MongoDB alongside Lift-MongoDB-Record.
    - Tutorial/Intro
    - Source/Downloads
- Blue Eyes is a lightweight framework for building REST APIs with strong MongoDB integration including a DSL and Mock MongoDB for testing.
- mongo-scala-driver is a thin wrapper around mongo-java-driver to make working with MongoDB more Scala-like.

* Wiki
* Mailing list

Webcast - MongoDB and Scala

Haskell Language Center

The Haskell driver and its API documentation reside on Hackage

C Language Center

C Driver

The MongoDB C Driver is the 10gen-supported driver for MongoDB. It's written in pure C.

The driver's core API is stable as of the v0.4 release; however, the GridFS API may change somewhat in the v0.5 release.

* Primary Doc Page
* Tutorial
* C Driver README
* History
* JIRA
* Source Code

Download and build

The C driver is hosted at GitHub. You can download the latest stable version: v0.4:

Then consult the building docs for detailed instructions on building the driver.

CSharp Language Center

MongoDB C# / .NET Driver

The MongoDB C# Driver is the 10gen-supported C# / .NET driver for MongoDB.

* C# Driver Quick-Start
* C# Driver Tutorial
* C# Driver LINQ Tutorial
* C# Driver Serialization Tutorial
* API Documentation
* C# Driver README
* Source Code

> Several other C# drivers have been developed by the community. This is the "official" C# Driver supported by 10gen. It is similar in many ways to the various drivers that came before it, but it is **not** a direct replacement for any of them. Most people have found it easy to convert to using this driver, but you should expect to have to make some changes.

Downloading the Driver

The C# Driver is hosted at github.com. Instructions for downloading the source code are at: Download Instructions

You can also download binary builds in either .msi or .zip formats from:
http://github.com/mongodb/mongo-csharp-driver/downloads.

Note: if you download the .zip file Windows might require you to "Unblock" the help file. If Windows asks "Do you want to open this file?" when you double click on the CSharpDriverDocs.chm file, clear the check box next to "Always ask before opening this file" before pressing the Open button. Alternatively, you can right click on the CSharpDriverDocs.chm file and select Properties, and then press the Unblock button at the bottom of the General tab. If the Unblock button is not present then the help file does not need to be unblocked.

Visual Studio Versions Supported

The current version of the C# Driver has been built and tested using

* Visual Studio 2010

* Visual Studio 2008

Questions and Bug Reports

Questions about the C# driver (or any other MongoDB topic) can be posted at the mongodb-user Google Group:

https://groups.google.com/group/mongodb-user

Bug reports can be created in JIRA at:

https://jira.mongodb.org/browse/CSHARP

See Also

* CSharp Community Projects

Presentations

* C# Development with MongoDB - MongoSF (May 2011)
* More C#-related presentations

CSharp Community Projects

Community Supported C# Drivers

* See also: the 10gen supported MongoDB C# driver

* mongodb-csharp driver
* simple-mongodb driver
* NoRM

Tools

* MongoDB.Emitter Document Wrapper
* log4net appender
* ASP.NET Membership and Role Providers for MongoDB
* ASP.NET User Administration
* MongoCola Administration Tool
* MongoRepository

F#

* F# Example

Community Articles

* Experimenting with MongoDB from C#
* Using MongoDB from C#
* Introduction to MongoDB for .NET
* Using Json.NET and Castle Dynamic Proxy with MongoDB
* Implementing a Blog Using ASP.NET MVC and MongoDB
* Intro Article using a Post and Comments Example
* Using the 10gen .NET driver from PowerShell
* Tutorial MongoDB con ASP.NET MVC - Ejemplo Práctico

Support

* http://groups.google.com/group/mongodb-csharp
* http://groups.google.com/group/mongodb-user
* IRC: #mongodb on freenode

See Also

* C++ Language Center

CSharp Driver LINQ Tutorial

Introduction

This tutorial covers the support for LINQ queries added in the 1.4 release of the C# driver.

You should already have read at least the quickstart introduction to the C# driver.

http://www.mongodb.org/display/DOCS/CSharp+Driver+Quickstart

Quickstart

First, add the following additional using statement to your program:

```
using MongoDB.Driver.Linq;
```

Then, get a reference to a collection variable in the usual way:

```
var collection = database.GetCollection<TDocument>("collectionname");
```

The basic idea behind writing a LINQ query is to start from a collection variable and begin the LINQ query by calling the AsQueryable<TDocument>() method. After that it's all standard LINQ.

For example:

```
var query =
    from e in collection.AsQueryable<Employee>()
    where e.FirstName == "John"
    select e;

foreach (var employee in query)
{
    // process employees named "John"
}
```

You can also write queries using lambda syntax. The previous query would be written using lambda syntax like this:

```
var query =
    collection.AsQueryable<Employee>()
    .Where(e => e.FirstName == "John")
    .Select(e => e); // this trivial projection is optional when using lambda syntax
```

The C# compiler translates all queries written using query syntax into lambda syntax internally anyway, so there is no performance advantage or penalty to choosing either style. You can also mix and match the styles, which can be useful when using query operators that are not supported by the query syntax.

All the code samples in this tutorial show both the query syntax and the lambda syntax for each query operator and supported where clauses.

Only LINQ queries that can be translated to an equivalent MongoDB query are supported. If you write a LINQ query that can't be translated you will get a runtime exception and the error message will indicate which part of the query wasn't supported.

Note: The 1.4 version of the C# driver requires that all where clauses that compare a field or property against a value have the constant on the right hand side. This restriction will be lifted in the next release.

Supported LINQ query operators

This section documents the supported LINQ query operators.

Any

Without a predicate Any just tests whether the collection has any documents.

```
var result =
    (from c in collection.AsQueryable<C>()
    select c)
    .Any();
// or
var result =
    collection.AsQueryable<C>()
    .Any();
```

Any (with predicate)

With a predicate Any tests whether the collection has any matching documents.

```
var result =
    (from c in collection.AsQueryable<C>()
    select c)
    .Any(c => c.X == 1);
// or
var result =
    collection.AsQueryable<C>()
    .Any(c => c.X == 1);
```

Note that the predicate can be provided either by a where clause or as an argument to Any, so the following are equivalent to the previous query.

```
var result =
    (from c in collection.AsQueryable<C>()
    where c.X == 1
    select c)
    .Any();
// or
var result =
    collection.AsQueryable<C>()
    .Where(c => c.X == 1)
    .Any();
```

Any with a predicate is not supported after a projection (at least not yet). So the following is not valid:

```
var result =
    collection.AsQueryable<C>()
    .Select(c => c.X)
    .Any(x => x == 1);
```

You can usually rewrite such a query by putting an equivalent where clause before the projection (in which case you can drop the projection).

Count

Without a predicate Count just returns the number of documents in the collection.

```
var result =
    (from c in collection.AsQueryable<C>()
    select c)
    .Count();
// or
var result =
    collection.AsQueryable<C>()
    .Count();
```

Count (with predicate)

With a predicate Count returns the number of documents that match the predicate.

```
var result =
    (from c in collection.AsQueryable<C>()
    select c)
    .Count(c => c.X == 1);
// or
var result =
    collection.AsQueryable<C>()
    .Count(c => c.X == 1);
```

Note that the predicate can be provided either by a where clause or as an argument to Count , so the following are equivalent to the previous query.

```
var result =
    (from c in collection.AsQueryable<C>()
    where c.X == 1
    select c)
    .Count();
// or
var result =
    collection.AsQueryable<C>()
    .Where(c => c.X == 1)
    .Count();
```

Count with a predicate is not supported after a projection (at least not yet). So the following is not valid:

```
var result =
    collection.AsQueryable<C>()
    .Select(c => c.X)
    .Count(x => x == 1);
```

You can usually rewrite such a query by putting an equivalent where clause before the projection (in which case you can drop the projection).

Distinct

Distinct returns the unique values of a field or property of the documents in the collection. You use a projection to identify the field or property whose distinct values you want.

```
var result =
    (from c in collection.AsQueryable<C>()
    select c.X)
    .Distinct();
// or
var result =
    collection.AsQueryable<C>()
    .Select(c => c.X)
    .Distinct();
```

The projection must select a particular field or property of the document. If the value of that field or property is represented in MongoDB as an array you can also use array indexing to select an item from the array.

```
var result =
    (from c in collection.AsQueryable<C>()
    select c.A[i])
    .Distinct();
// or
var result =
    collection.AsQueryable<C>()
    .Select(c => c.A[i])
    .Distinct();
```

ElementAt

ElementAt returns a particular document from a result set. Often you will combine this with a sort order.

```
var result =
    (from c in collection.AsQueryable<C>()
    where c.X > 0
    orderby c.X
    select c)
    .ElementAt(index);
// or
var result =
    collection.AsQueryable<C>()
    .Where(c => c.X > 0)
    .OrderBy(c => c.X)
    .ElementAt(index);
```

If the result set has fewer documents than index ElementAt throws an exception.

ElementAtOrDefault

ElementAtOrDefault is just like ElementAt except that if there are fewer documents than index it returns null instead of throwing an exception.

First

First returns the first document from a result set. Often you will combine this with a sort order.

```
var result =
    (from c in collection.AsQueryable<C>()
    where c.X > 0
    orderby c.X
    select c)
    .First();
// or
var result =
    collection.AsQueryable<C>()
    .Where(c => c.X > 0)
    .OrderBy(c => c.X)
    .First();
```

If the result set has no documents First throws an exception.

First (with predicate)

This overload of First allows you to provide a predicate as an argument to First. This is an alternative to using a where clause.

```
var result =
    (from c in collection.AsQueryable<C>()
    orderby c.X
    select c)
    .First(c => c.X > 0);
// or
var result =
    collection.AsQueryable<C>()
    .OrderBy(c => c.X)
    .First(c => c.X > 0);
```

First with a predicate is not supported after a projection (at least not yet). So the following is not valid:

```
var result =
    collection.AsQueryable<C>()
    .OrderBy(c => c.X)
    .Select(c => c.X)
    .Count(x => x > 0);
```

You can usually rewrite such a query by putting an equivalent where clause before the projection.

If the result set has no documents First with a predicate throws an exception.

FirstOrDefault

FirstOrDefault is just like First except that if there are no matching documents it returns null instead of throwing an exception.

FirstOrDefault (with predicate)

FirstOrDefault with a predicate is just like First with a predicate except that if there are no matching documents it returns null instead of throwing an exception.

Last

Last returns the last document from a result set. Often you will combine this with a sort order.

```
var result =
    (from c in collection.AsQueryable<C>()
    where c.X > 0
    orderby c.X
    select c)
    .Last();
// or
var result =
    collection.AsQueryable<C>()
    .Where(c => c.X > 0)
    .OrderBy(c => c.X)
    .Last();
```

If the result set has no documents Last throws an exception.

Last (with predicate)

This overload of Last allows you to provide a predicate as an argument to Last. This is an alternative to using a where clause.

```
var result =
    (from c in collection.AsQueryable<C>()
    orderby c.X
    select c)
    .Last(c => c.X > 0);
// or
var result =
    collection.AsQueryable<C>()
    .OrderBy(c => c.X)
    .Last(c => c.X > 0);
```

Last with a predicate is not supported after a projection (at least not yet). So the following is not valid:

```
var result =
    collection.AsQueryable<C>()
    .OrderBy(c => c.X)
    .Select(c => c.X)
    .Last(x => x > 0);
```

You can usually rewrite such a query by putting an equivalent where clause before the projection.

If the result set has no documents Last throws an exception.

LastOrDefault

LastOrDefault is just like Last except that if there are no matching documents it returns null instead of throwing an exception.

LastOrDefault (with predicate)

LastOrDefault with a predicate is just like Last with a predicate except that if there are no matching documents it returns null instead of throwing an exception.

LongCount

LongCount is just like Count except that the return value is a 64-bit integer instead of a 32-bit integer.

LongCount (with predicate)

LongCount with a predicate is just like Count with a predicate except that the return value is a 64-bit integer instead of a 32-bit integer.

Max

Max returns the maximum value of a field or property of the documents in the collection. You use a projection to identify the field or property whose maximum value you want.

```
var result =
    (from c in collection.AsQueryable<C>()
    select c.X)
    .Max();
// or
var result =
    collection.AsQueryable<C>()
    .Select(c => c.X)
    .Max();
```

The projection must select a particular field or property of the document. If the value of that field or property is represented in MongoDB as an array you can also use array indexing to select an item from the array.

```
var result =
    (from c in collection.AsQueryable<C>()
    select c.A[i])
    .Max();
// or
var result =
    collection.AsQueryable<C>()
    .Select(c => c.A[i])
    .Max();
```

Max (with selector)

This overload of Max lets you select the field or property whose maximum value you want as an argument to Max instead of to Select.

```
var result =
    (from c in collection.AsQueryable<C>()
    select c)
    .Max(c => c.X);
// or
var result =
    collection.AsQueryable<C>()
    .Max(c => c.X);
```

Min

Min returns the minimum value of a field or property of the documents in the collection. You use a projection to identify the field or property whose minimum value you want.

```
var result =
    (from c in collection.AsQueryable<C>()
    select c.X)
    .Min();
// or
var result =
    collection.AsQueryable<C>()
    .Select(c => c.X)
    .Min();
```

The projection must select a particular field or property of the document. If the value of that field or property is represented in MongoDB as an array you can also use array indexing to select an item from the array.

```
var result =
    (from c in collection.AsQueryable<C>()
    select c.A[i])
    .Min();
// or
var result =
    collection.AsQueryable<C>()
    .Select(c => c.A[i])
    .Min();
```

Min (with selector)

This overload of Min lets you select the field or property whose minimum value you want as an argument to Min instead of to Select.

```
var result =
    (from c in collection.AsQueryable<C>()
    select c)
    .Min(c => c.X);
// or
var result =
    collection.AsQueryable<C>()
    .Min(c => c.X);
```

OrderBy

OrderBy is used to specify an ascending sort order for the result set.

```
var query =
    from c in collection.AsQueryable<C>()
    orderby c.X
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .OrderBy(c => c.X);
```

OrderByDescending

OrderBy is used to specify a descending sort order for the result set.

```
var query =
    from c in collection.AsQueryable<C>()
    orderby c.X descending
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .OrderByDescending(c => c.X);
```

Select

Select is used to project a new result type from the matching documents. In the 1.4 version of the C# driver a projection must typically be the last operation (with a few exceptions like Distinct, Max and Min).

```
var query =
    from c in collection.AsQueryable<C>()
    select new { c.X, c.Y };
// or
var query =
    collection.AsQueryable<C>()
    .Select(c => new { c.X, c.Y });
```

Single

Single returns the first and only document from a result set.

```
var result =
    (from c in collection.AsQueryable<C>()
    where c.X > 0
    orderby c.X
    select c)
    .Single();
// or
var result =
    collection.AsQueryable<C>()
    .Where(c => c.X > 0)
    .OrderBy(c => c.X)
    .Single();
```

If the result set has no documents or multiple documents Single throws an exception.

Single (with predicate)

This overload of Single allows you to provide a predicate as an argument to Single . This is an alternative to using a where clause.

```
var result =
    (from c in collection.AsQueryable<C>()
    orderby c.X
    select c)
    .Single(c => c.X > 0);
// or
var result =
    collection.AsQueryable<C>()
    .OrderBy(c => c.X)
    .Single(c => c.X > 0);
```

Single with a predicate is not supported after a projection (at least not yet). So the following is not valid:

```
var result =
    collection.AsQueryable<C>()
    .OrderBy(c => c.X)
    .Select(c => c.X)
    .Single(x => x > 0);
```

You can usually rewrite such a query by putting an equivalent where clause before the projection.

If the result set has no documents or multiple documents Single throws an exception.

SingleOrDefault

SingleOrDefault is just like Single except that if there are no matching documents it returns null instead of throwing an exception.

SingleOrDefault (with predicate)

SingleOrDefault with a predicate is just like Single with a predicate except that if there are no matching documents it returns null instead of throwing an exception.

Skip

Use Skip to specify how many documents to skip from the beginning of the result set. Often you will combine Skip with a sort order.

```
var query =
    (from c in collection.AsQueryable<C>()
    orderby c.X
    select c)
    .Skip(100);
// or
var query =
    collection.AsQueryable<C>()
    .OrderBy(c => c.X)
    .Skip(100);
```

Take

Use Take to specify how many documents to return from the server. When combining Take with Skip often you will also specify a sort order.

```
var query =
    (from c in collection.AsQueryable<C>()
    orderby c.X
    select c)
    .Skip(100)
    .Take(100);
// or
var query =
    collection.AsQueryable<C>()
    .OrderBy(c => c.X)
    .Skip(100)
    .Take(100);
```

ThenBy

ThenBy is used to specify an additional ascending sort order for the result set.

```
var query =
    from c in collection.AsQueryable<C>()
    orderby c.X, c.Y
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .OrderBy(c => c.X)
    .ThenBy(c => c.Y);
```

ThenByDescending

ThenBy is used to specify an additional descending sort order for the result set.

```
var query =
    from c in collection.AsQueryable<C>()
    orderby c.X, c.Y descending
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .OrderBy(c => c.X)
    .ThenByDescending(c => c.Y);
```

Where

A where clause is used to specify which documents the query should return. A where clause is a C# expression that maps the query document type to a boolean value. If the expression returns true the document "matches" the query and is included in the result set.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.X > 0
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.X > 0);
```

Sometimes a predicate can be supplied in other places besides a where clause, and it is also possible to have multiple where clauses. When multiple predicates are involved they are combined into a single composite predicate by combining the individual predicates with the && operator.

For example, the following queries are equivalent:

```
var query =
    (from c in collection.AsQueryable<C>()
    where c.X > 0
    where c.Y > 0)
    .First(c.Z > 0);
// or
var query =
    (from c in collection.AsQueryable<C>()
    where c.X > 0 && c.Y > 0 && c.Z > 0)
    .First();
```

Supported where clauses

This section dcouments the supported where clauses.

As mentioned earlier, not all C# expressions are supported as a where clause. You can use this documentation as a guide to what is supported, or you can just try an expression and see if it works (a runtime exception is thrown if the where clause is not supported).

Where clauses are typically introduced using the Where query operator, but the same expressions are supported wherever a predicate is called for. In some cases multiple where clauses and predicates will be combined, in which case they are combined with the && operator.

Note: The 1.4 version of the C# driver requires that all where clauses that compare a field or property against a value have the constant on the right hand side. This restriction will be lifted in the next release.

And (&& operator)

Sub-expressions can be combined with the && operator to test whether all of them are true.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.X > 0 && c.Y > 0
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.X > 0 && c.Y > 0);
```

This is translated to the following MongoDB query:

```
{ X : { $gt : 0 }, Y : { $gt : 0 } }
```

In some cases the And query can't be flattened as shown, and the $and operator will be used. The following example matches documents where X is both a multiple of 2 and a multiple of 3:

```
var query =
    from c in collection.AsQueryable<C>()
    where (c.X % 2 == 0) && (c.X % 3 == 0)
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => (c.X % 2 == 0) && (c.X % 3 == 0));
```

This is translated to the following MongoDB query using $and:

```
{ $and : [{ X : { $mod : [2, 0] } }, { X : { $mod : [3, 0] } }] }
```

Any

This method is used to test whether an array field or property contains any items.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.A.Any()
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.A.Any());
```

matches any document where A has 1 or more items.

This is translated to the following MongoDB query:

```
{ A : { $ne : null, $not : { $size : 0 } } }
```

Boolean constant

This form is mostly for completeness. You will probably use it rarely. It allows a boolean constant to be used to either match or not match the document.

```
var query =
    from c in collection.AsQueryable<C>()
    where true
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => true);
```

This is translated to the following MongoDB query:

```
{ _id : { $exists : true } }
```

Which matches all documents since the _id is a mandatory field.

Boolean field or property

A boolean field or property of the document doesn't have to be compared to true, it can just be mentioned in the where clause and there is an implied comparison to true.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.B
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.B);
```

This is translated to the following MongoDB query:

```
{ B : true }
```

Contains (Enumerable method)

This method is used to test whether an array (or array-like) field or property contains a particular value:

```
var query =
    from c in collection.AsQueryable<C>()
    where c.A.Contains(123)
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.A.Contains(123));
```

This is transalated to the following MongoDB query:

```
{ A : 123 }
```

This translation relies on the way array fields are treated by the MongoDB query language.

Contains (string method)

This method is used to test whether a string field or property of the document contains a particular substring.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.S.Contains("abc")
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.S.Contains("abc"));
```

This is translated to the following MongoDB query (using regular expressions):

```
{ S : /abc/ }
```

ContainsAll (LINQ to MongoDB extension method)

This method is used to test whether an array (or array-like) field or property contains all of the provided values.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.A.ContainsAll(new[] { 1, 2, 3 })
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.A.ContainsAll(new[] { 1, 2, 3 }));
```

This is translated to the following MongoDB query:

```
{ A : { $all : [1, 2, 3] } }
```

ContainsAny (LINQ to MongoDB extension method)

This method is used to test whether an array (or array-like) field or property contains any of the provided values.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.A.ContainsAny(new[] { 1, 2, 3 })
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.A.ContainsAny(new[] { 1, 2, 3 }));
```

This is translated to the following MongoDB query:

```
{ A : { $in : [1, 2, 3] } }
```

Count method (array length)

This method is used to test whether an enumerable field or property has a certain count of items.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.L.Count() == 3
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.L.Count() == 3);
```

This is translated to the following MongoDB query:

```
{ L : { $size: 3 } }
```

Count property (array length)

This property is used to test whether a list (or list-like) field or property has a certain count of items.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.L.Count == 3
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.L.Count == 3);
```

This is translated to the following MongoDB query:

```
{ L : { $size: 3 } }
```

EndsWith (string method)

This method is used to test whether a string field or property of the document ends with a particular substring.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.S.EndsWith("abc")
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.S.EndsWith("abc"));
```

This is translated to the following MongoDB query (using regular expressions):

```
{ S : /abc$/ }
```

Enum comparisons (==, !=, <, <=, >, >=)

Enum fields or properties can be compared to constants of the same enum type. The relative comparison are based on the value of the underlying integer type.

```
public enum E { None, A, B };

var query =
    from c in collection.AsQueryable<C>()
    where c.E == E.A
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.E == E.A);
```

This is translated to the following MongoDB query:

```
{ E : 1 }
```

The LINQ implementation takes the representation of serialized values into account, so if you have configured your class map to store enums as string values instead of integer values the MongoDB query would instead be:

```
{ E : "A" }
```

In (LINQ to MongoDB extension method)

The In method is used to test whether a field or property is equal any of a set of provided values.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.X.In(new [] { 1, 2, 3 })
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c.X.In(new [] { 1, 2, 3 }));
```

This is translated to the following MongoDB query:

```
{ X : { $in : [1, 2, 3] } }
```

Inject

Inject is a pseudo-method that is used to inject a lower level MongoDB query into a LINQ query. The following query looks for X values that are larger than 0 and are 64-bit integers.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.X > 0 && Inject(Query.Type("X", BsonType.Int64))
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.X > 0 && Inject(Query.Type("X", BsonType.Int64)));
```

This is translated to the following MongoDB query:

```
{ X : { $gt : 0, $type : 18 } }
```

IsMatch (regular expression method)

This method is used to test whether a string field or property matches a regular expression.

```
var regex = new Regex("^abc");
var query =
    from c in collection.AsQueryable<C>()
    where regex.IsMatch(c.S)
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => regex.IsMatch(c.S));
```

This is translated to the following MongoDB query:

```
{ S : /^abc/ }
```

You can also use the static IsMatch method.

```
var query =
    from c in collection.AsQueryable<C>()
    where Regex.IsMatch(c.S, "^abc")
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => Regex.IsMatch(c.S, "^abc"));
```

This is translated to the following MongoDB query:

```
{ S : /^abc/ }
```

Length (array length)

This method is used to test whether an array (or array-like) field or property has a certain count of items.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.A.Length == 3
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.A.Length == 3);
```

This is translated to the following MongoDB query:

```
{ A : { $size: 3 } }
```

Mod (% operator)

This operator is used to test the result of the mod operator against a field or property of the document. The following query matches all the documents where X is odd.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.X % 2 == 1
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.X % 2 == 1);
```

This is translated to the following MongoDB query:

```
{ X : { $mod : [2, 1] } }
```

Not (! operator)

The ! operator is used to reverse the sense of a test.

```
var query =
    from c in collection.AsQueryable<C>()
    where !(c.X > 1)
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => !(c.X > 1));
```

This is translated into the following MongoDB query:

```
{ X : { $not : { $gt : 1 } } }
```

Note that !(c.X > 1) is not equivalent to (c.X <= 1) in cases where c.X is missing or does not have a numeric type.

Numeric comparisons (==, !=, <, <=, >, >=)

Numeric fields or properties can be compared using any of the above operators.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.X == 0 && c.Y < 100
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.X > 0 && c.Y < 100);
```

This is translated into the following MongoDB query:

```
{ X : 0, Y : { $lt : 100 } }
```

Or (|| operator)

Sub-expressions can be combined with the || operator to test whether any of them is true.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.X > 0 || c.Y > 0
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.X > 0 || c.Y > 0);
```

This is translated to the following MongoDB query:

```
{ $or : [{ X : { $gt : 0 } }, { Y : { $gt : 0 } }] }
```

StartsWith (string method)

This method is used to test whether a string field or property of the document starts with a particular substring.

```
var query =
    from c in collection.AsQueryable<C>()
    where c.S.StartsWith("abc")
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.S.StartsWith("abc"));
```

This is translated to the following MongoDB query (using regular expressions):

```
{ S : /^abc/ }
```

CSharp Driver Quickstart

Introduction

This quick-start provides just enough information to get you started using the C# driver. After you have gotten started you can refer to the rest of the documentation for more information.

Downloading the C# driver

You can download the C# driver here:

http://github.com/mongodb/mongo-csharp-driver/downloads

If you downloaded the .zip file, simply unzip it and place the contents anywhere you want.

If you downloaded the .msi file, double click on the .msi file to run the setup program, which will install the C# driver DLLs in the "C:\Program Files (x86)\MongoDB\CSharp Driver 1.x" directory (the exact path may vary on your system).

Add a reference to the C# driver DLLs

Right click on the References folder in Visual Studio's Solution Explorer and select "Add Reference...". Navigate to the folder where the C# driver DLLs were installed and add a reference to the following DLLs:

1. MongoDB.Bson.dll
2. MongoDB.Driver.dll

As an alternative you could use the NuGet package manager to add the C# driver package to your solution.

Add required using statements

As a minimum you will need the following using statements:

```
using MongoDB.Bson;
using MongoDB.Driver;
```

Additionally, you will frequently add one or more of these using statements:

```
using MongoDB.Driver.Builders;
using MongoDB.Driver.GridFS;
using MongoDB.Driver.Linq;
```

There are additional namespaces that would only be required in special cases.

Get a reference to the server object

The easiest way to get a reference to a server object is using a connection string:

```
var connectionString = "mongodb://localhost/?safe=true";
var server = MongoServer.Create(connectionString);
```

You should almost always add "safe=true" to your connection string.

If you want to store the server object in a global variable you can, but MongoServer.Create always returns the same instance of the server object when you call it with the same connection string so it's also OK to just call MongoServer.Create again whenever you need to.

Get a reference to a database object

To get a reference to a database object from the server object write this:

```
var database = server.GetDatabase("test"); // "test" is the name of the database
```

If you use more than one database call GetDatabase again for each database you want to use.

Decide if you want to work with the BsonDocument object model or with your own domain classes

There are two ways you can work with collections:

1. using the BsonDocument object model
2. using your own domain classes

You would use the BsonDocument object model when the data you are working with is so free form that it would be difficult or impossible to define domain classes for it.

Because it is so much easier to work with your own domain classes this quick-start will assume that you are going to do that. The C# driver can work with your domain classes provided that they:

1. Have a no-argument constructor
2. Define public read/write fields or properties for the data you want stored in the database

These requirements are essentially the same as those imposed by .NET's XmlSerializer.

In addition, if your domain class is going to be used as the root document it must contain an Id field or property (typically named "Id" although you can override that if necessary). Normally the Id will be of type ObjectId.

Get a reference to a collection object

For purposes of illustration let's assume you are using a domain class called Entity. You would get a reference to a collection containing Entity documents like this:

```
var collection = database.GetCollection<Entity>("entities"); // "entities" is the name of the
collection
```

Insert a document

Inserting a document is easy:

```
var entity = new Entity { Name = "Tom" };
collection.Insert(entity);
var id = entity.Id; // Insert will set the Id if necessary (as it was in this example)
```

Find an existing document

In this example we will read back an Entity assuming we know the Id value:

```
var query = Query.EQ("_id", id);
var entity = collection.FindOne(query);
```

Query.EQ is using the Query builder class to help you build the query. "_id" is the name of the field as stored in the database (normally the name of the field in the database is exactly the same as the name of the field or property in your domain class, but Id is an exception and is mapped to "_id" in the database).

Other query operators include: GT, GTE, In, LT, LTE, Near, NE, And, Or (and a few other more specialized ones).

Save a document

You can save changes to an existing document like this:

```
entity.Name = "Dick";
collection.Save(entity);
```

Update an existing document

An alternative to Save is Update. The difference is that Save sends the entire document back to the server, but Update sends just the changes. For example:

```
var query = Query.EQ("_id", id);
var update = Update.Set("Name", "Harry"); // update modifiers
collection.Update(query, update);
```

This example uses the Update builder to easily build the update modifiers.

Remove an existing document

To remove an existing document from a collection you write:

```
var query = Query.EQ("_id", id);
collection.Remove(query);
```

You do NOT need to call Connect or Disconnect

The C# driver has a connection pool to use connections to the server efficiently. There is no need to call Connect or Disconnect; just let the driver take care of the connections (calling Connect is harmless, but calling Disconnect is bad because it closes all the connections in the connection pool).

Full sample program

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using MongoDB.Bson;
using MongoDB.Driver;
using MongoDB.Driver.Builders;


namespace ConsoleApplication1
{
    public class Entity
    {
        public ObjectId Id { get; set; }
        public string Name { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var connectionString = "mongodb://localhost/?safe=true";
var server = MongoServer.Create(connectionString);
            var database = server.GetDatabase("test");
            var collection = database.GetCollection<Entity>("entities");

            var entity = new Entity { Name = "Tom" };
            collection.Insert(entity);
            var id = entity.Id;

            var query = Query.EQ("_id", id);
            entity = collection.FindOne(query);

            entity.Name = "Dick";
            collection.Save(entity);

            var update = Update.Set("Name", "Harry");
            collection.Update(query, update);

            collection.Remove(query);
        }
    }
}
```

CSharp Driver Serialization Tutorial

Introduction

This document refers to version 1.2 of the C# Driver.

This section of the C# Driver Tutorial discusses serialization (and deserialization) of instances of C# classes to and from BSON documents. Serialization is the process of mapping an object to a BSON document that can be saved in MongoDB, and deserialization is the reverse process of reconstructing an object from a BSON document. For that reason the serialization process is also often referred to as "Object Mapping."

Serialization is handled by the BSON Library. The BSON Library has an extensible serialization architecture, so if you need to take control of serialization you can. The BSON Library provides a default serializer which should meet most of your needs, and you can supplement the default serializer in various ways to handle your particular needs.

The main way the default serializer handles serialization is through "class maps". A class map is a structure that defines the mapping between a class and a BSON document. It contains a list of the fields and properties of the class that participate in serialization and for each one defines the required serialization parameters (e.g., the name of the BSON element, representation options, etc...).

The default serializer also has built in support for many .NET data types (primitive values, arrays, lists, dictionaries, etc...) for which class maps are not used.

Before an instance of a class can be serialized a class map must exist. You can either create this class map yourself or simply allow the class map to be created automatically when first needed (called "automapping"). You can exert some control over the automapping process either by decorating your classes with serialization related attributes or by using initialization code (attributes are very convenient to use but for those who prefer to keep serialization details out of their domain classes be assured that anything that can be done with attributes can also be done without them).

Creating a class map

To create a class map in your intialization code write:

```
BsonClassMap.RegisterClassMap<MyClass>();
```

which results in MyClass being automapped and registered. In this case you could just as well have allowed the class to be automapped by the serializer (when first serialized or deserialized). The one case where you must call RegisterClassMap yourself (even without arguments) is when you are using a polymorphic class hierarchy: in this case you must register all the known subclasses to guarantee that the discriminators get registered.

If you want to control the creation of the class map you can provide your own initialization code in the form of a lambda expression:

```
BsonClassMap.RegisterClassMap<MyClass)(cm => {
    cm.MapProperty(c => c.SomeProperty);
    cm.MapProperty(c => c.AnotherProperty);
});
```

When your lamba expression is executed the cm (short for class map) parameter is passed an empty class map for you to fill in. In this example two properties are added to the class map by calling the MapProperty method. The arguments to MapProperty are themselves lambda expressions which identify the property of the class. The advantage of using a lambda expression instead of just a string parameter with the name of the property is that Intellisense and compile time checking ensure that you can't misspell the name of the property.

It is also possible to use automapping and then override some of the results. We will see examples of that later on.

Note that a class map must only be registered once (an exception will be thrown if you try to register the same class map more than once). Usually you call RegisterClassMap from some code path that is known to execute only once (the Main method, the Application_Start event handler, etc...). If you must call RegisterClassMap from a code path that executes more than once, you can use IsClassMapRegistered to check whether a class map has already been registered for a class:

```
if (!BsonClassMap.IsClassMapRegistered(typeof(MyClass))) {
    // register class map for MyClass
}
```

Conventions

When automapping a class there are a lot of decisions that need to be made. For example:

- Which fields or properties of the class should be serialized
- Which field or property of the class is the "Id"
- What element name should be used in the BSON document
- If the class is being used polymorphically what discriminator values are used
- What should happen if a BSON document has elements we don't recognize
- Does the field or property have a default value
- Should the default value be serialized or ignored
- Should null values be serialized or ignored

Answers to these questions are represented by a set of "conventions". For each convention there is a default convention that is the most likely one you will be using, but you can override individual conventions (and even write your own) as necessary.

If you want to use your own conventions that differ from the defaults simply create an instance of ConventionProfile and set the values you want to override and then register that profile (in other words, tell the default serializer when your special conventions should be used). For example:

```
var myConventions = new ConventionProfile();
// override any conventions you want to be different
BsonClassMap.RegisterConventions(
    myConventions,
    t => t.FullName.StartsWith("MyNamespace.")
);
```

The second parameter is a filter function that defines when this convention profile should be used. In this case we are saying that any classes whose full names begin with "MyNamespace." should use myConventions.

ConventionProfile provides the following methods to allow you to set individual conventions:

- SetDefaultValueConvention
- SetElementNameConvention
- SetExtraElementsMemberConvention
- SetIdGeneratorConvention
- SetIdMemberConvention
- SetIgnoreExtraElementsConvention
- SetIgnoreIfNullConvention
- SetMemberFinderConvention
- SetSerializeDefaultValueConvention

Field or property level serialization options

There are many ways you can control serialization. The previous section discussed conventions, which are a convenient way to control serialization decisions for many classes at once. You can also control serialization at the individual class or field or property level.

Serialization can be controlled either by decorating your classes and fields or properties with serialization related attributes or by writing code to initialize class maps appropriately. For each aspect of serialization you can control we will be showing both ways.

Element name

To specify an element name using attributes, write:

```
public class MyClass {
    [BsonElement("sp")]
    public string SomeProperty { get; set; }
}
```

The same result can be achieved without using attributes with the following initialization code:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.SomeProperty).SetElementName("sp");
});
```

Note that we are first automapping the class and then overriding one particular piece of the class map. If you didn't call AutoMap first then GetMemberMap would throw an exception because there would be no member maps.

Element order

If you want precise control over the order of the elements in the BSON document you can use the Order named parameter to the BsonElement attribute:

```
public class MyClass {
    [BsonElement("sp", Order = 1)]
    public string SomeProperty { get; set; }
}
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.SomeProperty).SetElementName("sp").SetOrder(1);
});
```

Any fields or properties that do not have an explicit Order will occur after those that do have an Order.

Identifying the Id field or property

To identify which field or property of a class is the Id you can write:

```
public class MyClass {
    [BsonId]
    public string SomeProperty { get; set; }
}
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.SetIdMember(cm.GetMemberMap(c => c.SomeProperty));
});
```

When not using AutoMap, you can also map a field or property and identify it as the Id in one step as follows:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.MapIdProperty(c => c.SomeProperty);
    // mappings for other fields and properties
});
```

Selecting an IdGenerator to use for an Id field or property

When you Insert a document the C# driver checks to see if the Id member has been assigned a value, and if not, generates a new unique value for it. Since the Id member can be of any type, the driver requires the help of a matching IdGenerator to check whether the Id has a value assigned to it and to generate a new value if necessary. The driver has the following IdGenerators built-in:

- BsonObjectIdGenerator
- CombGuidGenerator
- GuidGenerator
- NullIdChecker

- ObjectIdGenerator
- StringObjectIdGenerator
- ZeroIdChecker<T>

Some of these IdGenerators are used automatically for commonly used Id types:

- BsonObjectIdGenerator is used for BsonObjectId
- GuidGenerator is used for Guid
- ObjectIdGenerator is used for ObjectId
- StringObjectIdGenerator is used for strings represented externally as ObjectId

To select an IdGenerator to use for your Id field or property write:

```
public class MyClass {
    [BsonId(IdGenerator = typeof(CombGuidGenerator))]
    public Guid Id { get; set; }
}
```

Or using intialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.IdMemberMap.SetIdGenerator(CombGuidGenerator.Instance);
});
```

You could also say that you want to use the CombGuidGenerator for all Guids. In this case you would write:

```
BsonSerializer.RegisterIdGenerator(
    typeof(Guid),
    CombGuidGenerator.Instance
);
```

The NullIdChecker and ZeroIdChecker<T> IdGenerators can be used when you don't have an IdGenerator for an Id type but you want to enforce that the Id is not null or zero. These pseudo-IdGenerators throw an exception if their GenerateId method is called. You can select it for an individual member just like a CombGuidGenerator was selected in the previous example, or you can turn on one or both of these IdGenerators for all types as follows:

```
BsonSerializer.UseNullIdChecker = true; // used for reference types
BsonSerializer.UseZeroIdChecker = true; // used for value types
```

Note: in version 1.0 of the C# Driver NullIdChecker and ZeroIdChecker<T> were always used, but it was decided that their use should be optional, since null and zero are valid values for an Id as far as the server is concerned, so they should only be considered an error if the developer has specifically said they should be.

Ignoring a field or property

When constructing a class map manually you can ignore a field or property simply by not adding it to the class map. When using AutoMap you need a way to specify that a field or property should be ignored. To do so using attributes write:

```
public class MyClass {
    [BsonIgnore]
    public string SomeProperty { get; set; }
}
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.UnmapProperty(c => c.SomeProperty);
});
```

In this case AutoMap will have initially added the property to the class map automatically but then UnmapProperty will remove it.

Ignoring null values

By default null values are serialized to the BSON document as a BSON Null. An alternative is to serialize nothing to the BSON document when the field or property has a null value. To specify this using attributes write:

```
public class MyClass {
    [BsonIgnoreIfNull]
    public string SomeProperty { get; set; }
}
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.SomeProperty).SetIgnoreIfNull(true);
});
```

Default values

You can specify a default value for a field or property as follows:

```
public class MyClass {
    [BsonDefaultValue("abc")]
    public string SomeProperty { get; set; }
}
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.SomeProperty).SetDefaultValue("abc");
});
```

You can also control whether default values are serialized or not (the default is yes). To not serialize default values using attributes write:

```
public class MyClass {
    [BsonDefaultValue("abc", SerializeDefaultValue = false)]
    public string SomeProperty { get; set; }
}
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.SomeProperty).SetDefaultValue("abc", false);
});
```

or equivalently:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.SomeProperty)
        .SetDefaultValue("abc")
        .SetSerializeDefaultValue(false);
});
```

Ignoring a member based on a ShouldSerializeXyz method

Sometimes the decision whether to serialize a member or not is more complicated than just whether the value is null or equal to the default value.

You can write a method that determines whether a value should be serialized. Usually the method for member Xyz is named ShouldSerializeXyz. If you follow this naming convention then AutoMap will automatically detect the method and use it. For example:

```
public class Employee {
    public ObjectId Id { get; set; }
    [BsonDateTimeOptions(DateOnly = true)]
    public DateTime DateOfBirth { get; set; }

    public bool ShouldSerializeDateOfBirth() {
        return DateOfBirth > new DateTime(1900, 1, 1);
    }
}
```

Or using initialization code instead of naming conventions:

```
BsonClassMap.RegisterClassMap<Employee>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.DateOfBirth).SetShouldSerializeMethod(
        obj => ((Employee) obj).DateOfBirth > new DateTime(1900, 1, 1)
    );
});
```

Identifying required fields

Normally, the deserializer doesn't care if the document being deserialized doesn't have a matching element for every field or property of the class. The members that don't have a matching element simply get assigned their default value (or null if they don't have a default value).

If you want to make an element in the document be required, you can mark an individual field or property like this:

```
public class MyClass {
    public ObjectId Id { get; set; }
    [BsonRequired]
    public string X { get; set; }
}
```

Or using initialization code instead attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.X).SetIsRequired(true);
});
```

Serialization Options

Serialization of some classes can be more finely controlled using serialization options (which are represented using classes that implement the IBsonSerializationOptions interface). Whether a class uses serialization options or not, and which ones, depends on the particular class involved. The following sections describe the available serialization option classes and the classes that use them.

DateTimeSerializationOptions

These serialization options control how a DateTime is serialized. For example:

```
public class MyClass {
    [BsonDateTimeOptions(DateOnly = true)]
    public DateTime DateOfBirth { get; set; }
    [BsonDateTimeOptions(Kind = DateTimeKind.Local)]
    public DateTime AppointmentTime { get; set; }
}
```

Here we are specifying that the DateOfBirth value holds a date only (so the TimeOfDay component must be zero). Additionally, because this is a date only, no timezone conversions at all will be performed. The AppointmentTime value is in local time and will be converted to UTC when it is serialized and converted back to local time when it is deserialized.

You can specify the same options using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.DateOfBirth)
        .SetSerializationOptions(
            new DateTimeSerializationOptions { DateOnly = true });
    cm.GetMemberMap(c => c.AppointmentTime)
        .SetSerializationOptions(
            new DateTimeSerializationOptions { Kind = DateTimeKind.Local });
});
```

DateTimeSerializationOptions are supported by the serializers for the following classes: BsonDateTime and DateTime.

DictionarySerializationOptions

When serializing dictionaries there are several alternative ways that the contents of the dictionary can be represented. The different ways are represented by the DictionaryRepresentation enumeration:

```
public enum DictionaryRepresentation {
    Dynamic,
    Document,
    ArrayOfArrays,
    ArrayOfDocuments
}
```

A dictionary represented as a `Document` will be stored as a BsonDocument, and each entry in the dictionary will be represented by a BsonElement with the name equal to the key of the dictionary entry and the value equal to the value of the dictionary entry. This representation can only be used when all the keys in a dictionary are strings that are valid element names.

A dictionary represented as an `ArrayOfArrays` will be stored as a BsonArray of key/value pairs, where each key/value pair is stored as a nested two-element BsonArray where the two elements are the key and the value of the dictionary entry. This representation can be used even when the keys of the dictionary are not strings. This representation is very general and compact, and is the default representation when `Document` does not apply. One problem with this representation is that it is difficult to write queries against it, which motivated the introduction in the 1.2 version of the driver of the `ArrayOfDocuments` representation.

A dictionary represented as an `ArrayOfDocuments` will be stored as a BsonArray of key/value pairs, where each key/value pair is stored as a nested two-element BsonDocument of the form { `k` : key, `v` : value }. This representation is just as general as the `ArrayOfArrays` representation, but because the keys and values are tagged with element names it is much easier to write queries against it. For backward compatibility reasons this is not the default representation.

If the `Dynamic` representation is specified, the dictionary key values are inspected before serialization, and if all the keys are strings which are also valid element names, then the `Document` representation will be used, otherwise the `ArrayOfArrays` representation will be used.

If no other representation for a dictionary is specified, then `Dynamic` is assumed.

You can specify a DictionarySerializationOption as follows:

```
public class C {
    public ObjectId Id;
    [BsonDictionaryOptions(DictionaryRepresentation.ArrayOfDocuments)]
    public Dictionary<string, int> Values;
}
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<C>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.Values)
        .SetSerializationOptions(DictionarySerializationOptions.ArrayOfDocuments);
});
```

DictionarySerializationOptions are suppored by the serializers for the following classes: the generic classes and interfaces Dictionary, IDictionary, SortedDictionary and SortedList, and the non-generic classes and interfaces Hashtable, IDictionary, ListDictionary, OrderedDictionary and SortedList.

RepresentationSerializationOptions

For some .NET primitive types you can control what BSON type you want used to represent the value in the BSON document. For example, you can specify whether a char value should be represented as a BSON Int32 or as a one-character BSON String:

```
public class MyClass {
    [BsonRepresentation(BsonType.Int32)]
    public char RepresentAsInt32 { get; set; }
    [BsonRepresentation(BsonType.String)]
    public char RepresentAsString { get; set; }
}
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.RepresentAsInt32)
        .SetRepresentation(BsonType.Int32);
    cm.GetMemberMap(c => c.RepresentAsString)
        .SetRepresentation(BsonType.String);
});
```

One case that deserves special mention is representing a string externally as an ObjectId. For example:

```
public class Employee {
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }
    // other properties
}
```

In this case the serializer will convert the ObjectId to a string when reading data from the database and will convert the string back to an ObjectId when writing data to the database (the string value must be a valid ObjectId). Typically this is done when you want to keep your domain classes free of any dependencies on the C# driver, so you don't want to declare the Id as an ObjectId. String serves as a neutral representation that is at the same time easily readable for debugging purposes. To keep your domain classes free of dependencies on the C# driver you also won't want to use attributes, so you can accomplish the same thing using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<Employee>(cm => {
    cm.AutoMap();
    cm.IdMemberMap.SetRepresentation(BsonType.ObjectId);
});
```

Class level serialization options

There are several serialization options that are related to the class itself instead of to any particular field or property. You can set these class level options either by decorating the class with serialization related attributes or by writing initialization code. As usual, we will show both ways in the examples.

Ignoring extra elements

When a BSON document is deserialized the name of each element is used to look up a matching field or property in the class map. Normally, if no matching field or property is found, an exception will be thrown. If you want to ignore extra elements during deserialization, use the following attribute:

```
[BsonIgnoreExtraElements]
public MyClass {
    // fields and properties
}
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.SetIgnoreExtraElements(true);
});
```

Supporting extra elements

You can design your class to be capable of handling any extra elements that might be found in a BSON document during deserialization. To do so, you must have a property of type BsonDocument and you must identify that property as the one that should hold any extra elements that are found (or you can name the property "ExtraElements" so that the default ExtraElementsMemberConvention will find it automatically). For example:

```
public MyClass {
    // fields and properties
[BsonExtraElements]
    public BsonDocument CatchAll { get; set; }
}
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.SetExtraElementsMember(cm.GetMemberMap(c => c.CatchAll));
});
```

When a BSON document is deserialized any extra elements found will be stored in the extra elements BsonDocument property. When the class is serialized the extra elements will be serialized also. One thing to note though is that the serialized class will probably not have the elements in exactly the same order as the original document. All extra elements will be serialized together when the extra elements member is serialized.

Polymorphic classes and discriminators

When you have a class hierarchy and will be serializing instances of varying classes to the same collection you need a way to distinguish one from another. The normal way to do so is to write some kind of special value (called a "discriminator") in the document along with the rest of the elements that you can later look at to tell them apart. Since there are potentially many ways you could discriminate between actual types, the default serializer uses conventions for discriminators. The default serializer provides two standard discriminators: ScalarDiscriminatorConvention and HierarchicalDiscriminatorConvention. The default is the HierarchicalDiscriminatorConvention, but it behaves just like the ScalarDiscriminatorConvention until certain options are set to trigger its hierarchical behavior (more on this later).

The default discriminator conventions both use an element named "_t" to store the discriminator value in the BSON document. This element will normally be the second element in the BSON document (right after the "_id"). In the case of the ScalarDiscriminatorConvention the value of "_t" will be a single string. In the case of the HierarchicalDiscriminatorConvention the value of "_t" will be an array of discriminator values, one for each level of the class inheritance tree (again, more on this later).

While you will normally be just fine with the default discriminator convention, you might have to write a custom discriminator convention if you must inter-operate with data written by another driver or object mapper that uses a different convention for its discriminators.

Setting the discriminator value

The default value for the discriminator is the name of the class (without the namespace part). You can specify a different value using attributes:

```
[BsonDiscriminator("myclass")]
public MyClass {
    // fields and properties
}
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.SetDiscriminator("myclass");
});
```

Specifying known types

When deserializing polymorphic classes it is important that the serializer know about all the classes in the hierarchy before deserialization begins. If you ever see an error message about an "Unknown discriminator" it is because the deserializer can't figure out the class for that discriminator. If you are mapping your classes programmatically simply make sure that all classes in the hierarchy have been mapped before beginning deserialization. When using attributes and automapping you will need to inform the serializer about known types (i.e. subclasses) it should create class maps for. Here is an example of how to do this:

```
[BsonKnownTypes(typeof(Cat), typeof(Dog)]
public class Animal {
}

[BsonKnownTypes(typeof(Lion), typeof(Tiger)]
public class Cat : Animal {
}

public class Dog : Animal {
}

public class Lion : Cat {
}

public class Tiger : Cat {
}
```

The BsonKnownTypes attribute lets the serializer know what subclasses it might encounter during deserialization, so when Animal is automapped the serializer will also automap Cat and Dog (and recursively, Lion and Tiger as well).

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<Animal>();
BsonClassMap.RegisterClassMap<Cat>();
BsonClassMap.RegisterClassMap<Dog>();
BsonClassMap.RegisterClassMap<Lion>();
BsonClassMap.RegisterClassMap<Tiger>();
```

Scalar and hierarchical discriminators

Normally a discriminator is simply the name of the class (although it could be different if you are using a custom discriminator convention or have explicitly specified a discriminator for a class). So a collection containing a mix of different type of Animal documents might look like:

```
{ _t : "Animal", ... }
{ _t : "Cat", ... }
{ _t : "Dog", ... }
{ _t : "Lion", ... }
{ _t : "Tiger", ... }
```

Sometimes it can be helpful to record a hierarchy of discriminator values, one for each level of the hierarchy. To do this, you must first mark a base class as being the root of a hierarchy, and then the default HierarchicalDiscriminatorConvention will automatically record discriminators as array values instead.

To identify Animal as the root of a hierarchy use the BsonDiscriminator attribute with the RootClass named parameter:

```
[BsonDiscriminator(RootClass = true)]
[BsonKnownTypes(typeof(Cat), typeof(Dog)]
public class Animal {
}

// the rest of the hierarchy as before
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<Animal>(cm => {
    cm.AutoMap();
    cm.SetIsRootClass(true);
});
BsonClassMap.RegisterClassMap<Cat>();
BsonClassMap.RegisterClassMap<Dog>();
BsonClassMap.RegisterClassMap<Lion>();
BsonClassMap.RegisterClassMap<Tiger>();
```

Now that you have identified Animal as a root class, the discriminator values will look a little bit different:

```
{ _t : "Animal", ... }
{ _t : ["Animal", "Cat"], ... }
{ _t : ["Animal", "Dog"], ... }
{ _t : ["Animal", "Cat", "Lion"], ... }
{ _t : ["Animal", "Cat", "Tiger"], ... }
```

The main reason you might choose to use hierarchical discriminators is because it makes it possibly to query for all instances of any class in the hierarchy. For example, to read all the Cat documents we can write:

```
var query = Query.EQ("_t", "Cat");
var cursor = collection.FindAs<Animal>(query);
foreach (var cat in cursor) {
    // process cat
}
```

This works because of the way MongoDB handles queries against array values.

Customizing serialization

There are several ways you can customize serialization:

1. Supplementing the default serializer
2. Make a class responsible for its own serialization
3. Write a custom serializer
4. Write a custom Id generator
5. Write a custom convention

Supplementing the default serializer

You can register your own serialization provider to supplement the default serializer. Register it like this:

```
IBsonSerializationProvider myProvider;
BsonSerializer.RegisterSerializationProvider(myProvider);
```

You should register your provider as early as possible. Your provider will be called first before the default serializer. You can delegate handling of any types your custom provider isn't prepared to handle to the default serializer by returning null from GetSerializer.

Make a class responsible for its own serialization

One way you can customize how a class is serialized is to make it responsible for its own serialization. You do so by implementing the IBsonSerializable interface:

```
public class MyClass : IBsonSerializable {
    // implement Deserialize method
// implement Serialize method
}
```

You also must implement the GetDocumentId and SetDocumentId methods. If your class is never used as a root document these methods can just be stubs that throw a NotSupportedException. Otherwise, return true from GetDocumentId if the value passed in has an Id, and set the Id value in SetDocumentId.

There is nothing else you have to do besides implementing this interface. The BSON Library automatically checks whether objects being serialized implement this interface and if so routes serialization calls directly to the classes.

This can be a very efficient way to customize serialization, but it does have the drawback that it pollutes your domain classes with serialization details, so there is also the option of writing a custom serializer as described next.

Write a custom serializer

A custom serializer can handle serialization of your classes without requiring any changes to those classes. This is a big advantage when you either don't want to modify those classes or can't (perhaps because you don't have control over them). You must register your custom serializer so that the BSON Library knows of its existence and can call it when appropriate.

To implement and register a custom serializer you would:

```
// MyClass is the class for which you are writing a custom serializer
public MyClass {
}

// MyClassSerializer is the custom serializer for MyClass
public MyClassSerializer : IBsonSerializer {
    // implement Deserialize
// implement Serialize
}

// register your custom serializer
BsonSerializer.RegisterSerializer(
    typeof(MyClass),
    new MyClassSerializer()
);
```

You also must implement the GetDocumentId and SetDocumentId methods. If your class is never used as a root document these methods can just be stubs that throw a NotSupportedException. Otherwise, return true from GetDocumentId if the value passed in has an Id, and set the Id value in SetDocumentId.

If you write a custom serializer you will have to become familiar with the BsonReader and BsonWriter abstract classes, which are not documented here, but are relatively straightforward to use. Look at the existing serializers in the driver for examples of how BsonReader and BsonWriter are used.

You may want to derive your custom serializer from BsonBaseSerializer, but be aware that this is an internal class that is slightly more likely to change than other core classes. If that concerns you, just implement IBsonSerializer directly.

To debug a custom serializer you can either Insert a document containing a value serialized by your custom serializer into some collection and then use the mongo shell to examine what the resulting document looks like. Alternatively you can use the ToJson method to see the result of the serializer without having to Insert anything into a collection as follows:

```
// assume a custom serializer has been registered for class C
var c = new C();
var json = c.ToJson();
// inspect the json string variable to see how c was serialized
```

Write a custom Id generator

You can write your own IdGenerator. For example, suppose you wanted to generate integer Employee Ids:

```
public class EmployeeIdGenerator : IIdGenerator {
    // implement GenerateId
// implement IsEmpty
}
```

You can specify that this generator be used for Employee Ids using attributes:

```
public class Employee {
    [BsonId(IdGenerator = typeof(EmployeeIdGenerator)]
    public int Id { get; set; }
    // other fields or properties
}
```

Or using initialization code instead of attributes:

```
BsonClassMap.RegisterClassMap<Employee >(cm => {
    cm.AutoMap();
    cm.IdMember.SetIdGenerator(new EmployeeIdGenerator());
});
```

Alternatively, you can get by without an Id generator at all by just assigning a value to the Id property before calling Insert or Save.

Write a custom convention

Earlier in this tutorial we discussed replacing one or more of the default conventions. You can either replace them with one of the provided alternatives or you can write your own convention. Writing your own convention varies slightly from convention to convention.

As an example we will write a custom convention to find the Id member of a class (the default convention looks for a member named "Id"). Our custom convention will instead consider any public property whose name ends in "Id" to be the Id for the class. We can implement this convention as follows:

```
public class EndsWithIdConvention : IIdMemberConvention {
    public string FindIdMember(Type type) {
        foreach (var property in type.GetProperties()) {
            if (property.Name.EndsWith("Id")) {
                return property.Name;
            }
        }
        return null;
    }
}
```

And we can configure this convention to be used with all of our own classes by writing:

```
var myConventions = new ConventionProfile();
myConventions.SetIdMemberConvention(new EndsWithIdConvention());
BsonClassMap.RegisterConventions(
    myConventions,
    t => t.FullName.StartsWith("MyNamespace.")
);
```

Warning: because GetProperties is not guaranteed to return properties in any particular order this convention as written will behave unpredictably for a class that contains more than one property whose name ends in "Id".

CSharp Driver Tutorial

C# Driver version v1.4.x

This tutorial is for v1.4.x of the C# Driver.

Introduction

This tutorial introduces the 10gen supported C# Driver for MongoDB. The C# Driver consists of two libraries: the BSON Library and the C# Driver. The BSON Library can be used independently of the C# Driver if desired. The C# Driver requires the BSON Library.

You may also be interested in the C# Driver Serialization Tutorial. It is a separate tutorial because it covers quite a lot of material.

Downloading

The C# Driver is available in source and binary form. While the BSON Library can be used independently of the C# Driver they are both stored in the same repository.

The source may be downloaded from github.com.

We use msysgit as our Windows git client. It can be downloaded from: http://code.google.com/p/msysgit/.

To clone the repository run the following commands from a git bash shell:

```
$ cd <parentdirectory>
$ git config --global core.autocrlf true
$ git clone git://github.com/mongodb/mongo-csharp-driver.git
$ cd mongo-csharp-driver
$ git config core.autocrlf true
```

You must set the global setting for core.autocrlf to true before cloning the repository. After you clone the repository, we recommend you set the local setting for core.autocrlf to true (as shown above) so that future changes to the global setting for core.autocrlf do not affect this repository. If you then want to change your global setting for core.autocrlf to false run:

```
$ git config --global core.autocrlf false
```

The typical symptom of problems with the setting for core.autocrlf is git reporting that an entire file has been modified (because of differences in the line endings). It is rather tedious to change the setting of core.autocrlf for a repository after it has been created, so it is important to get it right from the start.

You can download a zip file of the source files (without cloning the repository) by clicking on the Downloads button at:

http://github.com/mongodb/mongo-csharp-driver

You can download binaries (in both .msi and .zip formats) from:

http://github.com/mongodb/mongo-csharp-driver/downloads

Building

We are currently building the C# Driver with Visual Studio 2010. The name of the solution file is `CSharpDriver-2010.sln`.

Dependencies

The unit tests depend on NUnit 2.5.9, which is included in the dependencies folder of the repository. You can build the C# Driver without installing NUnit, but you must install NUnit before running the unit tests (unless you use a different test runner).

Running unit tests

There are two projects containing unit tests:

1. BsonUnitTests
2. DriverUnitTests

The BsonUnitTests do not connect to a MongoDB server. The DriverUnitTests connect to an instance of MongoDB running on the default port on localhost.

An easy way to run the unit tests is to set one of the unit test projects as the startup project and configure the project settings as follows (using BsonUnitTests as an example):

- On the Debug tab:
    1. Set Start Action to Start External Program
    2. Set external program to: `C:\Program Files (x86)\NUnit 2.5.9\bin\net-2.0\nunit.exe`
    3. Set command line aguments to: `BsonUnitTests.csproj /config:Debug /run`
    4. Set working directory to: the directory where BsonUnitTest.csproj is located

Repeat the above steps for the Release configuration (using /config:Release instead) if you also want to run unit tests for Release builds.

The exact location of the nunit.exe program might vary slightly on your machine.

To run the DriverUnitTests perform the same steps (modified as necessary).

Installing

If you want to install the C# Driver on your machine you can use the setup program (see above for download instructions). The setup program is very simple and just:

1. Copies the DLLs to `C:\Program Files (x86)\MongoDB\CSharpDriver 1.4.x`
2. Installs the DLLs in the Global Assembly Cache (the GAC)
3. Configures Visual Studio to include the C# Driver DLLs in the .NET tab of the Add Reference dialog

If you downloaded the binaries zip file simply extract the files and place them wherever you want them to be.

Note: if you download the .zip file Windows might require you to "Unblock" the help file. If Windows asks "Do you want to open this file?" when you double click on the CSharpDriverDocs.chm file, clear the check box next to "Always ask before opening this file" before pressing the Open button. Alternatively, you can right click on the CSharpDriverDocs.chm file and select Properties, and then press the Unblock button at the bottom of the General tab. If the Unblock button is not present then the help file does not need to be unblocked.

References and namespaces

To use the C# Driver you must add references to the following DLLs:

1. `MongoDB.Bson.dll`
2. `MongoDB.Driver.dll`

As a minimum add the following using statements to your source files:

```
using MongoDB.Bson;
using MongoDB.Driver;
```

Additionally you will frequently add some of the following using statements:

```
using MongoDB.Driver.Builders;
using MongoDB.Driver.GridFS;
using MongoDB.Driver.Linq;
```

In some cases you might add some of the following using statements if you are using some of the optional parts of the C# Driver:

```
using MongoDB.Bson.IO;
using MongoDB.Bson.Serialization;
using MongoDB.Bson.Serialization.Attributes;
using MongoDB.Bson.Serialization.Conventions;
using MongoDB.Bson.Serialization.IdGenerators;
using MongoDB.Bson.Serialization.Options;
using MongoDB.Bson.Serialization.Serializers;
using MongoDB.Driver.Wrappers;
```

The BSON Library

The C# Driver is built on top of the BSON Library, which handles all the details of the BSON specification, including: I/O, serialization, and an in-memory object model of BSON documents.

The important classes of the BSON object model are: BsonType, BsonValue, BsonElement, BsonDocument and BsonArray.

BsonType

This enumeration is used to specify the type of a BSON value. It is defined as:

```
public enum BsonType {
    Double = 0x01,
    String = 0x02,
    Document = 0x03,
    Array = 0x04,
    Binary = 0x05,
    Undefined = 0x06,
    ObjectId = 0x07,
    Boolean = 0x08,
    DateTime = 0x09,
    Null = 0x0a,
    RegularExpression = 0x0b,
    JavaScript = 0x0d,
    Symbol = 0x0e,
    JavaScriptWithScope = 0x0f,
    Int32 = 0x10,
    Timestamp = 0x11,
    Int64 = 0x12,
    MinKey = 0xff,
    MaxKey = 0x7f
}
```

BsonValue and subclasses

BsonValue is an abstract class that represents a typed BSON value. There is a concrete subclass of BsonValue for each of the values defined by the BsonType enum. There are several ways to obtain an instance of BsonValue:

- Use a public constructor (if available) of a subclass of BsonValue
- Use a static Create method of BsonValue
- Use a static Create method of a subclass of BsonValue
- Use a static property of a subclass of BsonValue
- Use an implicit conversion to BsonValue

The advantage of using the static Create methods is that they can return a pre-created instance for frequently used values. They can also return null (which a constructor cannot) which is useful for handling optional elements when creating BsonDocuments using functional construction. The static properties refer to pre-created instances of frequently used values. Implicit conversions allow you to use primitive .NET values wherever a BsonValue is expected, and the .NET value will automatically be converted to a BsonValue.

BsonValue has the following subclasses:

- `BsonArray`
- `BsonBinaryData`
- `BsonBoolean`
- `BsonDateTime`
- `BsonDocument`
- `BsonDouble`
- `BsonInt32`
- `BsonInt64`
- `BsonJavaScript`
- `BsonJavaScriptWithScope` (a subclass of `BsonJavaScript`)
- `BsonMaxKey`
- `BsonMinKey`
- `BsonNull`
- `BsonObjectId`
- `BsonRegularExpression`
- `BsonString`
- `BsonSymbol`
- `BsonTimestamp`
- `BsonUndefined`

BsonType property

BsonValue has a property called BsonType that you can use to query the actual type of a BsonValue. The following example shows several ways to determine the type of a BsonValue:

```
BsonValue value;
if (value.BsonType == BsonType.Int32) {
    // we know value is an instance of BsonInt32
}
if (value is BsonInt32) {
    // another way to tell that value is a BsonInt32
}
if (value.IsInt32) {
    // the easiest way to tell that value is a BsonInt32
}
```

As[Type] Properties

BsonValue has a number of properties that cast a BsonValue to one of its subclasses or a primitive .NET type:

- `AsBoolean (=> bool)`
- `AsBsonArray`
- `AsBsonBinaryData`
- `AsBsonDateTime`
- `AsBsonDocument`
- `AsBsonJavaScript // also works if BsonType == JavaScriptWithScope`
- `AsBsonJavaScriptWithScope`
- `AsBsonMaxKey`
- `AsBsonMinKey`
- `AsBsonNull`
- `AsBsonRegularExpression`
- `AsBsonSymbol`
- `AsBsonTimestamp`
- `AsBsonUndefined`
- `AsBsonValue`
- `AsByteArray (=> byte[])`
- `AsDateTime (=> DateTime)`
- `AsDouble (=> double)`
- `AsGuid (=> Guid)`
- `AsInt32 (=> int)`
- `AsInt64 (=> long)`
- `AsNullableBoolean (=> bool?)`
- `AsNullableDateTime (=> DateTime?)`
- `AsNullableDouble (=> double?)`
- `AsNullableGuid (=> Guid?)`
- `AsNullableInt32 (=> int?)`
- `AsNullableInt64 (=> long?)`
- `AsNullableObjectId (=> ObjectId?)`
- `AsObjectId (=> ObjectId)`
- `AsRegex (=> Regex)`
- `AsString (=> string)`

It is important to note that these all are casts, not conversions. They will throw an InvalidCastException if the BsonValue is not of the corresponding type. See also the To[Type] methods which do conversions, and the Is[Type] properties which you can use to query the type of a BsonValue before attempting to use one of the As[Type] properties.

Sample code using these properties:

```
BsonDocument document;
string name = document["name"].AsString;
int age = document["age"].AsInt32;
BsonDocument address = document["address"].AsBsonDocument;
string zip = address["zip"].AsString;
```

Is[Type] Properties

BsonValue has the following boolean properties you can use to test what kind of BsonValue it is:

- `IsBoolean`
- `IsBsonArray`
- `IsBsonBinaryData`
- `IsBsonDateTime`
- `IsBsonDocument`
- `IsBsonJavaScript`

- `IsBsonJavaScriptWithScope`
- `IsBsonMaxKey`
- `IsBsonMinKey`
- `IsBsonNull`
- `IsBsonRegularExpression`
- `IsBsonSymbol`
- `IsBsonTimestamp`
- `IsBsonUndefined`
- `IsDateTime`
- `IsDouble`
- `IsGuid`
- `IsInt32`
- `IsInt64`
- `IsNumeric` (true if type is Double, Int32 or Int64)
- `IsObjectId`
- `IsString`

Sample code:

```
BsonDocument document;
int age = -1;
if (document.Contains["age"] && document["age"].IsInt32) {
    age = document["age"].AsInt32;
}
```

To[Type] conversion methods

The following methods are available to do limited conversions between BsonValue types:

- `ToBoolean`
- `ToDouble`
- `ToInt32`
- `ToInt64`

The `ToBoolean` method never fails. It uses JavaScript's definition of truthiness: false, 0, 0.0, NaN, BsonNull, BsonUndefined and "" are false, and everything else is true (include the string "false").

The `ToBoolean` method is particularly useful when the documents you are processing might have inconsistent ways of recording true/false values:

```
if (employee["ismanager"].ToBoolean()) {
    // we know the employee is a manager
// works with many ways of recording boolean values
}
```

The `ToDouble`, `ToInt32`, and `ToInt64` methods never fail when converting between numeric types, though the value might be truncated if it doesn't fit in the target type. A string can be converted to a numeric type, but an exception will be thrown if the string cannot be parsed as a value of the target type.

Static Create methods

Because BsonValue is an abstract class you cannot create instances of BsonValue (only instances of concrete subclasses). BsonValue has a static `Create` method that takes an argument of type `object` and determines at runtime the actual type of BsonValue to create. Subclasses of BsonValue also have static Create methods tailored to their own needs.

Implicit conversions

Implicit conversions are defined from the following .NET types to BsonValue:

- `bool`
- `byte[]`
- `DateTime`
- `double`
- `Enum`
- `Guid`
- `int`
- `long`
- `ObjectId`
- `Regex`
- `string`

These eliminate the need for almost all calls to BsonValue constructors or Create methods. For example:

```
BsonValue b = true; // b is an instance of BsonBoolean
BsonValue d = 3.14159; // d is an instance of BsonDouble
BsonValue i = 1; // i is an instance of BsonInt32
BsonValue s = "Hello"; // s is an instance of BsonString
```

BsonMaxKey, BsonMinKey, BsonNull and BsonUndefined

These classes are singletons, so only a single instance of each class exists. You refer to these instances using the static Value property of each class:

```
document["status"] = BsonNull.Value;
document["priority"] = BsonMaxKey.Value;
```

Note that C# null and BsonNull.Value are two different things. The latter is an actual C# object that represents a BSON null value (it's a subtle difference, but plays an important role in functional construction).

ObjectId and BsonObjectId

ObjectId is a struct that holds the raw value of a BSON ObjectId. BsonObjectId is a subclass of BsonValue whose Value property is of type ObjectId.

Here are some common ways of creating ObjectId values:

```
var id1 = new ObjectId(); // same as ObjectId.Empty
var id2 = ObjectId.Empty; // all zeroes
var id3 = ObjectId.GenerateNewId(); // generates new unique Id
var id4 = ObjectId.Parse("4dad901291c2949e7a5b6aa8"); // parses a 24 hex digit string
```

Note that the first example behaves differently in C# than in JavaScript. In C# it creates an ObjectId of all zeroes, but in JavaScript it generates a new unique Id. This difference can't be avoided because in C# the default constructor of a value type always initializes the value to all zeros.

BsonElement

A BsonElement is a name/value pair, where the value is a BsonValue. It is used as the building block of BsonDocument, which consists of zero or more elements. You will rarely create BsonElements directly, as they are usually created indirectly as needed. For example:

```
document.Add(new BsonElement("age", 21)); // OK, but next line is shorter
document.Add("age", 21); // creates BsonElement automatically
```

BsonDocument

A BsonDocument is a collection of name/value pairs (represented by BsonElements). It is an in-memory object model of a BSON document. There are three ways to create and populate a BsonDocument:

1. Create a new document and call Add and Set methods
2. Create a new document and use the fluent interface Add and Set methods
3. Create a new document and use C#'s collection initializer syntax (recommended)

BsonDocument constructor

BsonDocument has the following constructors:

- `BsonDocument()`
- `BsonDocument(string name, BsonValue value)`
- `BsonDocument(BsonElement element)`
- `BsonDocument(Dictionary<string, object> dictionary)`
- `BsonDocument(Dictionary<string, object> dictionary, IEnumerable<string> keys)`
- `BsonDocument(IDictionary dictionary)`
- `BsonDocument(IDictionary dictionary, IEnumerable<string> keys)`
- `BsonDocument(IDictionary<string, object> dictionary)`
- `BsonDocument(IDictionary<string, object> dictionary, IEnumerable<string> keys)`
- `BsonDocument(IEnumerabe<BsonElement> elements)`

* `BsonDocument(params BsonElement[] elements)`
* `BsonDocument(bool allowDuplicateNames)`

The first two are the ones you are most likely to use. The first creates an empty document, and the second creates a document with one element (in both cases you can of course add more elements).

All the constructors (except the one with allowDuplicateNames) simply call the Add method that takes the same parameters, so refer to the coresponding Add method for details about how the new document is initially populated.

A BsonDocument normally does not allow duplicate names, but if you must allow duplicate names call the constructor with the allowDuplicateNames parameter and pass in true. It is **not** recommended that you allow duplicate names, and this option exists only to allow handling existing BSON documents that might have duplicate names. MongoDB makes no particular guarantees about whether it supports documents with duplicate names, so be cautious about sending any such documents you construct to the server.

Create a new document and call Add and Set methods

This is a traditional step by step method to create and populate a document using multiple C# statements. For example:

```
BsonDocument book = new BsonDocument();
book .Add("author", "Ernest Hemingway");
book .Add("title", "For Whom the Bell Tolls");
```

Create a new document and use the fluent interface Add and Set methods

This is similar to the previous approach but the fluent interface allows you to chain the various calls to Add so that they are all a single C# statement. For example:

```
BsonDocument book = new BsonDocument()
    .Add("author", "Ernest Hemingway")
    .Add("title", "For Whom the Bell Tolls");
```

Create a new document and use C#'s collection initializer syntax (recommended)

This is the recommended way to create and initialize a BsonDocument in one statement. It uses C#'s collection initializer syntax:

```
BsonDocument book = new BsonDocument {
    { "author", "Ernest Hemingway" },
    { "title", "For Whom the Bell Tolls" }
};
```

The compiler translates this into calls to the matching Add method:

```
BsonDocument book = new BsonDocument();
    book.Add("author", "Ernest Hemingway");
    book.Add("title", "For Whom the Bell Tolls");
```

A common mistake is to forget the inner set of braces. This will result in a compilation error. For example:

```
BsonDocument bad = new BsonDocument {
    "author", "Ernest Hemingway"
};
```

is translated by the compiler to:

```
BsonDocument bad = new BsonDocument();
bad.Add("author");
bad.Add("Ernest Hemingway");
```

which results in a compilation error because there is no Add method that takes a single string argument.

Creating nested BSON documents

Nested BSON documents are created by setting the value of an element to a BSON document. For example:

```
BsonDocument nested = new BsonDocument {
    { "name", "John Doe" },
    { "address", new BsonDocument {
        { "street", "123 Main St." },
        { "city", "Centerville" },
        { "state", "PA" },
        { "zip", 12345}
    }}
};
```

This creates a top level document with two elements ("name" and "address"). The value of "address" is a nested BSON document.

Add methods

BsonDocument has the following overloaded Add methods:

- `Add(BsonElement element)`
- `Add(Dictionary<string, object> dictionary)`
- `Add(Dictionary<string, object> dictionary, IEnumerable<string> keys)`
- `Add(IDictionary dictionary)`
- `Add(IDictionary dictionary, IEnumerable<string> keys)`
- `Add(IDictionary<string, object> dictionary)`
- `Add(IDictionary<string, object> dictionary, IEnumerable<string> keys)`
- `Add(IEnumerable<BsonElement> elements)`
- `Add(string name, BsonValue value)`
- `Add(string name, BsonValue value, bool condition)`

It is important to note that sometimes the Add methods **don't** add a new element. If the value supplied is null (or the condition supplied in the last overload is false) then the element isn't added. This makes it really easy to handle optional elements without having to write any if statements or conditional expressions.

For example:

```
BsonDocument document = new BsonDocument {
    { "name", name },
    { "city", city }, // not added if city is null
{ "dob", dob, dobAvailable } // not added if dobAvailable is false
};
```

is more compact and readable than:

```
BsonDocument document = new BsonDocument();
document.Add("name", name);
if (city != null) {
    document.Add("city", city);
}
if (dobAvailable) {
    document.Add("dob", dob);
}
```

If you want to add a BsonNull if a value is missing you have to say so. A convenient way is to use C#'s null coalescing operator as follows:

```
BsonDocument = new BsonDocument {
    { "city", city ?? BsonConstants.Null }
};
```

The IDictionary overloads initialize a BsonDocument from a dictionary. Each key in the dictionary becames the name of a new element, and each value is mapped to a matching BsonValue and becomes the value of the new element. The overload with the keys parameter lets you select which dictionary entries to load (you might also use the keys parameter to control the order in which the elements are loaded from the dictionary).

Accessing BsonDocument elements

The recommended way to access BsonDocument elements is to use one of the following indexers:

- `BsonValue this[int index]`
- `BsonValue this[string name]`
- `BsonValue this[string name, BsonValue defaultValue]`

Note that the return value of the indexers is BsonValue, not BsonElement. This actually makes BsonDocuments much easier to work with (if you ever need to get the actual BsonElements use GetElement).

We've already seen samples of accessing BsonDocument elements. Here are some more:

```
BsonDocument book;
string author = book["author"].AsString;
DateTime publicationDate = book["publicationDate"].AsDateTime;
int pages = book["pages", -1].AsInt32; // default value is -1
```

BsonArray

This class is used to represent BSON arrays. While arrays happen to be represented externally as BSON documents (with a special naming convention for the elements), the BsonArray class is unrelated to the BsonDocument class because they are used very differently.

Constructors

BsonArray has the following constructors:

- `BsonArray()`
- `BsonArray(IEnumerable<bool> values)`
- `BsonArray(IEnumerable<BsonValue> values)`
- `BsonArray(IEnumerable<DateTime> values)`
- `BsonArray(IEnumerable<double> values)`
- `BsonArray(IEnumerable<int> values)`
- `BsonArray(IEnumerable<long> values)`
- `BsonArray(IEnumerable<ObjectId> values)`
- `BsonArray(IEnumerable<string> values)`
- `BsonArray(IEnumerable values)`

All the constructors with a parameter call the matching Add method. The multiple overloads are needed because C# does not provide automatic conversions from IEnumerable<T> to IEnumerable<object>.

Add and AddRange methods

Bson Array has the following Add methods:

- `BsonArray Add(BsonValue value)`
- `BsonArray AddRange(IEnumerable<bool> values)`
- `BsonArray AddRange(IEnumerable<BsonValue> values)`
- `BsonArray AddRange(IEnumerable<DateTime> values)`
- `BsonArray AddRange(IEnumerable<double> values)`
- `BsonArray AddRange(IEnumerable<int> values)`
- `BsonArray AddRange(IEnumerable<long> values)`
- `BsonArray AddRange(IEnumerable<ObjectId> values)`
- `BsonArray AddRange(IEnumerable<string> values)`
- `BsonArray AddRange(IEnumerable values)`

Note that the Add method takes a single parameter. To create and initialize a BsonArray with multiple values use any of the following approaches:

```
// traditional approach
BsonArray a1 = new BsonArray();
a1.Add(1);
a2.Add(2);

// fluent interface
BsonArray a2 = new BsonArray().Add(1).Add(2);

// values argument
int[] values = new int[] { 1, 2 };
BsonArray a3 = new BsonArray(values);

// collection initializer syntax
BsonArray a4 = new BsonArray { 1, 2 };
```

Indexer

Array elements are accessed using an integer index. Like BsonDocument, the type of the elements is BsonValue. For example:

```
BsonArray array = new BsonArray { "Tom", 39 };
string name = array[0].AsString;
int age = array[1].AsInt32;
```

The C# Driver

Up until now we have been focusing on the BSON Library. The remainder of this tutorial focuses on the C# Driver.

Thread safety

Only a few of the C# Driver classes are thread safe. Among them: MongoServer, MongoDatabase, MongoCollection and MongoGridFS. Common classes you will use a lot that are not thread safe include MongoCursor and all the classes from the BSON Library (except BsonSymbolTable which is thread safe). A class is not thread safe unless specifically documented as being thread safe.

All static properties and methods of all classes are thread safe.

MongoServer class

This class serves as the root object for working with a MongoDB server. You will create one instance of this class for each server you connect to. The connections to the server are handled automatically behind the scenes (a connection pool is used to increase efficiency).

When you are connecting to a replica set you will still use only one instance of MongoServer, which represents the replica set as a whole. The driver automatically finds all the members of the replica set and identifies the current primary. MongoServer has several properties you can use to find out more about the current state of the replica set (such as Primary, Secondaries, etc...).

Instances of this class are thread safe.

Connection strings

The easiest way to connect to a MongoDB server is to use a connection string. The standard connection string format is:

```
mongodb://[username:password@]hostname[:port][/[database][?options]]
```

The username and password should only be present if you are using authentication on the MongoDB server. These credentials will be the default credentials for all databases. To authenticate against the admin database append "(admin)" to the username. If you are using different credentials with different databases pass the appropriate credentials to the GetDatabase method.

The port number is optional and defaults to 27017.

To connect to a replica set specify the seed list by providing multiple hostnames (and port numbers if required) separated by commas. For example:

```
mongodb://server1,server2:27017,server2:27018
```

This connection string specifies a seed list consisting of three servers (two of which are on the same machine but on different port numbers).

The C# Driver is able to connect to a replica set even if the seed list is incomplete. It will find the primary server even if it is not in the seed list as long as at least one of the servers in the seed list responds (the response will contain the full replica set and the name of the current primary).

The options part of the connection string is used to set various connection options. For example, to turn SafeMode on by default for all operations, you could use:

```
mongodb://localhost/?safe=true
```

You should almost always add "safe=true" to your connection string.

As another example, suppose you wanted to connect directly to a member of a replica set regardless of whether it was the current primary or not (perhaps to monitor its status or to issue read only queries against it). You could use:

```
mongodb://server2/?connect=direct;slaveok=true
```

The full documentation for connection strings can be found at:

http://www.mongodb.org/display/DOCS/Connections

Create method

To obtain an instance of MongoServer use one of the Create methods:

- `MongoServer Create()`
- `MongoServer Create(MongoConnectionStringBuilder builder)`
- `MongoServer Create(MongoServerSettings settings)`
- `MongoServer Create(MongoUrl url)`
- `MongoServer Create(string connectionString)`
- `MongoServer Create(Uri uri)`

For example:

```
string connectionString = "mongodb://localhost";
MongoServer server = MongoServer.Create(connectionString);
```

Create maintains a table of MongoServer instances it has returned before, so if you call Create again with the same parameters you get the same instance back again.

The recommended way to call Create is with a connection string in the MongoDB URL format. MongoConnectionStringBuilder is provided for compatibility with how .NET handles SQL Server connection strings, but we recommend you use the URL format instead.

GetDatabase method

You can navigate from an instance of MongoServer to an instance of MongoDatabase (see next section) using one of the following GetDatabase methods or indexers:

- `MongoDatabase GetDatabase(MongoDatabaseSettings settings)`
- `MongoDatabase GetDatabase(string databaseName)`
- `MongoDatabase GetDatabase(string databaseName, MongoCredentials credentials)`
- `MongoDatabase GetDatabase(string databaseName, MongoCredentials credentials, SafeMode safeMode)`
- `MongoDatabase GetDatabase(string databaseName, SafeMode safeMode)`
- `MongoDatabase this[MongoDatabaseSettings settings]`
- `MongoDatabase this[string databaseName]`
- `MongoDatabase this[string databaseName, MongoCredentials credentials]`
- `MongoDatabase this[string databaseName, MongoCredentials credentials, SafeMode safeMode]`
- `MongoDatabase this[string databaseName, SafeMode safeMode]`

Sample code:

```
MongoServer server = MongoServer.Create(); // connect to localhost
MongoDatabase test = server.GetDatabase("test");
MongoCredentials credentials = new MongoCredentials("username", "password");
MongoDatabase salaries = server.GetDatabase("salaries", credentials);
```

Most of the database settings are inherited from the server object, and the provided overloads of GetDatabase let you override a few of the most commonly used settings. To override other settings, call CreateDatabaseSettings and change any settings you want before calling GetDatabase, like this:

```
var databaseSettings = server.CreateDatabaseSettings("test");
databaseSettings.SlaveOk = true;
var database = server[databaseSettings];
```

GetDatabase maintains a table of MongoDatabase instances it has returned before, so if you call GetDatabase again with the same parameters you get the same instance back again.

RequestStart/RequestDone methods

Sometimes a series of operations needs to be performed on the same connection in order to guarantee correct results. This is rarely the case, and most of the time there is no need to call RequestStart/RequestDone. An example of when this might be necessary is when a series of Inserts are called in rapid succession with SafeMode off, and you want to query that data in a consistent manner immediately thereafter (with SafeMode off the writes can queue up at the server and might not be immediately visible to other connections). Using RequestStart you can force a query to be on the same connection as the writes, so the query won't execute until the server has caught up with the writes.

A thread can temporarily reserve a connection from the connection pool by using RequestStart and RequestDone. For example:

```
server.RequestStart(database);
// a series of operations that must be performed on the same connection
server.RequestDone();
```

The database parameter simply indicates some database which you intend to use during this request. This allows the server to pick a connection that is already authenticated for that database (if you are not using authentication then this optimization won't matter to you). You are free to use any other databases as well during the request.

There is actually a slight problem with this example: if an exception is thrown while performing the operations then RequestDone is never called. You could put the call to RequestDone in a finally block, but even easier is to use the C# using statement:

```
using (server.RequestStart(database)) {
    // a series of operations that must be performed on the same connection
}
```

This works because RequestStart returns a helper object that implements IDisposable and calls RequestDone for you.

RequestStart increments a counter (for this thread) and RequestDone decrements the counter. The connection that was reserved is not actually returned to the connection pool until the count reaches zero again. This means that calls to RequestStart/RequestDone can be nested and the right thing will happen.

Other properties and methods

MongoServer has the following properties:

- `Arbiters`
- `BuildInfo`
- `ConnectionAttempt`
- `IndexCache`
- `Instance` (and `Instances`)
- `MaxServerCount`
- `Passives`
- `Primary`
- `ReplicaSetName`
- `RequestNestingLevel`
- `Secondaries`
- `SequentialId`
- `ServerCount`
- `Settings`
- `State`

MongoServer has the following additional methods:

- `Connect`
- `CopyDatabase`
- `DatabaseExists`
- `Disconnect`
- `DropDatabase`
- `FetchDBRef`
- `FetchDBRefAs`
- `GetDatabaseNames`
- `GetLastError`
- `Ping`
- `Reconnect`
- `Shutdown`
- `UnregisterServer`
- `VerifyState`

MongoDatabase class

This class represents a database on a MongoDB server. Normally there will be only one instance of this class per database, unless you are using different settings to access the same database, in which case there will be one instance for each set of settings.

Instances of this class are thread safe.

GetCollection method

This method returns an object representing a collection in a database. When we request a collection object, we also specify the default document type for the collection. For example:

```
MongoDatabase hr = server.GetDatabase("hr");
MongoCollection<Employee> employees =
    hr.GetCollection<Employee>("employees");
```

A collection is not restricted to containing only one kind of document. The default document type simply makes it more convenient to work with that kind of document, but you can always specify a different kind of document when required.

Most of the collection settings are inherited from the database object, and the provided overloads of GetCollection let you override a few of the most commonly used settings. To override other settings, call CreateCollectionSettings and change any settings you want before calling GetCollection , like this:

```
var collectionSettings = database.CreateCollectionSettings<TDocument>("test");
collectionSettings.SlaveOk = true;
var collection = database.GetCollection(collectionSettings);
```

GetCollection maintains a table of instances it has returned before, so if you call GetCollection again with the same parameters you get the same instance back again.

Other properties and methods

MongoDatabase has the following properties:

- `CommandCollection`
- `Credentials`
- `GridFS`
- `Name`
- `Server`
- `Settings`

MongoDatabase has the following additional methods:

- `AddUser`
- `CollectionExists`
- `CreateCollection`
- `Drop`
- `DropCollection`
- `Eval`
- `FetchDBRef`
- `FetchDBRefAs`
- `FindAllUsers`
- `FindUser`
- `GetCollectionNames`
- `GetCurrentOp`
- `GetGridFS`
- `GetSisterDatabase`
- `GetStats`
- `RemoveUser`
- `RenameCollection`
- `RequestDone`
- `RequestStart`
- `RunCommand`
- `RunCommandAs`

MongoCollection<TDefaultDocument> class

This class represents a collection in a MongoDB database. The <TDefaultDocument> type parameter specifies the type of the default document for this collection.

Instances of this class are thread safe.

Insert<TDocument> method

To insert a document in the collection create an object representing the document and call Insert. The object can be an instance of BsonDocument or of any class that can be successfully serialized as a BSON document. For example:

```
MongoCollection<BsonDocument> books =
    database.GetCollection<BsonDocument>("books");
BsonDocument book = new BsonDocument {
    { "author", "Ernest Hemingway" },
    { "title", "For Whom the Bell Tolls" }
};
books.Insert(book);
```

If you have a class called Book the code might look like:

```
MongoCollection<Book> books = database.GetCollection<Book>("books");
Book book = new Book {
    Author = "Ernest Hemingway",
    Title = "For Whom the Bell Tolls"
};
books.Insert(book);
```

InsertBatch method

You can insert more than one document at a time using the InsertBatch method. For example:

```
MongoCollection<BsonDocument> books;
BsonDocument[] batch = {
    new BsonDocument {
        { "author", "Kurt Vonnegut" },
        { "title", "Cat's Cradle" }
    },
    new BsonDocument {
        { "author", "Kurt Vonnegut" },
        { "title", "Slaughterhouse-Five" }
    }
};
books.InsertBatch(batch);
```

When you are inserting multiple documents InsertBatch can be much more efficient than Insert, specially when using SafeMode.

FindOne and FindOneAs methods

To retrieve documents from a collection use one of the various Find methods. FindOne is the simplest. It returns the first document it finds (when there are many documents in a collection you can't be sure which one it will be). For example:

```
MongoCollection<Book> books;
Book book = books.FindOne();
```

If you want to read a document that is not of the <TDefaultDocument> type use the FindOneAs method, which allows you to override the type of the returned document. For example:

```
MongoCollection<Book> books;
BsonDocument document = books.FindOneAs<BsonDocument>();
```

In this case the default document type of the collection is Book, but we are overriding that and specifying that the result be returned as an instance of BsonDocument.

Find and FindAs methods

The Find and FindAs methods take a query that tells the server which documents to return. The query parameter is of type IMongoQuery. IMongoQuery is a marker interface that identifies classes that can be used as queries. The most common ways to construct a query are to either use the Query builder class or to create a QueryDocument yourself (a QueryDocument is a subclass of BsonDocument that also implements IMongoQuery and can therefore be used as a query object). Also, by using the QueryWrapper class the query can be of any type that can be successfully serialized to a BSON document, but it is up to you to make sure that the serialized document represents a valid query object.

One way to query is to create a QueryDocument object yourself:

```
MongoCollection<BsonDocument> books;
var query = new QueryDocument("author", "Kurt Vonnegut");
foreach (BsonDocument book in books.Find(query)) {
    // do something with book
}
```

Another way to query is to use the Query Builder (recommended):

```
MongoCollection<BsonDocument> books;
var query = Query.EQ("author", "Kurt Vonnegut");
foreach (BsonDocument book in books.Find(query)) {
    // do something with book
}
```

Yet another way to query is to use an anonymous class as the query, but in this case we must wrap the anonymous object:

```
MongoCollection<BsonDocument> books;
var query = Query.Wrap(new { author = "Kurt Vonnegut" });
foreach (BsonDocument book in books.Find(query)) {
    // do something with book
}
```

If you want to read a document of a type that is not the default document type use the FindAs method instead:

```
MongoCollection<BsonDocument> books;
var query = Query.EQ("author", "Kurt Vonnegut");
foreach (Book book in books.FindAs<Book>(query)) {
    // do something with book
}
```

Save<TDocument> method

The Save method is a combination of Insert and Update. If the Id member of the document has a value, then it is assumed to be an existing document and Save calls Update on the document (setting the Upsert flag just in case it actually is a new document after all). Otherwise it is assumed to be a new document and Save calls Insert after first assigning a newly generated unique value to the Id member.

For example, you could correct an error in the title of a book using:

```
MongoCollection<BsonDocument> books;
var query = Query.And(
    Query.EQ("author", "Kurt Vonnegut"),
    Query.EQ("title", "Cats Craddle")
);
BsonDocument book = books.FindOne(query);
if (book != null) {
    book["title"] = "Cat's Cradle";
    books.Save(book);
}
```

The TDocument class must have an Id member to be used with the Save method. If it does not you can call Insert instead of Save to insert the document.

Update method

The Update method is used to update existing documents. The code sample shown for the Save method could also have been written as:

```
MongoCollection<BsonDocument> books;
var query = new QueryDocument {
    { "author", "Kurt Vonnegut" },
    { "title", "Cats Craddle" }
};
var update = new UpdateDocument {
    { "$set", new BsonDocument("title", "Cat's Cradle") }
};
BsonDocument updatedBook = books.Update(query, update);
```

or using Query and Update builders:

```
MongoCollection<BsonDocument> books;
var query = Query.And(
    Query.EQ("author", "Kurt Vonnegut"),
    Query.EQ("title", "Cats Craddle")
);
var update = Update.Set("title", "Cat's Cradle");
BsonDocument updatedBook = books.Update(query, update);
```

FindAndModify method

Use FindAndModify when you want to find a matching document and update it in one atomic operation. FindAndModify always updates a single document, and you can combine a query that matches multiple documents with a sort criteria that will determine exactly which matching document is updated. In addition, FindAndModify will return the matching document (either as it was before the update or after) and if you wish you can specify which fields of the matching document to return.

Using the example documented here:

http://www.mongodb.org/display/DOCS/findAndModify+Command

the call to FindAndModify would be written in C# as:

```
var jobs = database.GetCollection("jobs");
var query = Query.And(
    Query.EQ("inprogress", false),
    Query.EQ("name", "Biz report")
);
var sortBy = SortBy.Descending("priority");
var update = Update.
    .Set("inprogress", true)
    .Set("started", DateTime.UtcNow);
var result = jobs.FindAndModify(
    query,
    sortBy,
    update,
    true // return new document
);
var chosenJob = result.ModifiedDocument;
```

MapReduce method

Map/Reduce is a way of a aggregating data from a collection. Every document in a collection (or some subset if an optional query is provided) is sent to the map function, which calls emit to produce intermediate values. The intermediate values are then sent to the reduce function to be aggregated.

This example is taken from page 87 of MongoDB: The Definitive Guide, by Kristina Chodorow and Michael Dirolf. It counts how many times each key is found in a collection.

```
var map =
    "function() {" +
    "    for (var key in this) {" +
    "        emit(key, { count : 1 });" +
    "    }" +
    "}";

var reduce =
    "function(key, emits) {" +
    "    total = 0;" +
    "    for (var i in emits) {" +
    "        total += emits[i].count;" +
    "    }" +
    "    return { count : total };" +
    "}";

var mr = collection.MapReduce(map, reduce);
foreach (var document in mr.GetResults()) {
    Console.WriteLine(document.ToJson());
}
```

Other properties and methods

MongoCollection has the following properties:

- `Database`
- `FullName`
- `Name`
- `Settings`

MongoCollection has the following additional methods:

- `Count`
- `CreateIndex`
- `Distinct`
- `Drop`
- `DropAllIndexes`
- `DropIndex`
- `DropIndexByName`
- `EnsureIndex`
- `Exists`
- `Find`
- `FindAll`
- `FindAllAs`
- `FindAndModify`
- `FindAndRemove`
- `FindAs`
- `FindOne`
- `FindOneAs`
- `FindOneById`
- `FindOneByIdAs`
- `GeoNear`
- `GeoNearAs`
- `GetIndexes`
- `GetStats`
- `GetTotalDataSize`
- `GetTotalStorageSize`
- `Group`
- `IndexExists`
- `IndexExistsByName`
- `IsCapped`
- `MapReduce`
- `ReIndex`
- `Remove`
- `RemoveAll`
- `ResetIndexCache`
- `Save`
- `Validate`

MongoCursor<TDocument> class

The Find method (and its variations) don't immediately return the actual results of a query. Instead they return a cursor that can be enumerated to retrieve the results of the query. The query isn't actually sent to the server until we attempt to retrieve the first result (technically, when MoveNext is called for the first time on the enumerator returned by GetEnumerator). This means that we can control the results of the query in interesting ways by modifying the cursor before fetching the results.

Instances of MongoCursor are not thread safe, at least not until they are frozen (see below). Once they are frozen they are thread safe because they are read-only (in particular, GetEnumerator is thread safe so the same cursor *could* be used by multiple threads).

Enumerating a cursor

The most convenient way to consume the results of a query is to use the C# foreach statement. For example:

```
var query = Query.EQ("author", "Ernest Hemingway");
var cursor = books.Find(query);
foreach (var book in cursor) {
    // do something with book
}
```

You can also use any of the extensions methods defined by LINQ for IEnumerable<T> to enumerate a cursor:

```
var query = Query.EQ("author", "Ernest Hemingway");
var cursor = books.Find(query);
var firstBook = cursor.FirstOrDefault();
var lastBook = cursor.LastOrDefault();
```

Note that in the above example the query is actually sent to the server twice (once when FirstOrDefault is called and again when LastOrDefault is called).

It is important that a cursor cleanly release any resources it holds. The key to guaranteeing this is to make sure the Dispose method of the enumerator is called. The foreach statement and the LINQ extension methods all guarantee that Dispose will be called. Only if you enumerate the cursor manually are you responsible for calling Dispose.

Modifying a cursor before enumerating it

A cursor has several properties that can be modified before it is enumerated to control the results returned. There are two ways to modify a cursor:

1. modify the properties directly
2. use the fluent interface to set the properties

For example, if we want to skip the first 100 results and limit the results to the next 10, we could write:

```
var query = Query.EQ("status", "pending");
var cursor = tasks.Find(query);
cursor.Skip = 100;
cursor.Limit = 10;
foreach (var task in cursor) {
    // do something with task
}
```

or using the fluent interface:

```
var query = Query.EQ("status", "pending");
foreach (var task in tasks.Find(query).SetSkip(100).SetLimit(10)) {
    // do something with task
}
```

The fluent interface works well when you are setting only a few values. When setting more than a few you might prefer to use the properties approach.

Once you begin enumerating a cursor it becomes "frozen" and you can no longer change any of its properties. So you must set all the properties before you start enumerating it.

Modifiable properties of a cursor

The following properties of a cursor are modifiable:

- `BatchSize` (`SetBatchSize`)
- `Fields` (`SetFields`)
- `Flags` (`SetFlags`)
- `Limit` (`SetLimit`)
- `Options` (`SetOption` and `SetOptions`)
- `SerializationOptions` (`SetSerializationOptions`)
- `Skip` (`SetSkip`)
- `SlaveOk` (`SetSlaveOk`)

The method names in parenthesis are the corresponding fluent interface methods.

The fluent interface also supports additional options that aren't used very frequently and are not exposed as properties:

- `SetHint`
- `SetMax`
- `SetMaxScan`
- `SetMin`
- `SetShowDiskLoc`
- `SetSnapshot`
- `SetSortOrder`

Other methods

MongoCursor has a few methods used for some special purpose operations:

- `Clone`
- `Count`
- `Explain`
- `Size`

SafeMode class

There are various levels of SafeMode, and this class is used to represent those levels. SafeMode applies only to operations that don't already return a value (so it doesn't apply to queries or commands). It applies to the following MongoCollection methods: Insert, Remove, Save and Update.

The gist of SafeMode is that after an Insert, Remove, Save or Update message is sent to the server it is followed by a GetLastError command so the driver can verify that the operation succeeded. In addition, when using replica sets it is possible to verify that the information has been replicated to some minimum number of secondary servers.

The SafeMode class is no longer immutable. The properties have been made settable to facilitate creation of new instances using object initializer syntax. While it is no longer immutable, SafeMode instances can now be "frozen" to make them immutable at run time. A SafeMode instance is not thread safe until it has been frozen, at which point it becomes thread safe.

Constructors

SafeMode has the following constructors:

- `SafeMode(bool enabled)`
- `SafeMode(bool enabled, bool fsync)`
- `SafeMode(bool enabled, bool fsync, int w)`
- `SafeMode(bool enabled, bool fsync, int w TimeSpan wtimeout)`
- `SafeMode(int w)`
- `SafeMode(int w, TimeSpan wtimeout)`
- `SafeMode(SafeMode other)`

Because SafeMode now has more properties than it originally did, we are no longer adding constructors with different combinations of parameters (there would just be too many). The existing constructors are kept for backward compatibility. The recommended way to instantiate an instance of SafeMode is to use the last constructor to initialize all properties from another instance of SafeMode and then use object initializer syntax to set the properties you want to change. For example:

```
var safeMode = new SafeMode(other) { WMode = "majority" };
```

which creates a new instance of SafeMode with all the same properties as the "other" instance with the WMode changed to "majority".

Properties

SafeMode has the following properties:

* `Enabled`
* `FSync`
* `IsFrozen`
* `J`
* `W`
* `WMode`
* `WTimeout`

Methods

SafeMode has the following methods:

* `Clone`
* `Freeze`
* `FrozenCopy`

CSharp getLastError and SafeMode

In the C# driver SafeMode can be set at different levels.

1. At the server level via the connection string:

```
var connectionString = "mongodb://hostname/?safe=true;w=2;wtimeout=30s";
var server = MongoServer.Create(connectionString);
```

2. At the database level:

```
var safemode = SafeMode.W2; // default timeout
// or
var safeMode = SafeMode.Create(2, TimeSpan.FromSeconds(30)); // 30 second timeout
var database = server.GetDatabase("test", safeMode);
```

3. At the collection level:

```
var collection = database.GetCollection("test", safeMode);
```

4. At the operation level:

```
var safeModeResult = collection.Insert(document, safeMode);
```

Each level inherits the setting from the level above it unless overridden.

getLastError is called automatically when any SafeMode other than false is used. An exception is thrown if there was an error, otherwise the SafeModeResult has the information returned by GetLastError.

See Also

* Connections
* getLastError Command
* Replica Set Design Concepts

Erlang Language Center

* Driver Download
  * https://github.com/mongodb/mongodb-erlang
* API Docs
* Design of the Erlang Driver post on blog.mongodb.org

Third Party Frameworks and Libs

- Mongrel - A record/document mapper that maps Erlang records to MongoDB documents
    - API Documentation for Mongrel

Tools and Libraries

- Talend Adapters

Driver Syntax Table

The wiki generally gives examples in JavaScript, so this chart can be used to convert those examples to any language.

JavaScript	Python	PHP	Ruby
`[ ]`	[]	`array()`	[]
{}	{}	`new stdClass`	{}
`{ x : 1 }`	{"x": 1}	`array('x' => 1)`	{'x' => 1}
`connect("www.example.net")`	`Connection("www.example.net")`	`new Mongo("www.example.net")`	`Connection.new("www.exa`
`cursor.next()`	`cursor.next()`	`$cursor->getNext()`	`cursor.next_document()`
`cursor.hasNext()`	*	`$cursor->hasNext()`	`cursor.has_next?`
`collection.findOne()`	`collection.find_one()`	`$collection->findOne()`	`collection.find_one()`
`db.eval()`	`db.eval()`	`$db->execute()`	`db.eval()`

* does not exist in that language

Javascript Language Center

MongoDB can be

- Used by clients written in Javascript;
- Uses Javascript internally server-side for certain options such as map/reduce;
- Has a shell that is based on Javascript for administrative purposes.

node.JS and V8

See the node.JS page.

SpiderMonkey

The MongoDB shell extends SpiderMonkey. See the MongoDB shell documentation.

Narwhal

- http://github.com/sergi/narwhal-mongodb

MongoDB Server-Side Javascript

Javascript may be executed in the MongoDB server processes for various functions such as query enhancement and map/reduce processing. See Server-side Code Execution.

node.JS

Node.js is used to write event-driven, scalable network programs in server-side JavaScript. It is similar in purpose to Twisted, EventMachine, etc. It runs on Google's V8.

Web Frameworks

- ExpressJS Mature web framework with MongoDB session support.
- Connect Connect is a middleware framework for node, shipping with over 11 bundled middleware and a rich choice of 3rd-party middleware.

mongodb.org supported drivers

- node-mongodb-native - Native async Node interface to MongoDB.

3rd Party ORM/ODM

- Mongoskin - The future layer for node-mongodb-native.
- Mongoose - Asynchronous JavaScript Driver with optional support for Modeling.
- Mongolia - Lightweight MongoDB ORM/Driver Wrapper.

3rd Party Drivers

- node-mongodb - Async Node interface to MongoDB (written in C)
- Mongolian DeadBeef - A node.js driver that attempts to closely approximate the MongoDB shell.

Presentations

- An introduction to the mongo node.js driver - June 2011
- Using MongoDB with node.js - June 2011
- Node.js & MongoDB - Webinar June 2011
- A beautiful marriage: MongoDB and node.js - MongoNYC June 2011
- Rapid Realtime App Development with Node.JS & MongoDB - MongoSF May 2011

JVM Languages

Redirection Notice
This page should redirect to Java Language Center in about 3 seconds.

Python Language Center

This is an overview of the available tools for using Python with MongoDB. Those wishing to skip to more detailed discussion should check out the Python Driver Tutorial.

- Python Driver
- Python tools
    - ORM Like Layers
    - Framework Tools
- Alternative drivers
- Presentations
- Tutorials

Python Driver

PyMongo is the recommended way to work with MongoDB from Python.

- Installation
- Tutorial
- API Documentation
- Changelog
- Source Code

Python tools

ORM Like Layers

Because MongoDB is so easy to use the basic Python driver is often the best solution for many applications. However, if you need data validation, associations and other high-level data modeling functionality then ORM like layers may be desired.

- ORM like layers

Framework Tools

Several tools and adapters for integration with various Python frameworks and libraries also exist.

- Framework Tools

Alternative drivers

- Alternative driver list

Presentations

- MongoDB & Python - Workshop materials from PyCon 2012
- PyCon Poster - PyCon 2012
- Realtime Analytics using MongoDB, Python, Gevent, and ZeroMQ - Rick Copeland's presentation from Mongo Seattle (December 2011)
- MongoDB with Python, Pylons, and Pyramid - Niall O'Higgins' presentation from MongoSF (May 2011)
- Python Development with MongoDB - Bernie Hackett's presentation from MongoSF (May 2011)
- Building a Social Graph with MongoDB at Eventbrite - Brian Zambrano's presentation from MongoSV (December 2010)
- More Python-related presenatations

Tutorials

- Python Driver Tutorial
- Write a Tumblelog Application with Django MongoDB Engine
- Write a Tumblelog Application with Flask and MongoEngine

PHP Language Center

Using MongoDB in PHP

To access MongoDB from PHP you will need:

- The MongoDB server running - the server is the "mongo**d**" file, not the "mongo" client (note the "d" at the end)
- The MongoDB PHP driver installed

Installing the PHP Driver

*NIX

Run:

```
sudo pecl install mongo
```

Open your php.ini file and add to it:

```
extension=mongo.so
```

It is recommended to add this to the section with the other "extensions", but it will work from anywhere within the php.ini file.

Restart your web server (Apache, nginx, etc.) for the change to take effect.

See the installation docs for configuration information and OS-specific installation instructions.

Note

`pecl` requires that pear be installed. For those using apt-get, you may need to run the following:

```
sudo apt-get install php5-dev php5-cli php-pear
```

Windows

- Download the correct driver for your environment from http://github.com/mongodb/mongo-php-driver/downloads. Thread safe is for

running PHP as an Apache module (typical installation), non-thread safe is for CGI
- Unzip and add the php_mongo.dll file to your PHP extensions directory (usually the "ext" folder in your PHP installation.)
- Add to your php.ini:

```
extension=php_mongo.dll
```

- Restart your web server (Apache, IIS, etc.) for the change to take effect

For more information, see the Windows section of the installation docs.

Using the PHP Driver

To get started, see the Tutorial. Also check out the API Documentation.

See Also

- PHP Libraries, Frameworks, and Tools for working with Drupal, Cake, Symfony, and more from MongoDB.
- Admin UIs
- If you are using Eclipse, you can get Content Assist working by downloading the mongo_version.zip package.

Installing the PHP Driver

Redirection Notice
This page should redirect to http://www.php.net/manual/en/mongo.installation.php.

PHP Libraries, Frameworks, and Tools

- Libraries and Frameworks
  - CakePHP
  - Codeigniter
  - Doctrine
  - Drupal
  - Fat-Free Framework
  - Kohana Framework
  - Lithium
  - Symfony 2
  - TechMVC
  - Vork
  - Yii
  - Zend Framework
- Stand-Alone Tools
  - ActiveMongo
  - MapReduce API
  - Mongofilesystem
  - Mandango
  - MongoDB Pagination
  - MongoDb PHP ODM
  - Mongodloid
  - MongoQueue
  - MongoRecord
  - Morph
  - simplemongophp
- Blogs & HOWTOs
  - How to batch import JSON data output from FFprobe for motion stream analysis

The PHP community has created a huge number of libraries to make working with MongoDB easier and integrate it with existing frameworks.

Libraries and Frameworks

CakePHP

- MongoDB datasource for CakePHP. There's also an introductory blog post on using it with Mongo.

Codeigniter

- MongoDB-Codeigniter-Driver

Doctrine

ODM (Object Document Mapper) is an experimental Doctrine MongoDB object mapper. The Doctrine\ODM\Mongo namespace is an experimental project for a PHP 5.3 MongoDB Object Mapper. It allows you to easily write PHP 5 classes and map them to collections in MongoDB. You just work with your objects like normal and Doctrine will transparently persist them to Mongo.

This project implements the same "style" of the Doctrine 2 ORM project interface so it will look very familiar to you and it has lots of the same features and implementations.

- Documentation - API, Reference, and Cookbook
- Official blog post
- Screencast
- Blog post on using it with Symfony
- Bug tracker

Drupal

- MongoDB Integration - Views (query builder) backend, a watchdog implementation (logging), and field storage.

Fat-Free Framework

Fat-Free is a powerful yet lightweight PHP 5.3+ Web development framework designed to help you build dynamic and robust applications - fast!

Kohana Framework

- Mango at github
  An ActiveRecord-like library for PHP, for the Kohana PHP Framework.
  See also PHP Language Center#MongoDb PHP ODM further down.

Lithium

Lithium supports Mongo out-of-the-box.

- Tutorial on creating a blog backend.

Symfony 2

- Symfony 2 Logger
  A centralized logger for Symfony applications. See the blog post.

- sfMongoSessionStorage - manages session storage via MongoDB with symfony.

- sfStoragePerformancePlugin - This plugin contains some extra storage engines (MongoDB and Memcached) that are currently missing from the Symfony (>= 1.2) core.

TechMVC

An extensive MVC 2 based PHP framework which supports MongoDB directly with only PHPMongo extension. Hosted at http://sourceforge.net/projects/techmvc/ and demo example available at http://demo.techmvc.techunits.com/.

Vork

Vork, the high-performance enterprise framework for PHP natively supports MongoDB as either a primary datasource or used in conjunction with an RDBMS. Designed for scalability & Green-IT, Vork serves more traffic with fewer servers and can be configured to operate without any disk-IO.

Vork provides a full MVC stack that outputs semantically-correct XHTML 1.1, complies with Section 508 Accessibility guidelines & Zend-Framework coding-standards, has SEO-friendly URLs, employs CSS-reset for cross-browser display consistency and is written in well-documented object-oriented E_STRICT PHP5 code.

An extensive set of tools are built into Vork for ecommerce (cc-processing, SSL, PayPal, AdSense, shipment tracking, QR-codes), Google Maps, translation & internationalization, Wiki, Amazon Web Services, Social-Networking (Twitter, Meetup, ShareThis, YouTube, Flickr) and much more.

Yii

- YiiMongoDbSuite is an almost complete, ActiveRecord like support for MongoDB in Yii It originally started as a fork of MongoRecord extension written by tyohan, to fix some major bugs, and add full featured suite for MongoDB developers.

Zend Framework

- Shanty Mongo is a prototype mongodb adapter for the Zend Framework. It's intention is to make working with mongodb documents as natural and as simple as possible. In particular allowing embeded documents to also have custom document classes.

- ZF Cache Backend
  A ZF Cache Backend for MongoDB. It support tags and auto-cleaning.

- There is a Zend_Nosql_Mongo component proposal.

Stand-Alone Tools

ActiveMongo

ActiveMongo is a really simple ActiveRecord for MongoDB in PHP.

There's a nice introduction to get you started at http://crodas.org/activemongo.php.

MapReduce API

A MapReduce abstraction layer. See the blog post.

- MongoDB-MapReduce-PHP at github

Mongofilesystem

Filesystem based on MongoDB GridFS. Mongofilesystem will help you use MongoDB GridFS like a typical filesystem, using the familiar PHP commands.

Mandango

Mandango is a simple, poweful and ultrafast Object Document Mapper (ODM) for PHP and MongoDB..

MongoDB Pagination

PHP MongoDB Pagination is the pagination plugin for MongoDB released under MIT License. Simple to install & use. It has been developed under TechMVC 3.0.4, but it's compatible with any 3rd party framework (e.g. Zend (tested)).

MongoDb PHP ODM

MongoDb PHP ODM is a simple object wrapper for the Mongo PHP driver classes which makes using Mongo in your PHP application more like ORM, but without the suck. It is designed for use with Kohana 3 but will also integrate easily with any PHP application with almost no additional effort.

Mongodloid

A nice library on top of the PHP driver that allows you to make more natural queries (`$query->query('a == 13 AND b >= 8 && c % 3 == 4');`), abstracts away annoying $-syntax, and provides getters and setters.

- Project Page
- Downloads
- Documentation

MongoQueue

MongoQueue is a PHP queue that allows for moving tasks and jobs into an asynchronous process for completion in the background. The queue is managed by Mongo

MongoQueue is an extraction from online classifieds site Oodle. Oodle uses MongoQueue to background common tasks in order to keep page response times low.

MongoRecord

MongoRecord is a PHP Mongo ORM layer built on top of the PHP Mongo PECL extension

MongoRecord is an extraction from online classifieds site Oodle. Oodle’s requirements for a manageable, easy to understand interface for dealing with the super-scalable Mongo datastore was the primary reason for MongoRecord. It was developed to use with PHP applications looking to add Mongo’s scaling capabilities while dealing with a nice abstraction layer.

Morph

A high level PHP library for MongoDB. Morph comprises a suite of objects and object primitives that are designed to make working with MongoDB in PHP a breeze.

- Morph at github

simplemongophp

Very simple layer for using data objects see blog post

- simplemongophp at github

Blogs & HOWTOs

How to batch import JSON data output from FFprobe for motion stream analysis

FFprobe is a stream analyzer that optionally reports in JSON. This example is a PHP script that reads JSON from STDIN, makes an object using json_decode, and inserts the object into a MongDB database. This script could be used with any program that outputs a JSON stream. A bash script will be used to batch process all files within the current directory. For example, the data may be used for analysis and logging of a day's shoot.

- Batch import Multimedia Stream Data into MongoDB with FFprobe web site
- Code and Sample Output at github

PHP - Storing Files and Big Data

> **Redirection Notice**
> This page should redirect to http://www.php.net/manual/en/class.mongogridfs.php.

Troubleshooting the PHP Driver

> **Redirection Notice**
> This page should redirect to http://www.php.net/manual/en/mongo.trouble.php.

Ruby Language Center

This is an overview of the available tools and suggested practices for using Ruby with MongoDB. Those wishing to skip to more detailed discussion should check out the Ruby Driver Tutorial, Getting started with Rails or Rails 3, and MongoDB Data Modeling and Rails. There are also a number of good external resources worth checking out.

- Ruby Driver
  - Installing / Upgrading
  - BSON
- Object Mappers
- Notable Projects
- Presentations

Ruby Driver

> Install the bson_ext gem for any performance-critical applications.

The MongoDB Ruby driver is the 10gen-supported driver for MongoDB. It's written in pure Ruby, with a recommended C extension for speed. The driver is optimized for simplicity. It can be used on its own, but it also serves as the basis of several object mapping libraries, such as MongoMapper.

- Tutorial
- Ruby Driver README
- API Documentation
- Source Code

Installing / Upgrading

The ruby driver is hosted at Rubygems.org. Before installing the driver, make sure you're using the latest version of rubygems (currently 1.6.0 as of Feb 2012):

```
$ gem update --system
```

Then install the gems:

```
$ gem install mongo
```

To stay on the bleeding edge, check out the latest source from github:

```
$ git clone git://github.com/mongodb/mongo-ruby-driver.git
$ cd mongo-ruby-driver/
```

Then, install the driver from there:

```
$ rake gem:install
```

BSON

In versions of the Ruby driver prior to 0.20, the code for serializing to BSON existed in the mongo gem. Now, all BSON serialization is handled by the required bson gem.

```
gem install bson
```

For significantly improved performance, install the bson_ext gem. Using compiled C instead of Ruby, this gem speeds up BSON serialization greatly.

```
gem install bson_ext
```

If you're running on Windows, you'll need the Ruby DevKit installed in order to compile the C extensions.

As long it's in Ruby's load path, `bson_ext` will be loaded automatically when you require `bson`.

Note that beginning with version 0.20, the mongo_ext gem is no longer used.

To learn more about the Ruby driver, see the Ruby Tutorial.

Object Mappers

Because MongoDB is so easy to use, the basic Ruby driver can be the best solution for many applications.

But if you need validations, associations, and other high-level data modeling functions then an Object Document Mapper may be needed.

In the context of a Rails application these provide functionality equivalent to, but distinct from, ActiveRecord. Because Mongo is a document-based database, these mappers are called Object Document Mappers (ODM) as opposed to Object Relational Mappers (ORM).

Several mappers are available:

- MongoMapper from John Nunemaker
- Mongoid from Durran Jordan
- Mongomatic from Ben Myles
- MongoODM from Carlos Paramio
- MongoModel from Sam Pohlenz
- DriverAPILayer from Alexey Petrushin

All the mappers build on top of the basic Ruby driver and so some knowledge of that is useful, especially if you work with a custom Mongo configuration.

Notable Projects

Tools for working with MongoDB in Ruby are being developed daily. A partial list can be found in the Projects and Libraries section of our external resources page.

If you're working on a project that you'd like to have included, let us know.

Presentations

> *MongoDB from Ruby - MongoSF (May 2011)*
> *MongoDB & Ruby Presentations*

Ruby Tutorial

> **Redirection Notice**
> This page should redirect to http://api.mongodb.org/ruby/current/file.TUTORIAL.html.

This tutorial gives many common examples of using MongoDB with the Ruby driver. If you're looking for information on data modeling, see MongoDB Data Modeling and Rails. Links to the various object mappers are listed on our object mappers page.

Interested in GridFS? Checkout GridFS in Ruby.

As always, the latest source for the Ruby driver can be found on github.

Installation

The mongo-ruby-driver gem is served through Rubygems.org. To install, make sure you have the latest version of rubygems.

```
gem update --system
```

Next, install the mongo rubygem:

```
gem install mongo
```

The required `bson` gem will be installed automatically.

For optimum performance, install the bson_ext gem:

```
gem install bson_ext
```

After installing, you may want to look at the examples directory included in the source distribution. These examples walk through some of the basics of using the Ruby driver.

The full API documentation can be viewed here.

A Quick Tour

Using the RubyGem

All of the code here assumes that you have already executed the following Ruby code:

```
require 'rubygems'  # not necessary for Ruby 1.9
require 'mongo'
```

Making a Connection

An Mongo::Connection instance represents a connection to MongoDB. You use a Connection instance to obtain an Mongo:DB instance, which represents a named database. The database doesn't have to exist - if it doesn't, MongoDB will create it for you.

You can optionally specify the MongoDB server address and port when connecting. The following example shows three ways to connect to the database "mydb" on the local machine:

```
db = Mongo::Connection.new.db("mydb")
db = Mongo::Connection.new("localhost").db("mydb")
db = Mongo::Connection.new("localhost", 27017).db("mydb")
```

At this point, the db object will be a connection to a MongoDB server for the specified database. Each DB instance uses a separate socket connection to the server.

If you're trying to connect to a replica set, see Replica Sets in Ruby.

Listing All Databases

```
connection = Mongo::Connection.new # (optional host/port args)
connection.database_names.each { |name| puts name }
connection.database_info.each { |info| puts info.inspect}
```

Dropping a Database

```
connection.drop_database('database_name')
```

Authentication (Optional)

MongoDB can be run in a secure mode where access to databases is controlled through name and password authentication. When run in this mode, any client application must provide a name and password before doing any operations. In the Ruby driver, you simply do the following with the connected mongo object:

```
auth = db.authenticate(my_user_name, my_password)
```

If the name and password are valid for the database, `auth` will be `true`. Otherwise, it will be `false`. You should look at the MongoDB log for further information if available.

Getting a List Of Collections

Each database has zero or more collections. You can retrieve a list of them from the db (and print out any that are there):

```
db.collection_names.each { |name| puts name }
```

and assuming that there are two collections, name and address, in the database, you would see

```
name
address
```

as the output.

Getting a Collection

You can get a collection to use using the `collection` method:

```
coll = db.collection("testCollection")
```

This is aliased to the [] method:

```
coll = db["testCollection"]
```

Once you have this collection object, you can now do things like insert data, query for data, etc.

Inserting a Document

Once you have the collection object, you can insert documents into the collection. For example, lets make a little document that in JSON would be represented as

```
{
   "name" : "MongoDB",
   "type" : "database",
   "count" : 1,
   "info" : {
               x : 203,
               y : 102
             }
}
```

Notice that the above has an "inner" document embedded within it. To do this, we can use a Hash or the driver's OrderedHash (which preserves key order) to create the document (including the inner document), and then just simply insert it into the collection using the `insert()` method.

```
doc = {"name" => "MongoDB", "type" => "database", "count" => 1,
       "info" => {"x" => 203, "y" => '102'}}
coll.insert(doc)
```

Updating a Document

We can update the previous document using the `update` method. There are a couple ways to update a document. We can rewrite it:

```
doc["name"] = "MongoDB Ruby"
coll.update({"_id" => doc["_id"]}, doc)
```

Or we can use an atomic operator to change a single value:

```

coll.update({"_id" => doc["_id"]}, {"$set" => {"name" => "MongoDB Ruby"}})
```

Read more about updating documents.

Finding the First Document In a Collection using `find_one()`

To show that the document we inserted in the previous step is there, we can do a simple `find_one()` operation to get the first document in the collection. This method returns a single document (rather than the `Cursor` that the `find()` operation returns).

```
my_doc = coll.find_one()
puts my_doc.inspect
```

and you should see:

```
{"_id"=>#<BSON::ObjectID:0x118576c ...>, "name"=>"MongoDB", "info"=>{"x"=>203, "y"=>102}, "type"=>
"database", "count"=>1}
```

Note the `_id` element has been added automatically by MongoDB to your document.

Adding Multiple Documents

To demonstrate some more interesting queries, let's add multiple simple documents to the collection. These documents will have the following form:

```
{
   "i" : value
}
```

Here's how to insert them:

```
100.times { |i| coll.insert("i" => i) }
```

Notice that we can insert documents of different "shapes" into the same collection. These records are in the same collection as the complex record we inserted above. This aspect is what we mean when we say that MongoDB is "schema-free".

Counting Documents in a Collection

Now that we've inserted 101 documents (the 100 we did in the loop, plus the first one), we can check to see if we have them all using the `count()` method.

```
puts coll.count()
```

and it should print `101`.

Using a Cursor to get all of the Documents

To get all the documents from the collection, we use the `find()` method. `find()` returns a `Cursor` object, which allows us to iterate over the set of documents that matches our query. The Ruby driver's Cursor implemented Enumerable, which allows us to use `Enumerable#each`, {{Enumerable#map}, etc. For instance:

```
coll.find().each { |row| puts row.inspect }
```

and that should print all 101 documents in the collection.

Getting a Single Document with a Query

We can create a *query* hash to pass to the `find()` method to get a subset of the documents in our collection. For example, if we wanted to find the document for which the value of the "i" field is 71, we would do the following ;

```
coll.find("i" => 71).each { |row| puts row.inspect }
```

and it should just print just one document:

```
{"_id"=>#<BSON::ObjectID:0x117de90 ...>, "i"=>71}
```

Getting a Set of Documents With a Query

We can use the query to get a set of documents from our collection. For example, if we wanted to get all documents where "i" > 50, we could write:

```
coll.find("i" => {"$gt" => 50}).each { |row| puts row }
```

which should print the documents where i > 50. We could also get a range, say 20 < i <= 30:

```
coll.find("i" => {"$gt" => 20, "$lte" => 30}).each { |row| puts row }
```

Selecting a subset of fields for a query

Use the `:fields` option. If you just want fields "a" and "b":

```
coll.find("i" => {"$gt" => 50}, :fields => ["a", "b"]).each { |row| puts row }
```

Querying with Regular Expressions

Regular expressions can be used to query MongoDB. To find all names that begin with 'a':

```
coll.find({"name" => /^a/})
```

You can also construct a regular expression dynamically. To match a given search string:

```
  search_string = params['search']

  # Constructor syntax
  coll.find({"name" => Regexp.new(search_string)})

  # Literal syntax
  coll.find({"name" => /#{search_string}/})
```

Although MongoDB isn't vulnerable to anything like SQL-injection, it may be worth checking the search string for anything malicious.

Creating An Index

MongoDB supports indexes, and they are very easy to add on a collection. To create an index, you specify an index name and an array of field names to be indexed, or a single field name. The following creates an ascending index on the "i" field:

```
# create_index assumes ascending order; see method docs
# for details
coll.create_index("i")
```

To specify complex indexes or a descending index you need to use a slightly more complex syntax - the index specifier must be an Array of [field name, direction] pairs. Directions should be specified as Mongo::ASCENDING or Mongo::DESCENDING:

```
# explicit "ascending"
coll.create_index([["i", Mongo::ASCENDING]])
```

Creating and querying on a geospatial index

First, create the index on a field containing long-lat values:

```
  people.create_index([["loc", Mongo::GEO2D]])
```

Then get a list of the twenty locations nearest to the point 50, 50:

```
people.find({"loc" => {"$near" => [50, 50]}}, {:limit => 20}).each do |p|
  puts p.inspect
end
```

Getting a List of Indexes on a Collection

You can get a list of the indexes on a collection using `coll.index_information()`.

Database Administration

A database can have one of three profiling levels: off (:off), slow queries only (:slow_only), or all (:all). To see the database level:

```
puts db.profiling_level   # => off (the symbol :off printed as a string)
db.profiling_level = :slow_only
```

Validating a collection will return an interesting hash if all is well or raise an exception if there is a problem.

```
p db.validate_collection('coll_name')
```

See Also

- Ruby Driver Official Docs
- MongoDB Koans A path to MongoDB enlightenment via the Ruby driver.
- MongoDB Manual

Replica Sets in Ruby

Redirection Notice
This page should redirect to http://api.mongodb.org/ruby/current/file.REPLICA_SETS.html.

Here follow a few considerations for those using the Ruby driver with MongoDB and replica sets.

- Setup
- Connection Failures
- Recovery
- Testing
- Further Reading

Setup

First, make sure that you've configured and initialized a replica set.

Connecting to a replica set from the Ruby driver is easy. If you only want to specify a single node, simply pass that node to `Connection.new`:

```
  @connection = Connection.new('foo.local', 27017)
```

If you want to pass in multiple seed nodes, use `Connection.multi`:

```
  @connection = Connection.multi([['n1.mydb.net', 27017],
     ['n2.mydb.net', 27017], ['n3.mydb.net', 27017]])
```

In both cases, the driver will attempt to connect to a master node and, when found, will merge any other known members of the replica set into the seed list.

Connection Failures

Imagine that our master node goes offline. How will the driver respond?

At first, the driver will try to send operations to what was the master node. These operations will fail, and the driver will raise a **ConnectionFailure** exception. It then becomes the client's responsibility to decide how to handle this.

If the client decides to retry, it's not guaranteed that another member of the replica set will have been promoted to master right away, so it's still possible that the driver will raise another **ConnectionFailure**. However, once a member has been promoted to master, typically within a few seconds, subsequent operations will succeed.

The driver will essentially cycle through all known seed addresses until a node identifies itself as master.

Recovery

Driver users may wish to wrap their database calls with failure recovery code. Here's one possibility:

```
# Ensure retry upon failure
def rescue_connection_failure(max_retries=5)
    success = false
    retries = 0
    while !success
      begin
        yield
        success = true
      rescue Mongo::ConnectionFailure => ex
        retries += 1
        raise ex if retries >= max_retries
        sleep(1)
      end
    end
  end
end

# Wrapping a call to #count()
rescue_connection_failure do
  @db.collection('users').count()
end
```

Of course, the proper way to handle connection failures will always depend on the individual application. We encourage object-mapper and application developers to publish any promising results.

Testing

The Ruby driver (>= 1.0.6) includes some unit tests for verifying replica set behavior. They reside in **tests/replica_sets**. You can run them individually with the following rake tasks:

```
rake test:replica_set_count
rake test:replica_set_insert
rake test:pooled_replica_set_insert
rake test:replica_set_query
```

Make sure you have a replica set running on localhost before trying to run these tests.

Further Reading

- Replica Sets
- [Replics Set Configuration]

GridFS in Ruby

Redirection Notice
This page should redirect to http://api.mongodb.org/ruby/current/file.GridFS.html.

GridFS, which stands for "Grid File Store," is a specification for storing large files in MongoDB. It works by dividing a file into manageable chunks and storing each of those chunks as a separate document. GridFS requires two collections to achieve this: one collection stores each file's metadata (e.g., name, size, etc.) and another stores the chunks themselves. If you're interested in more details, check out the GridFS

Specification.

Prior to version 0.19, the MongoDB Ruby driver implemented GridFS using the GridFS::GridStore class. This class has been deprecated in favor of two new classes: Grid and GridFileSystem. These classes have a much simpler interface, and the rewrite has resulted in a significant speed improvement. **Reads are over twice as fast, and write speed has been increased fourfold**. 0.19 is thus a worthwhile upgrade.

The Grid class

The Grid class represents the core GridFS implementation. Grid gives you a simple file store, keyed on a unique ID. This means that duplicate filenames aren't a problem. To use the Grid class, first make sure you have a database, and then instantiate a Grid:

```
@db = Mongo::Connection.new.db('social_site')

@grid = Grid.new(@db)
```

Saving files

Once you have a Grid object, you can start saving data to it. The data can be either a string or an IO-like object that responds to a #read method:

```
# Saving string data
id = @grid.put("here's some string / binary data")

# Saving IO data and including the optional filename
image = File.open("me.jpg")
id2   = @grid.put(image, :filename => "me.jpg")
```

Grid#put returns an object id, which you can use to retrieve the file:

```
# Get the string we saved
file = @grid.get(id)

# Get the file we saved
image = @grid.get(id2)
```

File metadata

There are accessors for the various file attributes:

```
  image.filename
# => "me.jpg"

  image.content_type
# => "image/jpg"

  image.file_length
# => 502357

  image.upload_date
# => Mon Mar 01 16:18:30 UTC 2010

  # Read all the image's data at once
  image.read

  # Read the first 100k bytes of the image
  image.read(100 * 1024)
```

When putting a file, you can set many of these attributes and write arbitrary metadata:

```
# Saving IO data
  file = File.open("me.jpg")
  id2  = @grid.put(file,
           :filename     => "my-avatar.jpg",
           :content_type => "application/jpg",
           :_id          => 'a-unique-id-to-use-in-lieu-of-a-random-one',
           :chunk_size   => 100 * 1024,
           :metadata     => {'description' => "taken after a game of ultimate"})
```

Safe mode

A kind of safe mode is built into the GridFS specification. When you save a file, and MD5 hash is created on the server. If you save the file in safe mode, an MD5 will be created on the client for comparison with the server version. If the two hashes don't match, an exception will be raised.

```
  image = File.open("me.jpg")
  id2   = @grid.put(image, "my-avatar.jpg", :safe => true)
```

Deleting files

Deleting a file is as simple as providing the id:

```
  @grid.delete(id2)
```

The GridFileSystem class

GridFileSystem is a light emulation of a file system and therefore has a couple of unique properties. The first is that filenames are assumed to be unique. The second, a consequence of the first, is that files are versioned. To see what this means, let's create a GridFileSystem instance:

Saving files

```
  @db = Mongo::Connection.new.db("social_site")

  @fs = GridFileSystem.new(@db)
```

Now suppose we want to save the file 'me.jpg.' This is easily done using a filesystem-like API:

```ruby
image = File.open("me.jpg")
@fs.open("me.jpg", "w") do |f|
  f.write image
end
```

We can then retrieve the file by filename:

```ruby
image = @fs.open("me.jpg", "r") {|f| f.read }
```

No problems there. But what if we need to replace the file? That too is straightforward:

```ruby
image = File.open("me-dancing.jpg")
@fs.open("me.jpg", "w") do |f|
  f.write image
end
```

But a couple things need to be kept in mind. First is that the original 'me.jpg' will be available until the new 'me.jpg' saves. From then on, calls to the #open method will always return the most recently saved version of a file. But, and this the second point, old versions of the file won't be deleted. So if you're going to be rewriting files often, you could end up with a lot of old versions piling up. One solution to this is to use the :delete_old options when writing a file:

```ruby
image = File.open("me-dancing.jpg")
@fs.open("me.jpg", "w", :delete_old => true) do |f|
  f.write image
end
```

This will delete all but the latest version of the file.

Deleting files

When you delete a file by name, you delete all versions of that file:

```ruby
@fs.delete("me.jpg")
```

Metadata and safe mode

All of the options for storing metadata and saving in safe mode are available for the GridFileSystem class:

```ruby
image = File.open("me.jpg")
@fs.open('my-avatar.jpg', w,
           :content_type => "application/jpg",
           :metadata     => {'description' => "taken on 3/1/2010 after a game of ultimate"},
           :_id          => 'a-unique-id-to-use-instead-of-the-automatically-generated-one',
           :safe         => true) { |f| f.write image }
```

Advanced Users

Astute code readers will notice that the Grid and GridFileSystem classes are merely thin wrappers around an underlying GridIO class. This means that it's easy to customize the GridFS implementation presented here; just use GridIO for all the low-level work, and build the API you need in an external manager class similar to Grid or GridFileSystem.

Rails - Getting Started

Using Rails 3? See Rails 3 - Getting Started

This tutorial describes how to set up a simple Rails application with MongoDB, using MongoMapper as an object mapper. We assume you're using Rails versions prior to 3.0.

Using a Rails Template
All of the configuration steps listed below, and more, are encapsulated in this Rails template (raw version), based on a similar one by Ben Scofield. You can create your project with the template as follows:

```
rails project_name -m "http://gist.github.com/219223.txt"
```

Be sure to replace **project_name** with the name of your project.

If you want to set up your project manually, read on.

Configuration

1. We need to tell MongoMapper which database we'll be using. Save the following to **config/initializers/database.rb**:

```
MongoMapper.database = "db_name-#{Rails.env}"
```

Replace **db_name** with whatever name you want to give the database. The **Rails.env** variable will ensure that a different database is used for each environment.

2. If you're using Passenger, add this code to **config/initializers/database.rb**.

```
if defined?(PhusionPassenger)
   PhusionPassenger.on_event(:starting_worker_process) do |forked|
     MongoMapper.connection.connect_to_master if forked
   end
end
```

3. Clean out **config/database.yml**. This file should be blank, as we're not connecting to the database in the traditional way.

4. Remove ActiveRecord from environment.rb.

```
config.frameworks -= [:active_record]
```

5. Add MongoMapper to the environment. This can be done by opening **config/environment.rb** and adding the line:

```
config.gem 'mongo_mapper'
```

Once you've done this, you can install the gem in the project by running:

```
rake gems:install
rake gems:unpack
```

Testing

It's important to keep in mind that with MongoDB, we cannot wrap test cases in transactions. One possible work-around is to invoke a **teardown** method after each test case to clear out the database.

To automate this, I've found it effective to modify **ActiveSupport::TestCase** with the code below.

```
# Drop all columns after each test case.
def teardown
  MongoMapper.database.collections.each do |coll|
    coll.remove
  end
end

# Make sure that each test case has a teardown
# method to clear the db after each test.
def inherited(base)
  base.define_method teardown do
    super
  end
end
```

This way, all test classes will automatically invoke the teardown method. In the example above, the teardown method clears each collection. We might also choose to drop each collection or drop the database as a whole, but this would be considerably more expensive and is only necessary if our tests manipulate indexes.

Usually, this code is added in **test/test_helper.rb**. See the aforementioned rails template for specifics.

Coding

If you've followed the foregoing steps (or if you've created your Rails with the provided template), then you're ready to start coding. For help on that, you can read about modeling your domain in Rails.

Rails 3 - Getting Started

It's not difficult to use MongoDB with Rails 3. Most of it comes down to making sure that you're not loading ActiveRecord and understanding how to use Bundler, the new Ruby dependency manager.

Install the Rails 3

If you haven't done so already, install Rails 3.

```
# Use sudo if your setup requires it
gem install rails
```

Configure your application

The important thing here is to avoid loading ActiveRecord. One way to do this is with the `--skip-active-record` switch. So you'd create your app skeleton like so:

```
rails new my_app --skip-active-record
```

Alternatively, if you've already created your app (or just want to know what this actually does), have a look at `config/application.rb` and change the first lines from this:

```
require "rails/all"
```

to this:

```
require "action_controller/railtie"
require "action_mailer/railtie"
require "active_resource/railtie"
require "rails/test_unit/railtie"
```

It's also important to make sure that the reference to active_record in the generator block is commented out:

```
# Configure generators values. Many other options are available, be sure to check the documentation.
# config.generators do |g|
#   g.orm             :active_record
#   g.template_engine :erb
#   g.test_framework  :test_unit, :fixture => true
# end
```

As of this this writing, it's commented out by default, so you probably won't have to change anything here.

Bundle and Initialize

The final step involves bundling any gems you'll need and then creating an initializer for connecting to the database.

Bundling

Edit `Gemfile`, located in the Rails root directory. By default, our `Gemfile` will only load Rails:

```
gem "rails", "3.0.0"
```

Normally, using MongoDB will simply mean adding whichever OM framework you want to work with, as these will require the "mongo" gem by default.

```
# Edit this Gemfile to bundle your application's dependencies.

source 'http://gemcutter.org'

gem "rails", "3.0.0"
gem "mongo_mapper"
```

However, there's currently an issue with loading bson_ext, as the current gemspec isn't compatible with the way Bundler works. We'll be fixing that soon; just pay attention to this issue.

In the meantime, you can use the following work-around:

```
# Edit this Gemfile to bundle your application's dependencies.

require 'rubygems'
require 'mongo'
source 'http://gemcutter.org'

gem "rails", "3.0.0"
gem "mongo_mapper"
```

Requiring `rubygems` and `mongo` before running the `gem` command will ensure that `bson_ext` is loaded. If you'd rather not load `rubygems`, just make sure that both `mongo` and `bson_ext` are in your load path when you require `mongo`.

Once you've configured your `Gemfile`, run the bundle installer:

```
bundle install
```

Initializing

Last item is to create an initializer to connect to MongoDB. Create a Ruby file in `config/initializers`. You can give it any name you want; here we'll call it `config/initializers/mongo.rb`:

```
MongoMapper.connection = Mongo::Connection.new('localhost', 27017)
MongoMapper.database = "#myapp-#{Rails.env}"

if defined?(PhusionPassenger)
   PhusionPassenger.on_event(:starting_worker_process) do |forked|
     MongoMapper.connection.connect if forked
   end
end
```

Running Tests

A slight modification is required to get `rake test` working (thanks to John P. Wood). Create a file `lib/tasks/mongo.rake` containing the following:

```
namespace :db do
  namespace :test do
    task :prepare do
      # Stub out for MongoDB
    end
  end
end
```

Now the various `rake test` tasks will run properly. See John's post for more details.

Conclusion

That should be all. You can now start creating models based on whichever OM you've installed.

MongoDB Data Modeling and Rails

This tutorial discusses the development of a web application on Rails and MongoDB. MongoMapper will serve as our object mapper. The goal is to provide some insight into the design choices required for building on MongoDB. To that end, we'll be constructing a simple but non-trivial social news application. The source code for **newsmonger** is available on github for those wishing to dive right in.

Assuming you've configured your application to work with MongoMapper, let's start thinking about the data model.

Modeling Stories

A news application relies on stories at its core, so we'll start with a Story model:

```
class Story
  include MongoMapper::Document

  key :title,     String
  key :url,       String
  key :slug,      String
  key :voters,    Array
  key :votes,     Integer, :default => 0
  key :relevance, Integer, :default => 0

  # Cached values.
  key :comment_count, Integer, :default => 0
  key :username,      String

  # Note this: ids are of class ObjectId.
  key :user_id,   ObjectId
  timestamps!

  # Relationships.
  belongs_to :user

  # Validations.
  validates_presence_of :title, :url, :user_id
end
```

Obviously, a story needs a title, url, and user_id, and should belong to a user. These are self-explanatory.

Caching to Avoid N+1

When we display our list of stories, we'll need to show the name of the user who posted the story. If we were using a relational database, we could perform a join on users and stores, and get all our objects in a single query. But MongoDB does not support joins and so, at times, requires bit of denormalization. Here, this means caching the 'username' attribute.

A Note on Denormalization

Relational purists may be feeling uneasy already, as if we were violating some universal law. But let's bear in mind that MongoDB collections are not equivalent to relational tables; each serves a unique design objective. A normalized table provides an atomic, isolated chunk of data. A document, however, more closely represents an object as a whole. In the case of a social news site, it can be argued that a username is intrinsic to the story being posted.

What about updates to the username? It's true that such updates will be expensive; happily, in this case, they'll be rare. The read savings achieved in denormalizing will surely outweigh the costs of the occasional update. Alas, this is not hard and fast rule: ultimately, developers must evaluate their applications for the appropriate level of normalization.

Fields as arrays

With a relational database, even trivial relationships are blown out into multiple tables. Consider the votes a story receives. We need a way of recording which users have voted on which stories. The standard way of handling this would involve creating a table, 'votes', with each row referencing user_id and story_id.

With a document database, it makes more sense to store those votes as an array of user ids, as we do here with the 'voters' key.

For fast lookups, we can create an index on this field. In the MongoDB shell:

```
db.stories.ensureIndex('voters');
```

Or, using MongoMapper, we can specify the index in **config/initializers/database.rb**:

```
Story.ensure_index(:voters)
```

To find all the stories voted on by a given user:

```
Story.all(:conditions => {:voters => @user.id})
```

Atomic Updates

Storing the `voters` array in the `Story` class also allows us to take advantage of atomic updates. What this means here is that, when a user votes on a story, we can

1. ensure that the voter hasn't voted yet, and, if not,
2. increment the number of votes and
3. add the new voter to the array.

MongoDB's query and update features allows us to perform all three actions in a single operation. Here's what that would look like from the shell:

```
// Assume that story_id and user_id represent real story and user ids.
db.stories.update({_id: story_id, voters: {'$ne': user_id}},
  {'$inc': {votes: 1}, '$push': {voters: user_id}});
```

What this says is "get me a story with the given id whose `voters` array does not contain the given user id and, if you find such a story, perform two atomic updates: first, increment `votes` by 1 and then push the user id onto the `voters` array."

This operation highly efficient; it's also reliable. The one caveat is that, because update operations are "fire and forget," you won't get a response from the server. But in most cases, this should be a non-issue.

A MongoMapper implementation of the same update would look like this:

```
  def self.upvote(story_id, user_id)
    collection.update({'_id' => story_id, 'voters' => {'$ne' => user_id}},
      {'$inc' => {'votes' => 1}, '$push' => {'voters' => user_id}})
  end
```

Modeling Comments

In a relational database, comments are usually given their own table, related by foreign key to some parent table. This approach is occasionally necessary in MongoDB; however, it's always best to try to embed first, as this will achieve greater query efficiency.

Linear, Embedded Comments

Linear, non-threaded comments should be embedded. Here are the most basic MongoMapper classes to implement such a structure:

```
class Story
  include MongoMapper::Document
  many :comments
end
```

```
class Comment
  include MongoMapper::EmbeddedDocument
  key :body, String

  belongs_to :story
end
```

If we were using the Ruby driver alone, we could save our structure like so:

```
@stories  = @db.collection('stories')
  @document = {:title => "MongoDB on Rails",
               :comments => [{:body     => "Revelatory! Loved it!",
                              :username => "Matz"
                             }
                            ]
              }
  @stories.save(@document)
```

Essentially, comments are represented as an array of objects within a story document. This simple structure should be used for any one-to-many relationship where the many items are linear.

Nested, Embedded Comments

But what if we're building threaded comments? An admittedly more complicated problem, two solutions will be presented here. The first is to represent the tree structure in the nesting of the comments themselves. This might be achieved using the Ruby driver as follows:

```
@stories  = @db.collection('stories')
  @document = {:title => "MongoDB on Rails",
               :comments => [{:body     => "Revelatory! Loved it!",
                              :username => "Matz",
                              :comments => [{:body     => "Agreed.",
                                             :username => "rubydev29"
                                            }
                                           ]
                             }
                            ]
              }
  @stories.save(@document)
```

Representing this structure using MongoMapper would be tricky, requiring a number of custom mods.

But this structure has a number of benefits. The nesting is captured in the document itself (this is, in fact, how Business Insider represents comments). And this schema is highly performant, since we can get the story, and all of its comments, in a single query, with no application-side processing for constructing the tree.

One drawback is that alternative views of the comment tree require some significant reorganizing.

Comment collections

We can also represent comments as their own collection. Relative to the other options, this incurs a small performance penalty while granting us the greatest flexibility. The tree structure can be represented by storing the unique path for each leaf (see Mathias's original post on the idea). Here are the relevant sections of this model:

```
class Comment
  include MongoMapper::Document

  key :body,       String
  key :depth,      Integer, :default => 0
  key :path,       String,  :default => ""

  # Note: we're intentionally storing parent_id as a string
  key :parent_id,  String
  key :story_id,   ObjectId
  timestamps!

  # Relationships.
  belongs_to :story

  # Callbacks.
  after_create :set_path

  private

  # Store the comment's path.
  def set_path
    unless self.parent_id.blank?
      parent        = Comment.find(self.parent_id)
      self.story_id = parent.story_id
      self.depth    = parent.depth + 1
      self.path     = parent.path + ":" + parent.id
    end
    save
  end
```

The path ends up being a string of object ids. This makes it easier to display our comments nested, with each level in order of karma or votes. If we specify an index on story_id, path, and votes, the database can handle half the work of getting our comments in nested, sorted order.

The rest of the work can be accomplished with a couple grouping methods, which can be found in the newsmonger source code.

It goes without saying that modeling comments in their own collection also facilitates various site-wide aggregations, including displaying the latest, grouping by user, etc.

Unfinished business

Document-oriented data modeling is still young. The fact is, many more applications will need to be built on the document model before we can say anything definitive about best practices. So the foregoing should be taken as suggestions, only. As you discover new patterns, we encourage you to document them, and feel free to let us know about what works (and what doesn't).

Developers working on object mappers and the like are encouraged to implement the best document patterns in their code, and to be wary of recreating relational database models in their apps.

Ruby External Resources

There are a number of good resources appearing all over the web for learning about MongoDB and Ruby. A useful selection is listed below. If you know of others, do let us know.

- Screencasts
- Presentations
- Articles
- Projects
- Libraries

Screencasts

Introduction to MongoDB - Part I
An introduction to MongoDB via the MongoDB shell.

Introduction to MongoDB - Part II
In this screencast, Joon You teaches how to use the Ruby driver to build a simple Sinatra app.

Introduction to MongoDB - Part III
For the final screencast in the series, Joon You introduces MongoMapper and Rails.

RailsCasts: MongoDB & MongoMapper
Ryan Bates' RailsCast introducing MongoDB and MongoMapper.

RailsCasts: Mongoid
Ryan Bates' RailsCast introducing Mongoid.

Presentations

Introduction to MongoDB (Video)
Mike Dirolf's introduction to MongoDB at Pivotal Labs, SF.

MongoDB: A Ruby Document Store that doesn't rhyme with 'Ouch' (Slides)
Wynn Netherland's introduction to MongoDB with some comparisons to CouchDB.

MongoDB (is) for Rubyists (Slides)
Kyle Banker's presentation on why MongoDB is for Rubyists (and all human-oriented programmers).

Articles

Why I Think Mongo is to Databases What Rails was to Frameworks
What if a key-value store mated with a relational database system?
Mongo Tips
John Nunemaker's articles on MongoDB and his Mongo Tips blog.

A series of articles on aggregation with MongoDB and Ruby:
1. Part I: Introduction of Aggregation in MongoDB
2. Part II: MongoDB Grouping Elaborated
3. Part III: Introduction to Map-Reduce in MongoDB

Does the MongoDB Driver Support Feature X?
An explanation of how the MongoDB drivers usually automatically support new database features.

Projects

Simple Pub/Sub

A very simple pub/sub system.

Mongo Queue
An extensible thread safe job/message queueing system that uses mongodb as the persistent storage engine.

Resque-mongo
A port of the Github's Resque to MongoDB.

Mongo Admin
A Rails plugin for browsing and managing MongoDB data. See the live demo.

Sinatra Resource
Resource Oriented Architecture (REST) for Sinatra and MongoMapper.

Shorty
A URL-shortener written with Sinatra and the MongoDB Ruby driver.

NewsMonger
A simple social news application demonstrating MongoMapper and Rails.

Data Catalog API
From Sunlight Labs, a non-trivial application using MongoMapper and Sinatra.

Watchtower
An example application using Mustache, MongoDB, and Sinatra.

Shapado
A question and answer site similar to Stack Overflow. Live version at shapado.com.

Libraries

ActiveExpando
An extension to ActiveRecord to allow the storage of arbitrary attributes in MongoDB.

ActsAsTree (MongoMapper)
ActsAsTree implementation for MongoMapper.

Machinist adapter (MongoMapper)
Machinist adapter using MongoMapper.

Mongo-Delegate
A delegation library for experimenting with production data without altering it. A quite useful pattern.

Remarkable Matchers (MongoMapper)
Testing / Matchers library using MongoMapper.

OpenIdAuthentication, supporting MongoDB as the datastore
Brandon Keepers' fork of OpenIdAuthentication supporting MongoDB.

MongoTree (MongoRecord)
MongoTree adds parent / child relationships to MongoRecord.

Merb_MongoMapper
a plugin for the Merb framework for supporting MongoMapper models.

Mongolytics (MongoMapper)
A web analytics tool.

Rack-GridFS
A Rack middleware component that creates HTTP endpoints for files stored in GridFS.

Frequently Asked Questions - Ruby

> **Redirection Notice**
> This page should redirect to http://api.mongodb.org/ruby/1.1.5/file.FAQ.html.

This is a list of frequently asked questions about using Ruby with MongoDB. If you have a question you'd like to have answered here, please add it in the comments.

- Can I run [insert command name here] from the Ruby driver?
- Does the Ruby driver support an EXPLAIN command?

- I see that BSON supports a symbol type. Does this mean that I can store Ruby symbols in MongoDB?
- Why can't I access random elements within a cursor?
- Why can't I save an instance of TimeWithZone?
- I keep getting CURSOR_NOT_FOUND exceptions. What's happening?
- I periodically see connection failures between the driver and MongoDB. Why can't the driver retry the operation automatically?

Can I run [insert command name here] from the Ruby driver?

Yes. You can run any of the available database commands from the driver using the DB#command method. The only trick is to use an OrderedHash when specifying the command. For example, here's how you'd run an asynchronous fsync from the driver:

```
# This command is run on the admin database.
@db = Mongo::Connection.new.db('admin')

# Build the command.
cmd = OrderedHash.new
cmd['fsync'] = 1
cmd['async'] = true

# Run it.
@db.command(cmd)
```

It's important to keep in mind that some commands, like `fsync`, must be run on the `admin` database, while other commands can be run on any database. If you're having trouble, check the command reference to make sure you're using the command correctly.

Does the Ruby driver support an EXPLAIN command?

Yes. `explain` is, technically speaking, an option sent to a query that tells MongoDB to return an explain plan rather than the query's results. You can use `explain` by constructing a query and calling explain at the end:

```
@collection = @db['users']
result = @collection.find({:name => "jones"}).explain
```

The resulting explain plan might look something like this:

```
{"cursor"=>"BtreeCursor name_1",
 "startKey"=>{"name"=>"Jones"},
 "endKey"=>{"name"=>"Jones"},
 "nscanned"=>1.0,
 "n"=>1,
 "millis"=>0,
 "oldPlan"=>{"cursor"=>"BtreeCursor name_1",
             "startKey"=>{"name"=>"Jones"},
             "endKey"=>{"name"=>"Jones"}
 },
 "allPlans"=>[{"cursor"=>"BtreeCursor name_1",
               "startKey"=>{"name"=>"Jones"},
               "endKey"=>{"name"=>"Jones"}}]
 }
```

Because this collection has an index on the "name" field, the query uses that index, only having to scan a single record. "n" is the number of records the query will return. "millis" is the time the query takes, in milliseconds. "oldPlan" indicates that the query optimizer has already seen this kind of query and has, therefore, saved an efficient query plan. "allPlans" shows all the plans considered for this query.

I see that BSON supports a symbol type. Does this mean that I can store Ruby symbols in MongoDB?

You can store Ruby symbols in MongoDB, but only as values. BSON specifies that document keys must be strings. So, for instance, you can do this:

```
@collection = @db['test']

boat_id = @collection.save({:vehicle  => :boat})
car_id  = @collection.save({"vehicle" => "car"})

@collection.find_one('_id' => boat_id)
{"_id" => ObjectID('4bb372a8238d3b5c8c000001'), "vehicle" => :boat}


@collection.find_one('_id' => car_id)
{"_id" => ObjectID('4bb372a8238d3b5c8c000002'), "vehicle" => "car"}
```

Notice that the symbol values are returned as expected, but that symbol keys are treated as strings.

Why can't I access random elements within a cursor?

MongoDB cursors are designed for sequentially iterating over a result set, and all the drivers, including the Ruby driver, stick closely to this directive. Internally, a Ruby cursor fetches results in batches by running a MongoDB `getmore` operation. The results are buffered for efficient iteration on the client-side.

What this means is that a cursor is nothing more than a device for returning a result set on a query that's been initiated on the server. Cursors are not containers for result sets. If we allow a cursor to be randomly accessed, then we run into issues regarding the freshness of the data. For instance, if I iterate over a cursor and then want to retrieve the cursor's first element, should a stored copy be returned, or should the cursor re-run the query? If we returned a stored copy, it may not be fresh. And if the the query is re-run, then we're technically dealing with a new cursor.

To avoid those issues, we're saying that anyone who needs flexible access to the results of a query should store those results in an array and then access the data as needed.

Why can't I save an instance of TimeWithZone?

MongoDB stores times in UTC as the number of milliseconds since the epoch. This means that the Ruby driver serializes Ruby Time objects only. While it would certainly be possible to serialize a TimeWithZone, this isn't preferable since the driver would still deserialize to a Time object.

All that said, if necessary, it'd be easy to write a thin wrapper over the driver that would store an extra time zone attribute and handle the serialization/deserialization of TimeWithZone transparently.

I keep getting CURSOR_NOT_FOUND exceptions. What's happening?

The most likely culprit here is that the cursor is timing out on the server. Whenever you issue a query, a cursor is created on the server. Cursor naturally time out after ten minutes, which means that if you happen to be iterating over a cursor for more than ten minutes, you risk a CURSOR_NOT_FOUND exception.

There are two solutions to this problem. You can either:

1. Limit your query. Use some combination of `limit` and `skip` to reduce the total number of query results. This will, obviously, bring down the time it takes to iterate.

2. Turn off the cursor timeout. To do that, invoke `find` with a block, and pass `:timeout => true`:

```
  @collection.find({}, :timeout => false) do |cursor|
    cursor.each do |document
      # Process documents here
    end
  end
```

I periodically see connection failures between the driver and MongoDB. Why can't the driver retry the operation automatically?

A connection failure can indicate any number of failure scenarios. Has the server crashed? Are we experiencing a temporary network partition? Is there a bug in our ssh tunnel?

Without further investigation, it's impossible to know exactly what has caused the connection failure. Furthermore, when we do see a connection failure, it's impossible to know how many operations prior to the failure succeeded. Imagine, for instance, that we're using safe mode and we send an `$inc` operation to the server. It's entirely possible that the server has received the `$inc` but failed on the call to `getLastError`. In that case, retrying the operation would result in a double-increment.

Because of the indeterminacy involved, the MongoDB drivers will not retry operations on connection failure. How connection failures should be handled is entirely dependent on the application. Therefore, we leave it to the application developers to make the best decision in this case.

The drivers will reconnect on the subsequent operation.

Java Language Center

Java Driver

- Java Driver
  - Basics
  - Specific Topics and How-To
- Third Party Frameworks and Libs
  - POJO Mappers
  - Code Generation
  - Misc
- Clojure
- Groovy
- JavaScript (Rhino)
- JRuby
- Scala
- Hadoop
- Presentations

Basics

- Download the Java Driver
- Tutorial
- API Documentation
- Release Notes

Specific Topics and How-To

- Concurrency
- Saving Objects
- Data Types

Third Party Frameworks and Libs

POJO Mappers

- Morphia - Type-Safe Wrapper with DAO/Datastore abstractions
- Mungbean (w/clojure support)
- Spring MongoDB – Provides Spring users with a familiar data access features including rich POJO mapping.
- DataNucleus JPA/JDO – JPA/JDO wrapper
- lib-mongomapper JavaBean Mapper (No annotations)
- mongo-jackson-mapper Uses jackson (annotations) to map to/from POJOs and has a simple wrapper around DBCollection to simply this.
- Kundera - JPA compliant ORM. Works with multiple datastores.
- Jongo - Query in Java as in Mongo shell (using strings), unmarshall results into Java objects (using Jackson)

Code Generation

- Sculptor - mongodb-based DSL -> Java (code generator)
- GuicyData - DSL -> Java generator with Guice integration
  - Blog Entries

Misc

- log4mongo a log4j appender
- mongo-java-logging a Java logging handler
- (Experimental, Type4) JDBC driver
- Metamodel data exploration and querying library
- Mongodb Java REST server based on Jetty

Clojure

- Congo Mongo
- monger

Groovy

- Groovy Tutorial for MongoDB

- MongoDB made more Groovy
- GMongo, a Groovy wrapper to the mongodb Java driver
    - GMongo 0.5 Release Writeup

JavaScript (Rhino)

- MongoDB-Rhino - A toolset to provide full integration between the Rhino JavaScript engine for the JVM and MongoDB. Uses the MongoDB Java driver.

JRuby

- jmongo A thin ruby wrapper around the mongo-java-driver for vastly better jruby performance.

If there is a project missing here, just add a comment or email the list and we'll add it.

Scala

- Scala Language Center

Hadoop

- There is an early stage MongoDB Hadoop integration plugin available, which supports reading data from MongoDB into Hadoop and writing data from Hadoop out to MongoDB.
- Part of the mongo-hadoop distribution includes a MongoDB sink for Flume to allow logging into MongoDB

Presentations

- Building a Mongo DSL in Scala at Hot Potato - Lincoln Hochberg's Presentation from MongoSF (April 2010)
- Java Development - Brendan McAdams' Presentation from MongoNYC (May 2010)
- Java Development - James Williams' Presentation from MongoSF (April 2010)
- Morphia: Easy Java Persistance for MongoDB - Scott Hernandez' presentation at MongoSF (May 2011)
- Spring Source and MongoDB - Chris Richardson's presentation at MongoSV (December 2010)
- Using MongoDB with Scala - Brendan McAdams' Presentation at the New York Scala Enthusiasts (August 2010)
- More Java-related presentations

Java Driver Concurrency

The Java MongoDB driver is thread safe. If you are using in a web serving environment, for example, you should create a single Mongo instance, and you can use it in every request. The Mongo object maintains an internal pool of connections to the database (default pool size of 10). For every request to the DB (find, insert, etc) the java thread will obtain a connection from the pool, execute the operation, and release the connection. This means the connection (socket) used may be different each time.

Additionally in the case of a replica set with slaveOk option turned on, the read operations will be distributed evenly across all slaves. This means that within the same thread, a write followed by a read may be sent to different servers (master then slave). In turns the read operation may not see the data just written since replication is asynchronous. If you want to ensure complete consistency in a "session" (maybe an http request), you would want the driver to use the same socket, which you can achieve by using a "consistent request". Call requestStart() before your operations and requestDone() to release the connection back to the pool:

```
DB db...;
db.requestStart();
db.requestEnsureConnection();

code....

db.requestDone();
```

DB and DBCollection are completely thread safe. In fact, they are cached so you get the same instance no matter what.

WriteConcern option for single write operation

Since by default a connection is given back to the pool after each request, you may wonder how calling getLastError() works after a write. You should actually use a write concern like `WriteConcern.SAFE` instead of calling getLastError() manually. The driver will then call getLastError() before putting the connection back in the pool.

```
DBCollection coll...;
coll.insert(..., WriteConcern.SAFE);

// is equivalent to
DB db...;
DBCollection coll...;
db.requestStart();
coll.insert(...);
DBObject err = db.getLastError();
db.requestDone();
```

Java - Saving Objects Using DBObject

The Java driver provides a DBObject interface to save custom objects to the database.

For example, suppose one had a class called Tweet that they wanted to save:

```
public class Tweet implements DBObject {
    /* ... */
}
```

Then you can say:

```
Tweet myTweet = new Tweet();
myTweet.put("user", userId);
myTweet.put("message", msg);
myTweet.put("date", new Date());

collection.insert(myTweet);
```

When a document is retrieved from the database, it is automatically converted to a DBObject. To convert it to an instance of your class, use DBCollection.setObjectClass():

```
collection.setObjectClass(Tweet.class);

Tweet myTweet = (Tweet)collection.findOne();
```

If for some reason you wanted to change the message you can simply take that tweet and save it back after updating the field.

```
Tweet myTweet = (Tweet)collection.findOne();
myTweet.put("message", newMsg);

collection.save(myTweet);
```

Java Tutorial

Introduction

This page is a brief overview of working with the MongoDB Java Driver.

For more information about the Java API, please refer to the online API Documentation for Java Driver

A Quick Tour

Using the Java driver is very simple. First, be sure to include the driver jar `mongo.jar` in your classpath. The following code snippets come from the `examples/QuickTour.java` example code found in the driver.

Making A Connection

To make a connection to a MongoDB, you need to have at the minimum, the name of a database to connect to. The database doesn't have to exist - if it doesn't, MongoDB will create it for you.

Additionally, you can specify the server address and port when connecting. The following example shows three ways to connect to the database *mydb* on the local machine :

```
import com.mongodb.Mongo;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;

Mongo m = new Mongo();
// or
Mongo m = new Mongo( "localhost" );
// or
Mongo m = new Mongo( "localhost" , 27017 );

DB db = m.getDB( "mydb" );
```

At this point, the db object will be a connection to a MongoDB server for the specified database. With it, you can do further operations.

Note: The Mongo object instance actually represents a pool of connections to the database; you will only need one object of class Mongo even with multiple threads. See the concurrency doc page for more information.

The Mongo class is designed to be thread safe and shared among threads. Typically you create only 1 instance for a given DB cluster and use it across your app. If for some reason you decide to create many mongo intances, note that:

- all resource usage limits (max connections, etc) apply per mongo instance
- to dispose of an instance, make sure you call mongo.close() to clean up resources

Authentication (Optional)

MongoDB can be run in a secure mode where access to databases is controlled through name and password authentication. When run in this mode, any client application must provide a name and password before doing any operations. In the Java driver, you simply do the following with the connected mongo object :

```
boolean auth = db.authenticate(myUserName, myPassword);
```

If the name and password are valid for the database, `auth` will be `true`. Otherwise, it will be `false`. You should look at the MongoDB log for further information if available.

Most users run MongoDB without authentication in a trusted environment.

Getting A List Of Collections

Each database has zero or more collections. You can retrieve a list of them from the db (and print out any that are there) :

```
Set<String> colls = db.getCollectionNames();

for (String s : colls) {
    System.out.println(s);
}
```

and assuming that there are two collections, name and address, in the database, you would see

```
name
address
```

as the output.

Getting A Collection

To get a collection to use, just specify the name of the collection to the getCollection`(String collectionName)` method:

```
DBCollection coll = db.getCollection("testCollection")
```

Once you have this collection object, you can now do things like insert data, query for data, etc

Inserting a Document

Once you have the collection object, you can insert documents into the collection. For example, lets make a little document that in JSON would be represented as

```
{
   "name" : "MongoDB",
   "type" : "database",
   "count" : 1,
   "info" : {
               x : 203,
               y : 102
             }
}
```

Notice that the above has an "inner" document embedded within it. To do this, we can use the BasicDBObject class to create the document (including the inner document), and then just simply insert it into the collection using the `insert()` method.

```
BasicDBObject doc = new BasicDBObject();

        doc.put("name", "MongoDB");
        doc.put("type", "database");
        doc.put("count", 1);

        BasicDBObject info = new BasicDBObject();

        info.put("x", 203);
        info.put("y", 102);

        doc.put("info", info);

        coll.insert(doc);
```

Finding the First Document In A Collection using `findOne()`

To show that the document we inserted in the previous step is there, we can do a simple findOne`()` operation to get the first document in the collection. This method returns a single document (rather than the DBCursor that the find`()` operation returns), and it's useful for things where there only is one document, or you are only interested in the first. You don't have to deal with the cursor.

```
DBObject myDoc = coll.findOne();
System.out.println(myDoc);
```

and you should see

```
{ "_id" : "49902cde5162504500b45c2c" , "name" : "MongoDB" , "type" : "database" , "count" : 1 , "info"
: { "x" : 203 , "y" : 102}}
```

Note the `_id` element has been added automatically by MongoDB to your document. Remember, MongoDB reserves element names that start with "_"/"$" for internal use.

Adding Multiple Documents

In order to do more interesting things with queries, let's add multiple simple documents to the collection. These documents will just be

```
{
   "i" : value
}
```

and we can do this fairly efficiently in a loop

```
for (int i=0; i < 100; i++) {
    coll.insert(new BasicDBObject().append("i", i));
}
```

Notice that we can insert documents of different "shapes" into the same collection. This aspect is what we mean when we say that MongoDB is "schema-free"

Counting Documents in A Collection

Now that we've inserted 101 documents (the 100 we did in the loop, plus the first one), we can check to see if we have them all using the `getCount()` method.

```
System.out.println(coll.getCount());
```

and it should print `101`.

Using a Cursor to Get All the Documents

In order to get all the documents in the collection, we will use the `find()` method. The `find()` method returns a `DBCursor` object which allows us to iterate over the set of documents that matched our query. So to query all of the documents and print them out :

```
DBCursor cur = coll.find();

        while(cur.hasNext()) {
            System.out.println(cur.next());
        }
```

and that should print all 101 documents in the collection.

Getting A Single Document with A Query

We can create a *query* to pass to the `find()` method to get a subset of the documents in our collection. For example, if we wanted to find the document for which the value of the "i" field is 71, we would do the following ;

```
BasicDBObject query = new BasicDBObject();

        query.put("i", 71);

        cur = coll.find(query);

        while(cur.hasNext()) {
            System.out.println(cur.next());
        }
```

and it should just print just one document

```
{ "_id" : "49903677516250c1008d624e" , "i" : 71 }
```

You may commonly see examples and documentation in MongoDB which use $ Operators, such as this:

```
db.things.find({j: {$ne: 3}, k: {$gt: 10} });
```

These are represented as regular String keys in the Java driver, using embedded DBObjects:

```
        BasicDBObject query = new BasicDBObject();

        query.put("j", new BasicDBObject("$ne", 3));
        query.put("k", new BasicDBObject("$gt", 10));

        cur = coll.find(query);

        while(cur.hasNext()) {
            System.out.println(cur.next());
        }
```

Getting A Set of Documents With a Query

We can use the query to get a set of documents from our collection. For example, if we wanted to get all documents where "i" > 50, we could write :

```
query = new BasicDBObject();

        query.put("i", new BasicDBObject("$gt", 50));  // e.g. find all where i > 50

        cur = coll.find(query);

        while(cur.hasNext()) {
            System.out.println(cur.next());
        }
```

which should print the documents where i > 50. We could also get a range, say 20 < i <= 30 :

```
query = new BasicDBObject();

        query.put("i", new BasicDBObject("$gt", 20).append("$lte", 30));  // i.e.   20 < i <= 30

        cur = coll.find(query);

        while(cur.hasNext()) {
            System.out.println(cur.next());
        }
```

Creating An Index

MongoDB supports indexes, and they are very easy to add on a collection. To create an index, you just specify the field that should be indexed, and specify if you want the index to be ascending (1) or descending (-1). The following creates an ascending index on the "i" field :

```
coll.createIndex(new BasicDBObject("i", 1));  // create index on "i", ascending
```

Getting a List of Indexes on a Collection

You can get a list of the indexes on a collection :

```
List<DBObject> list = coll.getIndexInfo();

        for (DBObject o : list) {
            System.out.println(o);
        }
```

and you should see something like

```
{ "name" : "i_1" , "ns" : "mydb.testCollection" , "key" : { "i" : 1} }
```

Quick Tour of the Administrative Functions

Getting A List of Databases

You can get a list of the available databases:

```
Mongo m = new Mongo();

        for (String s : m.getDatabaseNames()) {
            System.out.println(s);
        }
```

Dropping A Database

You can drop a database by name using the `Mongo` object:

```
m.dropDatabase("my_new_db");
```

Java Types

- Object Ids
- Regular Expressions
- Dates/Times
- Database References
- Binary Data
- Timestamp Data
- Code Data
- Embedded Documents
- Arrays

Object Ids

`com.mongodb.ObjectId` is used to autogenerate unique ids.

```
ObjectId id = new ObjectId();
ObjectId copy = new ObjectId(id);
```

Regular Expressions

The Java driver uses `java.util.regex.Pattern` for regular expressions.

```
Pattern john = Pattern.compile("joh?n", CASE_INSENSITIVE);
BasicDBObject query = new BasicDBObject("name", john);

// finds all people with "name" matching /joh?n/i
DBCursor cursor = collection.find(query);
```

Dates/Times

The java.util.Date class is used for dates.

```
Date now = new Date();
BasicDBObject time = new BasicDBObject("ts", now);

collection.save(time);
```

Database References

`com.mongodb.DBRef` can be used to save database references.

```
DBRef addressRef = new DBRef(db, "foo.bar", address_id);
DBObject address = addressRef.fetch();

DBObject person = BasicDBObjectBuilder.start()
    .add("name", "Fred")
    .add("address", addressRef)
    .get();
collection.save(person);

DBObject fred = collection.findOne();
DBRef addressObj = (DBRef)fred.get("address");
addressObj.fetch()
```

Binary Data

An array of bytes (`byte[]`) as a value will automatically be wrapped as a Binary type.
Additionally the Binary class can be used to represent binary objects, which allows to pick a custom type byte.

Timestamp Data

A timestamp is a special object used by Mongo as an ID based on time, represented by a (time in second, incremental id) pair (it is used notably in the replication oplog).
A timestamp is represented by the BSONTimestamp class.

Code Data

A code object is used to represent JavaScript code, for example when saving executable functions into the system.js collection.
The Code and CodeWScope classes are used to represent this data.
Note that some methods (like map/reduce) take Strings but wrap them into Code objects in the driver.

Embedded Documents

Suppose we have a document that, in JavaScript, looks like:

```
{
    "x" : {
        "y" : 3
    }
}
```

The equivalent in Java is:

```
BasicDBObject y = new BasicDBObject("y", 3);
BasicDBObject x = new BasicDBObject("x", y);
```

Arrays

Anything that extends List in Java will be saved as an array.

So, if you are trying to represent the JavaScript:

```
{
    "x" : [
        1,
        2,
        {"foo" : "bar"},
        4
    ]
}
```

you could do:

```
ArrayList x = new ArrayList();
x.add(1);
x.add(2);
x.add(new BasicDBObject("foo", "bar"));
x.add(4);

BasicDBObject doc = new BasicDBObject("x", x);
```

Replica Set Semantics

The MongoDB Java driver handles failover in replicated setups with tunable levels of transparency to the user. By default, a `Mongo` connection object will ignore failures of secondaries, and only reads will throw `MongoExceptions` when the primary node is unreachable.

The level of exception reporting is tunable however, using a specific `WriteConcern`; you can set this on the Mongo/DB/Collection/Method level. Several levels are included as static options:

- `WriteConcern.NONE` : No exceptions thrown.
- `WriteConcern.NORMAL` : Exceptions are only thrown when the primary node is unreachable for a read, or the full replica set is unreachable.
- `WriteConcern.SAFE` : Same as the above, but exceptions thrown when there is a server error on writes or reads. Calls `getLastError()`.
- `WriteConcern.REPLICAS_SAFE` : Tries to write to two separate nodes. Same as the above, but will throw an exception if two writes are not possible.
- `WriteConcern.FSYNC_SAFE` : Same as `WriteConcern.SAFE`, but also waits for write to be written to disk.

Additional errors may appear in the log files, these are for reporting purposes and logged based on the logging settings.

Sample code is provided which illustrates some of these options. To quickly initialize a sample replica set, you can use the `mongo` shell:

```
> var rst = new ReplSetTest({ nodes : 3 })
> rst.startSet() // wait for processes to start
> rst.initiate() // wait for replica set initialization
```

Java client code demonstrating error handling is available :

- https://github.com/mongodb/mongo-snippets/blob/master/java/Test.java

C++ Language Center

A C++ driver is available for communicating with the MongoDB. As the database is written in C++, the driver actually uses some core MongoDB code -- this is the same driver that the database uses itself for replication.

The driver has been compiled successfully on Linux, OS X, Windows, and Solaris.

- Downloading the Driver
- Compiling and Linking
- MongoDB C++ Client Tutorial
- API Documentation
- Using BSON from C++
- SQL to C++ Mapping Chart
- HOWTO
  - Connecting
  - getLastError
  - Tailable Cursors
  - BSON Arrays in C++
- Mongo Database and C++ Driver Source Code (at github). See the client subdirectory for client driver related files.

Additional Notes

- The Building documentation covers compiling the entire database, but some of the notes there may be helpful for compiling client applications too.
- There is also a pure C driver for MongoDB. For true C++ apps we recommend using the C++ driver.

BSON Arrays in C++

```
// examples

using namespace mongo;
using namespace bson;

bo an_obj;

/** transform a BSON array into a vector of BSONElements.
    we match array # positions with their vector position, and ignore
    any fields with non-numeric field names.
*/
vector<be> a = an_obj["x"].Array();

be array = an_obj["x"];
assert( array.isABSONObj() );
assert( array.type() == Array );

// Use BSON_ARRAY macro like BSON macro, but without keys
BSONArray arr = BSON_ARRAY( "hello" << 1 << BSON( "foo" << BSON_ARRAY( "bar" << "baz" << "qux" ) ) );

// BSONArrayBuilder can be used to build arrays without the macro
BSONArrayBuilder b;
b.append(1).append(2).arr();

/** add all elements of the object to the specified vector */
bo myarray = an_obj["x"].Obj();
vector<be> v;
myarray.elems(v);
list<be> L;
myarray.elems(L)


/** add all values of the object to the specified collection.  If type mismatches,
    exception.
        template <class T>
        void Vals(vector<T> &) const;
        template <class T>
        void Vals(list<T> &) const;
*/

/** add all values of the object to the specified collection.  If type mismatches, skip.
        template <class T>
        void vals(vector<T> &) const;
        template <class T>
        void vals(list<T> &) const;
*/
```

C++ BSON Library

- Standalone Usage
- API Docs
- Short Class Names

The MongoDB C++ driver library includes a bson package that implements the BSON specification (see http://www.bsonspec.org/). This library can be used standalone for object serialization and deserialization even when one is not using MongoDB at all.

Include `bson/bson.h` or `db/jsobj.h` in your application (not both). `bson.h` is new and may not work in some situations, was is good for light header-only usage of BSON (see the bsondemo.cpp example).

Key classes:

- mongo::BSONObj (aka bson::bo) a BSON object
- mongo::BSONElement (bson::be) a single element in a bson object. This is a key and a value.
- mongo::BSONObjBuilder (bson::bob) used to make BSON objects

* mongo::BSONObjIterator (bson::bo::iterator) to enumerate BSON objects

Let's now create a BSON "person" object which contains name and age. We might invoke:

```
BSONObjBuilder b;
b.append("name", "Joe");
b.append("age", 33);
BSONObj p = b.obj();
```

Or more concisely:

```
BSONObj p = BSONObjBuilder().append("name", "Joe").append("age", 33).obj();
```

We can also create objects with a stream-oriented syntax:

```
BSONObjBuilder b;
b << "name" << "Joe" << "age" << 33;
BSONObj p = b.obj();
```

The macro BSON lets us be even more compact:

```
BSONObj p = BSON( "name" << "Joe" << "age" << 33 );
```

Use the GENOID helper to add an object id to your object. The server will add an _id automatically if it is not included explicitly.

```
BSONObj p = BSON( GENOID << "name" << "Joe" << "age" << 33 );
// result is: { _id : ..., name : "Joe", age : 33 }
```

GENOID should be at the beginning of the generated object. We can do something similar with the non-stream builder syntax:

```
BSONObj p =
  BSONObjBuilder().genOID().append("name","Joe").append("age",33).obj();
```

Standalone Usage

You can use the C++ BSON library without MongoDB. Most BSON methods under the `bson/` directory are header-only. They require boost, but headers only.

See the bsondemo.cpp example at github.com.

API Docs

* http://api.mongodb.org/cplusplus/

Short Class Names

Add

```
using namespace bson;
```

to your code to use the following shorter more C++ style names for the BSON classes:

```
// from bsonelement.h
namespace bson {
    typedef mongo::BSONElement be;
    typedef mongo::BSONObj bo;
    typedef mongo::BSONObjBuilder bob;
}
```

(Or one could use `bson::bo` fully qualified for example).

Also available is `bo::iterator` as a synonym for `BSONObjIterator`.

C++ Driver Compiling and Linking

The C++ driver is included in the MongoDB server source repository, and can also be downloaded as a separate, "standalone" tarball (see Downloads).   To compile the "standalone" C++ driver, run the scons command in the top-level directory of the driver sources, e.g.:

```
cd mongo-cxx-driver-nightly/
scons
```

You may wish to compile and link client/simple_client_demo.cpp to verify that everything compiles and links fine.

Linux

Using the prebuilt library

```
$ cd mongo/client
$ g++ simple_client_demo.cpp -lmongoclient -lboost_thread-mt -lboost_filesystem
-lboost_program_options
```

Using mongo_client_lib.cpp instead of a library

If you have a compatibility problem with the library, include mongo_client_lib.cpp in your project instead. For example:

```
g++ -I .. simple_client_demo.cpp mongo_client_lib.cpp -lboost_thread-mt -lboost_filesystem
```

Windows

Note: we tend to test MongoDB with Windows using Visual Studio 2010. 2008 works, although you may have to tweak settings in some cases.

Include mongoclient.lib in your application.

To build mongoclient.lib:

```
scons mongoclient
```

Alternatively, include client/mongo_client_lib.cpp in your project.

For Windows, see also:

- bson/bsondemo/bsondemo.vcxproj
- client/examples/simple_client_demo.vcxproj
- Prebuilt Boost Libraries
  - Prebuild Boost Libraries only necessary for versions < 2.1.1
  - Boost 1.49 source is now included directly in version 2.1.1+

- Building with Visual Studio 2010

Other notes for Windows:

- Compile with /MT
- You may wish to define _CRT_SECURE_NO_WARNINGS to avoid warnings on use of strncpy and such by the MongoDB client code.
- Include the WinSock library in your application : Linker.Input.Additional Dependencies - add ws2_32.lib.

Windows Troubleshooting

- error LNK2005: ___ already defined in msvcprt.lib(MSVCP100.dll) libboost_thread-vc100-mt-1_42.lib(thread.obj)
  - The boost library being linked against is expecting a /MT build. If this is a release build, try using /MT instead of /MD for compilation (under Properties.C++.Code Generation).

C++ Driver Download

Driver tarballs

The C++ client library can be found at http://dl.mongodb.org/dl/cxx-driver/.

Note: despite the word 'linux' in the filenames, these files are mostly source code and thus should be applicable to all operating systems.

From the server source code

If you have the full MongoDB source code, the driver code is part of it. This is available on github and also on the MongoDB Downloads page.

The full server source is quite large, so the tarballs above may be easier. Also if using the full server source tree, use "scons mongoclient" to build just the client library.

Next : Compiling and Linking

C++ getLastError

- string mongo::DBClientWithCommands::getLastError();
  - Get error result from the last operation on this connection. Empty string if no error.

- BSONObj DBClientWithCommands::getLastErrorDetailed();
  - Get the full last error object. See the getLastError Command page for details.

See client/simple_client_demo.cpp for an example.

See Also

- getLastError Command

C++ Tutorial

This document is an introduction to usage of the MongoDB database from a C++ program.

First, install Mongo -- see the Quickstart for details.

Next, you may wish to take a look at the Developer's Tour guide for a language independent look at how to use MongoDB. Also, we suggest some basic familiarity with the `mongo` shell -- the shell is one's primary database administration tool and is useful for manually inspecting the contents of a database after your C++ program runs.

Installing the Driver Library and Headers

A good source for general information about setting up a MongoDB development environment on various operating systems is the building page.

The normal database distribution used to include the C++ driver, but there were many problems with library version mismatches so now you have to build from source. You can either get the full source code for the database and just build the C++ driver or download the driver separately and build it.

Unix

For Unix, the Mongo driver library is `libmongoclient.a`. For either build, run `scons --help` to see all options.

Full Database Source Driver Build

To install the libraries, run:

```
scons --full install
```

`--full` tells the install target to include the library and header files; by default library and header files are installed in */usr/local*.

You can use `--prefix` to change the install path: `scons --prefix /opt/mongo --full install`. You can also specify `--sharedclient` to build a shared library instead of a statically linked library.

Driver Build

If you download the driver source code separately, you can build it by running `scons` (no options).

Windows

For more information on Boost setup see the Building for Windows page.

Compiling

The C++ drivers requires the pcre and boost libraries (with headers) to compile. Be sure they are in your include and lib paths. You can usually install them from your OS's package manager if you don't already have them.

Writing Client Code

Note: for brevity, the examples below are simply inline code. In a real application one will define classes for each database object typically.

Connecting

Let's make a tutorial.cpp file that connects to the database (see client/examples/tutorial.cpp for full text of the examples below):

```
#include <iostream>
#include "client/dbclient.h"

using namespace mongo;

void run() {
  DBClientConnection c;
  c.connect("localhost");
}

int main() {
  try {
    run();
    cout << "connected ok" << endl;
  } catch( DBException &e ) {
    cout << "caught " << e.what() << endl;
  }
  return 0;
}
```

If you are using gcc on Linux or OS X, you would compile with something like this, depending on location of your include files and libraries:

```
$ g++ tutorial.cpp -lmongoclient -lboost_thread -lboost_filesystem -lboost_program_options -o tutorial
$ ./tutorial
connected ok
$
```

- Depending on your boost version you might need to link against the *boost_system* library as well: **-lboost_system**.
- You may need to append "-mt" to boost_filesystem and boost_program_options. If using a recent boost, "-mt" is not needed anymore.
- Of course, you may need to use -I and -L to specify the locations of your mongo and boost headers and libraries.

BSON

The Mongo database stores data in BSON format. BSON is a binary object format that is JSON-like in terms of the data which can be stored (some extensions exist, for example, a Date datatype).

To save data in the database we must create objects of class BSONObj. The components of a BSONObj are represented as BSONElement objects. We use BSONObjBuilder to make BSON objects, and BSONObjIterator to enumerate BSON objects.

Let's now create a BSON "person" object which contains name and age. We might invoke:

```
BSONObjBuilder b;
b.append("name", "Joe");
b.append("age", 33);
BSONObj p = b.obj();
```

Or more concisely:

```
BSONObj p = BSONObjBuilder().append("name", "Joe").append("age", 33).obj();
```

We can also create objects with a stream-oriented syntax:

```
BSONObjBuilder b;
b << "name" << "Joe" << "age" << 33;
BSONObj p = b.obj();
```

The macro BSON lets us be even more compact:

```
BSONObj p = BSON( "name" << "Joe" << "age" << 33 );
```

Use the GENOID helper to add an object id to your object. The server will add an _id automatically if it is not included explicitly.

```
BSONObj p = BSON( GENOID << "name" << "Joe" << "age" << 33 );
// result is: { _id : ..., name : "Joe", age : 33 }
```

GENOID should be at the beginning of the generated object. We can do something similar with the non-stream builder syntax:

```
BSONObj p =
  BSONObjBuilder().genOID().append("name","Joe").append("age",33).obj();
```

Other helpers are listed here, see bsonmisc.h for a full list.

Inserting

We now save our person object in a persons collection in the database:

```
c.insert("tutorial.persons", p);
```

The first parameter to insert is the namespace. tutorial is the database and persons is the collection name.

Querying

Let's now fetch all objects from the persons collection, and display them. We'll also show here how to use count().

```
cout << "count:" << c.count("tutorial.persons") << endl;

  auto_ptr<DBClientCursor> cursor =
    c.query("tutorial.persons", BSONObj());
  while( cursor->more() )
      cout << cursor->next().toString() << endl;
```

BSONObj() is an empty BSON object -- it represents {} which indicates an empty query pattern (an empty query is a query for all objects).

We use BSONObj::toString() above to print out information about each object retrieved. BSONObj::toString is a diagnostic function which prints an abbreviated JSON string representation of the object. For full JSON output, use BSONObj::jsonString.

Let's now write a function which prints out the name (only) of all persons in the collection whose age is a given value:

```
void printIfAge(DBClientConnection&c, int age) {
  auto_ptr<DBClientCursor> cursor =
    c.query("tutorial.persons", QUERY( "age" << age ) );
  while( cursor->more() ) {
    BSONObj p = cursor->next();
    cout << p.getStringField("name") << endl;
  }
}
```

getStringField() is a helper that assumes the "name" field is of type string. To manipulate an element in a more generic fashion we can retrieve the particular BSONElement from the enclosing object:

```
BSONElement name = p["name"];
// or:
//BSONElement name = p.getField("name");
```

See the api docs, and jsobj.h, for more information.

Our query above, written as JSON, is of the form

```
{ age : <agevalue> }
```

Queries are BSON objects of a particular format -- in fact, we could have used the BSON() macro above instead of QUERY(). See class Query in dbclient.h for more information on Query objects, and the Sorting section below.

In the mongo shell (which uses javascript), we could invoke:

```
use tutorial;
db.persons.find( { age : 33 } );
```

Indexing

Let's suppose we want to have an index on age so that our queries are fast. We would use:

```
c.ensureIndex("tutorial.persons", fromjson("{age:1}"));
```

The ensureIndex method checks if the index exists; if it does not, it is created. ensureIndex is intelligent and does not repeat transmissions to the server; thus it is safe to call it many times in your code, for example, adjacent to every insert operation.

In the above example we use a new function, fromjson. fromjson converts a JSON string to a BSONObj. This is sometimes a convenient way to specify BSON. Alternatively we could have written:

```
c.ensureIndex("tutorial.persons", BSON( "age" << 1 ));
```

Sorting

Let's now make the results from printIfAge sorted alphabetically by name. To do this, we change the query statement from:

```
auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons", QUERY( "age" << age ) );
```

to

```
to  auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons", QUERY( "age" << age ).sort("name")
);
```

Here we have used Query::sort() to add a modifier to our query expression for sorting.

Updating

Use the update() method to perform a database update . For example the following update in the mongo shell :

```
> use tutorial
> db.persons.update( { name : 'Joe', age : 33 },
...                  { $inc : { visits : 1 } } )
```

is equivalent to the following C++ code:

```
db.update( "tutorial.persons" ,
           BSON( "name" << "Joe" << "age" << 33 ),
           BSON( "$inc" << BSON( "visits" << 1 ) ) );
```

Example

A simple example illustrating usage of BSON arrays and the "$nin" operator is available here .

Further Reading

This overview just touches on the basics of using Mongo from C++. There are many more capabilities. For further exploration:

- See the language-independent Developer's Tour;
- Experiment with the `mongo` shell;
- Review the doxygen API docs;
- See connecting pooling information in the API docs;
- See GridFS file storage information in the API docs;
- See the HOWTO pages under the C++ Language Center
- Consider getting involved to make the product (either C++ driver, tools, or the database itself) better!

List of helper functions

This is a non-exhaustive list of helper functions for use in the C++ stream syntax. An exhaustive list is here: bsonmisc.h

Typical example of stream syntax:

```
BSONObj p = BSON( "name" << "Joe" << "age" << 33 );
```

GENOID -the server will add an _id automatically if it is not included explicitly.

```
BSONObj p = BSON( GENOID << "name" << "Joe" << "age" << 33 );
// result is: { _id : ..., name : "Joe", age : 33 }
```

LT,GT,LTE,GTE, NE -less than, greater than, etc

```
BSONObj p = BSON( "age" << GT << 21 );
// result is:  { age : { $gt : 21 } }
```

DATENOW -translates to current date

```
BSONObj p = BSON( "created" << DATENOW );
// result is: { created : "2009-10-09 11:41:42" }
```

MINKEY,MAXKEY

```
BSONObj p = BSON( "a" << MINKEY);
// result is: { "a" : { "$minKey" : 1 } }
```

OR

```
OR(BSON("x" << GT << 7), BSON("y" << LT << 6))
// result is: {$or: [{x: {$gt: 7}}, {y: {$lt: 6}}]}
```

BSONNULL -translates to null value (will appear in mongoDB 2.1)

```
BSONObj p = BSON( "name" << "Methuselah" << "age" << BSONNULL );
// result is: { name : "Methuselah", age : null }
```

Connecting

The C++ driver includes several classes for managing collections under the parent class DBClientInterface.

DBClientConnection is our normal connection class for a connection to a single MongoDB database server (or shard manager). Other classes exist for connecting to a replica set.

See http://api.mongodb.org/cplusplus/ for details on each of the above classes.

Perl Language Center

Installing

> Start a MongoDB server instance (`mongod`) before installing so that the tests will pass. The `mongod` cannot be running as a slave for the tests to pass.
>
> Some tests may be skipped, depending on the version of the database you are running.

CPAN

```
$ sudo cpan MongoDB
```

The Perl driver is available through CPAN as the package `MongoDB`. It should build cleanly on *NIX and Windows (via Strawberry Perl). It is also available as an ActivePerl module.

Manual (Non-CPAN) Installation

If you would like to try the latest code or are contributing to the Perl driver, it is available at Github. There is also documentation generated after every commit.

You can see if it's a good time to grab the bleeding edge code by seeing if the build is green.

To build the driver, run:

```
$ perl Makefile.PL
$ make
$ make test # make sure mongod is running, first
$ sudo make install
```

Please note that the tests will not pass without a `mongod` process running.

Note that the Perl driver requires some libraries available in CPAN. As of April, 2010, these are Any::Moose, Class::Method::Modifiers, Data::Types, DateTime, File::Slurp, Test::Exception, Try::Tiny, boolean, and Module::Install. (Additionally, Tie::IxHash is required for testing.) On Debian or Ubuntu systems, these prerequisites can be easily installed with the following command:

```
$ sudo apt-get install libmodule-install-perl libany-moose-perl libclass-method-modifiers-perl
libdata-types-perl libdatetime-perl libfile-slurp-perl libtest-exception-perl libtry-tiny-perl
libboolean-perl libtie-ixhash-perl
```

Big-Endian Systems

The driver will work on big-endian machines, but the database will not. The tests assume that `mongod` will be running on localhost unless %ENV{MONGOD} is set. So, to run the tests, start the database on a little-endian machine (at, say, "example.com") and then run the tests with:

```
MONGOD=example.com make test
```

A few tests that require a database server on "localhost" will be skipped.

Next Steps

There is a tutorial and API documentation on CPAN.

If you're interested in contributing to the Perl driver, check out Contributing to the Perl Driver.

MongoDB Perl Tools

`BSON`

`BSON` is a pure-Perl BSON implementation.

`Entities::Backend::MongoDB`

`Entities::Backend::MongoDB` is a backend for the Entities user management and authorization system stores all entities and relations between them in a MongoDB database, using the MongoDB module. This is a powerful, fast backend that gives you all the features of MongoDB.

`MojoX::Session::Store::MongoDB`

`MojoX::Session::Store::MongoDB` is a store for `MojoX::Session` that stores a session in a MongoDB database. Created by Ask Bjørn Hansen.

`MongoDB::Admin`

`MongoDB::Admin` is a collection of MongoDB administrative functions. Created by David Burley.

`Mongoose`

`Mongoose` is an attempt to bring together the full power of Moose with MongoDB. Created by Rodrigo de Oliveira Gonzalez.

`MongoDBI`

MongoDBI is an Object-Document-Mapper (ODM) for MongoDB. It allows you to create Moose-based classes to interact with MongoDB databases.

At-a-glance, most will enjoy MongoDBI for its ability to easily model classes while leveraging the power of MongoDB's schemaless and expeditious document-based design, dynamic queries, and atomic modifier operations.

Also noteworthy is MongoDBI's ease-of-use, chainable search facilities (filters), automated indexing, moose-integration (inheritance support, etc), lean document updates via dirty field tracking, and ability for each class to be configured to use a different database and connection, etc.

`MongoDBx-Class`

`MongoDBx-Class` is an ORM for MongoDB databases. MongoDBx::Class takes advantage of the fact that Perl's MongoDB driver is Moose-based to extend and tweak the driver's behavior, instead of wrapping it. This means MongoDBx::Class does not define its own syntax, so you simply use it exactly as you would the MongoDB driver directly. That said, MongoDBx::Class adds some sugar that enhances and simplifies the syntax unobtrusively (either use it or don't). Thus, it is relatively easy to convert your current MongoDB applications to MongoDBx::Class. A collection in MongoDBx::Class isa('MongoDB::Collection'), a database in MongoDBx::Class isa('MongoDB::Database'), etc. Created by Ido Perlmuter.

`MongoX`

MongoX - DSL sugar for MongoDB

`OOP Perl CMS`

`OOP Perl CMS` is based on Khurt Williams' Object Oriented Perl methodology and can be used as a basic CMS framework or as a basis for your own CMS system. It uses Apache & mod_perl with MongoDB backend. Created by Waitman Gobble.

Contributing to the Perl Driver

The easiest way to contribute is to file bugs and feature requests on Jira.

If you would like to help code the driver, read on...

Finding Something to Help With

Fixing Bugs

You can choose a bug on Jira and fix it. Make a comment that you're working on it, to avoid overlap.

Writing Tests

The driver could use a lot more tests. We would be grateful for any and all tests people would like to write.

Adding Features

If you think a feature is missing from the driver, you're probably right. Check on IRC or the mailing list, then go ahead and create a Jira case and add the feature. The Perl driver was a bit neglected for a while (although it's now getting a lot of TLC) so it's missing a lot of things that the other drivers have. You can look through their APIs for ideas.

Contribution Guildlines

The best way to make changes is to create an account on [Github], fork the driver, make your improvements, and submit a merge request.

To make sure your changes are approved and speed things along:

- Write tests. Lots of tests.
- Document your code.
- Write POD, when applicable.

Bonus (for C programmers, particularly):

- Make sure your change works on Perl 5.8, 5.10, Windows, Mac, Linux, etc.

Code Layout

The important files:

```
| perl_mongo.c # serialization/deserialization
| mongo_link.c # connecting to, sending to, and receiving from the database
- lib
   - MongoDB
      | Connection.pm # connection, queries, inserts... everything comes through here
      | Database.pm
      | Collection.pm
      | Cursor.pm
      | OID.pm
      | GridFS.pm
      - GridFS
         | File.pm
- xs
   | Mongo.xs
   | Connection.xs
   | Cursor.xs
   | OID.xs
```

Perl Tutorial

Redirection Notice
This page should redirect to http://search.cpan.org/dist/MongoDB/lib/MongoDB/Tutorial.pod.

Online API Documentation

MongoDB API and driver documentation is available online. It is updated daily.

- Java Driver API Documentation
- C++ Driver API Documentation
- Python Driver API Documentation
- Ruby Driver API Documentation
- PHP Driver API Documentation

Writing Drivers and Tools

See Also

- Mongo Query Language
- mongosniff
- --objcheck command line parameter

Overview - Writing Drivers and Tools

This section contains information for developers that are working with the low-level protocols of Mongo - people who are writing drivers and higher-level tools.

Documents of particular interest :

BSON http://bsonspec.org	Description of the BSON binary document format. Fundamental to how Mongo and it's client software works.
Mongo Wire Protocol	Specification for the basic socket communications protocol used between Mongo and clients.
Mongo Driver Requirements	Description of what functionality is expected from a Mongo Driver
GridFS Specification	Specification of GridFS - a convention for storing large objects in Mongo
Mongo Extended JSON	Description of the extended JSON protocol for the REST-ful interface (ongoing development)

Additionally we recommend driver authors take a look at existing driver source code as an example.

- Back to driver development home page

- Drivers Home Page

Mongo Driver Requirements

This is a high-level list of features that a driver for MongoDB might provide. We attempt to group those features by priority. This list should be taken with a grain of salt, and probably used more for inspiration than as law that must be adhered to. A great way to learn about implementing a driver is by reading the source code of any of the existing drivers, especially the ones listed as "mongodb.org supported".

High priority

- BSON serialization/deserialization
- full cursor support (e.g. support OP_GET_MORE operation)
- close exhausted cursors via OP_KILL_CURSORS
- support for running database commands
- handle query errors
- convert all strings to UTF-8 (part of proper support for BSON)
- hint, explain, count, $where
- database profiling: set/get profiling level, get profiling info
- advanced connection management (replica sets, slave okay)
- automatic reconnection

Medium priority

- validate a collection in a database
- buffer pooling
- Tailable cursor support

A driver should be able to connect to a single server. By default this must be `localhost:27017`, and must also allow the server to be specified by hostname and port.

```
Mongo m = new Mongo();   // go to localhost, default port
```

```
Mongo m = new Mongo(String host, int port);
```

How the driver does this is up to the driver - make it idiomatic. However, a driver should make it explicit and clear what is going on.

Replica Sets

A driver must be able to support "Replica-Set" configurations, where more than one mongod servers are specified, and configured for hot-failover.

The driver should determine which of the nodes is the current master, and send all operations to that server. In the event of an error, either socket error or a "not a master" error, the driver must restart the determination process.

1. **Cluster Mode** Connect to master in master-slave cluster

   ```
   ServerCluster sc = new ServerCluster(INETAddr...); // again, give one and discover?
   Mongo m = new Mongo(sc);
   ```

   ## Connect to slave in read-only mode in master-slave cluster

   ```
   ServerCluster sc = new ServerCluster(INETAddr...); // again, give one and discover?
   sc.setTarget(...)
   Mongo m = new Mongo(sc);

   or maybe make it like *Default/Simple* w/ a flag?
   ```

Other than that, we need a way to get a DB object :

```
Mongo m = new Mongo();

DB db = m.getDB(name);
```

And a list of db names (useful for tools...) :

```
List<String> getDBNameList();
```

Database Object

Simple operations on a database object :

```
/**
 *  get name of database
 */
String    dbName = db.getName();

/**
 * Get a list of all the collection names in this database
 */
List<String> cols = db.getCollectionNames();

/**
 * get a collection object.  Can optionally create it if it
 *  doesn't exist, or just  be strict.  (XJDM has strictness as an option)
 */
Collection coll = db.getCollection(string);

/**
 * Create a collection w/ optional options.  Can fault
 * if the collection exists, or can just return it if it already does
 */
Collection coll = db.createCollection( string);
Collection coll = db.createCollection( string, options);

/**
 * Drop a collection by its name or by collection object.
 * Driver could invalidate any outstanding Collection objects
 * for that collection, or just hope for the best.
 */
boolean  b  = db.dropCollection(name);
boolean  b  = db.dropCollection(Collection);

/**
 * Execute a command on the database, returning the
 * BSON doc with the results
 */
Document d = db.executeCommand(command);

/**
 * Close the [logical] database
 */
void  db.close();

/**
 * Erase / drop an entire database
 */
bool dropDatabase(dbname)
```

Database Administration

These methods have to do with database metadata: profiling levels and collection validation. Each admin object is associated with a database. These methods could either be built into the Database class or provided in a separate Admin class whose instances are only available from a database instance.

```
/* get an admin object from a database object. */
Admin admin = db.getAdmin();

/**
 * Get profiling level. Returns one of the strings "off", "slowOnly", or
 * "all". Note that the database returns an integer. This method could
 * return an int or an enum instead --- in Ruby, for example, we return
 * symbols.
 */
String profilingLevel = admin.getProfilingLevel();

/**
 * Set profiling level. Takes whatever getProfilingLevel() returns.
 */
admin.setProfilingLevel("off");

/**
 * Retrieves the database's profiling info.
 */
Document profilingInfo = admin.getProfilingInfo();

/**
 * Returns true if collection is valid; raises an exception if not.
 */
boolean admin.validateCollection(collectionName);
```

Collection Basic Ops

```
/**
 *   full query capabilities - limit, skip, returned fields, sort, etc
 */
Cursor        find(...);

void          insert(...) // insert one or more objects into the collection, local variants on args
void          remove(query) // remove objects that match the query
void          update(selector, modifier)  // modify all objects that match selector w/ modifier object
void          updateFirst(selector, object)   // replace first object that match selector w/ specified
object
void          upsert(selector, object)   // replace first object that matches, or insert
long          getCount();
long          getCount(query);
```

Index Operations

```
void          createIndex( index_info)
void          dropIndex(name)
void          dropIndexes()
List<info>    getIndexInformation()
```

Misc Operations

```
document      explain(query)
options       getOptions();
string        getName();
void          close();
```

Cursor Object

```
document      getNextDocument()
iterator      getIterator()  // again, local to language
bool          hasMore()
void          close()
```

Spec, Notes and Suggestions for Mongo Drivers

Assume that the BSON objects returned from the database may be up to 16MB. This size may change over time.

See Also

- Driver Requirements
- BSON
- The main Database Internals page

Feature Checklist for Mongo Drivers

Functionality Checklist

This section lists tasks the driver author might handle.

Essential

- BSON serialization/deserialization
- Basic operations: `query`, `insert`, `update`, `remove`, `ensureIndex`, `findOne`, `limit`, `sort`
- Fetch more data from a cursor when necessary (dbGetMore)
- Sending of KillCursors operation when use of a cursor has completed (ideally for efficiently these are sent in batches)
- Convert all strings to utf8
- Authentication

Recommended

- automatic _id generation
- Database $cmd support and helpers
- Detect `{ $err: ... }` response from a db query and handle appropriately --see Error Handling in Mongo Drivers
- [Automatically connect to proper server, and failover], when connecting to a Replica Set
- `ensureIndex` commands should be cached to prevent excessive communication with the database. (Or, the driver user should be informed that `ensureIndex` is not a lightweight operation for the particular driver.)
- Support detecting max BSON size on connection (e.g., using `buildinfo` or `isMaster` commands) and allowing users to insert docs up to that size.

More Recommended

- lasterror helper functions
- `count()` helper function
- `$where` clause
- `eval()`
- File chunking (GridFS)
- hint fields
- explain helper

More Optional

- `addUser`, `logout` helpers
- Allow client user to specify Option_SlaveOk for a query
- Tailable cursor support
- In/out buffer pooling (if implementing in a garbage collected languages)

More Optional

- [connection pooling]
- Automatic reconnect on connection failure
- DBRef Support:
    - Ability to generate easily
    - Automatic traversal

See Also

- The Driver and Integration Center for information about the latest drivers
- The [top page] for this section
- The main Database Internals page
- The starting point for all Home

Conventions for Mongo Drivers

Interface Conventions

It is desirable to keep driver interfaces consistent when possible. Of course, idioms vary by language, and when they do adaptation is appropriate. However, when the idiom is the same, keeping the interfaces consistent across drivers is desirable.

Terminology

In general, use these terms when naming identifiers. Adapt the names to the normal "punctuation" style of your language `-- foo_bar` in C might be `fooBar` in Java.

- *database* - what does this mean?
- *collection*
- *index*

Driver Testing Tools

Object IDs

- driverOIDTest for testing toString

```
> db.runCommand( { "driverOIDTest" : new ObjectId() } )
{
        "oid" : ObjectId("4b8991f221752a6e61a88267"),
        "str" : "4b8991f221752a6e61a88267",
        "ok" : 1
}
```

Mongo Wire Protocol

Introduction

The Mongo Wire Protocol is a simple socket-based, request-response style protocol. Clients communicate with the database server through a regular TCP/IP socket.

> **Default Socket Port**
> The default port is 27017, but this is configurable and will vary.

Clients should connect to the database with a regular TCP/IP socket. Currently, there is no connection handshake.

To describe the message structure, a C-like `struct` is used. The types used in this document (`cstring`, `int32`, etc.) are the same as those defined in the BSON specification. The standard message header is typed as `MsgHeader`. Integer constants are in capitals (e.g. `ZERO` for the integer value of 0).

In the case where more than one of something is possible (like in a OP_INSERT or OP_KILL_CURSORS), we again use the notation from the BSON specification (e.g. `int64*`). This simply indicates that one or more of the specified type can be written to the socket, one after another.

Byte Ordering
Note that like BSON documents, all data in the mongo wire protocol is little-endian.

Messages Types and Formats

TableOfContents

There are two types of messages, client requests and database responses, each having a slightly different structure.

Standard Message Header

In general, each message consists of a standard message header followed by request-specific data. The standard message header is structured as follows :

```
struct MsgHeader {
    int32   messageLength; // total message size, including this
int32   requestID;      // identifier for this message
int32   responseTo;     // requestID from the original request
//   (used in reponses from db)
int32   opCode;         // request type - see table below
}
```

`messageLength` : This is the total size of the message in bytes. This total includes the 4 bytes that holds the message length.

`requestID` : This is a client or database-generated identifier that uniquely identifies this message. For the case of client-generated messages (e.g. CONTRIB:OP_QUERY and CONTRIB:OP_GET_MORE), it will be returned in the
`responseTo` field of the CONTRIB:OP_REPLY message. Along with the `reponseTo` field in responses, clients can use this to associate query responses with the originating query.

`responseTo` : In the case of a message from the database, this will be the requestID taken from the CONTRIB:OP_QUERY or CONTRIB:OP_GET_MORE messages from the client. Along with the `requestID` field in queries, clients can use this to associate query responses with the originating query.

opCode : Type of message. See the table below in the next section.

Request Opcodes

TableOfContents

The following are the currently supported opcodes :

Opcode Name	opCode value	Comment
OP_REPLY	1	Reply to a client request. responseTo is set
OP_MSG	1000	generic msg command followed by a string
OP_UPDATE	2001	update document
OP_INSERT	2002	insert new document
RESERVED	2003	formerly used for OP_GET_BY_OID
OP_QUERY	2004	query a collection
OP_GETMORE	2005	Get more data from a query. See Cursors
OP_DELETE	2006	Delete documents

OP_KILL_CURSORS	2007	Tell database client is done with a cursor

Client Request Messages

TableOfContents

Clients can send all messages except for CONTRIB:OP_REPLY. This is reserved for use by the database.

Note that only the CONTRIB:OP_QUERY and CONTRIB:OP_GET_MORE messages result in a response from the database. There will be no response sent for any other message.

You can determine if a message was successful with a getLastError command.

OP_UPDATE

The OP_UPDATE message is used to update a document in a collection. The format of a OP_UPDATE message is

```
struct OP_UPDATE {
    MsgHeader header;             // standard message header
int32     ZERO;               // 0 - reserved for future use
cstring   fullCollectionName; // "dbname.collectionname"
int32     flags;              // bit vector. see below
document  selector;           // the query to select the document
document  update;             // specification of the update to perform
}
```

`fullCollectionName` : The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

`flags` :

bit num	name	description
0	Upsert	If set, the database will insert the supplied object into the collection if no matching document is found.
1	MultiUpdate	If set, the database will update all matching objects in the collection. Otherwise only updates first matching doc.
2-31	Reserved	Must be set to 0.

`selector` : BSON document that specifies the query for selection of the document to update.

`update` : BSON document that specifies the update to be performed. For information on specifying updates see the documentation on Updating.

There is no response to an OP_UPDATE message.

OP_INSERT

The OP_INSERT message is used to insert one or more documents into a collection. The format of the OP_INSERT message is

```
struct {
    MsgHeader header;             // standard message header
int32     flags;              // bit vector - see below
cstring   fullCollectionName; // "dbname.collectionname"
document* documents;          // one or more documents to insert into the collection
}
```

`fullCollectionName` : The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

`documents` : One or more documents to insert into the collection. If there are more than one, they are written to the socket in sequence, one after another.

`flags` :

bit num	name	description

0	ContinueOnError	If set, the database will not stop processing a bulk insert if one fails (eg due to duplicate IDs). This makes bulk insert behave similarly to a series of single inserts, except lastError will be set if any insert fails, not just the last one. If multiple errors occur, only the most recent will be reported by getLastError. (new in 1.9.1)
1-31	Reserved	Must be set to 0.

There is no response to an OP_INSERT message.

OP_QUERY

The OP_QUERY message is used to query the database for documents in a collection. The format of the OP_QUERY message is :

```
struct OP_QUERY {
    MsgHeader header;                // standard message header
int32     flags;                  // bit vector of query options.  See below for details.
cstring   fullCollectionName;    // "dbname.collectionname"
int32     numberToSkip;          // number of documents to skip
int32     numberToReturn;        // number of documents to return
//  in the first OP_REPLY batch
document  query;                 // query object.  See below for details.
[ document  returnFieldSelector; ] // Optional. Selector indicating the fields
//  to return.  See below for details.
}
```

`flags` :

bit num	name	description
0	Reserved	Must be set to 0.
1	TailableCursor	Tailable means cursor is not closed when the last data is retrieved. Rather, the cursor marks the final object's position. You can resume using the cursor later, from where it was located, if more data were received. Like any "latent cursor", the cursor may become invalid at some point (CursorNotFound) – for example if the final object it references were deleted.
2	SlaveOk	Allow query of replica slave. Normally these return an error except for namespace "local".
3	OplogReplay	Internal replication use only - driver should not set
4	NoCursorTimeout	The server normally times out idle cursors after an inactivity period (10 minutes) to prevent excess memory use. Set this option to prevent that.
5	AwaitData	Use with TailableCursor. If we are at the end of the data, block for a while rather than returning no data. After a timeout period, we do return as normal.
6	Exhaust	Stream the data down full blast in multiple "more" packages, on the assumption that the client will fully read all data queried. Faster when you are pulling a lot of data and know you want to pull it all down. Note: the client is not allowed to not read all the data unless it closes the connection.
7	Partial	Get partial results from a `mongos` if some shards are down (instead of throwing an error)
8-31	Reserved	Must be set to 0.

`fullCollectionName` : The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

`numberToSkip` : Sets the number of documents to omit - starting from the first document in the resulting dataset - when returning the result of the query.

`numberToReturn` : Limits the number of documents in the **first** CONTRIB:OP_REPLY message to the query. However, the database will still establish a cursor and return the `cursorID` to the client if there are more results than `numberToReturn`. If the client driver offers 'limit' functionality (like the SQL **LIMIT** keyword), then it is up to the client driver to ensure that no more than the specified number of document are returned to the calling application. If `numberToReturn` is 0, the db will use the default return size. If the number is negative, then the database will return that number and close the cursor. No futher results for that query can be fetched. If `numberToReturn` is 1 the server will treat it as -1 (closing the cursor automatically).

`query` : BSON document that represents the query. The query will contain one or more elements, all of which must match for a document to be included in the result set. Possible elements include `$query`, `$orderby`, `$hint`, `$explain`, and `$snapshot`.

`returnFieldsSelector` : OPTIONAL BSON document that limits the fields in the returned documents. The returnFieldsSelector contains one or more elements, each of which is the name of a field that should be returned, and and the integer value 1. In JSON notation, a returnFieldsSelector to limit to the fields "a", "b" and "c" would be :

```
{ a : 1, b : 1, c : 1}
```

The database will respond to an OP_QUERY message with an CONTRIB:OP_REPLY message.

OP_GETMORE

The OP_GETMORE message is used to query the database for documents in a collection. The format of the OP_GETMORE message is :

```
struct {
    MsgHeader header;             // standard message header
int32     ZERO;               // 0 - reserved for future use
cstring   fullCollectionName; // "dbname.collectionname"
int32     numberToReturn;     // number of documents to return
int64     cursorID;           // cursorID from the OP_REPLY
}
```

`fullCollectionName` : The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

`numberToReturn` : Limits the number of documents in the **first** CONTRIB:OP_REPLY message to the query. However, the database will still establish a cursor and return the `cursorID` to the client if there are more results than `numberToReturn`. If the client driver offers 'limit' functionality (like the SQL **LIMIT** keyword), then it is up to the client driver to ensure that no more than the specified number of document are returned to the calling application. If `numberToReturn` is 0, the db will used the default return size.

`cursorID` : Cursor identifier that came in the CONTRIB:OP_REPLY. This must be the value that came from the database.

The database will respond to an OP_GETMORE message with an CONTRIB:OP_REPLY message.

OP_DELETE

The OP_DELETE message is used to remove one or more documents from a collection. The format of the OP_DELETE message is :

```
struct {
    MsgHeader header;             // standard message header
int32     ZERO;               // 0 - reserved for future use
cstring   fullCollectionName; // "dbname.collectionname"
int32     flags;              // bit vector - see below for details.
document  selector;           // query object.  See below for details.
}
```

`fullCollectionName` : The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

`flags` :

bit num	name	description
0	SingleRemove	If set, the database will remove only the first matching document in the collection. Otherwise all matching documents will be removed.
1-31	Reserved	Must be set to 0.

`selector` : BSON document that represent the query used to select the documents to be removed. The selector will contain one or more elements, all of which must match for a document to be removed from the collection. Please see $$$ TODO QUERY for more information.

There is no reponse to an OP_DELETE message.

OP_KILL_CURSORS

The OP_KILL_CURSORS message is used to close an active cursor in the database. This is necessary to ensure that database resources are reclaimed at the end of the query. The format of the OP_KILL_CURSORS message is :

```
struct {
    MsgHeader header;            // standard message header
int32     ZERO;              // 0 - reserved for future use
int32     numberOfCursorIDs; // number of cursorIDs in message
int64*    cursorIDs;         // sequence of cursorIDs to close
}
```

`numberOfCursorIDs` : The number of cursors that are in the message.

`cursorIDs` : "array" of cursor IDs to be closed. If there are more than one, they are written to the socket in sequence, one after another.

Note that if a cursor is read until exhausted (read until OP_QUERY or OP_GETMORE returns zero for the cursor id), there is no need to kill the cursor.

OP_MSG

Deprecated. OP_MSG sends a diagnostic message to the database. The database sends back a fixed resonse. The format is

```
struct {
    MsgHeader header;  // standard message header
cstring   message; // message for the database
}
```

Drivers do not need to implement OP_MSG.

Database Response Messages

TableOfContents

OP_REPLY

The OP_REPLY message is sent by the database in response to an CONTRIB:OP_QUERY or CONTRIB:OP_GET_MORE message. The format of an OP_REPLY message is:

```
struct {
    MsgHeader header;         // standard message header
int32     responseFlags;  // bit vector - see details below
int64     cursorID;       // cursor id if client needs to do get more's
int32     startingFrom;   // where in the cursor this reply is starting
int32     numberReturned; // number of documents in the reply
document* documents;      // documents
}
```

`responseFlags` :

bit num	name	description
0	CursorNotFound	Set when getMore is called but the cursor id is not valid at the server. Returned with zero results.
1	QueryFailure	Set when query failed. Results consist of one document containing an "$err" field describing the failure.
2	ShardConfigStale	Drivers should ignore this. Only mongos will ever see this set, in which case, it needs to update config from the server.
3	AwaitCapable	Set when the server supports the AwaitData Query option. If it doesn't, a client should sleep a little between getMore's of a Tailable cursor. Mongod version 1.6 supports AwaitData and thus always sets AwaitCapable.
4-31	Reserved	Ignore

`cursorID` : The cursorID that this OP_REPLY is a part of. In the event that the result set of the query fits into one OP_REPLY message, `cursorID` will be 0. This `cursorID` must be used in any CONTRIB:OP_GET_MORE messages used to get more data, and also must be closed by the client when no longer needed via a CONTRIB:OP_KILL_CURSORS message.

BSON

bsonspec.org

BSON is a bin-ary-en-coded seri-al-iz-a-tion of JSON-like doc-u-ments. BSON is designed to be lightweight, traversable, and efficient. BSON, like JSON, supports the embedding of objects and arrays within other objects and arrays. See bsonspec.org for the spec and more information in general.

BSON and MongoDB

MongoDB uses BSON as the data storage and network transfer format for "documents".

BSON at first seems BLOB-like, but there exists an important difference: the Mongo database understands BSON internals. This means that MongoDB can "reach inside " BSON objects, even nested ones. Among other things, this allows MongoDB to build indexes and match objects against query expressions on both top-level and nested BSON keys.

See also: the BSON blog post and BSON and Data Interchange

Language-Specific Examples

We often map from a language's "dictionary" type – which may be its native objects – to BSON. The mapping is particularly natural in dynamically typed languages:

```
JavaScript: {"foo" : "bar"}
Perl: {"foo" => "bar"}
PHP: array("foo" => "bar")
Python: {"foo" : "bar"}
Ruby: {"foo" => "bar"}
Java: DBObject obj = new BasicDBObject("foo", "bar");
```

C

```
bson b;
bson_buffer buf;
bson_buffer_init( &buf )
bson_append_string( &buf, "name", "Joe" );
bson_append_int( &buf, "age", 33 );
bson_from_buffer( &b, &buf );
bson_print( &b );
```

See http://github.com/mongodb/mongo-c-driver/blob/master/src/bson.h for more information.

C++

```
BSONObj p = BSON( "name" << "Joe" << "age" << 33 );
cout << p.toString() << endl;
cout << p["age"].number() << endl;
```

See the BSON section of the C++ Tutorial for more information.

Java

```
BasicDBObject doc = new BasicDBObject();
doc.put("name", "MongoDB");
doc.put("type", "database");
doc.put("count", 1);
BasicDBObject info = new BasicDBObject();
info.put("x", 203);
info.put("y", 102);
doc.put("info", info);
coll.insert(doc);
```

PHP

The PHP driver includes `bson_encode` and `bson_decode` functions. `bson_encode` takes any PHP type and serializes it, returning a string of bytes:

```
$bson = bson_encode(null);
$bson = bson_encode(true);
$bson = bson_encode(4);
$bson = bson_encode("hello, world");
$bson = bson_encode(array("foo" => "bar"));
$bson = bson_encode(new MongoDate());
```

Mongo-specific objects (`MongoId`, `MongoDate`, `MongoRegex`, `MongoCode`) will be encoded in their respective BSON formats. For other objects, it will create a BSON representation with the key/value pairs you would get by running `for ($object as $key => $value)`.

`bson_decode` takes a string representing a BSON object and parses it into an associative array.

Python

```
>>> from bson import BSON
>>> bson_string = BSON.encode({"hello": "world"})
>>> bson_string
'\x16\x00\x00\x00\x02hello\x00\x06\x00\x00\x00world\x00\x00'
>>> bson_string.decode()
{u'hello': u'world'}
```

PyMongo also supports "ordered dictionaries" through the **bson.son** module. The **BSON** class can handle **SON** instances using the same methods you would use for regular dictionaries. Python2.7's collections.OrderedDict is also supported.

Ruby

There are now two gems that handle BSON-encoding: bson and bson_ext. These gems can be used to work with BSON independently of the MongoDB Ruby driver.

```
irb
>> require 'rubygems'
=> true
>> require 'bson'
=> true
>> doc = {:hello => "world"}
>> bson = BSON.serialize(doc).to_s
=> "\026\000\000\000\002hello\000\006\000\000\000world\000\000"
>> BSON.deserialize(bson.unpack("C*"))
=> {"hello" => "world"}
```

The BSON class also supports ordered hashes. Simply construct your documents using the OrderedHash class, also found in the MongoDB Ruby Driver.

MongoDB Document Types

MongoDB uses BSON documents for three things:

1. Data storage (user documents). These are the regular JSON-like objects that the database stores for us. These BSON documents are

sent to the database via the INSERT operation. User documents have limitations on the "element name" space due to the usage of special characters in the JSON-like query language.
   a. A user document element name cannot begin with "$".
   b. A user document element name cannot have a "." in the name.
   c. The element name "_id" is reserved for use as a primary key id, but you can store anything that is unique in that field.
   The database expects that drivers will prevent users from creating documents that violate these constraints.
2. Query "Selector" Documents : Query documents (or selectors) are BSON documents that are used in QUERY, DELETE and UPDATE operations. They are used by these operations to match against documents. Selector objects have no limitations on the "element name" space, as they must be able to supply special "marker" elements, like "$where" and the special "command" operations.
3. "Modifier" Documents : Documents that contain 'modifier actions' that modify user documents in the case of an update (see Updating).

Mongo Extended JSON

Mongo's REST interface supports storage and retrieval of JSON documents. Special representations are used for BSON types that do not have obvious JSON mappings, and multiple representations are allowed for some such types. The REST interface supports three different modes for document output { Strict, JS, TenGen }, which serve to control the representations used. Mongo can of course understand all of these representations in REST input.

- **Strict** mode produces output conforming to the JSON spec http://www.json.org.
- **JS** mode uses some Javascript types to represent certain BSON types.
- **TenGen** mode uses some Javascript types and some 10gen specific types to represent certain BSON types.

The following BSON types are represented using special conventions:

Type	Strict	JS	TenGen	Explanati
data_binary	`{ "$binary" : "<bindata>", "$type" : "<t>" }`	`{ "$binary" : "<bindata>", "$type" : "<t>" }`	`{ "$binary" : "<bindata>", "$type" : "<t>" }`	<bindata> the base64 representa of a binary string. <t> the hexadecim representa of a single indicating t data type.
data_date	`{ "$date" : <date> }`	`Date( <date> )`	`Date( <date> )`	<date> is t JSON representa of a 64 bit signed inte for millisec since epoc (unsigned before vers 1.9.1).
data_timestamp	`{ "$timestamp" : {"t": <t>, "i": <i>} }`	`Timestamp( <t>, <i> )`	`Timestamp( <t>, <i> )`	<t> is the JSON representa of a 32 bit unsigned integer for millisecond since epoc <i> is a 32 unsigned integer for increment.

data_regex	`{ "$regex" : "<sRegex>", "$options" : "<sOptions>" }`	`/<jRegex>/<jOptions>`	`/<jRegex>/<jOptions>`	<sRegex> string of va JSON characters <jRegex> i string that contain val JSON characters unescapec characters may not contain unescapec characters <sOptions: a string containing letters of th alphabet. <jOptions> string that contain on the charac 'g', 'i', 'm' a 's' (added i v1.9). Beca the JS and TenGen representa support a limited ran options, ar nonconforr options wil dropped w converting this representa
data_oid	`{ "$oid" : "<id>" }`	`{ "$oid" : "<id>" }`	`ObjectId( "<id>" )`	<id> is a 2 character hexadecim string. Not that these representa require a data_oid v to have an associated field name "_id".
data_ref	`{ "$ref" : "<name>", "$id" : "<id>" }`	`{ "$ref" : "<name>" , "$id" : "<id>" }`	`Dbref( "<name>", "<id>" )`	<name> is string of va JSON characters <id> is a 2 character hexadecim string.

GridFS Specification

Introduction

GridFS is a storage specification for large objects in MongoDB. It works by splitting large object into small chunks, usually 256k in size. Each chunk is stored as a separate document in a `chunks` collection. Metadata about the file, including the filename, content type, and any optional information needed by the developer, is stored as a document in a `files` collection.

So for any given file stored using GridFS, there will exist one document in `files` collection and one or more documents in the `chunks` collection.

If you're just interested in using GridFS, see the docs on storing files. If you'd like to understand the GridFS implementation, read on.

Specification

Storage Collections

GridFS uses two collections to store data:

- `files` contains the object metadata
- `chunks` contains the binary chunks with some additional accounting information

In order to make more than one GridFS namespace possible for a single database, the files and chunks collections are named with a prefix. By default the prefix is `fs.`, so any default GridFS store will consist of collections named `fs.files` and `fs.chunks`. The drivers make it possible to change this prefix, so you might, for instance, have another GridFS namespace specifically for photos where the collections would be `photos.files` and `photos.chunks`.

Here's an example of the standard GridFS interface in Java:

```
/*
 * default root collection usage - must be supported
 */
GridFS myFS = new GridFS(myDatabase);              // returns a default GridFS (e.g. "fs" root
collection)
myFS.storeFile(new File("/tmp/largething.mpg"));   // saves the file into the "fs" GridFS store

/*
 * specified root collection usage - optional
 */

GridFS myContracts = new GridFS(myDatabase, "contracts");             // returns a GridFS where
"contracts" is root
myFS.retrieveFile("smithco", new File("/tmp/smithco_20090105.pdf"));  // retrieves object whose
filename is "smithco"
```

Note that the above API is for demonstration purposes only - this spec does not (at this time) recommend any API. See individual driver documentation for API specifics.

`files`

Documents in the `files` collection require the following fields:

```

{
  "_id" : <unspecified>,                  // unique ID for this file
"length" : data_number,                 // size of the file in bytes
"chunkSize" : data_number,              // size of each of the chunks.  Default is 256k
"uploadDate" : data_date,               // date when object first stored
"md5" : data_string                     // result of running the "filemd5" command on this file's
chunks
}
```

Any other desired fields may be added to the files document; common ones include the following:

```
{
  "filename" : data_string,               // human name for the file
"contentType" : data_string,            // valid mime type for the object
"aliases" : data_array of data_string,  // optional array of alias strings
"metadata" : data_object,               // anything the user wants to store
}
```

Note that the _id field can be of any type, per the discretion of the spec implementor.

`chunks`

The structure of documents from the `chunks` collection is as follows:

```
{
  "_id" : <unspecified>,         // object id of the chunk in the _chunks collection
"files_id" : <unspecified>,    // _id of the corresponding files collection entry
"n" : chunk_number,            // chunks are numbered in order, starting with 0
"data" : data_binary,          // the chunk's payload as a BSON binary type
}
```

Notes:

- The `_id` is whatever type you choose. As with any MongoDB document, the default will be a BSON object id.
- The `files_id` is a foreign key containing the `_id` field for the relevant `files` collection entry

Indexes

GridFS implementations should create a unique, compound index in the `chunks` collection for `files_id` and n. Here's how you'd do that from the shell:

```
db.fs.chunks.ensureIndex({files_id:1, n:1}, {unique: true});
```

This way, a chunk can be retrieved efficiently using it's `files_id` and n values:

```
cursor = db.fs.chunks.findOne({files_id: myFileID}).sort({n:1});
```

Implementing Authentication in a Driver

The current version of Mongo supports only very basic authentication. One authenticates a username and password in the context of a particular database. Once authenticated, the user has full read and write access to the database in question.

The `admin` database is special. In addition to several commands that are administrative being possible only on `admin`, authentication on `admin` gives one read and write access to all databases on the server. Effectively, `admin` access means root access to the db.

Note on a single socket we may authenticate for any number of databases, and as different users. This authentication persists for the life of the database connection (barring a `logout` command).

The Authentication Process

Authentication is a two step process. First the driver runs a `getnonce` command to get a nonce for use in the subsequent authentication. We can view a sample `getnonce` invocation from `dbshell`:

```
> db.$cmd.findOne({getnonce:1})
{ "nonce":"7268c504683936e1" , "ok":1
```

The nonce returned is a hex String.

The next step is to run an `authenticate` command for the database on which to authenticate. The authenticate command has the form:

```
{ authenticate : 1, user : username, nonce : nonce, key : digest }
```

where

- *username* is a username in the database's system.users collection;
- *nonce* is the nonce returned from a previous getnonce command;
- *digest* is the hex encoding of a MD5 message digest which is the MD5 hash of the concatenation of (*nonce*, *username*, *password_digest*), where *password_digest* is the user's password value in the pwd field of the associated user object in the database's system.users collection. pwd is the hex encoding of MD5(username + ":mongo:" + *password_text*).

Authenticate will return an object containing

```
{ ok : 1 }
```

when successful.

Details of why an authentication command failed may be found in the Mongo server's log files.

The following code from the Mongo Javascript driver provides an example implementation:

```
DB.prototype.addUser = function( username , pass ){
    var c = this.getCollection( "system.users" );

    var u = c.findOne( { user : username } ) || { user : username };
    u.pwd = hex_md5( username + ":mongo:" + pass );
    print( tojson( u ) );

    c.save( u );
}

DB.prototype.auth = function( username , pass ){
    var n = this.runCommand( { getnonce : 1 } );

    var a = this.runCommand(
        {
            authenticate : 1 ,
            user : username ,
            nonce : n.nonce ,
            key : hex_md5( n.nonce + username + hex_md5( username + ":mongo:" + pass ) )
        }
    );

    return a.ok;
}
```

Logout

Drivers may optionally implement the logout command which deauthorizes usage for the specified database for this connection. Note other databases may still be authorized.

Alternatively, close the socket to deauthorize.

```
> db.$cmd.findOne({logout:1})
{
    "ok" : 1.0
}
```

Replica Sets and Authentication

For drivers that support replica sets, extra care with replication is required.

When switching from one server in a replica set to another, for example in a failover situation, you must reauthenticate. Clients will likely want to cache authentication from the user so that the client can reauthenticate with the new server when appropriate.

Be careful also with operations such as Logout. If you log out from only some members of a replica set, that could be an issue.

Authenticating with a server in slave mode is allowed.

See Also

- Security and Authentication

Notes on Pooling for Mongo Drivers

Note that with the db write operations can be sent asynchronously or synchronously (the latter indicating a getlasterror request after the write).

When asynchronous, one must be careful to continue using the same connection (socket). This ensures that the next operation will not begin until after the write completes.

Pooling and Authentication

An individual socket connection to the database has associated authentication state. Thus, if you pool connections, you probably want a separate pool for each authentication case (db + username).

Pseudo-code

The following pseudo-code illustrates our recommended approach to implementing connection pooling in a driver's connection class. This handles authentication, grouping operations from a single "request" onto the same socket, and a couple of other gotchas:

```
class Connection:
   init(pool_size, addresses, auto_start_requests):
      this.pool_size = pool_size
      this.addresses = addresses
      this.auto_start_requests = auto_start_requests
      this.thread_map = {}
      this.locks = Lock[pool_size]
      this.sockets = Socket[pool_size]
      this.socket_auth = String[pool_size][]
      this.auth = {}

      this.find_master()

   find_master():
      for address in this.addresses:
         if address.is_master():
            this.master = address

   pick_and_acquire_socket():
      choices = random permutation of [0, ..., this.pool_size - 1]

      choices.sort(order: ascending,
                   value: size of preimage of choice under this.thread_map)

      for choice in choices:
         if this.locks[choice].non_blocking_acquire():
            return choice

      sock = choices[0]
      this.locks[sock].blocking_acquire()
      return sock

   get_socket():
      if this.thread_map[current_thread] >= 0:
         sock_number = this.thread_map[current_thread]
         this.locks[sock_number].blocking_acquire()
      else:
         sock_number = this.pick_and_lock_socket()
         if this.auto_start_requests or current_thread in this.thread_map:
            this.thread_map[current_thread] = sock_number
```

```
        if not this.sockets[sock_number]:
            this.sockets[sock_number] = Socket(this.master)

        return sock_number

    send_message_without_response(message):
        sock_number = this.get_socket()
        this.check_auth()
        this.sockets[sock_number].send(message)
        this.locks[sock_number].release()

    send_message_with_response(message):
        sock_number = this.get_socket()
        this.check_auth()
        this.sockets[sock_number].send(message)
        result = this.sockets[sock_number].receive()
        this.locks[sock_number].release()
        return result

    # start_request is only needed if auto_start_requests is False
    start_request():
        this.thread_map[current_thread] = -1

    end_request():
        delete this.thread_map[current_thread]

    authenticate(database, username, password):
        # TODO should probably make sure that these credentials are valid,
        # otherwise errors are going to be delayed until first op.
        this.auth[database] = (username, password)

    logout(database):
        delete this.auth[database]

    check_auth(sock_number):
        for db in this.socket_auth[sock_number]:
            if db not in this.auth.keys():
                this.sockets[sock_number].send(logout_message)
                this.socket_auth[sock_number].remove(db)
        for db in this.auth.keys():
            if db not in this.socket_auth[sock_number]:
                this.sockets[sock_number].send(authenticate_message)
                this.socket_auth[sock_number].append(db)

# somewhere we need to do error checking - if you get not master then everything
# in this.sockets gets closed and set to null and we call find_master() again.
```

```
# we also need to reset the socket_auth information - nothing is authorized yet
# on the new master.
```

See Also

- The Driver and Integration Center for information about the latest drivers
- The [top page] for this section
- The main Database Internals page
- The starting point for all Home

Driver and Integration Center

Connecting Drivers to Replica Sets

Ideally a MongoDB driver can connect to a cluster of servers which represent a replica set, and automatically find the right set member with which to communicate. Failover should be automatic too. The general steps are:

1. The user, when opening the connection, specifies host[:port] for one or more members of the set. Not all members need be specified -- in fact the exact members of the set might change over time. This list for the connect call is the *seed list*.
2. The driver then connects to all servers on the seed list, perhaps in parallel to minimize connect time. Send an ismaster command to each server.
3. When the server is in replSet mode, it will return a *hosts* field with all members of the set that are potentially eligible to serve data. The client should cache this information. Ideally this refreshes too, as the set's config could change over time.
4. Choose a server with which to communicate.
   a. If ismaster == true, that server is primary for the set. This server can be used for writes and immediately consistent reads.
   b. If secondary == true, that server is not primary, but is available for eventually consistent reads. In this case, you can use the *primary* field to see which server the master should be. (If primary is not set, you may want to poll other nodes at random; it is conceivable that the member to which we are talking is partitioned from the other members, and thus it cannot determine who is primary on its own. This is unlikely but possible.)
5. If an error occurs with the current connection, find the new primary and resume use there.

For example, if we run the ismaster command on a non-primary server, we might get something like:

```
> db.runCommand("ismaster")
{
        "ismaster" : false,
        "secondary" : true,
        "hosts" : [
                "ny1.acme.com",
                "ny2.acme.com",
                "sf1.acme.com"
        ],
        "passives" : [
             "ny3.acme.com",
             "sf3.acme.com"
        ],
        "arbiters" : [
            "sf2.acme.com",
        ]
        "primary" : "ny2.acme.com",
        "ok" : true
}
```

There are three servers with priority > 0 (*ny1*, *ny2*, and *sf1*), two passive servers (*ny3* and *sf3*), and an arbiter (*sf2*). The primary should be *ny2*, but the driver should call ismaster on that server before it assumes it is.

Error Handling in Mongo Drivers

If an error occurs on a query (or getMore operation), Mongo returns an error object instead of user data.

The error object has a first field guaranteed to have the reserved key `$err`. For example:

```
{ $err : "some error message" }
```

The `$err` value can be of any type but is usually a string.

Drivers typically check for this return code explicitly and take action rather than returning the object to the user. The query results flags include a set bit when $err is returned.

```
/* db response format

Query or GetMore: // see struct QueryResult
int resultFlags;
  int64 cursorID;
  int startingFrom;
  int nReturned;
  list of marshalled JSObjects;

*/

struct QueryResult : public MsgData {
  enum {
    ResultFlag_CursorNotFound = 1, /* returned, with zero results, when getMore is called but the
cursor id is not valid at the server. */
    ResultFlag_ErrSet = 2          /* { $err : ... } is being returned */
  };
  ...
};
```

See Also

- The Driver and Integration Center for information about the latest drivers
- The [top page] for this section
- The main Database Internals page
- The starting point for all Home

Developer Zone

- Tutorial
- Shell
- Manual
    - Databases
    - Collections
    - Indexes
    - Data Types and Conventions
    - GridFS
    - Inserting
    - Updating
    - Querying
    - Removing
    - Optimization
- Developer FAQ
- Cookbook

If you have a comment or question about anything, please contact us through IRC (freenode.net#mongodb) or the mailing list, rather than leaving a comment at the bottom of a page. It is easier for us to respond to you through those channels.

Introduction

MongoDB is a collection-oriented, schema-free document database.

By *collection-oriented*, we mean that data is grouped into sets that are called 'collections'. Each collection has a unique name in the database, and can contain an unlimited number of documents. Collections are analogous to tables in a RDBMS, except that they don't have any defined schema.

By *schema-free*, we mean that the database doesn't need to know anything about the structure of the documents that you store in a collection. In fact, you can store documents with different structure in the same collection if you so choose.

By *document*, we mean that we store data that is a structured collection of key-value pairs, where keys are strings, and values are any of a rich set of data types, including arrays and documents. We call this data format "BSON" for "Binary Serialized dOcument Notation."

MongoDB Operational Overview

MongoDB is a server process that runs on Linux, Windows and OS X. It can be run both as a 32 or 64-bit application. We recommend running in 64-bit mode, since Mongo is limited to a total data size of about 2GB for all databases in 32-bit mode.

The MongoDB process listens on port 27017 by default (note that this can be set at start time - please see Command Line Parameters for more information).

Clients connect to the MongoDB process, optionally authenticate themselves if security is turned on, and perform a sequence of actions, such as inserts, queries and updates.

MongoDB stores its data in files (default location is `/data/db/`), and uses memory mapped files for data management for efficiency.

MongoDB can also be configured for automatic data replication , as well as automatic fail-over .

For more information on MongoDB administration, please see Mongo Administration Guide.

MongoDB Functionality

As a developer, MongoDB drivers offer a rich range of operations:

- Queries: Search for documents based on either query objects or SQL-like "where predicates". Queries can be sorted, have limited return sizes, can skip parts of the return document set, and can also return partial documents.
- Inserts and Updates : Insert new documents, update existing documents.
- Index Management : Create indexes on one or more keys in a document, including substructure, deleted indexes, etc
- General commands : Any MongoDB operation can be managed via DB Commands over the regular socket.

cookbook.mongodb.org

> **Redirection Notice**
> This page should redirect to http://cookbook.mongodb.org.

Tutorial

- Running MongoDB
- Getting A Database Connection
- Dynamic Schema ("Schema Free")
- Inserting Data into A Collection
- Accessing Data From a Query
- Specifying What the Query Returns
- `findOne()` - Syntactic Sugar
- Limiting the Result Set via `limit()`
- More Help
- What Next

Running MongoDB

First, run through the Quickstart guide for your platform to get Mongo installed.

Getting A Database Connection

Let's now try manipulating the database with the database shell . (We could perform similar operations from any programming language using an appropriate driver. The shell is convenient for interactive and administrative use.)

Start the MongoDB JavaScript shell with:

```
# 'mongo' is shell binary. exact location might vary depending on
# installation method and platform
$ bin/mongo
```

By default the shell connects to database "test" on localhost. You then see:

```
MongoDB shell version: <whatever>
url: test
connecting to: test
type "help" for help
>
```

"`connecting to:`" tells you the name of the database the shell is using. To switch databases, type:

```
> use mydb
switched to db mydb
```

Switching to a database with the `use` command won't immediately create the database - the database is created lazily the first time data is inserted. This means that if you `use` a database for the first time it won't show up in the list provided by ``show dbs`` until data is inserted.

To see a list of handy commands, type `help`.

> **Tip for Developers with Experience in Other Databases**
> You may notice, in the examples below, that we never create a database or collection. MongoDB does not require that you do so. As soon as you insert something, MongoDB creates the underlying collection and database. If you query a collection that does not exist, MongoDB treats it as an empty collection.

Dynamic Schema ("Schema Free")

MongoDB has databases, collections, and indexes much like a traditional RDBMS. In some cases (databases and collections) these objects can be implicitly created, however once created they exist in a system catalog (db.systems.collections, db.system.indexes).

Collections contain (BSON) documents. Within these documents are fields. In MongoDB there is no predefinition of fields (what would be columns in an RDBMS). There is no schema for fields within documents – the fields and their value datatypes can vary. Thus there is no notion of an "alter table" operation which adds a "column". In practice, it is highly common for a collection to have a homogenous structure across documents; however this is not a requirement. This flexibility means that schema migration and augmentation are very easy in practice - rarely will you need to write scripts which perform "alter table" type operations. In addition to making schema migration flexible, this facility makes iterative software development atop the database easier.

Inserting Data into A Collection

Let's create a test collection and insert some data into it. We will create two objects, `j` and `t`, and then save them in the collection *things*.

In the following examples, '>' indicates commands typed at the shell prompt.

```
> j = { name : "mongo" };
{"name" : "mongo"}
> t = { x : 3 };
{ "x" : 3  }
> db.things.save(j);
> db.things.save(t);
> db.things.find();
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
>
```

A few things to note :

- We did not predefine the collection. The database creates it automatically on the first insert.
- The documents we store can have different fields - in fact in this example, the documents have no common data elements at all. In practice, one usually stores documents of the same structure within collections.
- Upon being inserted into the database, objects are assigned an object ID (if they do not already have one) in the field `_id`.
- When you run the above example, your ObjectID values will be different.

Let's add some more records to this collection:

```
> for (var i = 1; i <= 20; i++) db.things.save({x : 4, j : i});
> db.things.find();
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85b"), "x" : 4, "j" : 6 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85c"), "x" : 4, "j" : 7 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "j" : 8 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85e"), "x" : 4, "j" : 9 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85f"), "x" : 4, "j" : 10 }
{ "_id" : ObjectId("4c220a42f3924d31102bd860"), "x" : 4, "j" : 11 }
{ "_id" : ObjectId("4c220a42f3924d31102bd861"), "x" : 4, "j" : 12 }
{ "_id" : ObjectId("4c220a42f3924d31102bd862"), "x" : 4, "j" : 13 }
{ "_id" : ObjectId("4c220a42f3924d31102bd863"), "x" : 4, "j" : 14 }
{ "_id" : ObjectId("4c220a42f3924d31102bd864"), "x" : 4, "j" : 15 }
{ "_id" : ObjectId("4c220a42f3924d31102bd865"), "x" : 4, "j" : 16 }
{ "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
{ "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
has more
```

Note that not all documents were shown - the shell limits the number to 20 when automatically iterating a cursor. Since we already had 2 documents in the collection, we only see the first 18 of the newly-inserted documents.

If we want to return the next set of results, there's the `it` shortcut. Continuing from the code above:

```
{ "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
{ "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
has more
> it
{ "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "j" : 19 }
{ "_id" : ObjectId("4c220a42f3924d31102bd869"), "x" : 4, "j" : 20 }
```

Technically, find() returns a cursor object. But in the cases above, we haven't assigned that cursor to a variable. So, the shell automatically iterates over the cursor, giving us an initial result set, and allowing us to continue iterating with the **it** command.

But we can also work with the cursor directly; just how that's done is discussed in the next section.

Accessing Data From a Query

Before we discuss queries in any depth, lets talk about how to work with the results of a query - a cursor object. We'll use the simple `find()` query method, which returns everything in a collection, and talk about how to create specific queries later on.

In order to see all the elements in the collection when using the mongo shell, we need to explicitly use the cursor returned from the `find()` operation.

Lets repeat the same query, but this time use the cursor that `find()` returns, and iterate over it in a while loop :

```
> var cursor = db.things.find();
> while (cursor.hasNext()) printjson(cursor.next());
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85b"), "x" : 4, "j" : 6 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85c"), "x" : 4, "j" : 7 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "j" : 8 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85e"), "x" : 4, "j" : 9 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85f"), "x" : 4, "j" : 10 }
{ "_id" : ObjectId("4c220a42f3924d31102bd860"), "x" : 4, "j" : 11 }
{ "_id" : ObjectId("4c220a42f3924d31102bd861"), "x" : 4, "j" : 12 }
{ "_id" : ObjectId("4c220a42f3924d31102bd862"), "x" : 4, "j" : 13 }
{ "_id" : ObjectId("4c220a42f3924d31102bd863"), "x" : 4, "j" : 14 }
{ "_id" : ObjectId("4c220a42f3924d31102bd864"), "x" : 4, "j" : 15 }
{ "_id" : ObjectId("4c220a42f3924d31102bd865"), "x" : 4, "j" : 16 }
{ "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
{ "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
{ "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "j" : 19 }
{ "_id" : ObjectId("4c220a42f3924d31102bd869"), "x" : 4, "j" : 20 }
```

The above example shows cursor-style iteration. The `hasNext()` function tells if there are any more documents to return, and the `next()` function returns the next document. We also used the built-in `printjson()` method to render the document in a pretty JSON-style format.

When working in the JavaScript shell, we can also use the functional features of the language, and just call `forEach` on the cursor. Repeating the example above, but using `forEach()` directly on the cursor rather than the while loop:

```
> db.things.find().forEach(printjson);
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85b"), "x" : 4, "j" : 6 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85c"), "x" : 4, "j" : 7 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "j" : 8 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85e"), "x" : 4, "j" : 9 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85f"), "x" : 4, "j" : 10 }
{ "_id" : ObjectId("4c220a42f3924d31102bd860"), "x" : 4, "j" : 11 }
{ "_id" : ObjectId("4c220a42f3924d31102bd861"), "x" : 4, "j" : 12 }
{ "_id" : ObjectId("4c220a42f3924d31102bd862"), "x" : 4, "j" : 13 }
{ "_id" : ObjectId("4c220a42f3924d31102bd863"), "x" : 4, "j" : 14 }
{ "_id" : ObjectId("4c220a42f3924d31102bd864"), "x" : 4, "j" : 15 }
{ "_id" : ObjectId("4c220a42f3924d31102bd865"), "x" : 4, "j" : 16 }
{ "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
{ "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
{ "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "j" : 19 }
{ "_id" : ObjectId("4c220a42f3924d31102bd869"), "x" : 4, "j" : 20 }
```

In the case of a `forEach()` we must define a function that is called for each document in the cursor.

In the mongo shell, you can also treat cursors like an array :

```
> var cursor = db.things.find();
> printjson(cursor[4]);
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
```

When using a cursor this way, note that all values up to the highest accessed (cursor[4] above) are loaded into RAM at the same time. This is inappropriate for large result sets, as you will run out of memory. Cursors should be used as an iterator with any query which returns a large number of elements.

In addition to array-style access to a cursor, you may also convert the cursor to a true array:

```
> var arr = db.things.find().toArray();
> arr[5];
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
```

Please note that these array features are specific to mongo - The Interactive Shell, and not offered by all drivers.

MongoDB cursors are not snapshots - operations performed by you or other users on the collection being queried between the first and last call to `next()` of your cursor *may or may not* be returned by the cursor. Use explicit locking to perform a snapshotted query.

Specifying What the Query Returns

Now that we know how to work with the cursor objects that are returned from queries, lets now focus on how to tailor queries to return specific things.

In general, the way to do this is to create "query documents", which are documents that indicate the pattern of keys and values that are to be matched.

These are easier to demonstrate than explain. In the following examples, we'll give example SQL queries, and demonstrate how to represent the same query using MongoDB via the mongo shell. This way of specifying queries is fundamental to MongoDB, so you'll find the same general facility in any driver or language.

SELECT * FROM things WHERE name="mongo"

```
> db.things.find({name:"mongo"}).forEach(printjson);
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
```

SELECT * FROM things WHERE x=4

```
> db.things.find({x:4}).forEach(printjson);
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85b"), "x" : 4, "j" : 6 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85c"), "x" : 4, "j" : 7 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "j" : 8 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85e"), "x" : 4, "j" : 9 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85f"), "x" : 4, "j" : 10 }
{ "_id" : ObjectId("4c220a42f3924d31102bd860"), "x" : 4, "j" : 11 }
{ "_id" : ObjectId("4c220a42f3924d31102bd861"), "x" : 4, "j" : 12 }
{ "_id" : ObjectId("4c220a42f3924d31102bd862"), "x" : 4, "j" : 13 }
{ "_id" : ObjectId("4c220a42f3924d31102bd863"), "x" : 4, "j" : 14 }
{ "_id" : ObjectId("4c220a42f3924d31102bd864"), "x" : 4, "j" : 15 }
{ "_id" : ObjectId("4c220a42f3924d31102bd865"), "x" : 4, "j" : 16 }
{ "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
{ "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
{ "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "j" : 19 }
{ "_id" : ObjectId("4c220a42f3924d31102bd869"), "x" : 4, "j" : 20 }
```

The query expression is an document itself. A query document of the form { a:A, b:B, ... } means "where a==A and b==B and ...". More information on query capabilities may be found in the Queries and Cursors section of the Mongo Developers' Guide.

MongoDB also lets you return "partial documents" - documents that have only a subset of the elements of the document stored in the database. To do this, you add a second argument to the `find()` query, supplying a document that lists the elements to be returned.

To illustrate, lets repeat the last example `find({x:4})` with an additional argument that limits the returned document to just the "j" elements:

SELECT j FROM things WHERE x=4

```
> db.things.find({x:4}, {j:true}).forEach(printjson);
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "j" : 5 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85b"), "j" : 6 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85c"), "j" : 7 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85d"), "j" : 8 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85e"), "j" : 9 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85f"), "j" : 10 }
{ "_id" : ObjectId("4c220a42f3924d31102bd860"), "j" : 11 }
{ "_id" : ObjectId("4c220a42f3924d31102bd861"), "j" : 12 }
{ "_id" : ObjectId("4c220a42f3924d31102bd862"), "j" : 13 }
{ "_id" : ObjectId("4c220a42f3924d31102bd863"), "j" : 14 }
{ "_id" : ObjectId("4c220a42f3924d31102bd864"), "j" : 15 }
{ "_id" : ObjectId("4c220a42f3924d31102bd865"), "j" : 16 }
{ "_id" : ObjectId("4c220a42f3924d31102bd866"), "j" : 17 }
{ "_id" : ObjectId("4c220a42f3924d31102bd867"), "j" : 18 }
{ "_id" : ObjectId("4c220a42f3924d31102bd868"), "j" : 19 }
{ "_id" : ObjectId("4c220a42f3924d31102bd869"), "j" : 20 }
```

Note that the "_id" field is always returned.

`findOne()` - Syntactic Sugar

For convenience, the mongo shell (and other drivers) lets you avoid the programming overhead of dealing with the cursor, and just lets you retrieve one document via the `findOne()` function. `findOne()` takes all the same parameters of the `find()` function, but instead of returning a cursor, it will return either the first document returned from the database, or `null` if no document is found that matches the specified query.

As an example, lets retrieve the one document with `name=='mongo'`. There are many ways to do it, including just calling `next()` on the cursor (after checking for `null`, of course), or treating the cursor as an array and accessing the 0th element.

However, the `findOne()` method is both convenient and efficient:

```
> printjson(db.things.findOne({name:"mongo"}));
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
```

This is more efficient because the client requests a single object from the database, so less work is done by the database and the network. This is the equivalent of `find({name:"mongo"}).limit(1)`.

Another example of finding a single document by _id:

```
> var doc = db.things.findOne({_id:ObjectId("4c2209f9f3924d31102bd84a")});
> doc
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
```

Limiting the Result Set via `limit()`

You may limit the size of a query's result set by specifing a maximum number of results to be returned via the `limit()` method.

This is highly recommended for performance reasons, as it limits the work the database does, and limits the amount of data returned over the network. For example:

```
> db.things.find().limit(3);
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
```

More Help

In addition to the general "help" command, you can call help on `db` and `db.collection` (where "`collection`" is the name of a collection) to see a summary of methods available.

If you are curious about what a function is doing, you can type it without the ()s and the shell will print the source, for example:

```
> printjson
function (x) {
    print(tojson(x));
}
```

`mongo` is a full JavaScript shell, so any JavaScript function, syntax, or class can be used in the shell. In addition, MongoDB defines some of its own classes and globals (e.g., `db`). You can see the full API at http://api.mongodb.org/js/.

What Next

- After completing this tutorial the next step to learning MongoDB is to dive into the rest of the documentation for more details, you can see the new Manual for additional documentation.
- See also SQL to Mongo Mapping Chart
- The MongoDB Manual
- Tutorials in the MongoDB Manual

Manual

This is the MongoDB manual. Except where otherwise noted, all examples are in JavaScript for use with the mongo shell. There is a table available giving the equivalent syntax for each of the drivers.

- Connections
- Databases
    - Commands
        - getLog Command
        - movePrimary Command
        - setParameter Command
        - cloneCollection Command
        - Compact Command
        - Copy Database Commands
        - fsync Command
        - getLastError Command
        - Index-Related Commands
        - Viewing and Terminating Current Operation
        - Validate Command
        - Windows Service
        - getLastError_old
        - List of Database Commands
    - Mongo Metadata
- Collections
    - Capped Collections
    - createCollection Command
    - renameCollection Command
    - Using a Large Number of Collections
- Documents
- Data Types and Conventions
    - Dates
    - Timestamp data type
    - Internationalized Strings
    - Object IDs
        - ObjectId
    - Database References
- GridFS
    - When to use GridFS
- Indexes
    - Building indexes with replica sets
    - Index Versions
    - Geospatial Indexing
        - Geospatial Haystack Indexing
    - Indexing as a Background Operation
    - Multikeys
        - Using Multikeys to Simulate a Large Number of Indexes
    - Indexing Advice and FAQ

- Inserting
    - Legal Key Names
    - Schema Design
        - Tweaking performance by document bundling during schema design
    - Trees in MongoDB
- Optimization
    - Explain
    - Optimizing Object IDs
    - Optimizing Storage of Small Objects
    - Query Optimizer
- Querying
    - Mongo Query Language
    - Querying and nulls
    - Retrieving a Subset of Fields
    - Advanced Queries
    - Dot Notation (Reaching into Objects)
    - Full Text Search in Mongo
    - min and max Query Specifiers
    - OR operations in query expressions
    - Queries and Cursors
        - Tailable Cursors
    - Server-side Code Execution
    - Sorting and Natural Order
    - Aggregation
        - Aggregation Framework
            - Aggregation Framework - $group
            - Aggregation Framework - $limit
            - Aggregation Framework - $match
            - Aggregation Framework - $project
            - Aggregation Framework - $skip
            - Aggregation Framework - $sort
            - Aggregation Framework - $unwind
            - Aggregation Framework - Expression Reference
            - SQL to Aggregation Framework Mapping Chart
- Removing
- Updating
    - Atomic Operations
        - Atomic operation examples
        - How to Make an Auto Incrementing Field
    - findAndModify Command
    - Padding Factor
    - two-phase commit
    - Updating Data in Mongo
- MapReduce
    - Troubleshooting MapReduce
- Data Processing Manual

Connections

MongoDB is a database server: it runs in the foreground or background and waits for connections from the user. Thus, when you start MongoDB, you will see something like:

```
~/$ ./mongod
#
# some logging output
#
Tue Mar  9 11:15:43 waiting for connections on port 27017
Tue Mar  9 11:15:43 web admin interface listening on port 28017
```

It will stop printing output at this point but it hasn't frozen, it is merely waiting for connections on port 27017. Once you connect and start sending commands, it will continue to log what it's doing. You can use any of the MongoDB drivers or Mongo shell to connect to the database.

You *cannot* connect to MongoDB by going to http://localhost:27017 in your web browser. The database *cannot* be accessed via HTTP on port 27017.

Standard Connection String Format

The uri scheme described on this page is not yet supported by all of the drivers. Refer to a specific driver's documentation to see how much (if any) of the standard connection uri is supported. All drivers support an alternative method of specifying connections if this format is not supported.

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]
```

- `mongodb://` is a required prefix to identify that this is a string in the standard connection format.
- `username:password@` are optional. If given, the driver will attempt to login to a database after connecting to a database server.
- `host1` is the only required part of the URI. It identifies a server address to connect to.
- `:portX` is optional and defaults to :27017 if not provided.
- `/database` is the name of the database to login to and thus is only relevant if the `username:password@` syntax is used. If not specified the "admin" database will be used by default.
- `?options` are connection options. Note that if `database` is absent there is still a `/` required between the last host and the `?` introducing the options. Options are name=value pairs and the pairs are separated either by "&"" or ";".

As many hosts as necessary may be specified (for connecting to replica pairs/sets).

The options are:

Replica set:

- `replicaSet=name`
    - The driver verifies that the name of the replica set it connects to matches this name. Implies that the hosts given are a seed list, and the driver will attempt to find all members of the set.

Single server:

- `slaveOk=true|false`

Any configuration:

- `safe=true|false`
    - `true`: the driver sends a getLastError command after every update to ensure that the update succeeded (see also `w` and `wtimeoutMS`).
    - `false`: the driver does not send a getLastError command after every update.
- `w=n`
    - The driver adds { w : n } to the getLastError command. Implies `safe=true`.
- `wtimeoutMS=ms`
    - The driver adds { wtimeout : ms } to the getlasterror command. Implies `safe=true`.
- `fsync=true|false`
    - `true`: the driver adds { fsync : true } to the getlasterror command. Implies `safe=true`.
    - `false`: the driver does not not add fsync to the getlasterror command.
- `journal=true|false`
    - `true`: Sync to journal. Implies `safe=true`.
- `connectTimeoutMS=ms`
    - How long a connection can take to be opened before timing out.
- `socketTimeoutMS=ms`
    - How long a send or receive on a socket can take before timing out.

These options are not case sensitive.

Examples

Connect to a database server running locally on the default port:

```
mongodb://localhost
```

Connect and login to the admin database as user "fred" with password "foobar":

```
mongodb://fred:foobar@localhost
```

Connect and login to the "baz" database as user "fred" with password "foobar":

```
mongodb://fred:foobar@localhost/baz
```

Connect to a replica pair, with one server on example1.com and another server on example2.com:

```
mongodb://example1.com:27017,example2.com:27017
```

Connect to a replica set with three servers running on localhost (on ports 27017, 27018, and 27019):

```
mongodb://localhost,localhost:27018,localhost:27019
```

Connect to a replica set with three servers, sending all writes to the primary and distributing reads to the secondaries:

```
mongodb://host1,host2,host3/?slaveOk=true
```

Connect to localhost with safe mode on:

```
mongodb://localhost/?safe=true
```

Connect to a replica set with safe mode on, waiting for replication to succeed on at least two machines, with a two second timeout:

```
mongodb://host1,host2,host3/?safe=true;w=2;wtimeoutMS=2000
```

Connection Pooling

The server will use one thread per TCP connection, therefore it is highly recomended that your application use some sort of connection pooling. Luckily, most drivers handle this for you behind the scenes. One notable exception is setups where your app spawns a new process for each request, such as CGI and some configurations of PHP.

Databases

Each MongoDB server can support multiple *databases*. Each database is independent, and the data for each database is stored separately, for security and ease of management.

A database consists of one or more *collections*, the *documents* (objects) in those collections, and an optional set of security credentials for controlling access.

See Also

- Mongo Metadata

Commands

Introduction

The Mongo database has a concept of a *database command*. Database commands are ways to ask the database to perform special operations, or to request information about its current operational status.

- Introduction
- Privileged Commands
- Getting Help Info for a Command
- More Command Documentation

- List of Database Commands
- The MongoDB Manual has complete documentation of all MongoDB Database Commands.

A command is sent to the database as a query to a special collection namespace called `$cmd`. The database will return a single document with the command results - use `findOne()` for that if your driver has it.

The general command syntax is:

```
db.$cmd.findOne( { <commandname>: <value> [, options] } );
```

The shell provides a helper function for this:

```
db.runCommand( { <commandname>: <value> [, options] } );
```

For example, to check our database's current profile level setting, we can invoke:

```
> db.runCommand({profile:-1});
{
    "was" : 0.0 ,
    "ok" : 1.0
}
```

For many db commands, some drivers implement wrapper methods are implemented to make usage easier. For example, the mongo shell offers

```
> db.getProfilingLevel()
0.0
```

Let's look at what this method is doing:

```
> print( db.getProfilingLevel )
function () {
    var res = this._dbCommand({profile:-1});
    return res ? res.was : null;
}

> print( db._dbCommand )
function (cmdObj) {
    return this.$cmd.findOne(cmdObj);
}
```

Many commands have helper functions - see your driver's documentation for more information.

Privileged Commands

Certain operations are for the database administrator only. These privileged operations may only be performed on the special database named `admin`.

```
> use admin;
> db.runCommand("shutdown"); // shut down the database
```

If the db variable is not set to 'admin', you can use `adminCommand` (`_adminCommand` in versions earlier than 1.8) to switch to the right database automatically (and just for that operation):

```
> db.adminCommand("shutdown");
```

(For this particular command there is also a shell helper function, `db.shutdownServer`.)

Getting Help Info for a Command

Use `commandHelp` in shell to get help info for a command:

```
> db.commandHelp("datasize")
help for: datasize  example: { datasize:"blog.posts", keyPattern:{x:1}, min:{x:10}, max:{x:55} }
NOTE: This command may take awhile to run
```

(Help is not yet available for some commands.)

More Command Documentation

Showing first 15 of 17 results

- Commands Quick Reference Card

getLog Command

Log-Related Commands

With the `getLog` command you can return a list of the log categories available and then all the messages for those categories. This command was introduced in 1.9.2.

```
> db.adminCommand({getLog:"*|global|<cat>"})
```

Getting Categories

To get all possible log categories, pass `getLog` a "*" option.

```
> db.adminCommand({getLog:"*"})
{
    "names" : [
            "global",
            "rs",
            "startupWarnings" ],
    "ok" : 1
}
```

This output indicates that there are 3 log categories available: "global" (see below), replica set notices, and startup warnings.

"global" Category

The special "global" category includes all other categories. You can pass any category in to limit the returned lines.

```
> db.adminCommand({getLog:"global"})
{
    "log" : [
        "Thu Aug 18 13:57:05 [initandlisten] MongoDB starting : pid=654312892 port=27017
dbpath=/tmp/db1/ 64-bit host=localnose",
        "Thu Aug 18 13:57:05 [initandlisten] recover : no journal files present, no recovery needed",
        "Thu Aug 18 13:57:07 [websvr] admin web console waiting for connections on port 28017",
        "Thu Aug 18 13:57:07 [initandlisten] waiting for connections on port 27017",
        "Thu Aug 18 13:57:07 [initandlisten] connection accepted from 127.0.0.1:56703 #1",
        "Thu Aug 18 13:57:07 [rsStart] replSet STARTUP2",
        "Thu Aug 18 13:57:07 [rsSync] replSet SECONDARY",
        "Thu Aug 18 13:57:10 [initandlisten] connection accepted from 127.0.0.1:56704 #2"
    ],
    "ok" : 1
}
```

movePrimary Command

movePrimary

This command allows changing the primary shard for sharded database. The primary will hold all unsharded collections in that database.

mongos only
This command is only available on a sharded system through "mongos".

Drain Shard First

When using this with existing data with sharded collections you must be very careful. You must drain the shard which is the primary shard before running this command because it will move all collections in the database.

This command is meant to be run on an offline system and only in the case where you need to remove a shard.

```
> db.adminCommand({movePrimary:<dbname>, to:<shard-name>})

//enable sharding on the database, and move it to the shard01
> use test1 //new database
> db.adminCommand({enablesharding : "test1"})
> db.adminCommand({movePrimary : "test1", to : "shard01" })
//This will affect new collections on the test1 db
```

setParameter Command

Setting Runtime Parameters

This command allows changes to be made at runtime for some internal options as well some of the Command Line Parameters.

This command was added in 1.8 but some of the options have changed since then. Please pay attention to the version when they were made available.

```
> db.adminCommand({setParameter:1, option1:true, ...})
// real example to increase logging
> db.adminCommand({setParameter:1, logLevel:4})
```

```
//disallow table/collection scans
> db.adminCommand({setParameter:1, notablescan:true})
{ "was" : false, "ok" : 1 }
> db.foo.insert({_id:1, name:false})
> db.foo.find()
{ "_id" : 1 , "name" : false }
> db.foo.find({name:true})
error: { "$err" : "table scans not allowed:test.foo", "code" : 10111 }
> db.adminCommand({setParameter:1, notablescan:false})
{ "was" : true, "ok" : 1 }
> db.foo.find({name:true})
```

Options

option	value	description	version available
syncdelay	number	period between memory flushes	1.8
logLevel	int (0-5)	sets logging level	1.8
traceExceptions	true/false	logs stack trace on assert/error	2.1
quiet	true/false	sets logging quiet mode (more details)	1.8
notablescan	true/false	causes error if a table scan is required	1.8
journalCommitInterval	number (1-500 ms)	journal (group) commit window	1.9.1

Getting Parameter States

Much like the ability to set runtime parameters you can also get them.

```
> db.adminCommand({getParameter:1, syncdelay:true})
{ "syncdelay" : 60, "ok" : 1 }
```

cloneCollection Command

Copy a single collection from one server to another.

```
db.runCommand( { cloneCollection: <namespace>,
                 from: <host> [,query: <query_filter>] [,copyIndexes:<bool>] } );
```

Copies a collection from one server to another. Do not use on a single server as the destination is placed at the same `db.collection` (namespace) as the source.

The local copy of the namespace is emptied before the copying begins. Any existing data will be lost there.

Compact Command

- Running a compaction
- Killing a compaction
- Effects of a compaction
  - File System
- Details
- Replica Sets
- See Also

v2.0+

The compact command compacts and defragments a collection. Indexes are rebuilt at the same time. It is conceptually similar to `repairDatabase`, but works on a single collection rather than an entire database.

To run (example from the mongo shell):

```
> db.runCommand( { compact : 'mycollectionname' } )
OR
> db.mycollection.runCommand( "compact" )
```

The command will not return until compaction completes. You can view intermediate progress in either the `mongod` log file, or by running `db.currentOp()` in another shell instance.

Because this blocks all other activity, the compact command returns an error when invoked on a replica set primary. To force it to run on a replica set primary, include `force:true` in the command as a parameter.

Running a compaction

The compact command generally uses less disk space while performing its work than repairDatabase does (though in worst case can need just as much). Additionally, compact is faster.

Although faster than `repairDatabase`, this command blocks all other operations while running, and is still slow. Run during scheduled maintenance. If you run the command on a replica set secondary, the secondary will automatically demote itself to a "recovery" state until the compaction is complete.

Killing a compaction

At the beginning of compaction, indexes are dropped for the collection. At the end, indexes are rebuilt. Thus, if you kill the compaction in the middle of its operation, either with killOp or a server failure, indexes may be missing from your collection. If you are running with `--journal`, no data should ever be lost on a crash during compaction, although on startup the indexes will not be present. (Regardless of this, always do a backup before system maintenance such as this!) When the indexes are rebuilt, they will be in the 2.0 index format.

If there's a crash while the command is running, then as long as journaling is enabled, your data will be safe.

Additionally, if a compaction operation is interrupted, much of the existing free space in a collection may become un-reusable. In this scenario, it is recommended that compaction be run again, to completion, to restore use of this free space.

Effects of a compaction

> ⚠ This command fully compacts the collection resulting in no Padding Factor for existing documents, but the paddingFactor number is kept for the collection (unlike when a repair is done). Thus, updates which grow documents will be slower after compaction as documents will move much more often on updates (at least at first).

You may wish to run the `collstats` command (`db.collectionname.stats()` in the shell) before and after compaction to see how the storage space changes for the collection.

Compaction may increase the total size and number of your data files, especially when running compact for the first time on a collection. Even in this case, total collection storage space consumed will not increase.

File System

This operation will not reduce the amount of disk space used on the filesystem. Storage size is the amount of data allocated within the database files, not the size/number of the files on file system.

Details

- Compact may be run on replica secondaries and slaves. Compact commands do not replicate, so each host needs to be compacted separately.
- Currently, compact is a command issued to a mongod. In a sharded environment one would do each shard separately as a maintenance operation. (This is likely to change in future versions, along with other enhancements.)
- Capped collections cannot be compacted. (The documents of a capped collection are not subject to fragmentation, however.)

Replica Sets

- Compact each member separately.
- Ideally compaction runs on a secondary (*see comment regarding `force:true` above*).
- If `compact` is run on a secondary, the secondary will go into "recovering" state automatically (to prevent reads from being routed to it while compacting). Once the compaction is finished, it will automatically return to secondary state.
- A partial script demonstration how to automate step down and compaction can be found here.

See Also

- repairDatabase command

Copy Database Commands

MongoDB includes commands for copying a database from one server to another.

These options are faster than the alternative of doing a mongodump and mongorestore since they skip the step of creating a temporary on-disk BSON representation.

copydb Command

Copies an entire database from one name on one server to another name on another server. omit <from_hostname> to copy from one name to another on the same server. Cann must be sent/run on the "destination" server.

```
> // shell helper syntax:
> db.copyDatabase(<from_dbname>, <to_dbname>, <from_hostname>);

> // if you must authenticate with the source database
> db.copyDatabase(<from_dbname>, <to_dbname>, <from_hostname>, <username>, <password>);

> // pure command syntax (runnable from any driver):
> db.runCommand(
... {copydb: 1, fromhost: <hostname>, fromdb: <db>,
...  todb: <db>[, slaveOk: <bool>, username: <username>,
...  nonce: <nonce>, key: <key>]}";

> // command syntax for authenticating with the source:
> n = db.runCommand( { copydbgetnonce : 1, fromhost: ... } );
db.runCommand( { copydb : 1, fromhost: ..., fromdb: ..., todb: ..., username: ..., nonce: n.nonce,
key: <hash of username, nonce, password > } );
```

cloneDatabase

Similar to copydb but with a simplified syntax for the case where we simply want to copy a database to this server, and leave its name as-is.

```
> db.cloneDatabase(<from_hostname>);
```

The copy database command does not work when copying to or from a sharded collection.

Other Notes

- copyDatabase may be run against a slave/secondary (i.e., the source server can be a slave/secondary).
- copyDatabase does not snapshot in any sense: if the source database is changing during the copy, the destination will receive documents representing different points in time in the source during the copying interval.
- The command must be run on the destination server.
- The command does not lock either the source or the destination server for the duration of the operation. Each side will yield periodically to allow other reads/writes through.

fsync Command

v1.4+

fsync Command

The fsync command allows us to flush all pending writes to datafiles. More importantly, it also provides a lock option that makes backups easier.

The fsync command forces the database to flush all datafiles:

```
> use admin
> db.runCommand({fsync:1});
```

By default the command returns after synchronizing. To return immediately use:

```
> db.runCommand({fsync:1,async:true});
```

To fsync on a regular basis, use the --syncdelay command line option (see mongod --help output). By default a full flush is forced every 60 seconds.

Lock, Snapshot and Unlock

With journaling enabled, you may not need to lock at all for a snapshot. See the backups page.

The fsync command supports a lock option that allows one to safely snapshot the database's datafiles. While locked, all write operations are blocked, although read operations are still allowed. After snapshotting, use the unlock command to unlock the database and allow locks again. Example:

```
> use admin
switched to db admin
> db.runCommand({fsync:1,lock:1})
{
 "info" : "now locked against writes",
 "ok" : 1
}
> db.currentOp()
{
 "inprog" : [
 ],
 "fsyncLock" : 1
}

>// do some work here: for example, snapshot datafiles...
>// runProgram("/path/to/my-filesystem-snapshotting-script.sh")

> db.$cmd.sys.unlock.findOne();
{ "ok" : 1, "info" : "unlock requested" }
> // unlock is now requested.  it may take a moment to take effect.
> db.currentOp()
{ "inprog" : [ ] }
```

Caveats

While the database can be read while locked for snapshotting, if a write is attempted, this will block readers due to the database's use of a read/write lock. This should be addressed in the future : https://jira.mongodb.org/browse/SERVER-4243

Snapshotting Slaves

The above procedure works on replica slaves. The slave will not apply operations while locked. However, see the above caveats section.

Sharding

The fsync command can only be sent to a single node, not to the entire sharded cluster.

v2.0+ Notes

Shell Helpers

`db.fsyncLock()` and `db.fsyncUnlock()` are thin wrappers around the interface documented above. You can see their implementation below:

```
> db.fsyncLock
function () {
    return db.adminCommand({fsync:1, lock:true});
}
> db.fsyncUnlock
function () {
    return db.getSiblingDB("admin").$cmd.sys.unlock.findOne();
}
```

Calls to unlock now block

More significantly, the unlock command has slightly modified semantics. Prior to v2.0, the command would request an unlock and return immediately. This made it difficult to know for certain whether the database had, in fact, been unlocked.

This command now blocks until the database is unlocked. When it returns, you can be certain that the databased has synced the datafiles to disk and is again ready to be written to.

See Also

- Backups

getLastError Command

MongoDB does not wait for a response by default when writing to the database. Use the getLastError command to ensure that operations have succeeded.

Many of the drivers can invoke the getLastError command automatically on a write operation. Enabling this driver option is called "safe mode" ("write concern" in some drivers). When enabled, the driver piggybacks a getLastError message with each write message. It then awaits the result of the getLastError command before returning.

A few other alternative modes of operation exist. First, an application could not call getLastError at all. This might be appropriate if, for example, one is writing data to a log, and would not report the error to the user anyway. Another option would be to only call getLastError after a series of operations.

Running in the Shell

The `getlasterror` command checks for an error on the last database operation for this connection. Since it's a command, there are a few ways to invoke it:

```
> db.$cmd.findOne({getlasterror:1})
```

Or in the shell:

```
> db.runCommand("getlasterror")
```

Or you can use the shell helper:

```
> db.getLastError()
```

In the mongo shell, db.getLastError() returns the last error – null if no error. Use db.getLastErrorObj() to see the full error result. db.getLastErrorObj().err should be null if there is no error condition.

For more about commands, see the command documentation.

When to Use

getlasterror is primarily useful for write operations (although it is set after a command or query too). Write operations by default do not have a return code: this saves the client from waiting for client/server turnarounds during write operations. One can always call getLastError if one wants a return code.

If you're writing data to MongoDB on multiple connections, then it can sometimes be important to call `getlasterror` on one connection to be certain that the data has been committed to the database. For instance, if you're writing to connection #1 and want those writes to be reflected in reads from connection #2, you can assure this by calling `getlasterror` after writing to connection #1.

For maximum speed, skip the getLastError (or "safe mode") for noncritical writes. Use it when you need to confirm to the user that the write succeeded.

Options

`j`

v2.0+: When j:true is specified, the getlasterror call awaits the journal commit before returning. If the server is running without journaling, it returns immediately, and successfully.

```
> db.runCommand({getlasterror:1,j:true})
```

`w`

A client can block until a write operation has been replicated to N servers.

`wtimeout` may be used in conjunction with `w`. The default is no timeout (wait forever).

```
> db.getLastError(2, 5000) // w=2, timeout 5000ms
```

Note the above options can be combined: waiting for journal acknowledgement *and* acknowledgement that the write has reached a majority of a replica set can make sense.

Tagging And Majority

Now with MongoDB 2.0 there is the ability to control which nodes are written to. Instead of specifying the number of nodes which a write must be acknowledged by, you can specify rules based on tags in the configuration.

In addition to the tagging rules you can also specify a string of "majority":

```
>db.getLastError("majority") // waits for more than 50% of the configured nodes to acknowledge the
write (until replication is applied to the point of that write)
```

`fsync`

When running mongod without journaling (--nojournal), the fsync option forces the database to fsync all files before returning.

When running with journaling, the fsync option awaits the next group commit before returning.

```
> db.runCommand({getlasterror:1,fsync:true})
{ "err" : null, "n" : 0, "fsyncFiles" : 2, "ok" : 1 }
```

Use `j`, not `fsync`.

Combining

A good combination for highly critical writes is `j:true` and `w:"majority"`.

Return Value

The return value from the command is an object with various fields. The common fields are listed below; there may also be other fields.

- ok - true indicates the getLastError command completed successfully. This does NOT indicate there wasn't a last error.
- err - if non-null, indicates an error occurred. Value is a textual description of the error.
- code - if set, indicates the error code which occurred.
- connectionId - the id of the connection
- lastOp - the op-id from the last operation

For updates:

- n - if an update was done, this is the number of documents updated.

With `w:<n>/<tag>`

- wnote - if set indicates that something unusual happened that is related to using w:
- wtimeout - if timed out, set to the value true
- waited - if timed out, how long waited in milliseconds
- wtime - the time spent waiting for the operation to complete

Using getLastError from Drivers

The drivers support `getLastError` in the command form and many also offer a "safe" mode for operations. For more on "safe" mode, see each driver's documentation.

* C#

- C++
- Python. If you're using Python, you automatically call getlasterror on insert as follows:

  ```
  collection.save({"name": "MongoDB"}, safe=True)
  ```

  If the save doesn't succeed, an exception will be raised.

- Java
  Java supports various getLastError() semantics using `WriteConcern` Objects.

Mongo Shell REPL Behavior

The database shell performs a `resetError()` before each read/eval/print loop command evaluation - and automatically prints the error, if one occurred, after each evaluation. Thus, after an error, at the shell prompt `db.getLastError()` will return null. However, if called before returning to the prompt, the result is as one would expect:

```
> try { db.foo.findOne() } catch(e) { print("preverr:" + tojson(db.getPrevError())); print("lasterr:"
+ tojson(db.getLastError()));}
preverr:{"err" : "unauthorized" , "nPrev" : 1 , "ok" : 1}
lasterr:"unauthorized"
```

getPrevError Command

When performing bulk write operations, `resetError()` and `getPrevError()` can be an efficient way to check for success of the operation. For example if we are inserting 1,000 objects in a collection, checking the return code 1,000 times over the network is slow. Instead one might do something like this:

```
db.resetError();
for( loop 1000 times... )
    db.foo.save(something...);
if( db.getPrevError().err )
    print("didn't work!");
```

- getPrevError only holds 1 previous error, so whole batch may have to be retried in case of error.
- A better alternative is to use the "batch" insert method provided by many drivers, along with safe writes. This way "getLastError" is only called after each batch, but still writes will stop at the 1st error.
- getPrevError may be deprecated in the future.

See Also

* Replica Set Design Concepts
* Verifying Propagation of Writes with getLastError

Index-Related Commands

Operations:

* Create Index
* Dropping an Index
* ReIndex
* Index Namespace

Create Index

`ensureIndex()` is the helper function for this. Its implementation creates an index by adding its info to the `system.indexes` collection.

```
> use test
> db.myCollection.ensureIndex(<keypattern>);
> // same as:
> db.system.indexes.insert({ name: "name", ns: "namespaceToIndex",
  key: <keypattern> });
```

Note: Once you've inserted the index, all subsequent document inserts for the given collection will be indexed, as will all pre-existing documents in the collection. If you have a large collection, this can take a significant amount of time and will block other operations.

Options

See Indexing Options.

You can query system.indexes to see all indexes for a collection *`foo`* in db *`test`*:

```
>db.system.indexes.find( { ns: "test.foo" } );
```

In some drivers, `ensureIndex()` remembers if it has recently been called, and foregoes the insert operation in that case. Even if this is not the case, `ensureIndex()` is a cheap operation, so it may be invoked often to ensure that an index exists.

Dropping an Index

From the shell:

```
db.mycollection.dropIndex(<name_or_pattern>)
db.mycollection.dropIndexes()

// example:
t.dropIndex( { name : 1 } );
```

From a driver (raw command object form; many drivers have helpers):

```
{ deleteIndexes: <collection_name>, index: <index_name> }
// "*" for <index_name> will drop all indexes except _id
```

ReIndex

The reIndex command will rebuild all indexes for a collection.

```
db.myCollection.reIndex()
// same as:
db.runCommand( { reIndex : 'myCollection' } )
```

Usually this is unnecessary. You may wish to do this if the size of your collection has changed dramatically or the disk space used by indexes seems oddly large.

Since 1.8 indexes will automatically be compacted as they are updated.

reIndex is a blocking operation (indexes are rebuilt in the foreground) and will be slow for large collections. The command will **not** propagate to secondaries in a replica set. It is only intended to effect the database on the instance on which the command is run.

The repair database command recreates all indexes in the database.

Index Namespace

Each index has a namespace of its own for the btree buckets. The namespace is:

```
<collectionnamespace>.$<indexname>
```

This is an internal namespace that cannot be queried directly.

Viewing and Terminating Current Operation

- View Operation(s) in Progress
- Killing an In Progress Operation
- Sharded Databases
- See Also

View Operation(s) in Progress

```
> db.currentOp();
{ inprog: [ { "opid" : 18 , "op" : "query" , "ns" : "mydb.votes" ,
              "query" : "{ score : 1.0 }" , "inLock" : 1 }
          ]
}
>
> // to include idle connections in report:
> db.currentOp(true);
```

Fields:

- opid - an incrementing operation number. Use with killOp().
- active - if the operation is active, false if the operation is queued
- waitingForLock - if true, lock has been requested but not yet granted
- op - the operation type (query, update, etc.)
- ns - namespace for the operation (database + collection name) (Note: if the ns begins with a question mark, the operation has yielded.)
- query - the query spec, if operation is a query
- lockType - the type of lock the operation requires, either read or write or none. See concurrency page.
- client - address of the client who requested the operation
- desc - the type of connection. `conn` indicates a normal client connections. Other values indicate internal threads in the server.
- threadId - id of the thread
- numYields - the number of the times the operation has yielded (to some other operation)

From a driver one runs currentOp by executing the equivalent of:

```
db.$cmd.sys.inprog.find()
```

Killing an In Progress Operation

v1.4+

```
> db.killOp(1234/*opid*/)
> // same as: db.$cmd.sys.killop.findOne({op:1234})
```

Be careful about terminating internal operations, for example the a replication sync thread. Typically you only kill operations from **external clients**.

Sharded Databases

In a sharded environment, operations named "writebacklistener" will appear. These are long-lived socket connections between mongos and mongod. These can be ignored. See Sharding FAQ.

See Also

- How does concurrency work

Validate Command

Use this command to check that a collection is valid (not corrupt) and to get various statistics.

This command scans the entire collection and its indexes and will be very slow on large datasets.

option	description
full	Validates everything – new in 2.0.0
scandata	Validates basics (index/collection-stats) and document headers

> **Blocking**
> This is a blocking operation (like Repair Database Command).

From the `mongo` shell:

```
> db.foo.validate({full:true})
{
 "ns" : "test.foo",
 "firstExtent" : "0:109c00 ns:test.foo",
 "lastExtent" : "0:2e0d00 ns:test.foo",
 "extentCount" : 3,
 "extents" : [
  {
   "loc" : "0:109c00",
   "xnext" : "0:192000",
   "xprev" : "null",
   "nsdiag" : "test.foo",
   "size" : 12288,
   "firstRecord" : "0:109cb0",
   "lastRecord" : "0:10cb90"
  },
  {
   "loc" : "0:192000",
   "xnext" : "0:2e0d00",
   "xprev" : "0:109c00",
   "nsdiag" : "test.foo",
   "size" : 49152,
   "firstRecord" : "0:1920b0",
   "lastRecord" : "0:19df00"
  },
  {
   "loc" : "0:2e0d00",
   "xnext" : "null",
   "xprev" : "0:192000",
   "nsdiag" : "test.foo",
   "size" : 196608,
   "firstRecord" : "0:2e0db0",
   "lastRecord" : "0:30c820"
  }
 ],
 "datasize" : 224112,
 "nrecords" : 1001,
 "lastExtentSize" : 196608,
 "padding" : 1,
 "firstExtentDetails" : {
  "loc" : "0:109c00",
  "xnext" : "0:192000",
  "xprev" : "null",
  "nsdiag" : "test.foo",
  "size" : 12288,
  "firstRecord" : "0:109cb0",
  "lastRecord" : "0:10cb90"
 },
 "objectsFound" : 1001,
 "invalidObjects" : 0,
 "bytesWithHeaders" : 240128,
 "bytesWithoutHeaders" : 224112,
 "deletedCount" : 1,
 "deletedSize" : 17392,
 "nIndexes" : 1,
 "keysPerIndex" : {
  "test.foo.$_id_" : 1001
 },
 "valid" : true,
 "errors" : [ ],
 "ok" : 1
}
```

Example From Driver

From a driver one might invoke the driver's equivalent of:

```
> db.$cmd.findOne({validate:"foo" } );
```

```
> db.$cmd.findOne({validate:"foo", full:true});
```

Windows Service

mongod.exe has native support for installing and running as a Windows service.

2.1.1+ This feature is available in mongos.exe as well

Service-related command line options

The main command line options for working with MongoDB as a Windows service are:

--install	Install as a Windows service, then exit without starting the service
--remove	Remove the Windows service, stopping it first if required
--reinstall	Remove and then reinstall the Windows service

You may use the following options with --install, --reinstall and --remove: in the examples, {arg} is any text. Use quotes around the text if it includes spaces.

--serviceName {arg}	Specify the service name (used with "net start/stop", and usually does not include spaces)
--serviceDisplayName {arg}	Specify the service's display name
--serviceDescription {arg}	Specify the service's description
--serviceUser {arg}	Specify login account for running the service (account must have the "Log on as a service" right)
--servicePassword {arg}	Specify password for login account

More than one instance of a MongoDB program can be installed as a service by specifying unique serviceName and serviceDisplayName options for each one. The serviceDescription does not have to be unique but this may make identifying the separate instances easier. The serviceUser and servicePassword options are used with --install and --reinstall to configure the login account used by the Service Control Manager when starting the service. If no serviceUser is specified, the system account LocalService will be used.

Because the running service does not have a command window to display its log output, the --logpath option must be used when installing the service. In most cases you will also want to specify --logappend so that restarts of the service append to the existing log file.

An additional command line option is used internally by the install feature: this option is added to the command line that is used by the Windows Service Control Manager when starting the service, and is not used when installing or removing the service. Running mongod.exe from the command line with the --service option will fail, because it cannot communicate with the Service Control Manager if it was not started by the SCM.

--service	Indicates that the process is running as a service: do not use on the Windows command line

Both --remove and --reinstall will stop the service if it is currently running.

A config file may be specified using the --config or -f option and this may simplify updating parameters after the service is installed. Service-related options must not be specified in the config file; these options are only used when installing or removing the service and must be specified only on the command line.

Other options that you include on the command line with --install will be added to the command line that is used to run the service. For example, if you execute:

```
mongod --logpath d:\mongo\logs\logfilename.log --logappend --dbpath d:\mongo\data --install
```

this will cause a service to be created with service name "MongoDB" and display name "Mongo DB" that will execute the following command line:

```
mongod --logpath d:\mongo\logs\logfilename.log --logappend --dbpath d:\mongo\data --service
```

If any file specification includes spaces, put quotes around the file specification.

```
mongod --logpath "d:\my mongo\logs\my log file name.log" --logappend --dbpath "d:\my mongo\data"
--install
```

Installing on Windows 7

If installing on Windows 7, you need to make sure that you're running as an administrator. To do this, open the start menu, and in the search box enter "cmd.exe." When the executable appears, right-click on it and choose "Run as administrator."

At this point, you can install MongoDB as a service with the `--install` option as described above.

Installing mongos.exe as a Windows service in versions earlier than 2.1.1

Versions of mongos.exe earlier than 2.1.1 require an alternate method to install as a service. Using the Windows Resource Kit available here, you can configure a Windows host to start mongos at system startup. (Tested on a Windows 7 system, with MongoDB 1.8.1):

1. Assume your mongos config file is at C:\MongoDB\mongos.conf, and that you've tested that it works.
2. Start cmd.exe as an administrator.
3. Install srvany.exe as a service, like this: C:\Program Files\Windows Resource Kits\Tools>"C:\Program Files\Windows Resource Kits\Tools\instsrv.exe" mongos "C:\Program Files\Windows Resource Kits\Tools\srvany.exe"
   The output looks like this:

   ```
   The service was successfuly added\!

   Make sure that you go into the Control Panel and use
   the Services applet to change the Account Name and
   Password that this newly installed service will use
   for its Security Context.
   ```

4. Start regedit.exe
5. Find the subkey

   ```
   HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\mongos
   ```

6. Right click on mongos, select New > Key. Name the key Parameters.
7. Select the Parameters key.
8. Right click in the right window. Select New > String Value. Name the new value Application.
9. Modify Application's value to be the full path to mongos.exe (e.g., c:\mongodb\bin\mongos.exe)
10. Right click in the right window again. Select New > String Value. Name the new value AppParameters.
11. Modify AppParameters's value to be the arguments to mongos (e.g., --config C:\mongodb\mongos.conf)
12. To verify that the service can be started, try NET START mongos.

getLastError_old

Redirection Notice
This page should redirect to getLastError Command in about 3 seconds.

List of Database Commands

List of MongoDB Commands

See the Commands page for details on how to invoke a command.

Also: with v1.5+, run `mongod` with `--rest` enabled, and then go to http://localhost:28017/_commands

Commands Quick Reference Card

Mongo Metadata

The `<dbname>.system.*` namespaces in MongoDB are special and contain database system information. System collections include:

- `<dbname>.system.namespaces` lists all namespaces.
- `<dbname>.system.indexes` lists all indexes. Additional namespace / index metadata exists in the database.ns files, and is opaque.
- `<dbname>.system.profile` stores database profiling information.
- `<dbname>.system.users` lists users who may access the database.

Additionally, in the `local` database **only** there is replication information in system collections, e.g., `local.system.replset` contains the replica set configuration.

Information on the structure of a stored document is stored within the document itself. See BSON .

There are several restrictions on manipulation of objects in the system collections. Inserting in `system.indexes` adds an index, but otherwise that table is immutable (the special drop index command updates it for you). `system.users` is modifiable. `system.profile` is droppable.

Note: $ is a reserved character. Do not use it in namespace names or within field names. Internal collections for indexes use the $ character in their names. These collection store b-tree bucket data and are not in BSON format (thus direct querying is not possible).

Collections

MongoDB collections are essentially named groupings of documents. You can think of them as roughly equivalent to relational database tables.

Overview

A MongoDB collection is a collection of BSON documents. These documents usually have the same structure, but this is not a requirement since MongoDB is a *schema-free* (or more accurately, "dynamic schema") database. You may store a heterogeneous set of documents within a collection, as you do not need predefine the collection's "columns" or fields.

A collection is created when the first document is inserted.

Collection names should begin with letters or an underscore and may include numbers; $ is reserved. Collections can be organized in namespaces; these are named groups of collections defined using a dot notation. For example, you could define collections blog.posts and blog.authors, both reside under "blog". Note that this is simply an organizational mechanism for the user -- the collection namespace is flat from the database's perspective.

The maximum size of a collection name is 128 characters (including the name of the db and indexes). It is probably best to keep it under 80/90 chars.

Shell

Programmatically, we access these collections using the dot notation. For example, using the mongo shell:

```
if( db.blog.posts.findOne() )
    print("blog.posts exists and is not empty.");
```

Alternative ways to access collections are:

```
> db["mycol"].find()
> db.getCollection("mycol").find()
```

Note that though the underscore character is allowed, it has a special function in the shell if it is the 1st character: the shell considers the property to be an actual javascript value, not a collection name. Consequently it is not accessible using the dot notation, but it works fine with getCollection().

```
> db._mycol.find() --> error
> db.getCollection("_mycol").find() --> success
```

See also:

Capped Collections

Capped collections are fixed sized collections that have a very high performance auto-FIFO age-out feature (age out is based on insertion order). They are a bit like the "RRD" concept if you are familiar with that.

In addition, capped collections automatically, with high performance, maintain insertion order for the documents in the collection; this is very powerful for certain use cases such as logging.

Capped collections are not shard-able.

Creating

Unlike a standard collection, you must explicitly create a capped collection, specifying a collection size in bytes. The collection's data space is then preallocated. Note that the size specified includes database headers.

```
> db.createCollection("mycoll", {capped:true, size:100000})
```

Behavior

- Once the space is fully utilized, newly added documents will replace the oldest document in the collection.
- If you perform a `find()` on the collection with no ordering specified, the document will always be returned in insertion order. Reverse order is always retrievable with find().sort({$natural:-1}).

Usage and Restrictions

- You may insert new documents in the capped collection.
- You may update the existing documents in the collection. However, the documents must not grow in size. If they do, the update will fail. Note if you are performing updates, you will likely want to declare an appropriate index (given there is no _id index for capped collections by default).
- The database does not allow deleting documents from a capped collection. Use the `drop()` method to remove all rows from the collection. (After the drop you must explicitly recreate the collection.)
- Capped collection are not shard-able.

Warning

Capped collections do not have a unique index on `_id`. Replication requires unique _ids. Therefore, if you are using a capped collections **and** replication, you must ensure that you have unique _ids. Having duplicate _ids in your capped collection may cause replication to halt on slaves/secondaries and require manual intervention or a resync.

You may want to create a unique index on _id to prevent this issue, see the `autoIndexId` section below.

Applications

- **Logging.** Capped collections provide a high-performance means for storing logging documents in the database. Inserting documents in an unindexed capped collection will be close to the speed of logging to a filesystem. Additionally, with the built-in FIFO mechanism, you are not at risk of using excessive disk space for the logging.

- **Automatic Maintaining of Insertion Order.** Capped collections keep documents in their insertion order automatically, with no index being required for this property. The logging example above is a good example of a case where keeping items in order is important.

- **Caching.** If you wish to cache a small number of documents in the database, perhaps cached computations of information, the capped tables provide a convenient mechanism for this. Note that for this application you will typically want to use an index on the capped table as there will be more reads than writes.

- **Automatic Age Out.** If you know you want data to automatically "roll out" over time as it ages, a capped collection can be an easier way to support than writing manual removal via cron scripts. Ejection from the capped collection is also inexpensive compared to explicit remove operations.

Recommendations

- When appropriate, do not create indexes on a capped collection. If the collection will be written to much more than it is read from, it is better to have no indexes. Note that you may create indexes on a capped collection; however, you are then moving from "log speed" inserts to "database speed" inserts -- that is, it will still be quite fast by database standards.
- Use natural ordering to retrieve the most recently inserted elements from the collection efficiently. This is (somewhat) analogous to tail on a log file.

Options

size.

The size, in bytes, of the capped collection. This must be specified.

max

You may also optionally cap the number of documents in the collection. Once the limit is reached, items roll out on a least recently inserted basis.

Note: When specifying a cap on the number of documents, you must also cap on size. Be sure to leave enough room for your chosen number of documents or items will roll out faster than expected. You can use the `validate()` utility method to see how much space an existing collection uses, and from that estimate your size needs.

```
db.createCollection("mycoll", {capped:true, size:100000, max:100});
db.mycoll.validate();
```

autoIndexId

The `autoIndexId` field may be set to true or false to explicitly enable or disable automatic creation of a unique key index on the `_id` field.

> An index is **not** automatically created on _id for capped collections by default.

If you will be using the `_id` field, you should create an index on `_id`.

Given these are used sometimes without an `_id` index, it can be useful to insert documents without an _id field. Most drivers and the mongo shell add an _id client-side. See each driver's documentation for how to suppress this (behavior might vary by driver). In the mongo shell one could invoke:

```
> db.mycollection._mongo.insert(db.mycollection._fullName, myDocWithoutAnId)
```

Checking if a collection is capped

You can check if a collection is capped by using the isCapped() shell function. db.foo.isCapped()

Here is the definition.

```
> db.c.isCapped
function () {
    var e = this.exists();
    return e && e.options && e.options.capped ? true : false; }
```

Converting a collection to capped

You can convert a (non-capped) collection to a capped collection with the `convertToCapped` command:

```
> db.runCommand({"convertToCapped": "mycoll", size: 100000});
{ "ok": 1 }
```

Note that the `size` is in bytes.

> **Indexes not copied**
> No indexes will be created when the new capped collection is. If you want the old indexes you will need to recreate them after it has been converted.

See Also

- The Sorting and Natural Order section of this Guide
- Creating collections manually createCollection Command

createCollection Command

Use the createCollection command to create a collection explicitly. Often this is used to declare Capped Collections.

```
> // mongo shell
> db.createCollection("mycoll", {capped:true, size:100000}) // Size is in bytes
> show collections
```

Most drivers also have a create collection helper method. You can manually issue any command also.

```
> db.runCommand( {create:"mycoll", capped:true, size:100000} )
```

renameCollection Command

> **Sharding**
> The renameCollection command cannot be used with sharded collections.

This command can be used to rename an existing collection.

Shell:

```
> db.oldname.renameCollection("newname")
```

From drivers without a helper method one can invoke the generic command syntax:

```
> db.runCommand( { renameCollection: "mydb.oldname", to: "mydb.newname" }
```

This command is executed atomically and should be safe to use on a production DB. It changes the metadata associated with the given collection and then copies the index metadata from the old namespace to the new one. The amount of time to execute this command is constant and is not related to the size of the collection or the indexes. If there is an open cursor on a collection and it is renamed, the cursor will be invalidated and won't get any more data.

See Also

- https://jira.mongodb.org/browse/SERVER-4820
- https://jira.mongodb.org/browse/SERVER-4941

Using a Large Number of Collections

A technique one can use with MongoDB in certain situations is to have several collections to store information instead of a single collection. By doing this, certain repeating data no longer needs to be stored in every object, and an index on that key may be eliminated. More importantly for performance (depending on the problem), the data is then clustered by the grouping specified.

For example, suppose we are logging objects/documents to the database, and want to have M logs: perhaps a dev log, a debug log, an ops log, etc. We could store them all in one collection 'logs' containing objects like:

```
{ log : 'dev', ts : ..., info : ...  }
```

However, if the number of logs is not too high, it might be better to have a collection per log. We could have a 'logs.dev' collection, a 'logs.debug' collection, 'logs.ops', etc.:

```
// logs.dev:
{ ts : ..., info : ... }
```

Of course, this only makes sense if we do not need to query for items from multiple logs at the same time.

Generally, having a large number of collections has no significant performance penalty, and results in very good performance.

Limits

By default MongoDB has a limit of approximately 24,000 *namespaces* per database. Each namespace is 628 bytes, the `.ns` file is 16MB by default.

Each collection counts as a namespace, as does each index. Thus if every collection had one index, we can create up to 12,000 collections. The `--nssize` parameter allows you to increase this limit (see below).

Be aware that there is a certain minimum overhead per collection -- a few KB. Further, any index will require at least 8KB of data space as the b-tree page size is 8KB. Certain operations can get slow if there are a lot of collections and the meta data gets paged out.

`--nssize`

If more collections are required, run mongod with the `--nssize` parameter specified. This will make the <database>.ns file larger and support more collections. Note that `--nssize` sets the size used for newly created `.ns` files -- if you have an existing database and wish to resize, after running the db with `--nssize`, run the `db.repairDatabase()` command from the shell to adjust the size.

Maximum `.ns` file size is 2GB.

Documents

- Maximum Document Size

MongoDB can be thought of as a document-oriented database. By 'document', we mean structured documents, not freeform text documents. These documents can be thought of as objects but only the data of an object, not the code, methods or class hierarchy. Additionally, there is much less linking between documents in MongoDB data models than there is between objects in a program written in an object-oriented programming language.

In MongoDB the documents are conceptually JSON. More specifically the documents are represented in a format called BSON (standing for Binary JSON).

Documents are stored in Collections.

Maximum Document Size

MongoDB limits the data size of individual BSON objects/documents. At the time of this writing the limit is 16MB.

This limit is designed as a sanity-check; it is not a technical limit on document sizes. The thinking is that if documents are larger than this size, it is likely the schema is not ideal. Further it allows drivers to make some assumptions on the max size of documents.

The concept is that the maximum document size is a limit that ensures each document does not require an excessive amount of RAM from the machine, or require too much network bandwidth to fetch. For example, fetching a full 100MB document would take over 1 second to fetch over a gigabit ethernet connection. In this situation one would be limited to 1 request per second.

Over time, as computers grow in capacity, the limit will be adjusted upward.

For cases where larger sizes are required, use GridFS.

Data Types and Conventions

MongoDB (BSON) Data Types

Mongo uses special data types in addition to the basic JSON types of string, integer, boolean, double, null, array, and object. These types include date, object id, binary data, regular expression, and code. Each driver implements these types in language-specific ways, see your driver's documentation for details.

See BSON for a full list of database types.

Internationalization

- See Internationalized strings

Database References

- See Database References and Schema Design

Checking Types from the Shell

Floats and ints are treating as standard javascript numbers, and are thus hard to tell apart in the shell.

```
> // v1.8+ shell
> x
{
        "_id" : ObjectId("4dcd3ebc9278000000005158"),
        "d" : ISODate("2011-05-13T14:22:46.777Z"),
        "b" : BinData(0,""),
        "c" : "aa",
        "n" : 3,
        "e" : [ ],
        "n2" : NumberLong(33)
}
> x.d instanceof Date
true
> x.b instanceof BinData
true
> typeof x
object
> typeof x.b
object
> typeof x.n
number
> typeof x.n
number
> typeof x.n2
object
> x.n2 instanceof NumberLong
true
> typeof x.c
string
```

Dates

The BSON Date/Time data type is referred to as "UTC DateTime" in the BSON spec.

Note – There is a Timestamp data type but that is a special internal type for MongoDB that typically should not be used.

A BSON Date value stores the number of milliseconds since the Unix epoch (Jan 1, 1970) as a 64-bit integer. **v2.0+** : this number is signed so dates before 1970 are stored as a negative numbers.

Before MongoDB v2.0 dates were incorrectly interpreted as an unsigned integer, adversely affected sorting, range queries, and indexes on Date fields. Indexes are not recreated when upgrading. Thus if you created on index on Date values with pre v2.0 versions, and dates before 1970 are relevant to your application, please reindex.

In the shell

```
> x = new Date()
ISODate("2011-10-12T14:54:02.069Z")

> x.toString()
Wed Oct 12 2011 10:54:02 GMT-0400 (Eastern Daylight Time)

> d = ISODate()                      // like Date() but behaves more intuitively when used
> d = ISODate('YYYY-MM-DD hh:mm:ss') // without an explicit "new " prefix on construction,
// which Date() would require

> d.getMonth()
9
```

See Also

- https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Date

Timestamp data type

> This is not the normal Date datatype. This is a special type for internal MongoDB use.

BSON includes a timestamp data type with special semantics in MongoDB.

Timestamps are stored as 64 bit values which, on a single `mongod`, are guaranteed unique. The first 32 bits are a time_t value (seconds since the UTC epoch). The second 32 bits are an incrementing ordinal for operations within a given second.

MongoDB uses the Timestamp datatype as "OpTimes" in the replication oplog's `ts` field.

Timestamps have special semantics when null. If null, and the timestamp is one of the first two fields of the object, the timestamp will automatically be converted to a unique value. (It must be one of the first two top level fields of the document for performance reasons; the entire document is not scanned for timestamps.)

An example from the mongo shell follows (example works with shells v1.7.5 and higher).

```
> // not one of the first 2 fields
> db.foo.insert( { x : 1, y : new Timestamp() } )
> db.foo.find()
{ "_id" : ObjectId("4d1d4ce78b1a04eeb294c098"), "x" : 1, "y" : { "t" : 0, "i" : 0 } }

> // in first 2 fields, auto fill of value works
> db.foo.drop()
> db.foo.insert( { y : new Timestamp(), x : 3 } )
> // the shell displays timestamps as { t : ..., i : ... } where t is the time
> // component and i is the ordinal component
> db.foo.find()
{ "_id" : ObjectId("4d1d4cfd8b1a04eeb294c099"), "y" : { "t" : 1293765885000, "i" : 1 }, "x" : 3 }

> db.foo.drop()
> for( var i = 0; i < 10; i++ ) db.foo.insert({y:new Timestamp(), x : i})
> db.foo.find()
{ "_id" : ObjectId("4d1d4d178b1a04eeb294c09a"), "y" : { "t" : 1293765911000, "i" : 1 }, "x" : 0 }
{ "_id" : ObjectId("4d1d4d178b1a04eeb294c09b"), "y" : { "t" : 1293765911000, "i" : 2 }, "x" : 1 }
{ "_id" : ObjectId("4d1d4d178b1a04eeb294c09c"), "y" : { "t" : 1293765911000, "i" : 3 }, "x" : 2 }
{ "_id" : ObjectId("4d1d4d178b1a04eeb294c09d"), "y" : { "t" : 1293765911000, "i" : 4 }, "x" : 3 }
{ "_id" : ObjectId("4d1d4d178b1a04eeb294c09e"), "y" : { "t" : 1293765911000, "i" : 5 }, "x" : 4 }
{ "_id" : ObjectId("4d1d4d178b1a04eeb294c09f"), "y" : { "t" : 1293765911000, "i" : 6 }, "x" : 5 }
{ "_id" : ObjectId("4d1d4d178b1a04eeb294c0a0"), "y" : { "t" : 1293765911000, "i" : 7 }, "x" : 6 }
{ "_id" : ObjectId("4d1d4d178b1a04eeb294c0a1"), "y" : { "t" : 1293765911000, "i" : 8 }, "x" : 7 }
{ "_id" : ObjectId("4d1d4d178b1a04eeb294c0a2"), "y" : { "t" : 1293765911000, "i" : 9 }, "x" : 8 }
{ "_id" : ObjectId("4d1d4d178b1a04eeb294c0a3"), "y" : { "t" : 1293765911000, "i" : 10 }, "x" : 9 }
>
```

From MongoDB 2.1.0+, timestamps are displayed in the shell with a wrapper, as Timestamp(1293765911000, 1).

Internationalized Strings

MongoDB supports UTF-8 for strings in stored objects and queries. (Specifically, BSON strings are UTF-8.)

Generally, drivers for each programming language convert from the language's string format of choice to UTF-8 when serializing and deserializing BSON. For example, the Java driver converts Java Unicode strings to UTF-8 on serialization.

In most cases this means you can effectively store most international characters in MongoDB strings. A few notes:

- MongoDB regex queries support UTF-8 in the regex string.
- Currently, sort() on a string uses strcmp: sort order will be reasonable but not fully international correct. Future versions of MongoDB may support full UTF-8 sort ordering.

Object IDs

Documents in MongoDB required a key, `_id`, which uniquely identifies them.

The _id Field

Almost every MongoDB document has an _id field as its first attribute (there are a few exceptions for system collections and capped collections). The _id value can be of any type with type ObjectId being the most common. _id must be unique for each document in a collection. In most cases collections automatically have an _id index which includes a unique key constraint that enforces this.

If a user tries to insert a document without providing an `_id` field, the database will automatically generate an _object id_ and store it the _id field.

The _id value may be of any type, other than arrays, so long as it is a unique. If your document has a natural primary key that is immutable we recommend you use that in _id instead of the automatically generated ids. Arrays are not allowed _ids because they are Multikeys.

The BSON ObjectId Datatype

Although _id values can be of any type, a special BSON datatype is provided for object ids. This type is a 12-byte binary value designed to have a reasonably high probability of being unique when allocated. All of the officially-supported MongoDB drivers use this type by default for _id values. Also, the Mongo database itself uses this type when assigning _id values on inserts where no _id value is present.

In the MongoDB shell, ObjectId() may be used to create ObjectIds. `ObjectId(`*`string`*`)` creates an object ID from the specified hex string.

```
> x={ name: "joe" }
{ name : "joe" }
> db.people.save(x)
{ name : "joe" , _id : ObjectId( "47cc67093475061e3d95369d" ) }
> x
{ name : "joe" , _id : ObjectId( "47cc67093475061e3d95369d" ) }
> db.people.findOne( { _id: ObjectId( "47cc67093475061e3d95369d" ) } )
{ _id : ObjectId( "47cc67093475061e3d95369d" ) , name : "joe" }
> db.people.findOne( { _id: new ObjectId( "47cc67093475061e3d95369d" ) } )
{ _id : ObjectId( "47cc67093475061e3d95369d" ) , name : "joe" }
```

BSON ObjectID Specification

A BSON ObjectID is a 12-byte value consisting of a 4-byte timestamp (seconds since epoch), a 3-byte machine id, a 2-byte process id, and a 3-byte counter. Note that the timestamp and counter fields must be stored big endian unlike the rest of BSON. This is because they are compared byte-by-byte and we want to ensure a mostly increasing order. The format:

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr>
<tr><td colspan="4">time</td><td colspan="3">machine</td><td colspan="2">pid</td><td colspan="3">inc</td></tr>
</table>

- TimeStamp. This is a unix style timestamp. It is a signed int representing the number of seconds before or after January 1st 1970 (UTC).
- Machine. This is the first three bytes of the (md5) hash of the machine host name, or of the mac/network address, or the virtual machine id.
- Pid. This is 2 bytes of the process id (or thread id) of the process generating the object id.
- Increment. This is an ever incrementing value, or a random number if a counter can't be used in the language/runtime.

BSON ObjectIds can be any 12 byte binary string that is unique; however, the server itself and almost all drivers use the format above.

Sequence Numbers

Traditional databases often use increasing sequence numbers for primary keys. In MongoDB, the preferred approach is to use Object IDs instead. The concept is that in a very large cluster of machines, it is easier to create an object ID than have global, uniformly increasing sequence numbers.

However, sometimes you may want a sequence number. You can do this by creating "counter" documents and using the findAndModify Command.

```
function counter(name) {
    var ret = db.counters.findAndModify({query:{_id:name}, update:{$inc : {next:1}}, "new":true,
upsert:true});
    // ret == { "_id" : "users", "next" : 1 }
return ret.next;
}

db.users.insert({_id:counter("users"), name:"Sarah C."}) // _id : 1
db.users.insert({_id:counter("users"), name:"Bob D."}) // _id : 2
//repeat
```

See also "Insert if Not Present" section of the Atomic Operations page for another method.

UUIDs

The _id field can be of any type; however, it must be unique. Thus you can use UUIDs in the _id field instead of BSON ObjectIds (BSON ObjectIds are slightly smaller; they need not be worldwide unique, just unique for a single db cluster). When using UUIDs, your application must generate the UUID itself. Ideally the UUID is then stored in the [DOCS:BSON] type for efficiency – however you can also insert it as a hex string if you know space and speed will not be an issue for the use case.

See Also

- Optimizing Object IDs

ObjectId

Redirection Notice
This page should redirect to Object IDs.

Database References

- Simple Direct/Manual Linking
- DBRef
- See Also

As MongoDB is non-relational (no joins), references ("foreign keys") between documents are generally resolved client-side by additional queries to the server ("linking").

These links are always resolved client-side. Doing this directly/manually can be quite easy and is recommended.

There is also a DBRef mechanism which many drivers support which abstracts the linking concept somewhat. **Generally though direct/manual links are the recommended approach.**

Embedding objects is an alternative to linking and is often more appropriate and superior. See the schema design page and also the schema design presentation videos.

Simple Direct/Manual Linking

Generally, manually coded link resolution works just fine and is easy. We simply store the value that is present in _id in some other document in the database and then query it later. For example:

```
> // grab a random blog post:
> p = db.postings.findOne();
{
 "_id" : ObjectId("4b866f08234ae01d21d89604"),
 "author" : "jim",
 "title" : "Brewing Methods"
}
> // get more info on author of post p. this is the "linking" step.
> a = db.users.findOne( { _id : p.author } )
{ "_id" : "jim", "email" : "jim@gmail.com" }

> // inverse: given an author, find all blog posts for the author
> db.postings.find( {author : a._id } )
```

DBRef

DBRef is a more formal specification for creating references between documents. DBRefs (generally) include a collection name as well as an object id. Most developers only use DBRefs if the collection can change from one document to the next. If your referenced collection will always be the same, the manual references outlined above are more efficient.

A DBRef is a reference from one document (object) to another within a database. A database reference is a standard embedded (JSON/BSON) object: we are defining a convention, not a special type. By having a standard way to represent, drivers and data frameworks can add helper methods which manipulate the references in standard ways.

DBRef's have the advantage of allowing optional automatic **client-side** dereferencing with some drivers. In many cases, you can just get away with storing the _id as a reference then dereferencing manually as detailed in the "Simple Manual References" section above.

Syntax for a DBRef reference value is

```
{ $ref : <collname>, $id : <idvalue>[, $db : <dbname>] }
```

where `<collname>` is the collection name referenced (without the database name), and `<idvalue>` is the value of the _id field for the object referenced. `$db` is optional (currently unsupported by many of the drivers) and allows for references to documents in other databases (specified by `<dbname>`).

> The ordering for DBRefs does matter, fields must be in the order specified above.

The old BSON DBRef datatype is deprecated.

DBRef in Different Languages / Drivers

C#

Use the DBRef class. It takes the collection name and _id as parameters to the constructor. Then you can use the FollowReference method on the Database class to get the referenced document.

C++

The C++ driver does not yet provide a facility for automatically traversing DBRefs. However one can do it manually of course.

Java

Java supports DB references using the DBRef class.

Javascript (mongo shell)

Example:

```
> x = { name : 'Biology' }
{ "name" : "Biology" }
> db.courses.save(x)
> x
{ "name" : "Biology", "_id" : ObjectId("4b0552b0f0da7d1eb6f126a1") }
> stu = { name : 'Joe', classes : [ new DBRef('courses', x._id) ] }
// or we could write:
// stu = { name : 'Joe', classes : [ {$ref:'courses',$id:x._id} ] }
> db.students.save(stu)
> stu
{
        "name" : "Joe",
        "classes" : [
                {
                        "$ref" : "courses",
                        "$id" : ObjectId("4b0552b0f0da7d1eb6f126a1")
                }
        ],
        "_id" : ObjectId("4b0552e4f0da7d1eb6f126a2")
}
> stu.classes[0]
{ "$ref" : "courses", "$id" : ObjectId("4b0552b0f0da7d1eb6f126a1") }
> stu.classes[0].fetch()
{ "_id" : ObjectId("4b0552b0f0da7d1eb6f126a1"), "name" : "Biology" }
>
```

Perl

The Perl driver does not provide a facility for automatically traversing DBRefs, but there is a CPAN package for it: MongoDBx::AutoDeref. You can also traverse them manually, of course.

PHP

PHP supports DB references with the MongoDBRef class, as well as creation and deferencing methods at the database (MongoDB::createDBRef and MongoDB::getDBRef) and collection (MongoCollection::createDBRef and MongoCollection::getDBRef) levels.

Python

To create a DB reference in python use the bson.dbref.DBRef class. You can also use the dereference method on Database instances to make dereferencing easier.

Python also supports auto-ref and auto-deref - check out the auto_reference example.

Ruby

Ruby also supports DB references using the DBRef class and a dereference method on DB instances. For example:

```
@db   = Connection.new.db("blog")
@user = @db["users"].save({:name => "Smith"})
@post = @db["posts"].save({:title => "Hello World", :user_id => @user.id})
@ref  = DBRef.new("users", @post.user_id)
assert_equal @user, @db.dereference(@ref)
```

See Also

- Schema Design

GridFS

GridFS is a specification for storing large files in MongoDB. All of the mongodb.org supported drivers implement the GridFS spec.

- Rationale
- Implementation
- Language Support
- Command Line Tools
- See also

Rationale

The database supports native storage of binary data within BSON objects. However, BSON objects in MongoDB are limited in size (4MB older versions, 16MB in v1.7/1.8, higher limits in the future). The GridFS spec provides a mechanism for transparently dividing a large file among multiple documents. This allows us to efficiently store large objects, and in the case of especially large files, such as videos, permits range operations (e.g., fetching only the first N bytes of a file).

Implementation

To facilitate this, a standard is specified for the chunking of files. Each file has a metadata object in a files collection, and one or more chunk objects in a chunks collection. Details of how this is stored can be found in the GridFS Specification; however, you do not really need to read that, instead, just look at the GridFS API in each language's client driver or mongofiles tool.

Language Support

Most drivers include GridFS implementations; for languages not listed below, check the driver's API documentation. (If a language does not include support, see the GridFS specification -- implementing a handler is usually quite easy.)

Command Line Tools

Command line tools are available to write and read GridFS files from and to the local filesystem.

See also

- C++
- A PHP GridFS Blog Article
- Choosing a Shard Key contains a section describing how best to shard GridFS.

When to use GridFS

This page is under construction

When to use GridFS

- Lots of files. GridFS tends to handle large numbers (many thousands) of files better than many file systems.
- User uploaded files. When users upload files you tend to have a lot of files, and want them replicated and backed up. GridFS is a perfect place to store these as then you can manage them the same way you manage your data. You can also query by user, upload date, etc... directly in the file store, without a layer of indirection
- Files that often change. If you have certain files that change a lot - it makes sense to store them in GridFS so you can modify them in one place and all clients will get the updates. Also can be better than storing in source tree so you don't have to deploy app to update files.

When not to use GridFS

- Few small static files. If you just have a few small files for a website (js,css,images) its probably easier just to use the file system.
- Note that if you need to update a binary object atomically, and the object is under the document size limit for your version of MongoDB (16MB for 1.8), then you might consider storing the object manually within a single document. This can be accomplished using the BSON bindata type. Check your driver's docs for details on using this type.

Indexes

Indexes enhance query performance, often dramatically. It's important to think about the kinds of queries your application will need so that you can define relevant indexes. Once that's done, actually creating the indexes in MongoDB is relatively easy.

Indexes in MongoDB are conceptually similar to those in RDBMSes like MySQL. You will want an index in MongoDB in the same sort of situations where you would have wanted an index in MySQL.

Basics

An index is a data structure that collects information about the values of the specified fields in the documents of a collection. This data structure is used by Mongo's query optimizer to quickly sort through and order the documents in a collection. Formally speaking, these indexes are implemented as "B-Tree" indexes.

In the shell, you can create an index by calling the `ensureIndex()` function, and providing a document that specifies one or more keys to index. Referring back to our *examples* database from Mongo Usage Basics, we can index on the 'j' field as follows:

```
db.things.ensureIndex({j:1});
```

The `ensureIndex()` function only creates the index if it does not exist.

Once a collection is indexed on a key, random access on query expressions which match the specified key are fast. Without the index, MongoDB has to go through each document checking the value of specified key in the query:

```
db.things.find({j:2});  // fast - uses index
db.things.find({x:3});  // slow - has to check all because 'x' isn't indexed
```

You can run

```
db.things.getIndexes()
```

in the shell to see the existing indexes on the collection. Run

```
db.system.indexes.find()
```

to see all indexes for the database.

`ensureIndex` creates the index if it does not exist. A standard index build will block all other database operations. If your collection is large, the build may take many minutes or hours to complete - if you must build an index on a live MongoDB instance, we suggest that you build it in the background using the `background : true` option. This will ensure that your database remains responsive even while the index is being built. Note, however, that background indexing may still affect performance, particularly if your collection is large.

> If you use replication, background index builds will block operations on the secondaries. To build new indices on a live replica set, it is recommended you follow the steps described here.

In many cases, not having an index at all can impact performance almost as much as the index build itself. If this is the case, we recommend the application code check for the index at startup using the chosen mongodb driver's getIndex() function and terminate if the index cannot be found. A separate indexing script can then be explicitly invoked when safe to do so.

The _id Index

For all collections except capped collections, an index is automatically created for the `_id` field. This index is special and cannot be deleted. The _id index enforces uniqueness for its keys (except for some situations with sharding).

_id values are invariant.

Indexing on Embedded Fields("Dot Notation")

With MongoDB you can even index on a key inside of an embedded document. Reaching into sub-documents is referred to as Dot Notation. For example:

```
db.things.ensureIndex({"address.city": 1})
```

Compound Keys

In addition to single-key basic indexes, MongoDB also supports multi-key "compound" indexes. Just like basic indexes, you use the `ensureIndex()` function in the shell to create the index, but instead of specifying only a single key, you can specify several :

```
db.things.ensureIndex({j:1, name:-1});
```

When creating an index, the number associated with a key specifies the direction of the index, so it should always be 1 (ascending) or -1 (descending). Direction doesn't matter for single key indexes or for random access retrieval but is important if you are doing sorts or range queries on compound indexes.

If you have a compound index on multiple fields, you can use it to query on the beginning subset of fields. So if you have an index on

```
a,b,c
```

you can use it query on

```
a
```

```
a,b
```

```
a,b,c
```

v1.6+
Now you can also use the compound index to service any combination of equality and range queries from the constitute fields. If the first key of the index is present in the query, that index may be selected by the query optimizer. If the first key is not present in the query, the index will only be used if hinted explicitly. While indexes can be used in many cases where an arbitrary subset of indexed fields are present in the query, as a general rule the optimal indexes for a given query are those in which queried fields precede any non queried fields.

Indexing Array Elements

When a document's stored value for a index key field is an array, MongoDB indexes each element of the array. See the Multikeys page for more information.

Creation Options

The second argument for `ensureIndex` is a document/object representing the options. These options are explained below.

option	values	default
background	true/false	false. see doc page for caveats
dropDups	true/false	false
sparse	true/false	false
unique	true/false	false

v	index version. 0 = pre-v2.0, 1 = smaller/faster (current)	1 in v2.0. Default is used except in unusual situations.

`name` is also an option but need not be specified and will be deprecated in the future. The name of an index is generated by concatenating the names of the indexed fields and their direction (i.e., 1 or -1 for ascending or descending). Index names (including their namespace/database), are limited to 128 characters.

sparse:true

v1.8+

A sparse index can only have one field. SERVER-2193

A "sparse index" is an index that only includes documents with the indexed field.
Any document that is missing the sparsely indexed field will not be stored in the index; the index will therefor be *sparse* because of the missing documents when values are missing.

Sparse indexes, by definition, are not complete (for the collection) and behave differently than complete indexes. When using a "sparse index" for sorting (or in some cases just filtering) some documents in the collection may not be returned. This is because only documents in the index will be returned.

```
> db.people.ensureIndex({title : 1}, {sparse : true})
> db.people.save({name:"Jim"})
> db.people.save({name:"Sarah", title:"Princess"})
> db.people.find()
{ "_id" : ObjectId("4de6abd5da558a49fc5eef29"), "name" : "Jim" }
{ "_id" : ObjectId("4de6abdbda558a49fc5eef2a"), "name" : "Sarah", "title" : "Princess" }
> db.people.find().sort({title:1}) // only 1 doc returned because sparse
{ "_id" : ObjectId("4de6abdbda558a49fc5eef2a"), "name" : "Sarah", "title" : "Princess" }
> db.people.dropIndex({title : 1})
{ "nIndexesWas" : 2, "ok" : 1 }
> db.people.find().sort({title:1}) // no more index, returns all documents
{ "_id" : ObjectId("4de6abd5da558a49fc5eef29"), "name" : "Jim" }
{ "_id" : ObjectId("4de6abdbda558a49fc5eef2a"), "name" : "Sarah", "title" : "Princess" }
```

You can combine sparse with unique to produce a unique constraint that ignores documents with missing fields.

Note that MongoDB's sparse indexes are not block-level indexes. MongoDB sparse indexes can be thought of as dense indexes with a specific filter.

unique:true

MongoDB indexes may optionally impose a unique key constraint, which guarantees that no documents are inserted whose values for the indexed keys match those of an existing document.

Note that if a document is missing the indexed field, the value is treated as null for that document. If the index is unique, two documents cannot both have the null value. If you want to allow missing fields or nulls for your unique indexed field, be sure to also use the sparse option.

Here are some examples using unique index constraints:

```
// everyone's username must be unique:
db.things.ensureIndex({email:1},{unique:true});

// in this variation, it's ok to not have an email address,
// but if you have one, it must be unique:
db.things.ensureIndex({email:1},{unique:true,sparse:true});

// a compound index example. firstname+lastname combination must be unique:
db.things.ensureIndex({firstname: 1, lastname: 1}, {unique: true});
```

For non-sparse indexes (see the sparse option documentation), if a field is not present its value is treated as null for purposes of indexing. This is important to keep in mind when writing an application. For example:

```
db.customers.ensureIndex({firstname:1,lastname:1},{unique:true});
assert( db.customers.count() == 0 );
db.customers.insert({firstname:'jane',lastname:'doe'});   // ok
db.customers.insert({firstname:'jane',lastname:'doe'});   // dup key error
db.customers.insert({firstname:'jane',lastname:'smith'}); // ok
db.customers.insert({firstname:'jane'});                  // ok, treated as
{firstname:'jane',lastname:null}
db.customers.insert({lastname:'smith'});                  // ok
db.customers.insert({firstname:'john',lastname:'smith'}); // ok
db.customers.insert({firstname:'jane'});                  // dup key error
db.customers.insert({email:'sally@abc.com',age:33});      // ok, treated as {firstname:null,lastname:
null}
db.customers.insert({email:'pete@abc.com',age:39});       // dup key error
```

dropDups:true

A unique index cannot be created on a key that has pre-existing duplicate values. If you would like to create the index anyway, keeping the first document the database indexes and deleting all subsequent documents that have duplicate values, add the `dropDups` option.

```
db.things.ensureIndex({firstname : 1}, {unique : true, dropDups : true})
```

dropDups deletes data. A "fat finger" with drop dups could delete almost all data from a collection. Backup before using. Note also that if the field is missing in multiple records, that evaluates to null, and those would then be consider duplicates – in that case using sparse, or not using dropDups, would be very important.

background:true

By default, building an index blocks other database operations. v1.4+ has a background index build option – however this option has significant limitations in a replicated cluster (see doc page).

Dropping Indexes

To delete all indexes on the specified collection:

```
db.collection.dropIndexes();
```

To delete a single index:

```
db.collection.dropIndex({x: 1, y: -1})
```

Running directly as a command without helper:

```
// note: command was "deleteIndexes", not "dropIndexes", before MongoDB v1.3.2
// remove index with key pattern {y:1} from collection foo
db.runCommand({dropIndexes:'foo', index : {y:1}})
// remove all indexes:
db.runCommand({dropIndexes:'foo', index : '*'})
```

ReIndex

The reIndex command will rebuild all indexes for a collection.

```
db.myCollection.reIndex()
```

See here for more documentation: reIndex Command

Performance Notes

Updates

When you update an object, if the object fits in its previous allocation area, only those indexes whose keys have changed are updated. This improves performance. Note that if the object has grown and must move, all index keys must then update, which is slower.

How many indexes?

Indexes make retrieval by a key, including ordered sequential retrieval, very fast. Updates by key are faster too as MongoDB can find the document to update very quickly.

However, keep in mind that each index created adds a certain amount of overhead for inserts and deletes. In addition to writing data to the base collection, keys must then be added to the B-Tree indexes. Thus, indexes are best for collections where the number of reads is much greater than the number of writes. For collections which are write-intensive, indexes, in some cases, may be counterproductive. Most collections are read-intensive, so indexes are a good thing in most situations.

Using `sort()` without an Index

You may use `sort()` to return data in order without an index if the data set to be returned is small (less than 32 megabytes in version 2.0, or less than four megabytes in version 1.8 and earlier). For these cases it is best to use `limit()` and `sort()` together.

Additional Notes

Behaviors

- MongoDB indexes (and string equality tests in general) are case sensitive.
- Index information is kept in the system.indexes collection, run db.system.indexes.find() to see example data.

Using Documents as Keys

Indexed fields may be of any type, including (embedded) documents:

```
db.factories.insert( { name: "xyz", metro: { city: "New York", state: "NY" } } );
db.factories.ensureIndex( { metro : 1 } );
// this query can use the above index:
db.factories.find( { metro: { city: "New York", state: "NY" } } );
// this one too, as {city:"New York"} < {city:"New York",state:"NY"}
db.factories.find( { metro: { $gte : { city: "New York" } } } );

// this query does not match the document because the order of fields is significant
db.factories.find( { metro: { state: "NY" , city: "New York" } } );
```

An alternative to documents as keys is to create a compound index:

Document indexes like these cannot be used to search for queries using their (embedded) parts, like "state" in this example. When you search you must use a prefix, or the whole document, of the embedded document as if they were stored as opaque blobs; that is essentially how they are stored in the index entry. This is even more understandable when you think about how each embedded document could include any number of fields and/or types of data.

```
db.factories.ensureIndex( { "metro.city" : 1, "metro.state" : 1 } );
// these queries can use the above index:
db.factories.find( { "metro.city" : "New York", "metro.state" : "NY" } );
db.factories.find( { "metro.city" : "New York" } );
db.factories.find().sort( { "metro.city" : 1, "metro.state" : 1 } );
db.factories.find().sort( { "metro.city" : 1 } )
```

There are pros and cons to the two approaches. When using the entire (sub-)document as a key, compare order is predefined and is ascending key order in the order the keys occur in the BSON document. With compound indexes reaching in, you can mix ascending and descending keys, and the query optimizer will then be able to use the index for queries on solely the first key(s) in the index too.

Keys Too Large To Index

Index entries have a limitation on their maximum size (the sum of the values), currently 1024 bytes (prior to v2.0 the maximum size was 819 bytes). Documents whose fields have values (key size in index terminology) greater than this size cannot be indexed. You will see log messages similar to:

```
...Btree::insert: key too large to index, skipping...
```

Queries against this index will not return the unindexed documents. You can force a query to use another index, or really no index, using this special index hint:

```
db.myCollection.find({<key>: <value too large to index>}).hint({$natural: 1})
```

This will cause the document to be used for comparison of that field (or fields), rather than the index.

This limitation will eventually be removed (see SERVER-3372).

Presentations

Indexing and Query Optimization - Presentation from MongoSV (December 2011)

More Presentations

- Video introduction to indexing and the query optimizer
- More advanced video and accompanying slides, with many examples and diagrams
- Intermediate level webinar and accompanying slides
- Another set of intermediate level slides

Building indexes with replica sets

- Version 2.1.0 and later
- Version 2.0.x and earlier
    - Building an index one replica set member at a time

Version 2.1.0 and later

Indexes can be built in the foreground or background. Background indexes builds on the primary will result in background index builds on the secondaries.

Index built on primary	Index built on secondary	Index built on recovering member
Foreground	Foreground	Foreground
Background	Background	Foreground

Recovering members (for example, new members who are initial syncing or members in maintenance mode) will always build indexes in the foreground because they cannot handle reads anyway. Generally you want recovering members to catch up as quickly as possible.

Version 2.0.x and earlier

Building an index with a replica set would be straightforward except that index build operations are blocking. The build index operation replicates from primaries to secondaries. This section discusses how to build an index on a live system.

If the collection to index is tiny, simply index it – the time to index would be short enough that the obstruction is not significant. (Be sure to verify the collection is small.)

Note that the background indexing option has a significant limitation and runs in background mode only on the primary. Thus, for now, the following procedure is recommended instead.

Building an index one replica set member at a time

This is a manual process where we take the members of a set offline one at a time to build the index. Suppose we wish to create on index on the `name` field in our `logins` collection, and that we have a three member replica set using hostnames a, b, and c.

- First check the entire replica set is healthy by running `rs.status()` in the mongo shell.
- Next, build the necessary index on one secondary at a time.

- For each secondary:
    - Shutdown the `mongod` process on that member.
    - Verify the rest of the set is still healthy by running `rs.status()` while connected to one of the other members. Also, before continuing, verify the members have sufficient capacity while this member is down.
    - Restart the local `mongod` in "maintenance mode" by starting it without -~~replSet and on a different port number (~~-port so that

clients to not try to connect to it.
- Connect from the shell to the `mongod` mentioned above.
- Build the index:

```
db.logins.ensureIndex({name: 1})
```

- After completion, shut down the `mongod` process.
- Restart the `mongod` with its normal parameters which include the `--replSet` parameter.
- Use `rs.status()` to check that the cluster is healthy all around.
- Repeat for the next secondary.

Now that all the secondaries have the index you can choose to either build the index on the primary in background mode, which will take longer and put more load on primary, or follow the steps below to demote the primary so you can build in the foreground (the default option) more quickly and without affecting the primary.

Once you've built the index on each secondary, you'll have to step down the primary node and do the same. Connect to the primary and then run `rs.stepDown()`. Once the primary has safely stepped down, check to see that a new secondary has been elected primary.

Note that when a member that was primary goes offline on its shutdown above, there will be a short window, until another member takes over, where there is no primary for the entire cluster. Therefore, you may have to momentarily put your application into a maintenance mode before you step down the primary so that the stepping down does not result in application exceptions, poor user experience, etc.

With the primary in secondary mode, you can now follow the steps above to create the necessary index.

Index Versions

MongoDB v2.0 uses a new data structure for its indexes. The new {`v:1`} indexes are, on average, 25% smaller than the old {`v:0`} indexes. This usually results in significantly increased performance. In addition, signed (pre 1970) dates are supported in {`v:1`} indexes.

Backward-compatibility with {v:0} indexes is maintained. Thus, you can upgrade to MongoDB v2.0 seamlessly. However, to get the benefits of the new index format, you'll need to reindex.

Converting an Existing Index to {`v:1`} format

All operations that create a new index will result in a {`v:1`} index by default.

- Reindexing will result in a replacement of any {`v:0`} index of with a {`v:1`} type.
  - Due to SERVER-3866, reindexing on a secondary does not work in <=v2.0.0. **Do not reindex on a secondary as the drop phase of reindex is done and then the rest of the work aborts**. See jira for workaround.
- A compact command invocation will convert all indexes for the given collection to {`v:1`} type.

In v2.0.1, the following operations will convert {`v:0`} indexes to {`v:1`} type:

- The repair database command.
- Creation of a new replica set secondary from scratch (initial sync).

Rolling Back to MongoDB < v2.0

While MongoDB v2.0 supports the old index format, **old versions will not support the new format**. If you need to roll back to an old version, the server will run, but queries and other operations involving the newer indexes will log and return an error. Thus, you'll need to recreate any new index you'd like to use on an old server.

Versions >= 1.8.3 are aware of the index version field, but version <= 1.8.2 are not. So if you rollback a `[v:1}` index to 1.8.2 and re-index it its version will still be marked {`v: 1`} although it actual version is {`v: 0`}. Then if you upgrade again to 2.0 this index won't work even though they are marked as {`v: 1`} in `db.system.indexes`. So if you have to rollback to a version <= 1.8.2, you must delete the index then create it again (instead of simply re-indexing).

Building a {`v:0`} Index

You can still create a {`v:0`} index with MongoDB v2.0. To do so, add the option {`v:0`} in the index creation command. For example in the mongo shell:

```
> // defaults to a v:1 index
> db.foo.ensureIndex({name:1})
> db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "mydb.foo", "name" : "_id_" }
{ "v" : 1, "key" : { "name" : 1 }, "ns" : "mydb.foo", "name" : "name_1" }
> db.foo.dropIndex({name:1})
{ "nIndexesWas" : 2, "ok" : 1 }

> // create a v:0 index
> db.foo.ensureIndex({name:1},{v:0})
> db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "mydb.foo", "name" : "_id_" }
{ "v" : 0, "key" : { "name" : 1 }, "ns" : "mydb.foo", "name" : "name_1" }
```

Geospatial Indexing

v1.4+

MongoDB supports two-dimensional geospatial indexes. It is designed with location-based queries in mind, such as "find me the closest N items to my location." It can also efficiently filter on additional criteria, such as "find me the closest N museums to my location."

In order to use the index, you need to have a field in your object that an array where the first 2 elements are x,y coordinates (or y,x - just be consistent; it might be advisable to use order-preserving dictionaries/hashes in your client code, to ensure consistency).

> To make sure ordering is preserved from all languages use a 2 element array
>
> ```
> [ x, y ]
> ```

Some examples:

```
{ loc : [ 50 , 30 ] } //SUGGESTED OPTION
{ loc : { x : 50 , y : 30 } }
{ loc : { foo : 50 , y : 30 } }
{ loc : { lon : 40.739037, lat: 73.992964 } }
```

Creating the Index

```
db.places.ensureIndex( { loc : "2d" } )
```

By default, the index assumes you are indexing longitude/latitude and is thus configured for a [-180..180) value range.

> The index space bounds are inclusive of the lower bound and exclusive of the upper bound.

If you are indexing something else, you can specify some options:

```
db.places.ensureIndex( { loc : "2d" } , { min : -500 , max : 500 } )
```

that will scale the index to store values between -500 and 500. Bounded geospatial searches are currently limited to rectangular and circular areas with no "wrapping" at the outer boundaries. You cannot insert values outside the boundary interval `[min, max)`. For example, using the code above, the point `(-500, 500)` could not be inserted and would raise an error (the point `(-500, 499)`, however, would be fine).

> Pre-1.9 releases of mongo do not allow the insertion of points exactly at the boundaries.

```
db.places.ensureIndex( { loc : "2d" } , { bits : 26 } )
```

The bits parameter sets the precision of the 2D geo-hash values, the smallest "buckets" in which locations are stored. By default, precision is set to 26 bits which is equivalent to approximately 1 foot given `(longitude, latitude)` location values and default `(-180, 180)` bounds. To index other spaces which may have very large bounds, it may be useful to increase the number of bits up to the maximum of 32.

> You may only have 1 geospatial index per collection, for now. While MongoDB may allow to create multiple indexes, this behavior is unsupported. Because MongoDB can only use one index to support a single query, in most cases, having multiple geo indexes will produce undesirable behavior.

Implicit array expansion syntax is only supported in v1.9+, where `"foo.bar" : "2d"` may reference a nested field like so:

```
{ foo : [ { bar : [ ... ] } ] }
```

This restriction holds even if there are not multiple locations per document and the array size is 1. In older versions you will need to embed the nested location without a parent array :

```
{ foo : { bar : [ ... ] } }
```

Querying

The index can be used for exact matches:

```
db.places.find( { loc : [50,50] } )
```

Of course, that is not very interesting. More important is a query to find points near another point, but not necessarily matching exactly:

```
db.places.find( { loc : { $near : [50,50] } } )
```

The above query finds the closest points to (50,50) and returns them sorted by distance (there is no need for an additional sort parameter). Use limit() to specify a maximum number of points to return (a default limit of 100 applies if unspecified):

```
db.places.find( { loc : { $near : [50,50] } } ).limit(20)
```

You can also use $near with a maximum distance

```
db.places.find( { loc : { $near : [50,50] , $maxDistance : 5 } } ).limit(20)
```

> Prior to v1.9.1, geospatial indexes can be used for **exact** lookups only when there is no other criteria specified in the query and locations are specified in arrays. For any type of `$near`, `$within`, or `geoNear` query, this restriction does not apply and any additional search criteria can be used.

All distances in geospatial queries are specified in the same units as the document coordinate system (aside from spherical queries, discussed below). For example, if your indexed region is of size `[300, 300)`, representing a 300 x 300 meter field, and you have documents at locations

`(10, 20)` and `(10, 30)`, representing objects at points in meters (x, y), you could query for points `$near : [10, 20], $maxDistance : 10`. The distance unit is the same as in your coordinate system, and so this query looks for points up to 10 meters away.

> When using longitude and latitude, which are **angular** measures, distance is effectively specified in approximate units of "degrees," which vary by position on the globe but can very roughly be converted to distance using 69 miles per degree latitude or longitude. The maximum error in northern or southernmost populated regions is ~2x longitudinally - for many purposes this is acceptable. Spherical queries (below) take the curvature of the earth into account.

Compound Indexes

MongoDB geospatial indexes optionally support specification of secondary key values. If you are commonly going to be querying on both a location and other attributes at the same time, add the other attributes to the index. The other attributes are annotated within the index to make filtering faster. For example:

```
db.places.ensureIndex( { location : "2d" , category : 1 } );
db.places.find( { location : { $near : [50,50] }, category : 'coffee' } );
```

> Limits in geospatial queries are always applied to the geospatial component first - this can cause unexpected results when also re-sorting results by additional criteria, pending resolution of SERVER-4247.

geoNear Command

While the find() syntax above is typically preferred, MongoDB also has a geoNear command which performs a similar function. The geoNear command has the added benefit of returning the distance of each item from the specified point in the results, as well as some diagnostics for troubleshooting.

Valid options are: "near", "num", "maxDistance", "distanceMultiplier" and "query".

```
> db.runCommand( { geoNear : "places" , near : [50,50], num : 10 } );
> db.runCommand({geoNear:"asdf", near:[50,50]})
{
        "ns" : "test.places",
        "near" : "1100110000001111110000001111110000001111110000001111",
        "results" : [
                {
                        "dis" : 69.29646421910687,
                        "obj" : {
                                "_id" : ObjectId("4b8bd6b93b83c574d8760280"),
                                "y" : [
                                        1,
                                        1
                                ],
                                "category" : "Coffee"
                        }
                },
                {
                        "dis" : 69.29646421910687,
                        "obj" : {
                                "_id" : ObjectId("4b8bd6b03b83c574d876027f"),
                                "y" : [
                                        1,
                                        1
                                ]
                        }
                }
        ],
        "stats" : {
                "time" : 0,
                "btreelocs" : 1,
                "btreelocs" : 1,
                "nscanned" : 2,
                "nscanned" : 2,
                "objectsLoaded" : 2,
                "objectsLoaded" : 2,
                "avgDistance" : 69.29646421910687
        },
        "ok" : 1
}
```

The above command will return the 10 closest items to (50,50). (The `loc` field is automatically determined by checking for a 2d index on the collection.)

If you want to add an additional filter, you can do so:

```
> db.runCommand( { geoNear : "places" , near : [ 50 , 50 ], num : 10,
... query : { type : "museum" } } );
```

`query` can be any regular mongo query.

Bounds Queries

`$within` can be used instead of `$near` to find items within a shape. Results are **not** sorted by distance, which may result in faster queries when this sorting is not required. Shapes of type `$box` (rectangles), `$center` (circles), and `$polygon` (concave and convex polygons) are supported. All bounds queries implicitly include the border of the shape as part of the boundary, though due to floating-point inaccuracy this can't strictly be relied upon.

To query for all points within a rectangle, you must specify the lower-left and upper-right corners:

```
> box = [[40.73083, -73.99756], [40.741404,  -73.988135]]
> db.places.find({"loc" : {"$within" : {"$box" : box}}})
```

A circle is specified by a center point and radius:

```
> center = [50, 50]
> radius = 10
> db.places.find({"loc" : {"$within" : {"$center" : [center, radius]}}})
```

A polygon is specified by an array or object of points, where each point may be specified by either an array or an object. The last point in the polygon is implicitly connected to the first point in the polygon.

```
> polygonA = [ [ 10, 20 ], [ 10, 40 ], [ 30, 40 ], [ 30, 20 ] ]
> polygonB = { a : { x : 10, y : 20 }, b : { x : 15, y : 25 }, c : { x : 20, y : 20 } }
> db.places.find({ "loc" : { "$within" : { "$polygon" : polygonA } } })
> db.places.find({ "loc" : { "$within" : { "$polygon" : polygonB } } })
```

Polygon searches are strictly limited to looking for **points** inside polygons, polygon shapes in documents can't currently be indexed in MongoDB.

Polygon searches are supported in versions >= 1.9

The Earth is Round but Maps are Flat

The current implementation assumes an idealized model of a flat earth, meaning that an arcdegree of latitude (y) and longitude (x) represent the same distance everywhere. This is only true at the equator where they are both about equal to 69 miles or 111km. However, at the 10gen offices at { x : -74 , y : 40.74 } one arcdegree of longitude is about 52 miles or 83 km (latitude is unchanged). This means that something 1 mile to the north would seem closer than something 1 mile to the east.

New Spherical Model

v1.8+.

Spherical distances can be used by adding "Sphere" to the name of the query. For example, use `$nearSphere` or `$centerSphere` (`$boxSphere` and `$polygonSphere` don't make as much sense and so aren't supported). If you use the geoNear command to get distance along with the results, you just need to add `spherical:true` to the list of options.

There are a few caveats that you must be aware of when using spherical distances. The biggest is:

1. The code assumes that you are using decimal degrees in **(longitude, latitude)** order. This is the same order used for the GeoJSON spec . Using (latitude, longitude) will result in very incorrect results, but is often the ordering used elsewhere, so it is good to double-check. The names you assign to a location object (if using an object and not an array) are **completely ignored**, only the ordering is detected. A few examples:

   ```
   /* assuming longitude is 13, latitude is -50 */
   [13, -50] // ok
   { x : 13, y : -50 } // ok
   { lon : 13, lat : -50 } // ok

   /* wrong, will make lat = longitude and lon = latitude */
   { lat : -50, lon : 13 }
   ```

   As above, the use of order-preserving dictionaries is required for consistent results.

Also:

1. All distances use radians. This allows you to easily multiply by the radius of the earth (about 6371 km or 3959 miles) to get the distance in your choice of units. Conversely, divide by the radius of the earth when doing queries.
2. We don't currently handle wrapping at the poles or at the transition from -180° to +180° longitude, however we detect when a search would wrap and raise an error.
3. While the default Earth-like bounds are `[-180, 180)`, valid values for latitude are between -90° and 90°.

Spherical Example

Below is a simple example of a spherical distance query, demonstrating how to convert a specified range in kilometers to a maxDistance in radians as well as converting the returned distance results from radians back to kilometers. The same conversion of kilometer to radian distance bounds is required when performing bounded $nearSphere and $centerSphere queries.

```
> db.points.insert({ pos : { lon : 30, lat : 30 } })
> db.points.insert({ pos : { lon : -10, lat : -20 } })
> db.points.ensureIndex({ pos : "2d" })
>
> var earthRadius = 6378 // km
> var range = 3000 // km
>
> distances = db.runCommand({ geoNear : "points", near : [0, 0], spherical : true, maxDistance : range
/ earthRadius /* to radians */ }).results
[
 {
  "dis" : 0.3886630122897946,
  "obj" : {
   "_id" : ObjectId("4d9123026ccc7e2cf22925c4"),
   "pos" : {
    "lon" : -10,
    "lat" : -20
   }
  }
 }
]
> pointDistance = distances[0].dis * earthRadius // back to km
2478.89269238431
```

Multi-location Documents

v.1.9+

MongoDB now also supports indexing documents by multiple locations. These locations can be specified in arrays of sub-objects, for example:

```
> db.places.insert({ addresses : [ { name : "Home", loc : [55.5, 42.3] }, { name : "Work", loc :
[32.3, 44.2] } ] })
> db.places.ensureIndex({ "addresses.loc" : "2d" })
```

Multiple locations may also be specified in a single field:

```
> db.places.insert({ lastSeenAt : [ { x : 45.3, y : 32.2 }, [54.2, 32.3], { lon : 44.2, lat : 38.2 } ]
})
> db.places.ensureIndex({ "lastSeenAt" : "2d" })
```

By default, when performing `geoNear` or `$near`-type queries on collections containing multi-location documents, the same document may be returned multiple times, since `$near` queries return ordered results by distance. Queries using the `$within` operator by default do not return duplicate documents.

v2.0

In v2.0, this default can be overridden by the use of a `$uniqueDocs` parameter for `geoNear` and `$within` queries, like so:

```
> db.runCommand( { geoNear : "places" , near : [50,50], num : 10, uniqueDocs : false } )
```

```
> db.places.find( { loc : { $within : { $center : [[0.5, 0.5], 20], $uniqueDocs : true } } } )
```

Currently it is not possible to specify $uniqueDocs for $near queries

Whether or not uniqueDocs is true, when using a limit the limit is applied (as is normally the case) to the number of results returned (and not to the docs or locations). If running a geoNear query with uniqueDocs : true, the closest location in a document to the center of the search region will always be returned - this is not true for $within queries.

In addition, when using geoNear queries and multi-location documents, often it is useful to return not only distances, but also the location in the document which was used to generate the distance. In v2.0, to return the location alongside the distance in the geoNear results (in the field `loc`), specify `includeLocs : true` in the geoNear query. The location returned will be a copy of the location in the document used.

> If the location was an array, the location returned will be an object with "0" and "1" fields in v2.0.0 and v2.0.1.

```
> db.runCommand({ geoNear : "places", near : [ 0, 0 ], maxDistance : 20, includeLocs : true })
{
 "ns" : "test.places",
 "near" : "1100000000000000000000000000000000000000000000000000000",
 "results" : [
  {
   "dis" : 5.830951894845301,
   "loc" : {
    "x" : 3,
    "y" : 5
   },
   "obj" : {
    "_id" : ObjectId("4e52672c15f59224bdb2544d"),
    "name" : "Final Place",
    "loc" : {
     "x" : 3,
     "y" : 5
    }
   }
  },
  {
   "dis" : 14.142135623730951,
   "loc" : {
    "0" : 10,
    "1" : 10
   },
   "obj" : {
    "_id" : ObjectId("4e5266a915f59224bdb2544b"),
    "name" : "Some Place",
    "loc" : [
     [
      10,
      10
     ],
     [
      50,
      50
     ]
    ]
   }
  },
  {
   "dis" : 14.142135623730951,
   "loc" : {
    "0" : -10,
    "1" : -10
   },
   "obj" : {
    "_id" : ObjectId("4e5266ba15f59224bdb2544c"),
    "name" : "Another Place",
    "loc" : [
     [
      -10,
      -10
     ],
     [
      -50,
      -50
     ]
```

```
    ]
   }
  }
 ],
 "stats" : {
  "time" : 0,
  "btreelocs" : 0,
  "nscanned" : 5,
  "objectsLoaded" : 3,
  "avgDistance" : 11.371741047435734,
  "maxDistance" : 14.142157540259815
 },
```

```
  "ok" : 1
}
```

Sharded Collections

v1.8+. Creating a geospatial index for a sharded collection is supported with some caveats: see http://jira.mongodb.org/browse/SHARDING-83. There are no caveats for using geospatial indexes with unsharded collections in a sharded cluster.

Implementation

The current implementation encodes geographic hash codes atop standard MongoDB B-trees. Results of $near queries are exact. One limitation with this encoding, while fast, is that prefix lookups don't give exact results, especially around bit flip areas. MongoDB solves this by doing a grid-neighbor search after the initial prefix scan to pick up any straggler points. This generally ensures that performance remains very high while providing correct results.

Presentations

- Geospatial Indexing with MongoDB - MongoSF (May 2011)
- Storing and Querying location data with MongoDB - MongoSF (May 2011)
- Community blog posts
  - http://gissolved.blogspot.com/2009/05/populating-mongodb-with-pois.html
  - http://geokoder.com/mongodb-plugin-for-quantum-gis
  - http://www.paolocorti.net/2009/12/06/using-mongodb-to-store-geographic-data/

Geospatial Haystack Indexing

In addition to ordinary 2d geospatial indices, mongodb supports the use of bucket-based geospatial indexes. Called "Haystack indexing", these indices can accelerate small-region type longitude / latitude queries when additional criteria is also required. For example, "find all restaurants within 25 miles with name 'foo'". Haystack indices allow you to tune your bucket size to the distribution of your data, so that in general you search only very small regions of 2d space for a particular kind of document. They are **not** suited for finding the closest documents to a particular location, when the closest documents are far away compared to bucket size.

> ⚠ For now, only a single coordinate field and optional single additional field can be used in a haystack index.

To use haystack indexing, documents with a `(longitude, latitude)` position stored as a sub-document or array are required, with an optional additional field to be indexed. For example:

```
> db.foo.insert({ pos : { long : 34.2, lat : 33.3 }, type : "restaurant" })
> db.foo.insert({ pos : { long : 34.2, lat : 37.3 }, type : "restaurant" })
> db.foo.insert({ pos : { long : 59.1, lat : 87.2 }, type : "office" })
...
> db.foo.ensureIndex({ pos : "geoHaystack", type : 1 }, { bucketSize : 1 })
```

The bucketSize parameter is required, and determines the granularity of the bucket index - our value of 1 above creates an index where keys within 1 unit of longitude or latitude are stored in the same bucket.

The haystack index can only be used by a database command, it is not at present chosen by the query optimizer. As an example of finding all restaurants in a particular area with a given maximum distance of 6 degrees longitude / latitude, with a maximum of 30 results returned (by default, there is a 50 document result limit):

```
> db.runCommand({ geoSearch : "foo", near : [33, 33], maxDistance : 6, search : { type : "restaurant"
}, limit : 30 })
{
 "results" : [
  {
   "_id" : ObjectId("4d948a7db1670124ce358fb7"),
   "pos" : {
    "long" : 34.2,
    "lat" : 33.3
   },
   "type" : "restaurant"
  },
  {
   "_id" : ObjectId("4d948a84b1670124ce358fb8"),
   "pos" : {
    "long" : 34.2,
    "lat" : 37.3
   },
   "type" : "restaurant"
  }
 ],
 "stats" : {
  "time" : 0,
  "btreeMatches" : 2,
  "n" : 2
 },
 "ok" : 1
}
```

Spherical queries are not currently supported by haystack indices.

Indexing as a Background Operation

Prior to 2.1.0, slaves and replica secondaries build indexes in the foreground **even if** `background:true` is specified. The slave/secondary will block queries while the index builds on it. Indexing on the secondaries begins after the index is completely built on the primary.

v1.4+

By default the `ensureIndex()` operation is blocking, and stops other operations on the database from proceeding until completed.

To build an index in the background, add `background:true` to your index options. Examples:

```
> db.things.ensureIndex({x:1}, {background:true});
> db.things.ensureIndex({name:1}, {background:true, unique:true,
... dropDups:true});
```

Prior to 2.1.0, the index build operation is *not* a background build when it replicates to secondaries. This could be a significant issue if you are reading from your secondaries – in this case back up each replica set member one by one – see the doc page for this.

With background mode enabled, other non-indexing operations, including writes, will not be obstructed during index creation. The index is not used for queries until the build is complete. Further calls to ensureIndex() while the background indexing is progressing will fail with a message indicating a background operation is in progress.

Although the operation is 'background' in the sense that other operations may run concurrently, the command will not return to the shell prompt until completely finished. To do other operations at the same time, open a separate mongo shell instance.

Please note that background mode building uses an incremental approach to building the index which is slower than the default foreground mode: time to build the index will be greater. If the resulting index is larger than RAM, background indexing can be *much* slower.

While the index build progresses, it is possible to see that the operation is still in progress with the `db.currentOp()` command (will be shown as an insert to system.indexes). You may also use `db.killOp()` to terminate the index build on the primary.

While the build progresses, the index is visible in system.indexes, but it is not used for queries until building completes.

Notes

- Only one index build at a time, whether foreground or background, is permitted per collection.
- Some administrative operations, such as repairDatabase, are disallowed while a background indexing job is in progress.

Multikeys

MongoDB provides an interesting "multikey" feature that can automatically index arrays of an object's values. A good example is tagging. Suppose you have an article tagged with some category names:

```
$ dbshell
> db.articles.save( { name: "Warm Weather", author: "Steve",
                      tags: ['weather', 'hot', 'record', 'april'] } )

> db.articles.find()
{"name" : "Warm Weather" , "author" : "Steve" ,
 "tags" : ["weather","hot","record","april"] ,
 "_id"  : "497ce4051ca9ca6d3efca323"}
```

We can easily perform a query looking for a particular value in the `tags` array:

```
> db.articles.find( { tags: 'april' } )
{"name" : "Warm Weather" , "author" : "Steve" ,
 "tags" : ["weather","hot","record","april"] ,
 "_id"  : "497ce4051ca9ca6d3efca323"}
```

Further, we can index on the tags array. Creating an index on an array element indexes results in the database **indexing each element of the array**:

```
> db.articles.ensureIndex( { tags : 1 } )
true

> db.articles.find( { tags: 'april' } )
{"name" : "Warm Weather" , "author" : "Steve" ,
 "tags" : ["weather","hot","record","april"] ,
 "_id"  : "497ce4051ca9ca6d3efca323"}

> db.articles.find( { tags: 'april' } ).explain()
{"cursor" : "BtreeCursor tags_1" , "startKey" : {"tags" : "april"} ,
 "endKey" : {"tags" : "april"} , "nscanned" : 1 , "n" : 1 , "millis" : 0 }
```

Incrementally adding and removing keys

You can use $addToSet to add a new key to the array, and $pull to remove one.

```
> db.articles.update({name: "Warm Weather"},{$addToSet:{tags:"northeast"}});
> db.articles.find();
...
> db.articles.update({name: "Warm Weather"},{$pull:{tags:"northeast"}});
```

Embedded object fields in an array

The same technique can be used to find fields within objects embedded in arrays:

```
> // find posts where julie commented
> db.posts.find( { "comments.author" : "julie" } )
{"title" : "How the west was won",
 "comments" : [{"text" : "great!" , "author" : "sam"},
               {"text" : "ok" , "author" : "julie"}],
 "_id" : "497ce79f1ca9ca6d3efca325"}
```

Querying on all values in a given set

By using the $all query option, a set of values may be supplied each of which must be present in a matching object field. For example:

```
> db.articles.find( { tags: { $all: [ 'april', 'record' ] } } )
{"name" : "Warm Weather" , "author" : "Steve" ,
 "tags" : ["weather","hot","record","april"] ,
 "_id"  : "497ce4051ca9ca6d3efca323"}

> db.articles.find( { tags: { $all: [ 'april', 'june' ] } } )
> // no matches
```

Exact Array Matching with an Index

Since indexing an array, which creates the multikey index, only indexes each element in the array it is not possible to do an exact array match using an index on the array field. The index will be used to lookup a subset of the values (currently the first one) and then the document will be inspected for the exact match. This is also true for range queries, but the rules for types are bit more complicated since it follows the matching rules for single values.

```
> db.emails.find()
{ "_id" : ObjectId("4ef8ee415ca7c3e2a4a639a8"), "email_id" : [ 109, 97, 105, 108 ] }
{ "_id" : ObjectId("4ef8ee435ca7c3e2a4a639a9"), "email_id" : [ 109, 101, 64, 98 ] }
> db.emails.find({"email_id" : [ 109, 97, 105, 108 ]})
{ "_id" : ObjectId("4ef8ee415ca7c3e2a4a639a8"), "email_id" : [ 109, 97, 105, 108 ] }
> db.emails.find({"email_id" : [ 109, 97, 105, 108 ]}).explain()
{
        "cursor" : "BtreeCursor email_id_1 multi",
        "nscanned" : 2,
        "nscannedObjects" : 2,
        "n" : 1,
        "millis" : 0,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "isMultiKey" : true,
        "indexOnly" : false,
        "indexBounds" : {
                "email_id" : [
                        [
                                109,
                                109
                        ],
                        [
                                [
                                        109,
                                        97,
                                        105,
                                        108
                                ],
                                [
                                        109,
                                        97,
                                        105,
                                        108
                                ]
                        ]
                ]
        }
}
//Note the nscanned, nscannedObjects and n values.

//Notice the array or arrays and the behavior below
> db.emails.insert({ "_id" : ObjectId("4ef8ee435ca7c3e2a4a639a7"), "email_id" : [[ 109, 101, 64, 98 ]]
})
> db.emails.find()
{ "_id" : ObjectId("4ef8ee415ca7c3e2a4a639a8"), "email_id" : [ 109, 97, 105, 108 ] }
{ "_id" : ObjectId("4ef8ee435ca7c3e2a4a639a9"), "email_id" : [ 109, 101, 64, 98 ] }
{ "_id" : ObjectId("4ef8ee435ca7c3e2a4a639a7"), "email_id" : [ [ 109, 101, 64, 98 ] ] }
> db.emails.find({"email_id" : [ 109, 101, 64, 98 ]})
{ "_id" : ObjectId("4ef8ee435ca7c3e2a4a639a9"), "email_id" : [ 109, 101, 64, 98 ] }
{ "_id" : ObjectId("4ef8ee435ca7c3e2a4a639a7"), "email_id" : [ [ 109, 101, 64, 98 ] ] }
```

Caveats with Parallel Arrays

When using a **compound index**, at most one of indexed values in any document can be an array. So if we have an index on {`a: 1, b: 1`}, the following documents are both fine:

```
{a: [1, 2], b: 1}
{a: 1, b: [1, 2]}
```

This document, however, will fail to be inserted, with an error message "cannot index parallel arrays":

```
{a: [1, 2], b: [1, 2]}
```

The problem with indexing parallel arrays is that each value in the cartesian product of the compound keys would have to be indexed, which can get out of hand very quickly.

See Also

- The Multikeys section of the Full Text Search in Mongo document for information about this feature.

Using Multikeys to Simulate a Large Number of Indexes

One way to work with data that has a high degree of options for queryability is to use the multikey indexing feature where the keys are objects. For example:

```
> x = {
... _id : "abc",
... cost : 33,
... attribs : [
...          { color : 'red' },
...          { shape : 'rect' },
...          { color : 'blue' },
...          { avail : true } ]
... };
> db.foo.insert(x);
> db.foo.ensureIndex({attribs:1});
> db.foo.find( { attribs : {color:'blue'} } ); // uses index
> db.foo.find( { attribs : {avail:false} } );  // uses index
```

In addition to being able to have an unlimited number of attributes types, we can also add new types dynamically.

This is mainly useful for simply attribute lookups; the above pattern is not necessary helpful for sorting or certain other query types.

See Also

Discussion thread MongoDB for a chemical property search engine for a more complex real world example.

Indexing Advice and FAQ

We get a lot of questions about indexing. Here we provide answers to a number of these. There are a couple of points to keep in mind, though. First, indexes in MongoDB work quite similarly to indexes in MySQL, and thus many of the techniques for building efficient indexes in MySQL apply to MongoDB.

Second, and even more importantly, know that advice on indexing can only take you so far. The best indexes for your application should always be based on a number of important factors, including the kinds of queries you expect, the ratio of reads to writes, and even the amount of free memory on your system. This means that the best strategy for designing indexes will always be to profile a variety of index configurations with data sets similar to the ones you'll be running in production, and see which perform best. There's no substitute for good empirical analyses.

Note: if you're brand new to indexing, you may want to read this introductory article first.

* Using Multikeys to Simulate a Large Number of Indexes

Indexing Strategies

Here are some general principles for building smart indexes.

Create indexes to match your queries.

If you only query on a single key, then a single-key index will do. For instance, maybe you're searching for a blog post's slug:

```
db.posts.find({ slug : 'state-of-mongodb-2010' })
```

In this case, a unique index on a single key is best:

```
db.posts.ensureIndex({ slug: 1 }, {unique: true});
```

However, it's common to query on multiple keys and to sort the results. For these situations, compound indexes are best. Here's an example for querying the latest comments with a 'mongodb' tag:

```
db.comments.find({ tags : 'mongodb'}).sort({ created_at : -1 });
```

And here's the proper index:

```
db.comments.ensureIndex({tags : 1, created_at : -1});
```

Note that if we wanted to sort by `created_at` ascending, this index would be less effective.

One index per query.

It's sometimes thought that queries on multiple keys can use multiple indexes; this is not the case with MongoDB. If you have a query that selects on multiple keys, and you want that query to use an index efficiently, then a compound-key index is necessary.

Make sure your indexes can fit in RAM.

The shell provides a command for returning the total index size on a given collection:

```
db.comments.totalIndexSize();
65443
```

If your queries seem sluggish, you should verify that your indexes are small enough to fit in RAM. For instance, if you're running on 4GB RAM and you have 3GB of indexes, then your indexes probably aren't fitting in RAM. You may need to add RAM and/or verify that all the indexes you've created are actually being used.

Be careful about single-key indexes with low selectivity.

Suppose you have a field called 'status' where the possible values are 'new' and 'processeed'. If you add an index on 'status' then you've created a low-selectivity index, meaning that the index isn't going to be very helpful in locating records and might just be be taking up space.

A better strategy, depending on your queries, of course, would be to create a compound index that includes the low-selectivity field. For instance, you could have a compound-key index on 'status' and 'created_at.'

Another option, again depending on your use case, might be to use separate collections, one for each status. As with all the advice here, experimentation and benchmarks will help you choose the best approach.

Use `explain`.

MongoDB includes an `explain` command for determining how your queries are being processed and, in particular, whether they're using an index. `explain` can be used from of the drivers and also from the shell:

```
db.comments.find({ tags : 'mongodb'}).sort({ created_at : -1 }).explain();
```

This will return lots of useful information, including the number of items scanned, the time the query takes to process in milliseconds, which indexes the query optimizer tried, and the index ultimately used.

If you've never used `explain`, now's the time to start.

Understanding `explain`'s output.

There are three main fields to look for when examining the `explain` command's output:

- cursor: the value for cursor can be either `BasicCursor` or `BtreeCursor`. The second of these indicates that the given query is using an index.
- nscanned: he number of documents scanned.
- n: the number of documents returned by the query. You want the value of `n` to be close to the value of `nscanned`. What you want to avoid is doing a collection scan, that is, where every document in the collection is accessed. This is the case when `nscanned` is equal to the number of documents in the collection.
- millis: the number of milliseconds require to complete the query. This value is useful for comparing indexing strategies, indexed vs. non-indexed queries, etc.

Pay attention to the read/write ratio of your application.

This is important because, whenever you add an index, you add overhead to all insert, update, and delete operations on the given collection. If your application is read-heavy, as are most web applications, the additional indexes are usually a good thing. But if your application is write-heavy, then be careful when creating new indexes, since each additional index with impose a small write-performance penalty.

In general, **don't be cavalier about adding indexes**. Indexes should be added to complement your queries. Always have a good reason for adding a new index, and make sure you've benchmarked alternative strategies.

Indexing Properties

Here are a few properties of compound indexes worth keeping in mind (Thanks to Doug Green and Karoly Negyesi for their help on this).

These examples assume a compound index of three fields: a, b, c. So our index creation would look like this:

```
db.foo.ensureIndex({a: 1, b: 1, c: 1})
```

Here's some advice on using an index like this:

> This information is no longer strictly correct in 1.6.0+; compound indexes can now be used to service queries where range or filter fields are used within the compound index, not just fields used from left to right. Please run explain to see how the compound index is used.

1. The sort column must be the last column used in the index.

Good:

- find(a=1).sort(a)
- find(a=1).sort(b)
- find(a=1, b=2).sort(c)

Bad:

- find(a=1).sort(c)
- even though c is the last column used in the index, a is that last column used, so you can only sort on a or b.

2. The range query must also be the last column in an index. This is an axiom of 1 above.

Good:

- find(a=1,b>2)
- find(a>1 and a<10)
- find(a>1 and a<10).sort(a)

Bad:

- find(a>1, b=2)

3. Only use a range query or sort on one column.

Good:

- find(a=1,b=2).sort(c)
- find(a=1,b>2)
- find(a=1,b>2 and b<4)
- find(a=1,b>2).sort(b)

Bad:

- find(a>1,b>2)
- find(a=1,b>2).sort(c)

4. Conserve indexes by re-ordering columns used on equality (non-range) queries.

Imagine you have the following two queries:

- find(a=1,b=1,d=1)
- find(a=1,b=1,c=1,d=1)

A single index defined on a, b, c, and d can be used for both queries.
If, however, you need to sort on the final value, you might need two indexes

5. MongoDB's $ne or $nin operator's aren't efficient with indexes.

- When excluding just a few documents, it's better to retrieve extra rows from MongoDB and do the exclusion on the client side.

FAQ

I've started building an index, and the database has stopped responding. What's going on? What do I do?

Building an index can be an IO-intensive operation, especially you have a large collection. This is true on any database system that supports secondary indexes, including MySQL. If you'll need to build an index on a large collection in the future, you'll probably want to consider building the index in the background, a feature available beginning with 1.3.2. See the docs on background indexing for more info.

As for the long-building index, you only have a few options. You can either wait for the index to finish building or kill the current operation (see `killOp()`). If you choose the latter, the partial index will be deleted.

I'm using $ne or $nin in a query, and while it uses the index, it's still slow. What's happening?

The problem with $ne and $nin is that much of an index will match queries like these. If you need to use $nin, it's often best to make sure that an additional, more selective criterion is part of the query.

Inserting

When we insert data into MongoDB, that data will always be in document-form. Documents are data structure analogous to JSON, Python dictionaries, and Ruby hashes, to take just a few examples. Here, we discuss more about document-orientation and describe how to insert data into MongoDB.

Document-Orientation

Document-oriented databases store "documents" but by document we mean a structured document – the term perhaps coming from the phrase "XML document". However other structured forms of data, such as JSON or even nested dictionaries in various languages, have similar properties.

The documents stored in Mongo DB are JSON-like. JSON is a good way to store object-style data from programs in a manner that is language-independent and standards based.

To be efficient, MongoDB uses a format called BSON which is a binary representation of this data. BSON is faster to scan for specific fields than JSON. Also BSON adds some additional types such as a data data type and a byte-array (bindata) datatype. BSON maps readily to and from JSON and also to various data structures in many programming languages.

Client drivers serialize data to BSON, then transmit the data over the wire to the db. Data is stored on disk in BSON format. Thus, on a retrieval, the database does very little translation to send an object out, allowing high efficiency. The client driver unserialized a received BSON object to its native language format.

JSON

For example the following "document" can be stored in Mongo DB:

```
{ author: 'joe',
  created : new Date('03/28/2009'),
  title : 'Yet another blog post',
  text : 'Here is the text...',
  tags : [ 'example', 'joe' ],
  comments : [ { author: 'jim', comment: 'I disagree' },
               { author: 'nancy', comment: 'Good post' }
  ]
}
```

This document is a blog post, so we can store in a "posts" collection using the shell:

```
> doc = { author : 'joe', created : new Date('03/28/2009'), ... }
> db.posts.insert(doc);
```

MongoDB understands the internals of BSON objects -- not only can it store them, it can query on internal fields and index keys based upon them. For example the query

```
> db.posts.find( { "comments.author" : "jim" } )
```

is possible and means "find any blog post where at least one comment subjobject has author == 'jim'".

Mongo-Friendly Schema

Mongo can be used in many ways, and one's first instincts when using it are probably going to be similar to how one would write an application with a relational database. While this work pretty well, it doesn't harness the real power of Mongo. Mongo is designed for and works best with a rich object model.

Store Example

If you're building a simple online store that sells products with a relation database, you might have a schema like:

```
item
     title
     price
     sku
  item_features
     sku
     feature_name
     feature_value
```

You would probably normalize it like this because different items would have different features, and you wouldn't want a table with all possible features. You could model this the same way in mongo, but it would be much more efficient to do

```
item : {
           "title" : <title> ,
           "price" : <price> ,
           "sku"   : <sku>   ,
           "features" : {
              "optical zoom" : <value> ,
              ...
           }
  }
```

This does a few nice things

- you can load an entire item with one query
- all the data for an item is on the same place on disk, thus only one seek is required to load

Now, at first glance there might seem to be some issues, but we've got them covered.

- you might want to insert or update a single feature. mongo lets you operate on embedded files like:

```
db.items.update( { sku : 123 } , { "$set" : { "features.zoom" : "5" } } )
```

- Does adding a feature require moving the entire object on disk? No. mongo has a padding heuristic that adapts to your data so it will leave some empty space for the object to grow. This will prevent indexes from being changed, etc.

Bulk inserts

It is possible to insert many documents in a single db call. Consult your driver's documentation for how to do bulk inserts in that language. There is currently no support for bulk inserts from the mongo shell, see SERVER-2395.

MongoDB is quite fast at a series of singleton inserts. Thus one often does not need to use this specialized version of insert.

In v2.0+ one can set the `ContinueOnError` flag for bulk inserts to signal inserts should continue even if one or more from the batch fails. In that case, `getLastError` will be set if any insert fails, not just the last one. If multiple errors occur, only the most recent will be reported by `getLastError`.

For a sharded collection, `ContinueOnError` is implied and cannot be disabled.

Legal Key Names

Key names in inserted documents are limited as follows:

- The '$' character must not be the first character in the key name.
- The '.' character must not appear anywhere in the key name.

Schema Design

Schema design in MongoDB is very different from schema design in a relational DBMS. However it is still very important and the first step towards building an application.

In relational data models, conceptually is a "correct" design for a given entity relationship model independent of the use case. This is typically a third normal form normalization. One typically only diverges from this for performance reasons. In MongoDB, the schema design is not only a function of the data to be modeled but also of the use case. The schema design is optimized for our most common use case. This has pros and cons – that use case is then typically highly performant; however there is a bias in the schema which may make certain ad hoc queries a little less elegant than in the relational schema.

As we design the schema, the questions we must answer are:

1. When do we embed data versus linking (see below)? Our decisions here imply the answer to question #2:

2. How many collections do we have, and what are they?

3. When do we need atomic operations? These operations can be performed within the scope of a BSON document, but not across documents.

4. What indexes will we create to make query and updates fast?

5. How will we shard? What is the shard key?

Embedding and Linking

A key question when designing a MongoDB schema is when to *embed* and when to *link*. Embedding is the nesting of objects and arrays inside a BSON document. Links are references between documents.

There are no joins in MongoDB – distributed joins would be difficult on a 1,000 server cluster. Embedding is a bit like "prejoined" data. Operations within a document are easy for the server to handle; these operations can be fairly rich. Links in contrast must be processed client-side by the application; the application does this by issuing a follow-up query.

Generally, for "contains" relationships between entities, embedding should be be chosen. Use linking when not using linking would result in duplication of data.

Collections

Collections in MongoDB are analogous to tables in a relational database. Each collection contains *documents*. As mentioned above these documents can be fairly rich.

There is no explicit declaration of the fields within the documents of a collection. However there is a conceptual design from the schema designer of what those fields will be and how the documents in the collection will be structured. MongoDB does not require documents in a collection to have the same structure. However, in practice, most collections are highly homogenous. We can move away from this when we need to though; for example when adding a new field. In a situation such as that, an "alter table" style operation is not necessary.

Atomic Operations

Some problems require the ability to perform atomic operations. For example, simply incrementing a counter is often a case where one wants atomicity. MongoDB can also perform more complex operations such as that shown in the pseudocode below:

```
atomically {
  if( doc.credits > 5 ) {
    doc.credits -= 5;
    doc.debits += 5;
  }
}
```

Another example would be a user registration scenario. We would never want two users to register the same username simultaneously:

```
atomically {
  if( exists a document with username='jane' ) {
    print "username already in use please choose another";
  } else {
    insert a document with username='jane' in the users collection;
    print("thanks you have registered as user jane.");
  }
}
```

The key aspect here in terms of schema design is that our scope of atomicity / ACID properties is the document. Thus we need to assure that all fields relevant to the atomic operation are in the same document.

Indexes

MongoDB supports the declaration of indexes. Indexes in MongoDB are highly analogous to indexes in a relational database : they are needed for efficient query processing, and must be explicitly declared. Thus we need to think about what indexes we will define as part of the schema design process. Just like in a relational database, indexes can be added later – if we decide later to have a new one, we can do that.

Sharding

Another consideration for schema design is sharding. A BSON document (which may have significant amounts of embedding) resides on one and only one shard.

A collection may be sharded. When sharded, the collection has a shard key, which determines how the collection is partitioned among shards. Typically (but not always) queries on a sharded collection involve the shard key as part of the query expression.

The key here is that changing shard keys is difficult. You will want to choose the right key from the start.

Example

Let's consider an example, which is a content management system. The examples below use mongo shell syntax but could also be done in any programming language – just use the appropriate driver for that language.

Our content management system will have posts. Posts have authors. We'd like to support commenting and voting on posts. We'd also like posts to be taggable for searching.

One good schema design for this scenario would be to have two MongoDB collections, one called `posts` and one called `users`. This is what we will use for the example.

Our users have a few properties - a user id they registered with, their real name, and their karma. For example we could invoke:

```
> db.users.insert( { _id : "alex", name: { first:"Alex", last:"Benisson" }, karma : 1.0 } )
```

The `_id` field is always present in MongoDB, and is automically indexed with a unique key constraint. That is perfect for our usernames so we store them in the `_id` field. We don't have to though; we could instead make a username field and let the system automically generate a unique id.

Let's now consider posts. We'll assume some posts are already populated. Let's query one:

```
> db.posts.findOne()
{
  _id : ObjectId("4e77bb3b8a3e000000004f7a"),
  when : Date("2011-09-19T02:10:11.3Z",
  author : "alex",
  title : "No Free Lunch",
  text : "This is the text of the post.  It could be very long.",
  tags : [ "business", "ramblings" ],
  votes : 5,
  voters : [ "jane", "joe", "spencer", "phyllis", "li" ],
  comments : [
    { who : "jane", when : Date("2011-09-19T04:00:10.112Z"),
      comment : "I agree." },
    { who : "meghan", when : Date("2011-09-20T14:36:06.958Z"),
      comment : "You must be joking.  etc etc ..." }
  ]
}
```

It's interesting to contrast this with how we might design the same schema in a relation database. We would likely have a users collection and a posts collection. But in addition one would typically have a tags collection, a voters collection, and a comments collection. Grabbing all the information on a single post would then be a little involved. Here to grab the whole post we might execute:

```
> db.posts.findOne( { _id : ObjectId("4e77bb3b8a3e000000004f7a") } );
```

To get all posts written by alex:

```
> db.posts.find( { author : "alex" } )
```

If the above is a common query we would create an index on the author field:

```
> db.posts.ensureIndex( { author : 1 } )
```

The post documents can be fairly large. To get just the titles of the posts by alex :

```
> db.posts.find( { author : "alex" }, { title : 1 } )
```

We may want to search for posts by tag:

```
> // make and index of all tags so that the query is fast:
> db.posts.ensureIndex( { tags : 1 } )
> db.posts.find( { tags : "business" } )
```

What if we want to find all posts commented on by meghan?

```
> db.posts.find( { comments.who : "meghan" } )
```

Let's index that to make it fast:

```
> db.posts.ensureIndex( { "comments.who" : 1 } )
```

We track voters above so that no one can vote more than once. Suppose calvin wants to vote for the example post above. The following update operation will record calvin's vote. Because of the `$nin` sub-expression, if calvin has already voted, the update will have no effect.

```
> db.posts.update( { _id : ObjectId("4e77bb3b8a3e000000004f7a"),
                     voters : { $nin : "calvin" } },
                   { votes : { $inc : 1 }, voters : { $push : "calvin" } );
```

Note the above operation is atomic : if multiple users vote simultaneously, no votes would be lost.

Suppose we want to display the title of the latest post in the system as well as the full user name for the author. This is a case where we must use client-side linking:

```
> var post = db.posts.find().sort( { when : -1 } ).limit(1);
> var user = db.users.find( { _id : post.author } );
> print( post.title + " " + user.name.first + " " + user.name.last );
```

A final question we might ask about our example is how we would shard. If the users collection is small, we would not need to shard it at all. If posts is huge, we would shard it. We would need to choose a shard key. The key should be chosen based on the queries that will be common. We want those queries to involve the shard key.

- Sharding by _id is one option here.
- If finding the most recent posts is a very frequent query, we would then shard on the `when` field. (There is also an optimization trick which might work here.)

Summary of Best Practices

- "First class" objects, that are at top level, typically have their own collection.
- Line item detail objects typically are embedded.
- Objects which follow an object modelling "contains" relationship should generally be embedded.
- Many to many relationships are generally done by linking.
- Collections with only a few objects may safely exist as separate collections, as the whole collection is quickly cached in application server memory.
- Embedded objects are a bit harder to link to than "top level" objects in collections.
- It is more difficult to get a system-level view for embedded objects. When needed an operation of this sort is performed by using MongoDB's map/reduce facility.
- If the amount of data to embed is huge (many megabytes), you may reach the limit on size of a single object. See also GridFS.
- If performance is an issue, embed.

More Details

Choosing Indexes

A second aspect of schema design is index selection. As a general rule, *where you want an index in a relational database, you want an index in Mongo.*

- The `_id` field is automatically indexed.
- Fields upon which keys are looked up should be indexed.
- Sort fields generally should be indexed.

The MongoDB profiling facility provides useful information for where an index should be added that is missing.

Note that adding an index slows writes to a collection, but not reads. Use lots of indexes for collections with a high read : write ratio (assuming one does not mind the storage overage). For collections with more writes than reads, indexes are expensive as keys must be added to each index for each insert.

How Many Collections?

As Mongo collections are polymorphic, one could have a collection `objects` and put everything in it! This approach is taken by some object databases. This is not recommended in MongoDB for several reasons, mainly performance. Data within a single collection is roughly contiguous on disk. Thus, "table scans" of a collection are possible, and efficient. Just like in relational dbs, independent collections are very important for high throughput batch processing.

See Also

Books

- Document Design for MongoDB - O'Reilly Ebook

* More Books

Blog posts

* http://blog.fiesta.cc/post/11319522700/walkthrough-mongodb-data-modeling
* Blog post on dynamic schemas

Related Doc Pages

* Trees in MongoDB
* MongoDB Data Modeling and Rails
* Advanced Queries

Videos

* Schema Design Basics - MongoSF Presentation (May 2011)
* Schema Design at Scale - MongoSF Presentation (May 2011)
* Schema Design: Data as Documents - MongoSV Presentation (December 2010)

Schema Design Basics - Presentation from MongoSF (May 2011)

<#comment><#comment>

Schema Design by Example - Presentation at MongoSV (December 2011)

Tweaking performance by document bundling during schema design

Note: this page discusses performance tuning aspects – if you are just getting started skip this for later. If you have a giant collection of small documents that will require significant tuning, read on.

During schema design one consideration is when to embed entities in a larger document versus storing them as separate small documents. Tiny documents work fine and should be used when that is the natural way to go with the schema. However, in some circumstances, it can be better to group data into larger documents to improve performance.

Consider for example a collection which contains some documents that are fairly small. Documents are indicated in the figures below as squares. Related documents – perhaps all associated with some larger entity in our program, or else that correlate in their access, are indicated in figure 1 with the same color.

MongoDB caches data in pages, where the page size is that of the operating system's virtual memory manager (almost always 4KB). Page units are indicated by the black lines – for this example 8 boxes fit per page.

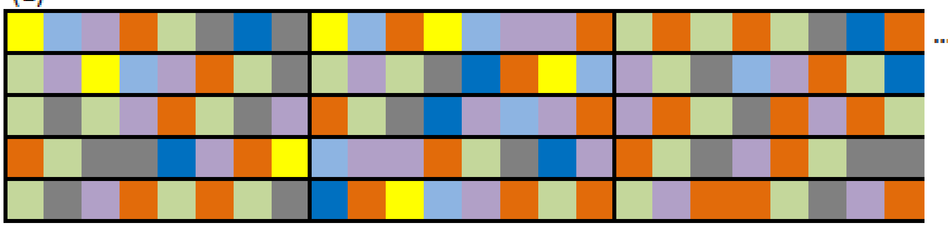
Let's suppose we wish to fetch all of the dark blue documents – indicates with stripes in figure 2. If this data is in RAM, we can (assuming we have an index) fetch them very efficiently. However note that the eight entities span eight pages, even though they could in theory fit on a single page.

With an alternate schema design we could "roll up" some of these entities into a larger document which includes an array of subdocuments. By doing that the items will be clustered together – a single BSON document in MongoDB is always stored contiguously. Figure 3 shows an example where the eight entities roll up into two documents (perhaps they could have rolled up to just one document; the point here is that it isn't essential that it be one, we are simply doing some bundling). In this example the two new documents are stored within three pages. While this isn't a huge reduction – eight to three – in many situations the documents are much smaller than a page – sometimes 100 documents fit within a single page. (The diagram example is not very granular to make reading easy.)

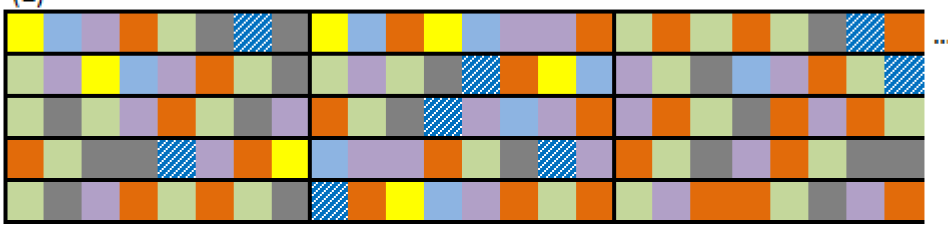
The benefits of this rolled-up schema design are

* Better RAM cache utilization. If we need to cache the dark blue items (but not the others), we can now cache three pages instead of eight. Note this is really only important if the data is too large to fit entirely in RAM – if it all fits, there is no gain here.
* Fewer disk seeks. If nothing was cached in RAM, less random i/o's are necessary to fetch the objects.
* Smaller index sizse. The common key the eight items contain can be stored in less copies, with less associated key entries in its corresponding index.

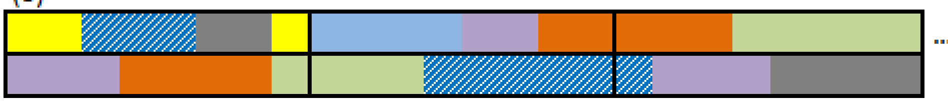
(1)



(2)



(3)



Caveats:

- Do not optimize prematurely; if grouping the entities would be awkward, don't do it. The goal of Mongo is to make development easier, not harder.
- Note we simply want to get to a document size that is on the order of the page cache unit size – about 4KB. If your documents are already of roughly that size, there is less benefit to the above (some still regarding random disk i/o, but no benefit for ram cache efficiency).
- If you often only need a subset of the items you would group, this approach could be inefficient compared to alternatives.

Trees in MongoDB

The best way to store a tree usually depends on the operations you want to perform; see below for some different options. In practice, most developers find that one of the "Full Tree in Single Document", "Parent Links", and "Array of Ancestors" patterns works best.

Patterns

Full Tree in Single Document

```
{
  comments: [
    {by: "mathias", text: "...", replies: []}
    {by: "eliot", text: "...", replies: [
      {by: "mike", text: "...", replies: []}
    ]}
  ]
}
```

Pros:

- Single document to fetch per page
- One location on disk for whole tree
- You can see full structure easily

Cons:

- Hard to search
- Hard to get back partial results
- Can get unwieldy if you need a huge tree. Further there is a limit on the size of documents in MongoDB – 16MB in v1.8 (limit may rise in future versions).

Parent Links

Storing all nodes in a single collection, with each node having the id of its parent, is a simple solution. The biggest problem with this approach is getting an entire subtree requires several query turnarounds to the database (or use of db.eval).

```
> t = db.tree1;

> t.find()
{ "_id" : 1 }
{ "_id" : 2, "parent" : 1 }
{ "_id" : 3, "parent" : 1 }
{ "_id" : 4, "parent" : 2 }
{ "_id" : 5, "parent" : 4 }
{ "_id" : 6, "parent" : 4 }

> // find children of node 4
> t.ensureIndex({parent:1})
> t.find( {parent : 4 } )
{ "_id" : 5, "parent" : 4 }
{ "_id" : 6, "parent" : 4 }
```

Child Links

Another option is storing the ids of all of a node's children within each node's document. This approach is fairly limiting, although ok if no operations on entire subtrees are necessary. It may also be good for storing graphs where a node has multiple parents.

```
> t = db.tree2
> t.find()
{ "_id" : 1, "children" : [ 2, 3 ] }
{ "_id" : 2 }
{ "_id" : 3, "children" : [ 4 ] }
{ "_id" : 4 }

> // find immediate children of node 3
> t.findOne({_id:3}).children
[ 4 ]

> // find immediate parent of node 3
> t.ensureIndex({children:1})
> t.find({children:3})
{ "_id" : 1, "children" : [ 2, 3 ] }
```

Array of Ancestors

Here we store all the ancestors of a node in an array. This makes a query like "get all descendents of x" fast and easy.

```
> t = db.mytree;

> t.find()
{ "_id" : "a" }
{ "_id" : "b", "ancestors" : [ "a" ], "parent" : "a" }
{ "_id" : "c", "ancestors" : [ "a", "b" ], "parent" : "b" }
{ "_id" : "d", "ancestors" : [ "a", "b" ], "parent" : "b" }
{ "_id" : "e", "ancestors" : [ "a" ], "parent" : "a" }
{ "_id" : "f", "ancestors" : [ "a", "e" ], "parent" : "e" }
{ "_id" : "g", "ancestors" : [ "a", "b", "d" ], "parent" : "d" }

> t.ensureIndex( { ancestors : 1 } )

> // find all descendents of b:
> t.find( { ancestors : 'b' })
{ "_id" : "c", "ancestors" : [ "a", "b" ], "parent" : "b" }
{ "_id" : "d", "ancestors" : [ "a", "b" ], "parent" : "b" }
{ "_id" : "g", "ancestors" : [ "a", "b", "d" ], "parent" : "d" }

> // get all ancestors of f:
> anc = db.mytree.findOne({_id:'f'}).ancestors
[ "a", "e" ]
> db.mytree.find( { _id : { $in : anc } } )
{ "_id" : "a" }
{ "_id" : "e", "ancestors" : [ "a" ], "parent" : "a" }
```

ensureIndex and MongoDB's multikey feature makes the above queries efficient.

In addition to the ancestors array, we also stored the direct parent in the node to make it easy to find the node's immediate parent when that is necessary.

Materialized Paths (Full Path in Each Node)

Materialized paths make certain query options on trees easy. We store the full path to the location of a document in the tree within each node. Usually the "array of ancestors" approach above works just as well, and is easier as one doesn't have to deal with string building, regular expressions, and escaping of characters. (Theoretically, materialized paths will be *slightly* faster.)

The best way to do this with MongoDB is to store the path as a string and then use regex queries. Simple regex expressions beginning with "^" can be efficiently executed. As the path is a string, you will need to pick a delimiter character -- we use ',' below. For example:

```
> t = db.tree
test.tree

> // get entire tree -- we use sort() to make the order nice
> t.find().sort({path:1})
{ "_id" : "a", "path" : "a," }
{ "_id" : "b", "path" : "a,b," }
{ "_id" : "c", "path" : "a,b,c," }
{ "_id" : "d", "path" : "a,b,d," }
{ "_id" : "g", "path" : "a,b,g," }
{ "_id" : "e", "path" : "a,e," }
{ "_id" : "f", "path" : "a,e,f," }
{ "_id" : "g", "path" : "a,b,g," }

> t.ensureIndex( {path:1} )

> // find the node 'b' and all its descendents:
> t.find( { path : /^a,b,/ } )
{ "_id" : "b", "path" : "a,b," }
{ "_id" : "c", "path" : "a,b,c," }
{ "_id" : "d", "path" : "a,b,d," }
{ "_id" : "g", "path" : "a,b,g," }

> // find the node 'b' and its descendents, where path to 'b' is not already known:
> nodeb = t.findOne( { _id : "b" } )
{ "_id" : "b", "path" : "a,b," }
> t.find( { path : new RegExp("^" + nodeb.path) } )
{ "_id" : "b", "path" : "a,b," }
{ "_id" : "c", "path" : "a,b,c," }
{ "_id" : "d", "path" : "a,b,d," }
{ "_id" : "g", "path" : "a,b,g," }
```

Ruby example: http://github.com/banker/newsmonger/blob/master/app/models/comment.rb

acts_as_nested_set

See http://ar.rubyonrails.org/classes/ActiveRecord/Acts/NestedSet/ClassMethods.html

This pattern is best for datasets that rarely change as modifications can require changes to many documents.

Use Cases

Find Nodes by a Partial Path

Suppose we want to find a given node in a tree, given some path through a portion of the tree to that node, and then get back that node, and perhaps also everything below it.

With a materialized paths approach we can do the above. The main thing that needs tweaking is to make the operation fast if there is a path "a..b..c..d..e" to a document and we want to find documents with a path "..b..c..d..". If we are starting from the very top it is easy (and described above in the materialized paths section). However here we aren't starting at the top. One approach is to use a combination of materialized path plus an array of the node's ancestors, something like:

```
{ path : ",a,b,c,d,e,",
  ancestor : ['a','b','c','d','e'] }
```

We could index on ancestors which will create a multikey index. Then we would do a query like the following to find nodes on path "...b,c,d..." with some efficiency:

```
find({ path : /,b,c,d,/, ancestor : 'd', <more_query_expressions_optionally> })
```

In the above the index on ancestor would be used and only docs from 'd' down need be inspected. The following could be tried which might be even better depending on how smart the query optimizer is:

```
find( { path : /,b,c,d,/, ancestor : { $all : ['a','d'] }, ... } )
```

See Also

- Sean Cribbs blog post (source of several ideas on this page).

Optimization

Additional Articles

- Optimizing Object IDs
- Optimizing Storage of Small Objects

Optimizing A Simple Example

This section describes proper techniques for optimizing database performance.

Let's consider an example. Suppose our task is to display the front page of a blog - we wish to display headlines of the 10 most recent posts. Let's assume the posts have a timestamp field `ts`.

The simplest thing we could write might be:

```
articles = db.posts.find().sort({ts:-1}); // get blog posts in reverse time order

for (var i=0; i< 10; i++) {
   print(articles[i].getSummary());
}
```

Optimization #1: Create an index

Our first optimization should be to create an index on the key that is being used for the sorting:

```
db.posts.ensureIndex({ts:1});
```

With an index, the database is able to sort based on index information, rather than having to check each document in the collection directly. This is much faster.

Optimization #2: Limit results

MongoDB cursors return results in groups of documents that we'll call 'chunks'. The chunk returned might contain more than 10 objects - in some cases, much more. These extra objects are a waste of network transmission and resources both on the app server and the database.

As we know how many results we want, and that we do not want all the results, we can use the `limit()` method for our second optimization.

```
articles = db.posts.find().sort({ts:-1}).limit(10); // 10 results maximum
```

Now, we'll only get 10 results returned to client.

Optimization #3: Select only relevant fields

The blog post object may be very large, with the post text and comments embedded. Much better performance will be achieved by selecting only the fields we need:

```
articles = db.posts.find({}, {ts:1,title:1,author:1,abstract:1}).sort({ts:-1}).limit(10);
articles.forEach( function(post) { print(post.getSummary()); } );
```

The above code assumes that the `getSummary()` method only references the fields listed in the `find()` method.

Note if you fetch only select fields, you have a partial object. An object in that form cannot be updated back to the database:

```
a_post = db.posts.findOne({}, Post.summaryFields);
a_post.x = 3;
db.posts.save(a_post); // error, exception thrown
```

Using the Profiler

MongoDB includes a database profiler which shows performance characteristics of each operation against the database. Using the profiler you can find queries (and write operations) which are slower than they should be; use this information, for example, to determine when an index is needed. See the Database Profiler page for more information.

Optimizing Statements that Use `count()`

To speed operations that rely on `count()`, create an index on the field involved in the count query expression.

```
db.posts.ensureIndex({author:1});
db.posts.find({author:"george"}).count();
```

Increment Operations

MongoDB supports simple object field increment operations; basically, this is an operation indicating "increment this field in this document at the server". This can be much faster than fetching the document, updating the field, and then saving it back to the server and are particularly useful for implementing real time counters. See the Updates section of the Mongo Developers' Guide for more information.

Circular Fixed Size Collections

MongoDB provides a special circular collection type that is pre-allocated at a specific size. These collections keep the items within well-ordered even without an index, and provide very high-speed writes and reads to the collection. Originally designed for keeping log files - log events are stored in the database in a circular fixed size collection - there are many uses for this feature. See the Capped Collections section of the Mongo Developers' Guide for more information.

Server Side Code Execution

Occasionally, for maximal performance, you may wish to perform an operation in process on the database server to eliminate client/server network turnarounds. These operations are covered in the Server-Side Processing section of the Mongo Developers' Guide.

Explain

A great way to get more information on the performance of your database queries is to use the explain plan feature. See the Explain doc page.

Hint

While the mongo query optimizer often performs very well, explicit "hints" can be used to force mongo to use a specified index, potentially improving performance in some situations. When you have a collection indexed and are querying on multiple fields (and some of those fields are indexed), pass the index as a hint to the query.

To set the hint for a particular query, call the `hint()` function on the cursor before accessing any data, and specify a document with the key to be used in the query:

```
db.collection.find({user:u, foo:d}).hint({user:1});
```

Be sure to Index
For the above hints to work, you need to have run `ensureIndex()` to index the collection on the user field.

Some other examples, for an index on {a:1, b:1} named "a_1_b_1":

```
db.collection.find({a:4,b:5,c:6}).hint({a:1,b:1});
db.collection.find({a:4,b:5,c:6}).hint("a_1_b_1");
```

To force the query optimizer to not use indexes (do a table scan), use:

```
> db.collection.find().hint({$natural:1})
```

See Also

- Query Optimizer
- currentOp()
- Sorting and Natural Order

Explain

- Basics
- Output Fields
    - cursor
    - nscanned
    - nscannedObjects
    - nYields
    - n
    - millis
    - scanAndOrder
    - indexOnly
    - isMultiKey
- With Sharding
    - clusteredType
    - shards
    - nChunkSkips
    - millisTotal
    - millisAvg
    - numQueries
    - numShards
- See Also

Basics

A great way to get more information on the performance of your database queries is to use the `$explain` feature. This will display "explain plan" information about a query from the database.

When using the mongo shell, invoke the `explain()` method on a cursor. The result will be a document that contains the explain output. *Note that explain runs the actual query to determine the result.* If the query is slow, the explain will be slow too.

Some of the options below may not appear depending on the version of mongodb you are running.

```
> db.collection.find(query).explain()
```

provides information such as the following:

```
{
    "cursor" : "BasicCursor",
    "indexBounds" : [ ],
    "nscanned" : 57594,
    "nscannedObjects" : 57594,
    "nYields" : 2 ,
    "n" : 3 ,
    "millis" : 108,
    "indexOnly" : false,
    "isMultiKey" : false,
    "nChunkSkips" : 0
}
```

Output Fields

cursor

The type of cursor used for scanning.

- BasicCursor - indicates a table scan style operation
- BtreeCursor - an index was used. When an index is used, indexBounds will be set to indicate the key bounds for scanning in the index.

nscanned

Number of items (documents or index entries) examined. Items might be objects or index keys. If a "covered index" is involved, nscanned may be higher than nscannedObjects.

nscannedObjects

Number of documents scanned.

nYields

Number of times this query yielded the read lock to let waiting writes execute.

n

Number of documents matched (on all criteria specified).

millis

Time for the query to run, in milliseconds.

scanAndOrder

True if the index could not be used for sorting.

indexOnly

True if the results of the query can be returned using only the index. See SERVER-5759 for more information on the behavior of this value.

isMultiKey

If true, a multikey index was used.

With Sharding

In a sharded deployment, `explain` outputs some additional fields. An example:

```
"clusteredType" : "ParallelSort",
 "shards" : {
  "shard1:27017" : [
   {
    "cursor" : "BtreeCursor a_1",
    "nscanned" : 1,
    "nscannedObjects" : 1,
    "n" : 1,
    "millis" : 0,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "isMultiKey" : false,
    "indexOnly" : false,
    "indexBounds" : {
     "a" : [
      [
       1,
       1
      ]
     ]
    }
   }
  ]
 },
 "n" : 1,
 "nChunkSkips" : 0,
 "nYields" : 0,
 "nscanned" : 1,
 "nscannedObjects" : 1,
 "millisTotal" : 0,
 "millisAvg" : 0,
 "numQueries" : 1,
 "numShards" : 1
```

clusteredType

- `ParallelSort` means all shards are accessed in parallel
- `SerialServer` means shards are queried one by one in order

shards

List of all the shards accessed during the query and the `explain` output for each shard.

nChunkSkips

The number of documents skipped because of active chunk migrations in a sharded system. Typically this will be zero. A number greater than zero is ok, but indicates a little bit of inefficiency.

millisTotal

Total time for the query to run, in milliseconds.

millisAvg

Average time for the query to run on each shard, in milliseconds.

numQueries

Total number of queries executed.

numShards

Total number of shards queried.

See Also

- Optimization
- Database Profiler
- Viewing and Terminating Current Operation

Optimizing Object IDs

- Use the collections 'natural primary key' in the _id field.
- When possible, use _id values that are roughly in ascending order.
- Store Binary GUIDs as BinData, rather than as hex encoded strings
- Extract insertion times from _id rather than having a separate timestamp field.
- Sort by _id to sort by insertion time
- See Also

The `_id` field in a MongoDB document is very important and is always indexed for normal collections. This page lists some recommendations. Note that it is common to use the BSON ObjectID datatype for _id's, but the values of an `_id` field can be of any type.

Use the collections 'natural primary key' in the _id field.

_id's can be any type, so if your objects have a natural unique identifier, consider using that in _id to both save space and avoid an additional index.

When possible, use _id values that are roughly in ascending order.

If the _id's are in a somewhat well defined order, on inserts the entire b-tree for the _id index need not be loaded. BSON ObjectIds have this property.

Store Binary GUIDs as BinData, rather than as hex encoded strings

BSON includes a binary data datatype for storing byte arrays. Using this will make the id values, and their respective keys in the _id index, twice as small.

Note that unlike the BSON Object ID type (see above), most UUIDs do not have a rough ascending order, which creates additional caching needs for their index.

```
> // mongo shell bindata info:
> help misc
        b = new BinData(subtype,base64str)  create a BSON BinData value
        b.subtype()                         the BinData subtype (0..255)
        b.length()                          length of the BinData data in bytes
        b.hex()                             the data as a hex encoded string
        b.base64()                          the data as a base 64 encoded string
        b.toString()
```

Extract insertion times from _id rather than having a separate timestamp field.

The BSON ObjectId format provides documents with a creation timestamp (one second granularity) for free. Almost all drivers implement methods for extracting these timestamps; see the relevant api docs for details. In the shell:

```
> // mongo shell ObjectId methods
> help misc
        o = new ObjectId()      create a new ObjectId
        o.getTimestamp()        return timestamp derived from first 32 bits of the OID
        o.isObjectId()
        o.toString()
        o.equals(otherid)
```

Sort by _id to sort by insertion time

BSON ObjectId's begin with a timestamp. Thus sorting by _id, when using the ObjectID type, results in sorting by time. Note: granularity of the timestamp portion of the ObjectID is to one second only.

```
> // get 10 newest items
> db.mycollection.find().sort({id:-1}).limit(10);
```

See Also

- Object IDs

Optimizing Storage of Small Objects

MongoDB records have a certain amount of overhead per object (BSON document) in a collection. This overhead is normally insignificant, but if your objects are tiny (just a few bytes, maybe one or two fields) it would not be. Below are some suggestions on how to optimize storage efficiently in such situations.

Using the _id Field Explicitly

Mongo automatically adds an object ID to each document and sets it to a unique value. Additionally this field in indexed. For tiny objects this takes up significant space.

The best way to optimize for this is to use _id explicitly. Take one of your fields which is unique for the collection and store its values in _id. By doing so, you have explicitly provided IDs. This will effectively eliminate the creation of a separate _id field. If your previously separate field was indexed, this eliminates an extra index too.

Using Small Field Names

Consider a record

```
{ last_name : "Smith", best_score: 3.9 }
```

The strings "last_name" and "best_score" will be stored in each object's BSON. Using shorter strings would save space:

```
{ lname : "Smith", score : 3.9 }
```

Would save 9 bytes per document. This of course reduces expressiveness to the programmer and is not recommended unless you have a collection where this is of significant concern.

Field names are not stored in indexes as indexes have a predefined structure. Thus, shortening field names will not help the size of indexes. In general it is not necessary to use short field names.

Combining Objects

Fundamentally, there is a certain amount of overhead per document in MongoDB. One technique is combining objects. In some cases you may be able to embed objects in other objects, perhaps as arrays of objects. If your objects are tiny this may work well, but will only make sense for certain use cases.

Query Optimizer

The MongoDB query optimizer generates query plans for each query and then executes the plan to return results. Thus, MongoDB supports ad hoc queries much like say, Oracle or MySQL.

The database uses an interesting approach to query optimization. Traditional approaches (which tend to be cost-based and statistical) are not used, as these approaches have a couple of potential issues.

First, the optimizer might consistently pick a bad query plan. For example, there might be correlations in the data of which the optimizer is unaware. In a situation like this, the developer might use a query hint.

Also with the traditional approach, query plans can change in production with negative results. No one thinks rolling out new code without testing is a good idea. Yet often in a production system a query plan can change as the statistics in the database change on the underlying data. The query plan in effect may be a plan that never was invoked in QA. If it is slower than it should be, the application could experience an outage.

The MongoDB query optimizer is different. It is not cost based -- it does not model the cost of various queries. Instead, the optimizer simply tries different query plans and learn which ones work well. Of course, when the system tries a really bad plan, it may take an extremely long time to run. To solve this, *when testing new plans, MongoDB executes multiple query plans in parallel*. As soon as one finishes, it terminates the other executions, and the system has learned which plan is good. This works particularly well given the system is non-relational, which makes the space of possible query plans much smaller (as there are no joins).

Sometimes a plan which was working well can work poorly -- for example if the data in the database has changed, or if the parameter values to the query are different. In this case, if the query seems to be taking longer than usual, the database will once again run the query in parallel to try different plans.

This approach adds a little overhead, but has the advantage of being much better at worst-case performance.

Testing of queries repeats after 1,000 operations and also after certain manipulations of a collection occur (such as adding an index).

Query plan selection is based on a "query pattern". For example the query

```
{ x : 123 }
```

is treated as the same pattern as { x : <anothervalue> }.

See Also

- MongoDB hint() and explain() operators

Querying

One of MongoDB's best capabilities is its support for dynamic (ad hoc) queries. Systems that support dynamic queries don't require any special indexing to find data; users can find data using any criteria. For relational databases, dynamic queries are the norm. If you're moving to MongoDB from a relational databases, you'll find that many SQL queries translate easily to MongoDB's document-based query language.

> In MongoDB, just like in an RDBMS, creating appropriate indexes for queries is quite important for performance. See the Indexes page for more info.

Query Expression Objects

MongoDB supports a number of query objects for fetching data. Queries are expressed as BSON documents which indicate a query pattern. For example, suppose we're using the MongoDB shell and want to return every document in the `users` collection. Our query would look like this:

```
db.users.find({})
```

In this case, our selector is an empty document, which matches every document in the collection. Here's a more selective example:

```
db.users.find({'last_name': 'Smith'})
```

Here our selector will match every document where the `last_name` attribute is 'Smith.'

MongoDB support a wide array of possible document selectors. For more examples, see the MongoDB Tutorial or the section on Advanced Queries. If you're working with MongoDB from a language driver, see the driver docs:

Query Options

Field Selection

In addition to the query expression, MongoDB queries can take some additional arguments. For example, it's possible to request only certain fields be returned. If we just wanted the social security numbers of users with the last name of 'Smith,' then from the shell we could issue this query:

```
// retrieve ssn field for documents where last_name == 'Smith':
db.users.find({last_name: 'Smith'}, {'ssn': 1});

  // retrieve all fields *except* the thumbnail field, for all documents:
db.users.find({}, {thumbnail:0});
```

Note the `_id` field is always returned even when not explicitly requested.

Sorting

MongoDB queries can return sorted results. To return all documents and sort by last name in ascending order, we'd query like so:

```
db.users.find({}).sort({last_name: 1});
```

Skip and Limit

MongoDB also supports **skip** and **limit** for easy paging. Here we skip the first 20 last names, and limit our result set to 10:

```
db.users.find().skip(20).limit(10);
db.users.find({}, {}, 10, 20); // same as above, but less clear
```

slaveOk (Querying Secondaries)

When querying a replica set, drivers route their requests to the master mongod by default; to perform a query against an (arbitrarily-selected) secondary, the query can be run with the slaveOk option. See your driver's for details on enabling slaveOk.

In the mongo shell, if you try reading from a secondary without specifying slaveOk, you will receive the error message

```
13435 not master and slaveok=false
```

In the mongo shell, to indicate slaveOk mode, enter the following:

```
rs.slaveOk(); // enable querying a secondary
db.users.find(...)
```

By indicating slaveOk, we are declaring "for my usage on this connection I am ok with eventually consistent reads".

Cursors

Database queries, performed with the find() method, technically work by returning a *cursor*. Cursors are then used to iteratively retrieve all the documents returned by the query. For example, we can iterate over a cursor in the mongo shell like this:

```
> var cur = db.example.find();
> cur.forEach( function(x) { print(tojson(x))});
{"n" : 1 , "_id" : "497ce96f395f2f052a494fd4"}
{"n" : 2 , "_id" : "497ce971395f2f052a494fd5"}
{"n" : 3 , "_id" : "497ce973395f2f052a494fd6"}
>
```

Quick Reference Card

Download the Query and Update Modifier Quick Reference Card

More info

This was just an introduction to querying in Mongo. More information:

- Mongo Query Language
- Querying and nulls
- Retrieving a Subset of Fields
- Advanced Queries
- Dot Notation (Reaching into Objects)
- Full Text Search in Mongo

- min and max Query Specifiers
- OR operations in query expressions
- Queries and Cursors
- Server-side Code Execution
- Sorting and Natural Order
- Aggregation

See Also

- Advanced Queries
- Indexes
- Queries and Cursors
- Query Optimizer

Mongo Query Language

Queries in MongoDB are expressed as JSON (BSON). Usually we think of query object as the equivalent of a SQL "WHERE" clause:

```
> db.users.find( { x : 3, y : "abc" } ).sort({x:1}); // select * from users where x=3 and y='abc'
order by x asc;
```

However, the MongoDB server actually looks at all the query parameters (ordering, limit, etc.) as a single object. In the above example from the mongo shell, the shell is adding some syntactic sugar for us. Many of the drivers do this too. For example the above query could also be written:

```
> db.users.find( { $query : { x : 3, y : "abc" }, $orderby : { x : 1 } } );
```

The possible specifies in the query object are:

- $query - the evaluation or "where" expression
- $orderby - sort order desired
- $hint - hint to query optimizer
- $explain - if true, return explain plan results instead of query results
- $snapshot - if true, "snapshot mode"

Querying and nulls

The `null` value in javascript carries different meanings. When a query involves `null`, this can have multiple interpretations.

Take the following examples:

```
> db.foo.insert( { x : 1, y : 1 } )
> db.foo.insert( { x : 2, y : "string" } )
> db.foo.insert( { x : 3, y : null } )
> db.foo.insert( { x : 4 } )

// Query #1
> db.foo.find( { "y" : null } )
{ "_id" : ObjectId("4dc1975312c677fc83b5629f"), "x" : 3, "y" : null }
{ "_id" : ObjectId("4dc1975a12c677fc83b562a0"), "x" : 4 }

// Query #2
> db.foo.find( { "y" : { $type : 10 } } )
{ "_id" : ObjectId("4dc1975312c677fc83b5629f"), "x" : 3, "y" : null }

// Query #3
> db.foo.find( { "y" : { $exists : false } } )
{ "_id" : ObjectId("4dc1975a12c677fc83b562a0"), "x" : 4 }
```

To summarize the three queries:

1. documents where y has the value `null` **or** where y does not exist
2. documents where y has the value `null`
3. documents where y does not exist

Retrieving a Subset of Fields

By default on a find operation, the entire object is returned. However we may also request that only certain fields be returned. This is somewhat analogous to the list of [column specifiers in a SQL SELECT statement (projection).

```
// select z from things where x="john"
db.things.find( { x : "john" }, { z : 1 } );
```

Field Negation

We can say "all fields except x" – for example to remove specific fields that you know will be large:

```
// get all posts about 'tennis' but without the comments field
db.posts.find( { tags : 'tennis' }, { comments : 0 } );
```

The `_id` Field

The `_id` field will be included by default.

If you do not want it, you must exclude it specifically. (Typically you will want to do that if using the covered index feature described below.)

```
// get all posts about 'tennis' but without the _id field
db.posts.find( { tags : 'tennis' }, { _id : 0 } );
```

Mixing Includes/Excludes

You cannot mix them, with the exception of the `_id` field. Note also that the $slice operator does not conflict with exclusions on other fields.

Covered Indexes

v1.8+

MongoDB can return data from the index only when the query only involves keys which are present in the index. Not inspecting the actual documents can speed up responses considerably since the index is compact in size and usually fits in RAM, or is sequentially located on disk.

Mongod will automatically use covered index when it can. But be sure that:

- you provide list of fields to return, so that it can determine that it can be covered by index
- you must explicitly exclude the _id field by using {_id: 0} (unless the index includes that)
- as soon as you insert one array value for one of the index keys, the index will immediately become a multikey index and this disables covered index functionality
- use Explain to determine if the covered index is used: the indexOnly field should be true

```
// do a login with a covered index, returning the users roles/groups
> db.users.ensureIndex( { username : 1, password : 1, roles : 1} );
> db.users.save({username: "joe", password: "pass", roles: 2})
> db.users.save({username: "liz", password: "pass2", roles: 4})
> db.users.find({username: "joe"}, {_id: 0, roles: 1})
{ "roles" : 2 }
> db.users.find({username: "joe"}, {_id: 0, roles: 1}).explain()
{
 "cursor" : "BtreeCursor username_1_password_1_roles_1",
...
 "indexOnly" : true,
...
}
```

Dot Notation

You can retrieve partial sub-objects via Dot Notation.

```
> t.find({})
{ "_id" : ObjectId("4c23f0486dad1c3a68457d20"), "x" : { "y" : 1, "z" : [ 1, 2, 3 ] } }
> t.find({}, {'x.y':1})
{ "_id" : ObjectId("4c23f0486dad1c3
a68457d20"), "x" : { "y" : 1 } }
```

Retrieving a Subrange of Array Elements

You can use the $slice operator to retrieve a subrange of elements in an array.

```
db.posts.find({}, {comments:{$slice: 5}}) // first 5 comments
db.posts.find({}, {comments:{$slice: -5}}) // last 5 comments
db.posts.find({}, {comments:{$slice: [20, 10]}}) // skip 20, limit 10
db.posts.find({}, {comments:{$slice: [-20, 10]}}) // 20 from end, limit 10
```

The examples above will return all fields and only the subset of elements based on the `$slice` for that field.

Filtering with `$slice` does not affect other fields inclusion/exclusion. It only applies within the array being sliced.

```
db.posts.find({}, {_id:1, comments:{$slice: 5}}) // first 5 comments, and the _id field only
```

See Also

- example slice1

Advanced Queries

- Intro
- Retrieving a Subset of Fields
- Conditional Operators
    - <, <=, >, >=
    - $all
    - $exists
    - $mod
    - $ne
    - $in
    - $nin
    - $nor
    - $or
    - $and
    - $size
    - $type
- Regular Expressions
- Value in an Array

> In MongoDB, just like in an RDBMS, creating appropriate indexes for queries is quite important for performance. See the Indexes page for more info.

Intro

MongoDB offers a rich query environment with lots of features. This page lists some of those features.

Queries in MongoDB are represented as JSON-style objects, very much like the documents we actually store in the database. For example:

```
// i.e., select * from things where x=3 and y="foo"
db.things.find( { x : 3, y : "foo" } );
```

Note that any of the operators on this page can be combined in the same query document. For example, to find all document where j is not equal to 3 and k is greater than 10, you'd query like so:

```
db.things.find({j: {$ne: 3}, k: {$gt: 10} });
```

Unless otherwise noted, the operations below can be used on array elements in the same way that they can be used on "normal" fields. For example, suppose we have some documents such as:

```
> db.things.insert({colors : ["blue", "black"]})
> db.things.insert({colors : ["yellow", "orange", "red"]})
```

Then we can find documents that aren't "red" using:

```
> db.things.find({colors : {$ne : "red"}})
{"_id": ObjectId("4dc9acea045bbf04348f9691"), "colors": ["blue","black"]}
```

Retrieving a Subset of Fields

See Retrieving a Subset of Fields

Conditional Operators

<, <=, >, >=

Use these special forms for greater than and less than comparisons in queries, since they have to be represented in the query document:

```
db.collection.find({ "field" : { $gt: value } } );   // greater than  : field > value
db.collection.find({ "field" : { $lt: value } } );   // less than  :  field < value
db.collection.find({ "field" : { $gte: value } } );  // greater than or equal to : field >= value
db.collection.find({ "field" : { $lte: value } } );  // less than or equal to : field <= value
```

For example:

```
db.things.find({j : {$lt: 3}});
db.things.find({j : {$gte: 4}});
```

You can also combine these operators to specify ranges:

```
db.collection.find({ "field" : { $gt: value1, $lt: value2 } } );    // value1 < field < value
```

$all

The `$all` operator is similar to `$in`, but instead of matching any value in the specified array all values in the array must be matched. For example, the object

```
{ a: [ 1, 2, 3 ] }
```

would be matched by

```
db.things.find( { a: { $all: [ 2, 3 ] } } );
```

but not

```
db.things.find( { a: { $all: [ 2, 3, 4 ] } } );
```

An array can have more elements than those specified by the `$all` criteria. `$all` specifies a minimum set of elements that must be matched.

$exists

Check for existence (or lack thereof) of a field.

```
db.things.find( { a : { $exists : true } } ); // return object if a is present
db.things.find( { a : { $exists : false } } ); // return if a is missing
```

Before v2.0, $exists is not able to use an index. Indexes on other fields are still used.
$exists is not very efficient even with an index, and esp. with {`$exists:true`} since it will effectively have to scan all indexed values.

$mod

The `$mod` operator allows you to do fast modulo queries to replace a common case for where clauses. For example, the following $where query:

```
db.things.find( "this.a % 10 == 1")
```

can be replaced by:

```
db.things.find( { a : { $mod : [ 10 , 1 ] } } )
```

$ne

Use $ne for "not equals".

```
db.things.find( { x : { $ne : 3 } } );
```

$in

The `$in` operator is analogous to the SQL `IN` modifier, allowing you to specify an array of possible matches.

```
db.collection.find( { field : { $in : array } } );
```

Let's consider a couple of examples. From our *things* collection, we could choose to get a subset of documents based upon the value of the 'j' key:

```
db.things.find({j:{$in: [2,4,6]}});
```

Suppose the collection `updates` is a list of social network style news items; we want to see the 10 most recent updates from our friends. We could invoke:

```
db.updates.ensureIndex( { ts : 1 } ); // ts == timestamp
var myFriends = myUserObject.friends; // let's assume this gives us an array of DBRef's of my friends
var latestUpdatesForMe = db.updates.find( { user : { $in : myFriends } } ).sort( { ts : -1 }
).limit(10);
```

The target field's value can also be an array; if so then the document matches if any of the elements of the array's value matches any of the $in field's values (see the Multikeys page for more information.

$nin

The `$nin` operator is similar to `$in` except that it selects objects for which the specified field does not have any value in the specified array. For example

```
db.things.find({j:{$nin: [2,4,6]}});
```

would match {`j:1,b:2`} but not {`j:2,c:9`}.

$nor

The `$nor` operator lets you use a boolean or expression to do queries. You give `$nor` a list of expressions, none of which can satisfy the query.

$or

v1.6+

The `$or` operator lets you use boolean or in a query. You give `$or` an array of expressions, any of which can satisfy the query.

Simple:

```
db.foo.find( { $or : [ { a : 1 } , { b : 2 } ] } )
```

With another field

```
db.foo.find( { name : "bob" , $or : [ { a : 1 } , { b : 2 } ] } )
```

The `$or` operator retrieves matches for each or clause individually and eliminates duplicates when returning results.

> `$or` can be nested as of v2.0, however nested `$or` clauses are not handled as efficiently by the query optimizer as top level `$or` clauses.

$and

v2.0+

The `$and` operator lets you use boolean and in a query. You give `$and` an array of expressions, all of which must match to satisfy the query.

```
db.foo.insert( { a: [ 1, 10 ] } )
db.foo.find( { $and: [ { a: 1 }, { a: { $gt: 5 } } ] } )
```

In the above example documents with an element of a having a value of a equal to 1 and a value of a greater than 5 will be returned. Thus the inserted document will be returned given the multikey semantics of MongoDB.

$size

The `$size` operator matches any array with the specified number of elements. The following example would match the object `{a:["foo"]}`, since that array has just one element:

```
db.things.find( { a : { $size: 1 } } );
```

You cannot use `$size` to find a range of sizes (for example: arrays with more than 1 element). If you need to query for a range, create an extra `size` field that you increment when you add elements. Indexes cannot be used for the $size portion of a query, although if other query expressions are included indexes may be used to search for matches on that portion of the query expression.

$type

The `$type` operator matches values based on their BSON type.

```
db.things.find( { a : { $type : 2 } } ); // matches if a is a string
db.things.find( { a : { $type : 16 } } ); // matches if a is an int
```

Possible types are:

Type Name	Type Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular expression	11
JavaScript code	13
Symbol	14
JavaScript code with scope	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

For more information on types and BSON in general, see http://www.bsonspec.org.

Regular Expressions

You may use regexes in database query expressions:

```
db.customers.find( { name : /acme.*corp/i } );
db.customers.find( { name : { $regex : 'acme.*corp', $options: 'i' } } );
```

If you need to combine the regex with another operator, you need to use the `$regex` clause. For example, to find the customers where name matches the above regex but does not include `'acmeblahcorp'`, you would do the following:

```
db.customers.find( { name : { $regex : /acme.*corp/i, $nin : ['acmeblahcorp'] } } );
```

Note that `$regex` queries are escaped slightly differently than JavaScript native regular expressions. For example:

```
> db.x.insert({someId : "123[456]"})
> db.x.find({someId : /123\[456\]/}) // use "\" to escape
> db.x.find({someId : {$regex : "123\\[456\\]"}}) // use "\\" to escape
```

An index on the field queried by regexp can increase performance significantly, as follows:

- Simple prefix queries (also called rooted regexps) like `/^prefix/` will make efficient use of the index (much like most SQL databases that use indexes for a `LIKE 'prefix%'` expression). This only works if the expression is left-rooted and the `i` (case-insensitivity) flag is not used.
- All other queries will not make an efficient use of the index: all values in the index will be scanned and tested against the regular expression.

> While `/^a/`, `/^a.*/`, and `/^a.*$/` are equivalent, they will have different performance characteristics. The latter two will be slower as they have to scan the whole string. The first format can stop scanning after the prefix is matched.

MongoDB uses PCRE for regular expressions. Valid flags are:

- `i` - Case insensitive. Letters in the pattern match both upper and lower case letters.
- `m` - Multiline. By default, Mongo treats the subject string as consisting of a single line of characters (even if it actually contains newlines). The "start of line" metacharacter (^) matches only at the start of the string, while the "end of line" metacharacter ($) matches only at the end of the string, or before a terminating newline.
  When `m` it is set, the "start of line" and "end of line" constructs match immediately following or immediately before internal newlines in the subject string, respectively, as well as at the very start and end. If there are no newlines in a subject string, or no occurrences of ^ or $ in a pattern, setting `m` has no effect.
- `x` - Extended. If set, whitespace data characters in the pattern are totally ignored except when escaped or inside a character class. Whitespace does not include the VT character (code 11). In addition, characters between an unescaped # outside a character class and the next newline, inclusive, are also ignored.
  This option makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence `(?(` which introduces a conditional subpattern.
- `s` - Dot all. **New in 1.9.0**. Allows the dot (`.`) to match all characters including new lines. By default, `/a.*b/` will not match the string `"apple\nbanana"`, but `/a.*b/s` will.

Note that javascript regex objects only support the `i` and `m` options (and `g` which is ignored by mongod). Therefore if you want to use other options, you will need to use the $regex, $options query syntax.

Value in an Array

To look for the value "red" in an array field `colors`:

```
db.things.find( { colors : "red" } );
```

That is, when "colors" is inspected, if it is an array, each value in the array is checked. This technique may be mixed with the embedded object technique below.

$elemMatch

v1.4+

Use $elemMatch to check if an element in an array matches the specified match expression.

```
> t.find( { x : { $elemMatch : { a : 1, b : { $gt : 1 } } } } )
{ "_id" : ObjectId("4b5783300334000000000aa9"),
  "x" : [ { "a" : 1, "b" : 3 }, 7, { "b" : 99 }, { "a" : 11 } ]
}
```

Note that a single array element must match all the criteria specified; thus, the following query is semantically different in that each criteria can match a different element in the x array:

```
> t.find( { "x.a" : 1, "x.b" : { $gt : 1 } } )
```

See the dot notation page for more.

> You only need to use this when more than one field must be matched in the array element.

Value in an Embedded Object

For example, to look `author.name=="joe"` in a postings collection with embedded author objects:

```
db.postings.find( { "author.name" : "joe" } );
```

See the dot notation page for more.

Meta operator: `$not`

The `$not` meta operator can be used to negate the check performed by a standard operator. For example:

```
db.customers.find( { name : { $not : /acme.*corp/i } } );
```

```
db.things.find( { a : { $not : { $mod : [ 10 , 1 ] } } } );
```

The `$not` meta operator can **only** affect *other operators*. The following **do not** work. For such a syntax use the $ne operator.

```
db.things.find( { a : { $not : true } } ); // syntax error
```

> $not is not supported for regular expressions specified using the {`$regex: ...`} syntax. When using $not, all regular expressions should be passed using the native BSON type (e.g. {`"$not": re.compile("acme.*corp")`} in PyMongo)

Javascript Expressions and `$where`

In addition to the structured query syntax shown so far, you may specify query expressions as Javascript. To do so, pass a string containing a Javascript expression to `find()`, or assign such a string to the query object member `$where`. The database will evaluate this expression for each object scanned. When the result is true, the object is returned in the query results.

For example, the following mongo shell statements all do the same thing:

```
> db.myCollection.find( { a : { $gt: 3 } } );
> db.myCollection.find( { $where: "this.a > 3" } );
> db.myCollection.find("this.a > 3");
> f = function() { return this.a > 3; } db.myCollection.find(f);
```

You may mix mongo query expressions and a $where clause. In that case you must use the $where syntax, e.g.:

```
> db.myCollection.find({registered:true, $where:"this.a>3"})
```

Javascript executes more slowly than the native operators listed on this page, but is very flexible. See the server-side processing page for more information.

Cursor Methods

count()

The `count()` method returns the number of objects matching the query specified. It is specially optimized to perform the count in the MongoDB server, rather than on the client side for speed and efficiency:

```
nstudents = db.students.find({'address.state' : 'CA'}).count();
```

Note that you can achieve the same result with the following, but the following is slow and inefficient as it requires all documents to be put into memory on the client, and then counted. Don't do this:

```
nstudents = db.students.find({'address.state' : 'CA'}).toArray().length; // VERY BAD: slow and uses
excess memory
```

On a query using skip() and limit(), count ignores these parameters by default. Use count(true) to have it consider the skip and limit values in the calculation.

```
n = db.students.find().skip(20).limit(10).count(true);
```

limit()

`limit()` is analogous to the LIMIT statement in MySQL: it specifies a maximum number of results to return. For best performance, use `limit()` whenever possible. Otherwise, the database may return more objects than are required for processing.

```
db.students.find().limit(10).forEach( function(student) { print(student.name + "<p>"); } );
```

In the shell (and most drivers), a limit of `0` is equivalent to setting no limit at all.

skip()

The `skip()` expression allows one to specify at which object the database should begin returning results. This is often useful for implementing "paging". Here's an example of how it might be used in a JavaScript application:

```
function printStudents(pageNumber, nPerPage) {
   print("Page: " + pageNumber);
   db.students.find().skip((pageNumber-1)*nPerPage).limit(nPerPage).forEach( function(student) {
print(student.name + "<p>"); } );
}
```

Paging Costs
Unfortunately skip can be (very) costly and requires the server to walk from the beginning of the collection, or index, to get to the offset/skip position before it can start returning the page of data (limit). As the page number increases skip will become slower and more cpu intensive, and possibly IO bound, with larger collections.

Range based paging provides better use of indexes but does not allow you to easily jump to a specific page.

snapshot()

Indicates use of snapshot mode for the query. Snapshot mode assures no duplicates are returned, or objects missed, which were present at both the start and end of the query's execution (even if the object were updated). If an object is new during the query, or deleted during the query, it may or may not be returned, even with snapshot mode.

Note that short query responses (less than 1MB) are effectively snapshotted.

Currently, snapshot mode may not be used with sorting or explicit hints.

For more information, see How to do Snapshotted Queries in the Mongo Database.

sort()

`sort()` is analogous to the ORDER BY statement in SQL - it requests that items be returned in a particular order. We pass `sort()` a key pattern which indicates the desired order for the result.

```
db.myCollection.find().sort( { ts : -1 } ); // sort by ts, descending order
```

`sort()` may be combined with the `limit()` function. In fact, if you do not have a relevant index for the specified key pattern, `limit()` is recommended as there is a limit on the size of sorted results when an index is not used. Without a `limit()`, or an index, a full in-memory sort must be done. However, using a `limit()` reduces the required memory footprint and increases the speed of the operation by using an optimized sorting algorithm.

batchSize()

`batchSize()` determines the number of documents MongoDB returns in each batch to the client. MongoDB considers batchSize(), limit(), and the size in bytes of each document when deciding how many documents to send in each batch.

```
db.myCollection.find().batchSize(10);
```

The shell and most drivers present results to your application code as if they came in a single batch, so you normally do not need to think about batchSize.

Meta query operators

$returnKey

Only return the index key:

```
db.foo.find()._addSpecial("$returnKey" , true )
```

$maxScan

Limit the number of items to scan:

```
db.foo.find()._addSpecial( "$maxScan" , 50 )
```

$orderby

Sort results:

```
db.foo.find()._addSpecial( "$orderby", {x : -1} )
// same as
db.foo.find().sort({x:-1})
```

$explain

Explain the query instead of actually returning the results:

```
db.foo.find()._addSpecial( "$explain", true )
// same as
db.foo.find().explain()
```

$snapshot

Snapshot query:

```
db.foo.find()._addSpecial( "$snapshot", true )
// same as
db.foo.find().snapshot()
```

$min and $max

Set index bounds (see min and max Query Specifiers for details):

```
db.foo.find()._addSpecial("$min" , {x: -20})._addSpecial("$max" , { x : 200 })
```

$showDiskLoc

Show disk location of results:

```
db.foo.find()._addSpecial("$showDiskLoc" , true)
```

$hint

Force query to use the given index:

```
db.foo.find()._addSpecial("$hint", {_id : 1})
```

$comment

You can put a $comment field on a query to make looking in the profiler logs simpler.

```
db.foo.find()._addSpecial( "$comment" , "some comment to help find a query" )
```

```
```

`group()`

The `group()` method is analogous to GROUP BY in SQL. `group()` is more flexible, actually, allowing the specification of arbitrary reduction operations. See the Aggregation section of the Mongo Developers' Guide for more information.

See Also

- Indexes
- Optimizing Queries (including explain() and hint())

SDASDSSDD

Dot Notation (Reaching into Objects)

- Dot Notation vs. Subobjects
- Array Element by Position
- Matching with $elemMatch
- See Also

MongoDB is designed for storing JSON-style objects. The database understands the structure of these objects and can reach into them to evaluate query expressions.

Let's suppose we have some objects of the form:

```
> db.persons.findOne()
{ name: "Joe", address: { city: "San Francisco", state: "CA" } ,
  likes: [ 'scuba', 'math', 'literature' ] }
```

Querying on a top-level field is straightforward enough using Mongo's JSON-style query objects:

```
> db.persons.find( { name : "Joe" } )
```

But what about when we need to reach into embedded objects and arrays? This involves a bit different way of thinking about queries than one would do in a traditional relational DBMS. To reach into embedded objects, we use a "dot notation":

```
> db.persons.find( { "address.state" : "CA" } )
```

Reaching into arrays is implicit: if the field being queried is an array, the database automatically assumes the caller intends to look for a value within the array:

```
> db.persons.find( { likes : "math" } )
```

We can mix these styles too, as in this more complex example:

```
> db.blogposts.findOne()
{ title : "My First Post", author: "Jane",
  comments : [{ by: "Abe", text: "First" },
              { by : "Ada", text : "Good post" } ]
}
> db.blogposts.find( { "comments.by" : "Ada" } )
```

We can also create indexes of keys on these fields:

```
db.persons.ensureIndex( { "address.state" : 1 } );
db.blogposts.ensureIndex( { "comments.by" : 1 } );
```

Dot Notation vs. Subobjects

Suppose there is an author id, as well as name. To store the author field, we can use an object:

```
> db.blog.save({ title : "My First Post", author: {name : "Jane", id : 1}})
```

If we want to find any authors named Jane, we use the notation above:

```
> db.blog.findOne({"author.name" : "Jane"})
```

To match only objects with these exact keys and values, we use an object:

```
db.blog.findOne({"author" : {"name" : "Jane", "id" : 1}})
```

Note that

```
db.blog.findOne({"author" : {"name" : "Jane"}})
```

will not match, as subobjects have to match exactly (it would match an object with one field: `{"name" : "Jane"}`). Note that the embedded document must also have the same key order, so:

```
db.blog.findOne({"author" : {"id" : 1, "name" : "Jane"}})
```

will not match, either. This can make subobject matching unwieldy in languages whose default document representation is unordered.

Array Element by Position

Array elements also may be accessed by specific array position:

```
// i.e. comments[0].by == "Abe"
> db.blogposts.find( { "comments.0.by" : "Abe" } )
```

(The above examples use the mongo shell's Javascript syntax. The same operations can be done in any language for which Mongo has a driver available.)

Matching with $elemMatch

Using the $elemMatch query operator (mongod >= 1.3.1), you can match an entire document within an array. This is best illustrated with an example. Suppose you have the following two documents in your collection:

```

// Document 1
{ "foo" : [
      {
        "shape" : "square",
        "color" : "purple",
        "thick" : false
      },
      {
        "shape" : "circle",
        "color" : "red",
        "thick" : true
      }
] }


// Document 2
{ "foo" : [
      {
        "shape" : "square",
        "color" : "red",
        "thick" : true
      },
      {
        "shape" : "circle",
        "color" : "purple",
        "thick" : false
      }
] }
```

You want to query for a purple square, and so you write the following:

```

db.foo.find({"foo.shape": "square", "foo.color": "purple"})
```

The problem with this query is that it will match the second in addition to matching the first document. In other words, the standard query syntax won't restrict itself to a single document within the `foo` array. As mentioned above, subobjects have to match exactly, so

```

db.foo.find({foo: {"shape": "square", "color": "purple"} } )
```

won't help either, since there's a third attribute specifying thickness.

To match an entire document within the foo array, you need to use $elemMatch. To properly query for a purple square, you'd use $elemMatch like so:

```
db.foo.find({foo: {"$elemMatch": {shape: "square", color: "purple"}}})
```

The query will return the first document, which contains the purple square you're looking for.

See Also

- Advanced Queries
- Multikeys

Full Text Search in Mongo

- Introduction
- Multikeys (Indexing Values in an Array)
- Text Search
- Comparison to Full Text Search Engines
- Real World Examples

Introduction

Mongo provides some functionality that is useful for text search and tagging.

Multikeys (Indexing Values in an Array)

The Mongo multikey feature can automatically index arrays of values. Tagging is a good example of where this feature is useful. Suppose you have an article object/document which is tagged with some category names:

```
obj = {
  name: "Apollo",
  text: "Some text about Apollo moon landings",
  tags: [ "moon", "apollo", "spaceflight" ]
}
```

and that this object is stored in `db.articles`. The command

```
db.articles.ensureIndex( { tags: 1 } );
```

will index all the tags on the document, and create index entries for "moon", "apollo" and "spaceflight" for that document.

You may then query on these items in the usual way:

```
> print(db.articles.findOne( { tags: "apollo" } ).name);
Apollo
```

The database creates an index entry for each item in the array. Note an array with many elements (hundreds or thousands) can make inserts very expensive. (Although for the example above, alternate implementations are equally expensive.)

Text Search

It is fairly easy to implement basic full text search using multikeys. What we recommend is having a field that has all of the keywords in it, something like:

```
{ title : "this is fun" ,
  _keywords : [ "this" , "is" , "fun" ]
}
```

Your code must split the title above into the keywords before saving. Note that this code (which is not part of Mongo DB) could do stemming, etc. too. (Perhaps someone in the community would like to write a standard module that does this...)

Comparison to Full Text Search Engines

MongoDB has interesting functionality that makes certain search functions easy. That said, it is not a dedicated full text search engine.

For example, dedicated engines provide the following capabilities:

- built-in text stemming
- ranking of queries matching various numbers of terms (can be done with MongoDB, but requires user supplied code to do so)
- bulk index building

Bulk index building makes building indexes fast, but has the downside of not being realtime. MongoDB is particularly well suited for problems where the search should be done in realtime. Traditional tools are often not good for this use case.

Real World Examples

The Business Insider web site uses MongoDB for its blog search function in production.

Mark Watson's opinions on Java, Ruby, Lisp, AI, and the Semantic Web - A recipe example in Ruby.

Full text search with MongoDB at Flowdock

min and max Query Specifiers

The `min()` and `max()` functions may be used in conjunction with an index to constrain query matches to those having index keys between the min and max keys specified. The `min()` and `max()` functions may be used individually or in conjunction. The index to be used may be specified with a `hint()` or one may be inferred from pattern of the keys passed to `min()` and/or `max()`.

```
db.f.find().min({name:"barry"}}.max({name:"larry"}).hint({name:1});
db.f.find().min({name:"barry"}}.max({name:"larry"});
db.f.find().min({last_name:"smith",first_name:"john"}};
```

> The currently supported way of using this functionality is as describe above. We document hereafter a way that we could potentially support in the future.

If you're using the standard query syntax, you must distinguish between the `$min` and `$max` keys and the query selector itself. See here:

```
db.f.find({$min: {name:"barry"}, $max: {name:"larry"}, $query:{}});
```

The min() value is included in the range and the max() value is excluded.

Normally, it is much preferred to use $gte and $lt rather than to use min and max, as min and max require a corresponding index. Min and max are primarily useful for compound keys: it is difficult to express the last_name/first_name example above without this feature (it can be done using $where).

min and max exist primarily to support the mongos (sharding) process.

OR operations in query expressions

Query objects in Mongo by default AND expressions together.

$in

The $in operator indicates a "where value in ..." expression. For expressions of the form x == a OR x == b, this can be represented as

```
{ x : { $in : [ a, b ] } }
```

$or

v1.5.3+

The `$or` operator lets you use a boolean or expression to do queries. You give $or a list of expressions, any of which can satisfy the query.

Simple:

```
db.foo.find( { $or : [ { a : 1 } , { b : 2 } ] } )
```

With another field

```
db.foo.find( { name : "bob" , $or : [ { a : 1 } , { b : 2 } ] } )
```

The `$or` operator retrieves matches for each or clause individually and eliminates duplicates when returning results.

$where

We can provide arbitrary Javascript expressiosn to the server via the $where operator. This provides a means to perform OR operations. For example in the mongo shell one might invoke:

```
db.mycollection.find( { $where : function() { return this.a == 3 || this.b == 4; } } );
```

The following syntax is briefer and also works; however, if additional structured query components are present, you will need the $where form:

```
db.mycollection.find( function() { return this.a == 3 || this.b == 4; } );
```

See Also

- Advanced Queries

Queries and Cursors

Queries to MongoDB return a cursor, which can be iterated to retrieve results. The exact way to query will vary with language driver. Details below focus on queries from the MongoDB shell (i.e. the `mongo` process).

The shell `find()` method returns a cursor object which we can then iterate to retrieve specific documents from the result. We use `hasNext()` and `next()` methods for this purpose.

```
for( var c = db.parts.find(); c.hasNext(); ) {
   print( c.next());
}
```

Additionally in the shell, `forEach()` may be used with a cursor:

```
db.users.find().forEach( function(u) { print("user: " + u.name); } );
```

Topics:

Array Mode in the Shell

Note that in some languages, like JavaScript, the driver supports an "array mode". Please check your driver documentation for specifics.

In the db shell, to use the cursor in array mode, use array index [] operations and the `length` property.

Array mode will load all data into RAM up to the highest index requested. Thus it should ***not*** be used for any query which can return very large amounts of data: you will run out of memory on the client.

You may also call `toArray()` on a cursor. `toArray()` will load all objects queries into RAM.

Getting a Single Item

The shell `findOne()` method fetches a single item. Null is returned if no item is found.

`findOne()` is equivalent in functionality to:

```
function findOne(coll, query) {
    var cursor = coll.find(query).limit(1);
    return cursor.hasNext() ? cursor.next() : null;
}
```

Tip: If you only need one row back and multiple match, `findOne()` is efficient, as it performs the `limit()` operation, which limits the objects returned from the database to one.

Querying Embedded Objects

To find an exact match of an entire embedded object, simply query for that object:

```
db.order.find( { shipping: { carrier: "usps" } } );
```

The above query will work if { carrier: "usps" } is an exact match for the entire contained shipping object. If you wish to match any sub-object with `shipping.carrier == "usps"`, use this syntax:

```
db.order.find( { "shipping.carrier" : "usps" } );
```

See the dot notation docs for more information.

Greater Than / Less Than

```
db.myCollection.find( { a : { $gt : 3 } } );
db.myCollection.find( { a : { $gte :3 } } );
db.myCollection.find( { a : { $lt :3 } } );
db.myCollection.find( { a : { $lte :3 } } ); // a <= 3
```

Latent Cursors and Snapshotting

A latent cursor has (in addition to an initial access) a latent access that occurs after an intervening write operation on the database collection (i.e., an insert, update, or delete). Under most circumstances, the database supports these operations.

Conceptually, a cursor has a current position. If you delete the item at the current position, the cursor automatically skips its current position forward to the next item.

MongoDB cursors do not provide a snapshot: if other write operations occur during the life of your cursor, it is unspecified if your application will see the results of those operations or not. See the snapshot docs for more information.

Execution of queries in batches

The MongoDB server returns query results to the client in batches. You can modify this behavior in two ways: You can specify batchSize(), which tells the server how many documents to return in each batch, or limit(), which determines the total number of documents to return for this query. (See Advanced Queries for setting limit and batchSize.)

If limit and batchSize are not specified, the first batch contains 101 documents, or enough documents to exceed 1 MB, whichever comes first. Otherwise, the server returns enough documents to satisfy the lesser of the batchSize or the limit. If the query matches more than that quantity of results and you would like them all to be returned, you need to either specify a larger limit or batchSize, or iterate through the result set to retrieve all results. Iterating a cursor will return enough documents in each batch to satisfy batchSize or to exceed 4 MB, whichever comes first.

A special case is if you sort a set of documents without an index. In that case, since MongoDB must load all the documents in order to sort them in memory, so it returns them all in the first batch.

Regardless of limit and batchSize, no batch will contain more than enough documents to exceed 4 MB.

Examples:

```
> // Insert 200 small documents
> for (var i = 0; i < 200; i++) { db.foo.insert({i: i}); }
> var cursor = db.foo.find()
> // Default batch size is 101 documents
> cursor.objsLeftInBatch()
101
> // Adding a large limit lets you get all docs at once
> var cursor = db.foo.find().limit(1000)
> cursor.objsLeftInBatch()
200
> // A small batchSize() can override the limit
> var cursor = db.foo.find().batchSize(10).limit(1000)
> cursor.objsLeftInBatch()
10
> // A small limit can override the batchSize
> var cursor = db.foo.find().batchSize(10).limit(5)
> cursor.objsLeftInBatch()
5
> // Insert 10 documents of one megabyte each
> var megabyte = '';
> for (var i = 0; i < 1024 * 1024; i++) { megabyte += 'a'; }
> for (var i = 0; i < 10; i++) { db.bar.insert({s:megabyte}); }
> // First batch stops after 1 MB
> var cursor = db.bar.find()
> cursor.objsLeftInBatch()
1
```

Performance implications

If, for example, you do

```
cursor = db.foo.find( { x : 1 } )
for ( i=0; i<100; i++ ) {
 printjson( cursor.next() );
}
```

The server will only find the first the first 100 results. If the result set is large, finding the first 100 results may be much faster than finding all results and printing the first 100.

Note that counts are performed against the entire result set, so for example

```
db.foo.find( { x : 1 } ).count()
```

Could be much slower than finding the first 100 results above.

Auditing allocated cursors

Information on allocated cursors may be obtained using the {`cursorInfo:1`} command.

```
db.runCommand({cursorInfo:1})
```

Closing and Timeouts

By default a cursor will timeout after 10 minutes of inactivity. The server will close the cursor if it isn't accessed in that time or it has been exhausted.

You can specify NoTimeout optionally for a query. If you do this please be careful to close the cursor manually – otherwise they will consume memory on the server.

See Also

- Advanced Queries

* Multikeys in the HowTo
* Mongo Wire Protocol

Tailable Cursors

> Tailable cursors are only allowed on capped collections and can only return objects in natural order. Tailable queries never use indexes.

A tailable cursor "tails" the end of a capped collection, much like the Unix "tail -f" command. They key idea is that if we "catch up" and have reached the end of the collection, our position is remembered rather than the cursor being closed. Thus, after new objects are inserted, we can resume retrieving from where we left off – which is then very inexpensive.

If the field you wish to "tail" is indexed, do not use tailable cursors; instead simply (re)query for { *field* : { $gt : *value* } } where `value` is where you last left off. This is normally quite efficient. Tailable cursors are for cases where having an index would be prohibitive (extremely high write collections). If performance is not problematic, use a normal query and cursor, tailable adds some complexity.

As no index will be used for the query, the initial scanning to find the first object to return will likely be quite costly. However once found, retrieving additional data from new inserts then becomes very inexpensive.

The cursor may become invalid if, for example, the last object returned is at the end of the collection and is deleted. Thus, you should be prepared to requery if the cursor is "dead". You can determine if a cursor is dead by checking its id. An id of zero indicates a dead cursor (use isDead in the c++ driver). In addition, the cursor will be in "dead" state after a query which returns no matches.

MongoDB replication uses tailable cursors to follow the end of the primary's replication op log collection. Writes to the oplog would be slower with an index. The tailable feature eliminates the need to create an index for replication's use case.

C++ example:

```
#include "client/dbclient.h"

using namespace mongo;

/* "tail" the namespace, outputting elements as they are added. Cursor blocks
 * waiting for data if no documents currently exist. For this to work something
 * field -- _id in this case -- should be increasing when items are added.
 */
void tail(DBClientBase& conn, const char *ns) {
  // minKey is smaller than any other possible value
BSONElement lastId = minKey.firstElement();

  Query query = Query().sort("$natural"); // { $natural : 1 } means in forward
// capped collection insertion order
while( 1 ) {
    auto_ptr<DBClientCursor> c =
      conn.query(ns, query, 0, 0, 0,
                 QueryOption_CursorTailable | QueryOption_AwaitData );
    while( 1 ) {
      if( !c->more() ) {
        if( c->isDead() ) {
          // we need to requery
break;
        }
        // No need to wait here, cursor will block for several sec with _AwaitData
 continue; // we will try more() again
}
      BSONObj o = c->next();
      lastId = o["_id"];
      cout << o.toString() << endl;
    }

    // prepare to requery from where we left off
query = QUERY( "_id" << GT << lastId ).sort("$natural");
  }
}
```

Javascript example:

```
var coll = db.some.capped.collection;
var lastVal = coll.find().sort({ '$natural' : 1 })
                         .limit( 1 ).next()[ 'increasing' ];
while(1){

  cursor = coll.find({ 'increasing' : { '$gte' : lastVal } });

  // tailable
cursor.addOption( 2 );
  // await data
cursor.addOption( 32 );

  // Waits several sec for more data
while( cursor.hasNext() ){
    var doc = cursor.next();
    lastVal = doc[ 'increasing' ];
    printjson( doc );
  }

}
```

See Also

- A detailed blog post on tailable cursors
- http://github.com/mongodb/mongo-snippets/blob/master/cpp-examples/tailable_cursor.cpp
- http://api.mongodb.org/cplusplus/2.0.3/dbclient_8h_source.html

Server-side Code Execution

Mongo supports the execution of code inside the database process.

Map/Reduce

MongoDB supports Javascript-based map/reduce operations on the server. See the map/reduce documentation for more information.

Using `db.eval()`

> Use map/reduce instead of db.eval() for long running jobs. db.eval blocks other operations!

`db.eval()` is used to evaluate a function (written in JavaScript) at the database server.
This is useful if you need to touch a lot of data lightly. In that scenario, network transfer of the data could be a bottleneck.

`db.eval()` returns the return value of the function that was invoked at the server. If invocation fails an exception is thrown.

For a trivial example, we can get the server to add 3 to 3:

```
> db.eval( function() { return 3+3; } );
6
>
```

Let's consider an example where we wish to erase a given field, `foo`, in every single document in a collection. A naive client-side approach would be something like

```
function my_erase() {
  db.things.find().forEach( function(obj) {
                     delete obj.foo;
                     db.things.save(obj);
                   } );
}

my_erase();
```

Calling `my_erase()` on the client will require the entire contents of the collection to be transmitted from server to client and back again.

Instead, we can pass the function to `eval()`, and it will be called in the runtime environment of the server. On the server, the `db` variable is set to the current database:

```
db.eval(my_erase);
```

Examples

```
> myfunc = function(x){ return x; };

> db.eval( myfunc, {k:"asdf"} );
{ k : "asdf"  }

> db.eval( myfunc, "asdf" );
"asdf"

> db.eval( function(x){ return x; }, 2 );
2.0
```

If an error occurs on the evaluation (say, a null pointer exception at the server), an exception will be thrown of the form:

```
{ dbEvalException: { errno : -3.0 , errmsg : "invoke failed" , ok : 0.0  } }
```

Example of using `eval()` to do equivalent of the Mongo `count()` function:

```
function mycount(collection) {
  return db.eval( function(){return db[collection].find({},{_id:ObjId()}).length();} );
}
```

Example of using `db.eval()` for doing an atomic increment, plus some calculations:

```
function inc( name , howMuch ){
    return db.eval(
        function(){
            var t = db.things.findOne( { name : name } );
            t = t || { name : name , num : 0 , total : 0 , avg : 0 };
            t.num++;
            t.total += howMuch;
            t.avg = t.total / t.num;
            db.things.save( t );
            return t;
        }
    );
}

db.things.remove( {} );
print( tojson( inc( "eliot" , 2 )) );
print( tojson( inc( "eliot" , 3 )) );
```

Limitations of `eval`

Write locks

It's important to be aware that by default `eval` takes a write lock. This means that you can't use `eval` to run other commands that themselves take a write lock. To take an example, suppose you're running a replica set and want to add a new member. You may be tempted to do something like this from a driver:

```
db.eval("rs.add('ip-address:27017')");
```

As we just mentioned, `eval` will take a write lock on the current node. Therefore, this won't work because you can't add a new replica set member if any of the existing nodes is write-locked.

The proper approach is to run the commands to add a node manually. `rs.add` simply queries the `local.system.replSet` collection, updates the config object, and run the `replSetReconfig` command. You can do this from the driver, which, in addition to not taking out the `eval` write lock, manages to more directly perform the operation.

In 1.7.2, a `nolock` option was added to `eval`. To use `nolock` you have to use the command interface directly:

```
db.runCommand({$eval: function() {return 42;}, nolock: true})
```

or with args

```
db.runCommand({$eval: function(x,y) {return x*y;}, args: [6,7], nolock: true})
```

Sharding

Note also that `eval` doesn't work with sharding. If you expect your system to later be sharded, it's probably best to avoid `eval` altogether.

Storing functions server-side

Note: we recommend *not* using server-side stored functions when possible. As these are code it is likely best to store them with the rest of your code in a version control system.

There is a special system collection called `system.js` that can store JavaScript functions to be reused. To store a function, you would do:

```
db.system.js.save( { _id : "foo" , value : function( x , y ){ return x + y; } } );
```

_id is the name of the function, and is unique per database.

Once you do that, you can use `foo` from any JavaScript context (db.eval, $where, map/reduce)

Here is an example from the shell:

```
>db.system.js.save({ "_id" : "echo", "value" : function(x){return x;} })
>db.eval("echo('test')")
test
```

See http://github.com/mongodb/mongo/tree/master/jstests/storefunc.js for a full example.

In MongoDB 2.1 you will also be able to load all the scripts saved in `db.system.js` into the shell using `db.loadServerScripts()`

```
>db.loadServerScripts()
>echo(3)
3
```

`$where` Clauses and Functions in Queries

In addition to the regular document-style query specification for `find()` operations, you can also express the query either as a string containing a SQL-style WHERE predicate clause, or a full JavaScript function. **Note: if a normal data-driven BSON query expression is possible, use that construction. Use $where only when you must it is significantly slower.**

When using this mode of query, the database will call your function, or evaluate your predicate clause, for each object in the collection.

In the case of the string, you must represent the object as "this" (see example below). In the case of a full JavaScript function, you use the normal JavaScript function syntax.

The following four statements in mongo - The Interactive Shell are equivalent:

```
db.myCollection.find( { a : { $gt: 3 } } );
db.myCollection.find( { $where: "this.a > 3" });
db.myCollection.find( "this.a > 3" );
db.myCollection.find( { $where: function() { return this.a > 3;}});
```

The first statement is the preferred form. It will be faster to execute because the query optimizer can easily interpret that query and choose an index to use.

You may mix data-style find conditions and a function. This can be advantageous for performance because the data-style expression will be evaluated first, and if not matched, no further evaluation is required. Additionally, the database can then consider using an index for that condition's field. To mix forms, pass your evaluation function as the `$where` field of the query object. For example, both of the following would work:

```
db.myCollection.find( { active: true, $where: function() { return obj.credits - obj.debits < 0; } } );
db.myCollection.find( { active: true, $where: "this.credits - this.debits < 0" } );
```

Do not write to the database from a `$where` expression.

Notes on Concurrency

If you don't use the "nolock" flag, db.eval() blocks the entire mongod process while running. Thus, its operations are atomic but prevent other operations from processing.

When more concurrency is needed consider using map/reduce instead of eval().

Running .js files via a mongo shell instance on the server

This is a good technique for performing batch administrative work. Run `mongo` on the server, connecting via the `localhost` interface. The connection is then very fast and low latency. This is friendlier than db.eval() as db.eval() blocks other operations.

Sorting and Natural Order

"Natural order" is defined as the database's native ordering of objects in a collection.

When executing a `find()` with no parameters, the database returns objects in forward natural order.

For standard tables, natural order is not particularly useful because, although the order is often close to insertion order, it is not *guaranteed* to be. However, for Capped Collections, natural order is guaranteed to be the insertion order. This can be very useful.

In general, the natural order feature is a very efficient way to store and retrieve data in insertion order (much faster than say, indexing on a timestamp field). But remember, the collection must be capped for this to work.

In addition to forward natural order, items may be retrieved in reverse natural order. For example, to return the 50 most recently inserted items (ordered most recent to less recent) from a capped collection, you would invoke:

```
> c=db.cappedCollection.find().sort({$natural:-1}).limit(50)
```

Sorting can also be done on arbitrary keys in any collection. For example, this sorts by 'name' ascending, then 'age' descending:

```
> c=db.collection.find().sort({name : 1, age : -1})
```

See Also

- The Capped Collections section of this Guide
- Advanced Queries
- The starting point for all Home

Aggregation

- Aggregation Framework
- Count
- Distinct
- Group
    - Examples
    - Using Group from Various Languages
- Map/Reduce
- See Also

Mongo includes utility functions which provide server-side `count, distinct,` and `group by` operations. More advanced aggregate functions can be crafted using MapReduce.

Aggregation Framework

v2.2+

This is a new module in the v2.2 release. You can test it via v2.1.0 in the 2.1 development (unstable) branch. Aggregation Framework

Count

`count()` returns the number of objects in a collection or matching a query. If a document selector is provided, only the number of matching documents will be returned.

`size()` is like `count()` but takes into consideration any limit() or skip() specified for the query.

```
db.collection.count(selector);
```

For example:

```
print( "# of objects: " + db.mycollection.count() );
print( db.mycollection.count( {active:true} ) );
```

`count` is faster if an index exists for the condition in the selector. For example, to make the count on `active` fast, invoke

```
db.mycollection.ensureIndex( {active:1} );
```

Distinct

The distinct command returns returns a list of distinct values for the given `key` across a collection.

Command is of the form:

```
{ distinct : <collection_name>, key : <key>[, query : <query>] }
```

although many drivers have a helper function for distinct.

```
> db.addresses.insert({"zip-code": 10010})
> db.addresses.insert({"zip-code": 10010})
> db.addresses.insert({"zip-code": 99701})

> // shell helper:
> db.addresses.distinct("zip-code");
[ 10010, 99701 ]

> // running as a command manually:
> db.runCommand( { distinct: 'addresses', key: 'zip-code' } )
{ "values" : [ 10010, 99701 ], "ok" : 1 }
```

`distinct` may also reference a nested key:

```
> db.comments.save({"user": {"points": 25}})
> db.comments.save({"user": {"points": 31}})
> db.comments.save({"user": {"points": 25}})

> db.comments.distinct("user.points");
[ 25, 31 ]
```

You can add an optional query parameter to distinct as well

```
> db.address.distinct( "zip-code" , { age : 30 } )
```

Note: the distinct command results are returned as a single BSON object. If the results could be large (> max document size – 4/16MB), use map/reduce instead.

> **Covered Index Use**
> Starting with 1.7.3 distinct can use an index not only to find the documents in the query, but also to return the data.

Group

Note: currently one must use map/reduce instead of group() in sharded MongoDB configurations.

`group` returns an array of grouped items. The command is similar to SQL's group by. The SQL statement

```
select a,b,sum(c) csum from coll where active=1 group by a,b
```

corresponds to the following in MongoDB:

```
db.coll.group(
           {key: { a:true, b:true },
            cond: { active:1 },
            reduce: function(obj,prev) { prev.csum += obj.c; },
            initial: { csum: 0 }
            });
```

Note: the result is returned as a single BSON object and for this reason must be fairly small – less than 10,000 keys, else you will get an exception. For larger grouping operations without limits, please use map/reduce .

`group` takes a single object parameter containing the following fields:

- *`key`*: Fields to group by.
- *`reduce`*: The `reduce` function aggregates (reduces) the objects iterated. Typical operations of a reduce function include summing and counting. `reduce` takes two arguments: the current document being iterated over and the aggregation counter object. In the example above, these arguments are named `obj` and `prev`.
- *`initial`*: initial value of the aggregation counter object.
- *`keyf`*: An optional **function** returning a "key object" to be used as the grouping key. Use this instead of *`key`* to specify a key that is not a single/multiple existing fields. Could be used to group by day of the week, for example. Set in lieu of *`key`*.
- *`cond`*: An optional condition that must be true for a row to be considered. This is essentially a `find()` query expression object. If null, the reduce function will run against all rows in the collection.
- *`finalize`*: An optional function to be run on each item in the result set just before the item is returned. Can either modify the item (e.g., add an average field given a count and a total) or return a replacement object (returning a new object with just _id and average fields). See jstests/group3.js for examples.

To order the grouped data, simply sort it client-side upon return. The following example is an implementation of `count()` using `group()`.

```
function gcount(collection, condition) {
  var res =
    db[collection].group(
           { key: {},
             initial: {count: 0},
             reduce: function(obj,prev){ prev.count++;},
             cond: condition } );
  // group() returns an array of grouped items.  here, there will be a single
// item, as key is {}.
return res[0] ? res[0].count : 0;
}
```

Examples

The examples assume data like this:

```
{ domain: "www.mongodb.org"
, invoked_at: {d:"2009-11-03", t:"17:14:05"}
, response_time: 0.05
, http_action: "GET /display/DOCS/Aggregation"
}
```

Show me stats for each http_action in November 2009:

```
db.test.group(
   { cond: {"invoked_at.d": {$gte: "2009-11", $lt: "2009-12"}}
   , key: {http_action: true}
   , initial: {count: 0, total_time:0}
   , reduce: function(doc, out){ out.count++; out.total_time+=doc.response_time }
   , finalize: function(out){ out.avg_time = out.total_time / out.count }
   } );

[
  {
    "http_action" : "GET /display/DOCS/Aggregation",
    "count" : 1,
    "total_time" : 0.05,
    "avg_time" : 0.05
  }
]
```

Show me stats for each domain for each day in November 2009:

```
db.test.group(
   { cond: {"invoked_at.d": {$gte: "2009-11", $lt: "2009-12"}}
   , key: {domain: true, invoked_at.d: true}
   , initial: {count: 0, total_time:0}
   , reduce: function(doc, out){ out.count++; out.total_time+=doc.response_time }
   , finalize: function(out){ out.avg_time = out.total_time / out.count }
   } );

[
  {
    "http_action" : "GET /display/DOCS/Aggregation",
    "count" : 1,
    "total_time" : 0.05,
    "avg_time" : 0.05
  }
]
```

Using Group from Various Languages

Some language drivers provide a group helper function. For those that don't, one can manually issue the db command for group. Here's an example using the Mongo shell syntax:

```
> db.foo.find()
{"_id" :  ObjectId( "4a92af2db3d09cb83d985f6f")  , "x" : 1}
{"_id" :  ObjectId( "4a92af2fb3d09cb83d985f70")  , "x" : 3}
{"_id" :  ObjectId( "4a92afdab3d09cb83d985f71")  , "x" : 3}

> db.$cmd.findOne({group : {
... ns : "foo",
... cond : {},
... key : {x : 1},
... initial : {count : 0},
... $reduce : function(obj,prev){prev.count++;}}})
{"retval" : [{"x" : 1 , "count" : 1},{"x" : 3 , "count" : 2}] , "count" : 3 , "keys" : 2 , "ok" : 1}
```

If you use the database command with `keyf` (instead of `key`) it must be prefixed with a $. For example:

```
db.$cmd.findOne({group : {
... ns : "foo",
... $keyf : function(doc) { return {"x" : doc.x}; },
... initial : {count : 0},
... $reduce : function(obj,prev) { prev.count++; }}})
```

Map/Reduce

MongoDB provides a MapReduce facility for more advanced aggregation needs. CouchDB users: please note that basic queries in MongoDB do not use map/reduce.

See Also

- jstests/eval2.js for an example of group() usage
- Advanced Queries

Aggregation Framework

The Aggregation Framework was first available in MongoDB 2.1.0. See an improved version of this documentation in the MongoDB Manual:

- Aggregation Documentation
- Aggregation Framework Operators

Contents:

Related Sub-Topics:

- Aggregation Framework - $group
- Aggregation Framework - $limit
- Aggregation Framework - $match
- Aggregation Framework - $project
- Aggregation Framework - $skip
- Aggregation Framework - $sort
- Aggregation Framework - $unwind
- Aggregation Framework - Expression Reference
- SQL to Aggregation Framework Mapping Chart

Overview

The MongoDB aggregation framework provides a means to calculate aggregate values without having to use Map-Reduce. Map-Reduce is very powerful, but it is also harder than necessary for simpler things such as totaling or averaging field values. For those familiar with SQL, the aggregation framework can be used to do the kind of thing that SQL does with group-by and distinct, as well as some simple forms of self-joins.

The aggregation framework also provides projection facilities that can be used to reshape data. This includes the ability to add computed fields, to create new virtual sub-objects, and to extract sub-fields and bring them to the top-level of results.

An introductory presentation from MongoSV 2011 is available here.

Using the Aggregation Framework

The aggregation framework relies on two key concepts: pipelines and expressions.

Pipelines

A pipeline is a sequence of operations that is applied to a stream of documents. For those familiar with linux command line shells, this is very similar to a pipe. In a linux shell, a pipe is a series of programs that each process a stream of characters. The MongoDB aggregation pipeline is a series of pipeline operators, each of which processes a stream of documents. Logically, the pipeline behaves as if a collection is being scanned, and each document found is passed into the top of the pipeline. Each operator in the pipeline can transform each document as it goes by, until they appear at the end of the pipeline. Pipeline operators need not produce one output document for every input document: operators may also generate new documents, or filter out documents, so that they do not go any further in the pipeline.

Pipeline Operators

The links in the list below link to detailed descriptions of each operator.

- $project - select columns or sub-columns, create computed values or sub-objects
- $match - filter documents out from the document stream
- $limit - limit the number of documents that pass through the document stream
- $skip - skip over a number of documents that pass through the document stream
- $unwind - unwind an array, subsituting each value in the array for the array within the same document
- $group - group documents by key and calculate aggregate values for the group
- $sort - sort documents by key

Pipeline operators appear in an array. Documents pass through these operators and come out at the other end.

> The pipeline cannot operate on collects with documents that contain one of the following "special fields:" `MinKey`, `MaxKey`, `EOO`, `Undefined`, `DBRef`, `Code`.

Expressions

Expressions are used to calculate values. In keeping with MongoDB's JSON heritage, expressions are defined in a prefix format using JSON.

Expressions are usually stateless, and are just evaluated when they are seen. These can do things such as adding the values of two fields together, or extracting the year from a date.

It is also possible to use accumulator expressions which retain state. These types of expressions are used in the $group operator in order to maintain counts, totals, maxima, etc, as documents go through the pipeline.

There are many examples in the operator documentation found above. For the complete list, see the Aggregation Framework - Expression Reference.

Invocation

Aggregation is invoked as a command with two operands:

- aggregate - provide the name of the collection to use at the head of the pipeline
- pipeline - an array of pipeline operators, each with its own operands; see the examples in the pipeline operator references above

As a command, invocation of the aggregation pipeline is the same in all drivers. Use your host programming language to create a database object, with the fields above, and then submit it as a command.

Here are some examples of pipelines that can be issued from the mongo shell. For the examples that follow, imagine an article collection made up of documents that look like this:

```
{
    title : "this is my title" ,
    author : "bob" ,
    posted : new Date() ,
    pageViews : 5 ,
    tags : [ "fun" , "good" , "fun" ] ,
    comments : [
        { author :"joe" , text : "this is cool" } ,
        { author :"sam" , text : "this is bad" }
    ],
    other : { foo : 5 }
});
```

The following example pivots data to create a set of author names grouped by tags applied to an article:

```
db.article.aggregate(
    { $project : {
 author : 1,
 tags : 1,
    }},
    { $unwind : "$tags" },
    { $group : {
 _id : { tags : 1 },
 authors : { $addToSet : "$author" }
    }}
);
```

Result Format

The result of a successful aggregation command is an document with two fields:

- result - an array of documents that came out of the pipeline
- ok - a field with the value 1, indicating success, or another value if there was an error

As a document, the result is subject to the current BSON Document size limit; see Maximum Document Size. If you expect a large result, use the $out pipeline operator to write it to a collection.

Optimizing Performance

Early Filtering

Logically, operation proceeds as if a collection scan is done to feed documents into the pipeline. For some pipelines, this may not be optimal.

If your aggregation operation does not require all of the data in a collection, you are likely to use a $match to filter out items you do not want to include. The aggregation framework recognizes matches, and will attempt to find a suitable index to use to access the matching elements of the collection. In the simplest case, where a $match appears first in the pipeline, pipeline will be fed with the result of a query (see Querying and Advanced Queries).

In order to take advantage of this, before execution, an optimization phase will try to re-arrange the pipeline so that any $match operators are moved as far towards the beginning as possible. As a simple example, if a pipeline begins with a $project that just renames fields, followed by a $match, the $match can be pushed in front of the $project by renaming the fields appropriately.

Over time, we expect to apply more of these kinds of optimizations, but is the initial release, put $match operators at the start of your pipeline whenever possible.

Memory for Cumulative Operators

Certain pipeline operators need to see their entire input set before they can produce any output. For example, $sort must see all of its input before producing its first output document. The current implementation does not go to disk in such cases, and all of the input must fit in memory to be sorted.

$group has similar characteristics, and must also see all of its input before anything can be produced. However, this usually doesn't require as much memory as sorting, because only one record needs to be kept for each unique key in the grouping specification.

The current implementation will log a warning if a cumulative operator consumes 5% or more of the physical memory on the host. Cumulative operators will signal an error if they consume 10% or more of the physical memory on the host.

Sharded Operation

The aggregation framework can be used on sharded collections.

When the source collection is sharded, the aggregation pipeline will be split into two parts. All of the operators up to and including the first $group or $sort are pushed to each shard. (If an early $match can exclude shards through the use of the shard key in the predicate, then these operators are only pushed to the relevant shards.) A second pipeline, consisting of the first $group or $sort and any remaining pipeline operators, is executed in mongos, using the results received from the shards.

For $sort, the results are merged. For $group, any "sub-totals" are brought in and combined; in some cases these may be structures. For example, the $avg expression maintains a total and count for each shard; these are combined in mongos and then divided.

> Using the aggregation framework in sharded operations will cause `mongos` instances to use more CPU resources than in previous versions, or without the aggregation framework. This modified performance profile may dictate alternate architecture decisions if you make use the aggregation framework extensively in a sharded environment.

Aggregation Framework - $group

Overview

In its most general form, $group is used to group documents together for the purpose of calculating an aggregate value based on their content. In practical terms, this could be something like calculating the average number of page views for each web page on a web site by day, or it could be something simpler like counting up how many distinct pages there are.

Specification

Grouping things together depends on defining how to bucket things into groups. This is done by defining an _id field for the group. This can be a single input field from the document stream, or a composite.

Once documents are put into a group, computed fields can be defined that aggregate values from members of the group. This is done using aggregate functions, which can find the sum, averge, minimum, maximum, etc, of values from the group.

An example will illustrate these features. Imagine a content management system that uses a collection of articles that looks something like this:

```
{
 author : "Dave Stevenson" ,
 body : "... content goes here ..." ,
 pageViews : 1200 ,
 tags : ['humor', 'baseball']
}
```

We can use $group to count up the number of articles each author has written, as well as find out how many cumulative page views each author has:

```
db.article.aggregate(
    { $group : {
 _id : "$author",
 docsPerAuthor : { $sum : 1 } ,  /* add one to "docsPerAuthor" for every document by this (_id) author
*/
 viewsPerAuthor : { $sum : "$pageViews" } /* sum the pageViews for every document by this (_id) author
*/
    }}
);
```

Each field created by the group must use one of the group aggregation functions (below) to aggregate the values for the documents in the group.

Specifying the Grouping Key Using _id

A $group operator must always have an _id field. This specifies the key to group things by. Since documents are grouped by this, it is naturally unique. (The grouping ensures that each bucket only occurs once.) Using _id is convenient if the pipeline later includes an [$out] stage, which will require a unique _id.

_id can be a dotted field path reference (prefixed with a dollar sign, '$'), a braced document expression containing multiple fields (an order-preserving concatenated key), or a single constant. Using a constant will create a single bucket, and can be used to count documents, or to add all the values for a field in all the documents in a collection.

If you need the output to have a different name, it can easily be renamed using a simple $project after the $group.

Group Aggregation Functions

$addToSet takes a field and returns an array of all the values found in that field among the documents in that group, except each unique value will only appear once.

$first remembers the first value it sees for its field-argument. This is only useful if there is a $sort ahead of the $group, otherwise the results are unpredictable.

$last remembers the last value it sees for its field-argument. This is only useful if there is a $sort ahead of the $group, otherwise the results are unpredictable.

$max takes a field and finds the highest value stored in that field among the documents in that group.

$min takes a field and finds the lowest value stored in that field among the documents in that group.

$avg takes a field and finds the average value stored in that field among the documents in the group.

$push takes a field and returns an array of all the values found in that field among the documents in that group, each value will appear as many times as it does among the documents.

$sum takes a field or value and adds up the values contained in that field by documents in that group or in the case of a value, adds that value for each document found in that group. $sum can be used to add all the values for a field in the group, or to count the members of the group by adding a 1 for each group member.

Notes

At present, groups are stored in memory. If there is a larger number of groups, this could cause problems.

The use of _id to specify the group key makes the output ready to use for $out operator which has not been implemented yet..

Aggregation Framework - $limit

- Overview
- Specification
- Notes

Overview

The $limit pipeline operator limits the number of JSON documents that passes through it.

Specification

$limit takes a single numeric value as a parameter. Once that many documents have passed through the pipeline operator, no more will.

Here's a simple example:

```
db.article.aggregate(
    { $limit : 5 }
);
```

This pipeline will only pass through the first 5 documents that are seen. $limit has no effect on the content of the documents that are passed through – all fields present will be passed through as they are.

Notes

Aggregation Framework - $match

- Overview
- Specification
- Notes

Overview

$match filters documents. Documents which do not match the specified predicate are filtered out and do not progress further along the aggregation pipeline. Documents which do match are passed along unchanged.

Specification

$match predicate syntax is always exactly the same as that for queries; See Querying and Advanced Queries.

Within a pipeline, the $match operator appears thusly:

```
db.article.aggregate(
    { $match : <match-predicate> }
);
```

Here is an example with a simple field equality test:

```
db.aggregate(
    { $match : { author : "dave" } }
);
```

On its own like this, this is the equivalent of db.article.find({ author : "dave" }).

Here is another example, this time with a range test:

```
db.article.aggregate(
    { $match : { score : { $gt : 50, $lte : 90 } }
);
```

Notes

$match should be placed as early in the aggregation pipeline as possible. This minimizes the number of documents after it, thereby minimizing later processing. Placing a $match at the very beginning of a pipeline will enable it to take advantage of indexes in exactly the same way as a regular query (find()/findOne()).

Aggregation Framework - $project

Overview

$project can be used to reshape a document stream by renaming fields, adding fields, or removing fields.

Specification

For the examples that follow, imagine an article collection made up of documents that look like this:

```
{
    title : "this is my title" ,
    author : "bob" ,
    posted : new Date() ,
    pageViews : 5 ,
    tags : [ "fun" , "good" , "fun" ] ,
    comments : [
        { author :"joe" , text : "this is cool" } ,
        { author :"sam" , text : "this is bad" }
    ],
    other : { foo : 5 }
});
```

Inclusion and Exclusion

$project can be used to include or exclude fields using the same syntax as the query field selection syntax (see Field Selection). In this mode, a field path specification is given a value of 1 or 0 to include or exclude that field.

If the first field to use this mode specifies a 1, then the projection will only include this and other fields explicitly included in this way, except for _id, which is always included by default. However, even in this inclusionary mode, _id may be excluded with _id:0; no other fields may be excluded in this mode. An inclusion will not create a field if it does not already exist in a document.

If the first field to use this mode specifies a 0, then the projection will include all fields except those which are explicitly excluded like this one. When fields are excluded in this way, no fields may be explicitly included using :1.

Dotted field paths may be used to include or exclude fields in nested documents. This syntax requires that the field paths be enclosed in quotes to conform to JSON syntax.

Here are some projection specifications for a collection of articles as exemplified above.

Field inclusion:

```
db.runCommand(
{ aggregate : "article", pipeline : [
    { $project : {
        title : 1 , /* include this field, if it exists */
 author : 1 , /* include this field, if it exists */
 "comments.author" : 1  /* include the "author" field in each document in "comments", if it exists */
    }}
]});
```

In inclusion mode, _id is always included by default. As a special case, inclusions may explicitly exclude _id.

```
db.article.aggregate(
    { $project : {
        _id : 0 , /* exclude _id */
        title : 1 , /* include this field, if it exists */
 author : 1 /* include this field, if it exists */
    }}
);
```

In exclusionary mode, all fields except those that are excluded will pass through:

```
db.article.aggregate(
 { $project : {
        comments : 0 ,/* exclude this field */
        other : 0 /* exclude this field */
    }}
);
```

Computed Fields

Projections may be used to add computed fields to the document stream passing through the pipeline.

A computed field may use any of the expressions from the Aggregation Framework - Expression Reference.

Here is a simple example that shows how to add a value to a field, creating a new computed field:

```
db.article.aggregate(
    { $project : {
        title : 1,
 doctoredPageViews : { $add:["$pageViews", 10] }
    }}
);
```

Note the expression must be enclosed in braces to make it look like an object for the sake of conforming to JavaScript syntax.

Renaming Fields

Field references are a direct expression without an operator, and can be used without surrounding braces to rename existing fields. Field references here (and in any expression) can be dotted paths. Here is a simple example:

```
db.article.aggregate(
    { $project : {
        title : 1 ,
 page_views : "$pageViews" , /* rename this field */
        florble : "$other.foo" /* expose this nested field as if it were top-level */
    }}
);
```

Document-Valued Fields

Another form of computed field is a newly created sub-document. Field renaming and expressions can be used to populate a new sub-document that only exists in the result set.

Here is an example:

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        stats : {  /* the value of this new field is a sub-document */
     pv : "$pageViews", /* rename this from the top-level */
     foo : "$other.foo", /* a dotted field path expression */
     dpv : { $add:["$pageViews", 10] } /* a regular computed expression */
 }
    }}
);
```

Notes

Field Ordering

The BSON specification specifies that field order matters, and is to be preserved. A projection will honor that, and fields will be output in the same order as they are input, regardless of the order of any inclusion or exclusion specifications.

When new computed fields are added via a projection, these always follow all fields from the original source, and will appear in the order they appear in the projection specification.

Aggregation Framework - $skip

- Overview
- Specification
- Notes

Overview

The $skip pipeline operator skips over a number of JSON documents before passing on the rest of its input.

Specification

$skip takes a single numeric value as a parameter. Once that many input documents have been skipped, the rest are passed through the pipeline operator.

Here's a simple example:

```
db.article.aggregate(
    { $skip : 5 }
);
```

This pipeline will skip over the first 5 documents that are seen, and then allow the rest to pass through. $skip has no effect on the content of the documents that are passed through – all fields present will be passed through as they are.

Notes

Aggregation Framework - $sort

- Overview
- Specification

- Notes

Overview

The $sort pipeline operator sorts its input documents.

Specification

```
db.<collection-name>.aggregate(
    { $sort : { <sort-key> } }
);
```

$sort requires a sort key. Sort key specifications are exactly the same as those for
indexing (see Indexes - Basics and Indexes - Compound Keys Indexes). A sort key is a series of field-value pairs, each of which specifies a 1 or -1 for its value to indicate an ascending or descending sort, as shown by this example:

```
db.sortex.aggregate(
    { $sort : { b : -1, a: 1 } }
);
```

Here, the output will be the sorted collection, sorted by "b" descending, and "a" ascending.

Notes

The $sort operator cannot produce any output documents until it has seen all of its input. It will exhaust the prior pipeline operator's output, sort what it has received, and then be ready to produce output.

At the present time, sorts must fit in memory, so sorting a large input stream may not work.

Aggregation Framework - $unwind

- Overview
- Specification
- Notes

Overview

$unwind peels off the elements of an array one by one, returning a stream of documents. For each source document, for each member of the specified array within it, a new document emerges, but with the specified array replaced by one of its elements.

Specification

$unwind takes a single string argument, that is the path to the array element to be unwound. The path must be prefixed with a dollar sign ('$').

For the examples that follow, imagine an article collection made up of documents that look like this:

```
{
    title : "this is my title" ,
    author : "bob" ,
    posted : new Date() ,
    pageViews : 5 ,
    tags : [ "fun" , "good" , "fun" ] ,
    comments : [
        { author :"joe" , text : "this is cool" } ,
        { author :"sam" , text : "this is bad" }
    ],
    other : { foo : 5 }
}
```

Here is an example that demonstrates the effect of $unwind:

```
db.article.aggregate(
    { $project : {
 author : 1 , /* include this field */
        title : 1 , /* include this field */
        tags : 1 /* include this field */
    }},
    { $unwind : "$tags" }
);
```

Note the dollar sign ('$') in front of the name of the field to be unwound.

If this pipeline were to be run on the single article above, this would be the result:

```
{
        "result" : [
                {
                        "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                        "title" : "this is my title",
                        "author" : "bob",
                        "tags" : "fun"
                },
                {
                        "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                        "title" : "this is my title",
                        "author" : "bob",
                        "tags" : "good"
                },
                {
                        "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                        "title" : "this is my title",
                        "author" : "bob",
                        "tags" : "fun"
                }
        ],
        "ok" : 1
}
```

Note that the single document has become three documents: each is identical except for the value of the "tags" field. Each value of tags is one of the values in the original "tags" array.

Notes

$unwind is most useful when combined with $group or [$filter].

The effects of an unwind can be undone with the $push $group aggregation function.

If the target field does not exist within an input document, the document is passed through unchanged.

If the target field within an input document is not an array, an error is generated.

If the target field within an input document is an empty array ("[]"), then the document is passed through unchanged.

Aggregation Framework - Expression Reference

Overview

Expressions are used to calculate values from fields and other values.

Expressions are expressed in a prefix form as JSON field-value pairs. The "fieldname" is really the expression name. Multiple operands are then provided in an array: $add:[5, 6, 7]. Most of the operators below are binary, and expect a pair of operands to be presented as an array. Some, such as $add, $and, and $or, will accept more operands. A few are unary, and for these the operands may optionally be provided without array syntax, e.g., $not:[false] and $not:false are equivalent.

Composition

Expressions may be composed hierarchically. An expression may appear wherever a value may. Building on the addition above, we could have $add:[5,{ $divide:[12, 2] }, 7].

Field References

Within an expression, field names must be prefixed by a dollar sign ('$'); this must also be quoted. The dollar sign is required in order to be able to tell the difference between a field reference and a string literal. For example, two fields can be concatenated with a dot in between them as follows: $add:['$fieldOne', '.', '$fieldTwo'].

Table of Expressions

Operator	Operand Type(s)	Result Type
$cmp	2 ints/longs/doubles/strings	int
$eq	2 ints/longs/doubles/strings	boolean
$gt	2 ints/longs/doubles/strings	boolean
$gte	2 ints/longs/doubles/strings	boolean
$lt	2 ints/longs/doubles/strings	boolean
$lte	2 ints/longs/doubles/strings	boolean
$ne	2 ints/longs/doubles/strings	boolean
$and	2 or more of any type	boolean
$not	any single operand	boolean
$or	2 or more of any type	boolean
$add	strings/dates/ints/longs/double	int/long/double/string/date
$divide	ints/longs/doubles	int/long/double
$mod	2 ints/longs	int/long
$multiply	ints/longs/doubles	int/long/double
$subtract	2 dates/ints/longs/doubles	int/long/double/date
$strcasecmp	2 strings	int
$substr	string, 2 ints (starting point and length)	string
$toUpper	string	string
$toLower	string	string
$dayOfMonth	date	int
$dayOfWeek	date	int
$dayOfYear	date	int
$hour	date	int
$minute	date	int
$month	date	int
$second	date	int
$week	date	int
$year	date	int

$isoDate	an object with named date attributes	date
$ifNull	field and a field or any value	value
$cond	array:a boolean, followed by any other two values	the type of the 2nd or 3rd operand, as appropriate

Comparison Operators

All comparison operators take either an array containing either a pair of strings or a pair of numbers and return a boolean (except $cmp which returns an int).

$cmp returns zero if the two values are equivalent, a negative number if the first is less than the second, and a positive number if the first is greater than the second.

$eq returns true if the two values are equivalent and false otherwise.

$gt returns true if the first value is greater than the second and false otherwise.

$gte returns true if the first value is greater than or equal to the second and false otherwise.

$lt returns true if the first value is less than the second and false otherwise.

$lte returns true if the first value is less than or equal to the second and false otherwise.

$ne returns true if the two values are not equivalent and false otherwise.

Boolean Operators

All boolean operators take booleans as their arguments and return booleans. Non-boolean values passed as input are converted to booleans as per BSON standards. So numeric values that are not zero treated as true, as a strings, dates, objects, arrays, etc. However, Null, undefined, and zero are treated as false.

$and takes an array and returns true if all of the values passed are true and false otherwise. Note: $and uses short-circuit logic, meaning it will stop evaluation when it encounters the first false expression.

$not returns the opposite of the boolean value it is passed (true if handed false and false if handed true).

$or takes an array and returns true if any of the values passed are true and false otherwise. Note: $or uses short-circuit logic, meaning it will stop evaluation when it encounters the first true expression.

Arithmetic Operators

$add takes an array of numbers and adds them together, returning their sum.

- If a string is present in the array, all the values will be appended to one another in the order they are passed and returned as a string.
- If a date is present in the array (and no strings are), all numeric values are treated as a number of days and added to the date, the resulting date is returned.

$divide takes an array containing a pair of numbers and returns the value of the first number divided by the second number.

$mod takes an array containing a pair of numbers and computes and returns the remainder of the first number divided by the second number.

$multiply takes an array of numbers and multiples them together, the resulting product is returned.

$subtract takes an array containing a pair of numbers and subtracts the second from the first, returning their difference.

- If a date is passed as the first entry in the array, the number will be treated as a number of days and removed from the date, the resulting date is returned.

String Operators

$add see Arithmetic Add

$strcasecmp takes in two strings and returns a Javascript long that is positive if the first string is "greater than" the second, is negative if the first string is "less than" the second, and 0 if they are the same string. Note: unlike $cmp the strings are capitalized before being compared, so $strcmp is case insensitive.

$substr takes a string and two numbers, the first number represents the number of characters to skip in the original string and the second is the number of characters to take from the original string. The resulting string is returned.

$toLower takes in a single string and returns the same string with all uppercase letters replace with their lowercase equivalents.

$toUpper takes in a single string and returns the same string with all lowercase letters replace with their uppercase equivalents.

Date Operators

All date operators, except $add, $subtract, and $isoDate, take a Date as their single argument and return a Javascript long.

$add see Arithmetic Add

$dayOfMonth returns the day of the month as a number between 1 and 31.

$dayOfWeek returns the day of the week as a number between 1 and 7.

$dayOfYear returns the day of the year as a number between 1 and 366.

$hour returns the hour between 0 and 23.

$isoDate converts a broken-down date into a date. Use as

```
$isoDate:{year: <year>, month: <month>, dayOfMonth: <dayOfMonth>,
          hour: <hour>, minute: <minute>, second: <second>}
```

All values are optional, with the following defaults:

operand	default value
year	1900
month	1
dayOfMonth	1
hour	0
minute	0
second	0

These values can appear in any order, and may themselves be expressions.

$minute* returns the minute between 0 and 59.

month returns the month as a number between 1 and 12.

second returns the second between 0 and 59.

subtract see Arithmetic Subtract

week returns the week of the year as a number between 0 and 53. Weeks start on Sundays and the days before the first Sunday of the year are in week 0.

year returns the four digit year.

Other

$ifNull takes an array of two expressions. If the first expression evaluates to a non-false value, it is returned. Otherwise, the second expression's value is returned.

$cond takes an array of three expressions, the first of which should evaluate to a boolean value. If the first expression is true, $cond evaluates and returns the second expression. If the first expression is false, $cond evaluates and returns the third expression.

SQL to Aggregation Framework Mapping Chart

SQL term	MongoDB agg framework term
WHERE	$match
GROUP BY	$group
SELECT	$project
ORDER BY	$sort

Note: you can try the agg framework by testing with the v2.1.0 build. Note that 2.1.x is an unstable branch and not for production. Once ready it will be promoted to v2.2.

MongoDB has a few means of performing aggregation and data processing. In v2.2+ the main means are the aggregation framework and the map/reduce command. In general, if your operation can be expressed in the aggregation framework, use that as it is usually easier than using map/reduce and also performs better.

The table below shows examples of a SQL aggregation statement and the analogous MongoDB agg framework statement. The examples are invocations from the mongo shell, however usage from other programming languages is very similar.

SQL Statement	Mongo Statement
`SELECT COUNT(*) FROM users`	`db.users.aggregate([ { $group: {_id:null, count:{$sum:1}} } ])`
`SELECT SUM(price) FROM orders`	`db.orders.aggregate([ { $group: {_id:null, total:{$sum: "$price"} } } ])`
`SELECT cust_id,SUM(price) FROM orders GROUP BY cust_id`	`db.orders.aggregate([ { $group: {_id: "$cust_id",total:{$sum: "$price"}} } ])`
`SELECT cust_id,SUM(price) FROM orders WHERE active= true GROUP BY cust_id`	`db.orders.aggregate([ { $match:{active:true} }, { $group:{_id: "$cust_id",total:{$sum: "$price"}} } ])`

Removing

Removing Objects from a Collection

To remove objects from a collection, use the `remove()` function in the mongo shell. (Other drivers offer a similar function, but may call the function "delete". Please check your driver's documentation).

`remove()` is like `find()` in that it takes a JSON-style query document as an argument to select which documents are removed. If you call `remove()` without a document argument, or with an empty document `{}`, it will remove all documents in the collection. Some examples :

```
db.things.remove({});    // removes all
db.things.remove({n:1}); // removes all where n == 1
```

The most efficient method to delete a specific document is to use item's document _id value as a criteria:

```
> // myobject is some document that is in our db.things collection
> db.things.remove({_id: myobject._id});
```

Note that passing an entire document to remove() would work, but is inefficient as in that case we are basically specifying every single field of the document as part of the query expression.

```
> db.things.remove(myobject); // don't do this
```

Isolation

If a simultaneous update (on the same collection) grows an object which matched the remove criteria, the updated object may not be removed (as the operations are happening at approximately the same time, this may not even be surprising). In situations where this is undesireable, pass {$atomic : true} in your filter expression:

```
db.videos.remove( { rating : { $lt : 3.0 }, $atomic : true } )
```

The remove operation is then isolated – however, it will also block other operations while executing. The collection must be unsharded to use this option.

Updating

MongoDB supports atomic, in-place updates as well as more traditional updates for replacing an entire document.

update()

`update()` replaces the document matching criteria entirely with objNew. If you only want to modify some fields, you should use the $ modifiers below.

Here's the MongoDB shell syntax for `update()`:

`db.collection.update(` *`criteria`*`,` *`objNew`*`,` *`upsert`*`,` *`multi`* `)`

Arguments:

- *`criteria`* - query which selects the record to update;
- *`objNew`* - updated object or $ operators (e.g., $inc) which manipulate the object
- *`upsert`* - if this should be an "upsert" operation; that is, if the record(s) do not exist, insert one. **Upsert only inserts a single document**.
- *`multi`* - indicates if all documents matching *`criteria`* should be updated rather than just one. Can be useful with the $ operators below.

> If you are coming from SQL, be aware that by default, update() only modifies the first matched object. If you want to modify all matched objects, you need to use the multi flag.

> It is important to understand that **upsert** can only insert a single document. Upsert means "update if present; insert (a single document) if missing".

`save()` in the mongo shell

The save() helper method in the mongo shell provides a shorthand syntax to perform an update of a single document with upsert semantics:

```
> // x is some JSON style object
> db.mycollection.save(x); // updates if exists; inserts if new
>
> // equivalent to:
> db.mycollection.update( { _id: x._id }, x, /*upsert*/ true );
```

Modifier Operations

Modifier operations are highly-efficient and useful when updating existing values; for instance, they're great for incrementing a number.

So, while a conventional implementation does work:

```
var j=myColl.findOne( { name: "Joe" } );
j.n++;
myColl.save(j);
```

a modifier update has the advantages of avoiding the latency involved in querying and returning the object. The modifier update also features operation atomicity and very little network data transfer.

To perform an atomic update, simply specify any of the special update operators (which always start with a '$' character) with a relevant update document:

```
db.people.update( { name:"Joe" }, { $inc: { n : 1 } } );
```

The preceding example says, "Find the first document where 'name' is 'Joe' and then increment 'n' by one."

> While not shown in the examples, most modifier operators will accept multiple field/value pairs when one wishes to modify multiple fields. For example, the following operation would set x to 1 and y to 2:
>
> ```
> { $set : { x : 1 , y : 2 } }
> ```
>
> Also, multiple operators are valid too:
>
> ```
> { $set : { x : 1 }, $inc : { y : 1 } }
> ```

$inc

```
{ $inc : { field : value } }
```

increments *field* by the number *value* if *field* is present in the object, otherwise sets *field* to the number *value*. This can also be used to decrement by using a negative *value*.

$set

```
{ $set : { field : value } }
```

sets *field* to *value*. All datatypes are supported with `$set`.

$unset

```
{ $unset : { field : 1} }
```

Deletes a given field. v1.3+

$push

```
{ $push : { field : value } }
```

appends *value* to *field*, if *field* is an existing array, otherwise sets *field* to the array [*value*] if *field* is not present. If *field* is present but is not an array, an error condition is raised.

Multiple arrays may be updated in one operation by comma separating the *field* : *value* pairs:

```
{ $push : { field : value, field2 : value2 } }
```

$pushAll

```
{ $pushAll : { field : value_array } }
```

appends each value in *value_array* to *field*, if *field* is an existing array, otherwise sets *field* to the array *value_array* if *field* is not present. If *field* is present but is not an array, an error condition is raised.

$addToSet and $each

```
{ $addToSet : { field : value } }
```

Adds value to the array only if its not in the array already, if *field* is an existing array, otherwise sets *field* to the array *value* if *field* is not present. If *field* is present but is not an array, an error condition is raised.

To add a list of several values to the set use the `$each` qualifier:

```
{ $addToSet : { a : { $each : [ 3 , 5 , 6 ] } } }
```

$pop

```
{ $pop : { field : 1  } }
```

removes the last element in an array (ADDED in 1.1)

```
{ $pop : { field : -1  } }
```

removes the first element in an array (ADDED in 1.1) |

$pull

```
{ $pull : { field : _value } }
```

removes all occurrences of *value* from *field*, if *field* is an array. If *field* is present but is not an array, an error condition is raised.

In addition to matching an exact value you can also use expressions ($pull is special in this way):

```
{ $pull : { field : {field2: value} } } removes array elements with field2 matching value
```

```
{ $pull : { field : {$gt: 3} } } removes array elements greater than 3
```

```
{ $pull : { field : {<match-criteria>} } } removes array elements meeting match criteria
```

Because of this feature, to use the embedded doc as a match criteria, you cannot do exact matches on array elements.

$pullAll

```
{ $pullAll : { field : value_array } }
```

removes all occurrences of each value in *value_array* from *field*, if *field* is an array. If *field* is present but is not an array, an error condition is raised.

$rename

Version 1.7.2+ only.

```
{ $rename : { old_field_name : new_field_name } }
```

Renames the field with name 'old_field_name' to 'new_field_name'. Does not expand arrays to find a match for 'old_field_name'.

$bit

v1.8+.

```
{$bit : { field : {and : 5}}}
{$bit : {field : {or : 43}}}
{$bit : {field : {and : 5, or : 2}}}
```

Does a bitwise update of *field*. Can only be used with integers.

The $ positional operator

v1.4+

The $ operator (by itself) means "position of the matched array item in the query". Use this to find an array member and then manipulate it. For example:

```
> t.find()
{ "_id" : ObjectId("4b97e62bf1d8c7152c9ccb74"), "title" : "ABC",
  "comments" : [ { "by" : "joe", "votes" : 3 }, { "by" : "jane", "votes" : 7 } ] }

> t.update( {'comments.by':'joe'}, {$inc:{'comments.$.votes':1}}, false, true )

> t.find()
{ "_id" : ObjectId("4b97e62bf1d8c7152c9ccb74"), "title" : "ABC",
  "comments" : [ { "by" : "joe", "votes" : 4 }, { "by" : "jane", "votes" : 7 } ] }
```

Currently the $ operator only applies to the **first** matched item in the query. For example:

```
> t.find();
{ "_id" : ObjectId("4b9e4a1fc583fa1c76198319"), "x" : [ 1, 2, 3, 2 ] }
> t.update({x: 2}, {$inc: {"x.$": 1}}, false, true);
> t.find();
{ "_id" : ObjectId("4b9e4a1fc583fa1c76198319"), "x" : [ 1, 3, 3, 2 ] }
```

The positional operator cannot be combined with an `upsert` since it requires a matching array element. If your update results in an insert then the "$" will literally be used as the field name.

> Using "$unset" with an expression "array.$" will result in the array item becoming `null`, not being removed. You can issue an update with "{$pull:{x:null}}" to remove all nulls. $pull can now do much of this so this example is now mostly historical.
>
> ```
> > t.insert({x: [1,2,3,4,3,2,3,4]})
> > t.find()
> { "_id" : ObjectId("4bde2ad3755d00000000710e"), "x" : [ 1, 2, 3, 4, 3, 2, 3, 4 ] }
> > t.update({x:3}, {$unset:{"x.$":1}})
> > t.find()
> { "_id" : ObjectId("4bde2ad3755d00000000710e"), "x" : [ 1, 2, null, 4, 3, 2, 3, 4 ] }
> ```

Upserts with Modifiers

You may use upsert with a modifier operation. In such a case, the modifiers will be applied to the update *`criteria`* member and the resulting object will be inserted. The following upsert example may insert the object {`name:"Joe",x:1,y:1`}.

```
db.people.update( { name:"Joe" }, { $inc: { x:1, y:1 } }, true );
```

There are some restrictions. A modifier may not reference the `_id` field, and two modifiers within an update may not reference the same field, for example the following is not allowed:

```
db.people.update( { name:"Joe" }, { $inc: { x: 1 }, $set: { x: 5 } } );
```

Pushing a Unique Value

To add a value to an array only if not already present:

Starting in 1.3.3, you can do

```
update( {_id:'joe'},{"$addToSet": { tags : "baseball" } } );
```

For older versions, add $ne : <value> to your query expression:

```
update( {_id:'joe', tags: {"$ne": "baseball"}},
        {"$push": { tags : "baseball" } } );
```

Checking the Outcome of an Update Request

As described above, a non-upsert update may or may not modify an existing object. An upsert will either modify an existing object or insert a new object. The client may determine if its most recent message on a connection updated an existing object by subsequently issuing a `getlasterror` command (`db.runCommand( "getlasterror" )`). If the result of the `getlasterror` command contains an `updatedExisting` field, the last message on the connection was an update request. If the `updatedExisting` field's value is true, that update request caused an existing object to be updated; if `updatedExisting` is false, no existing object was updated. An "`upserted`" field will contain the new `_id` value if an insert is performed (new as of 1.5.4).

Notes

Object Padding

When you update an object in MongoDB, the update occurs in-place if the object has not grown in size. This is good for insert performance if the collection has many indexes.

Mongo adaptively learns if objects in a collection tend to grow, and if they do, it adds some padding to prevent excessive movements. This statistic is tracked separately for each collection. More info here.

Blocking

Starting in 1.5.2, multi updates yield occasionally so you can safely update large amounts of data. If you want a multi update to be truly isolated (so no other writes happen while processing the affected documents), you can use the $atomic flag in the query expression. For example:

```
db.students.update({score: {$gt: 60}, $atomic: true}, {$set: {pass: true}}, false, true)
```

sharding
`$atomic` is not supported with Sharding. If used it results in no updates and the getLastError Command returns error information.

atomic
`$atomic` only means that the update will not yield for other operations while running. It does **not** mean that the operation is "all or nothing". See here.

Field (re)order

During an update the field order may be changed. There is no guarantee that the field order will be consistent, or the same, after an update. At the moment, if the update can be applied in place then the order will be the same (with additions applied at the end), but if a move is required for the document (if the currently allocated space is not sufficient for the update) then the fields will be reordered (alphanumerically).

See Also

- findandmodify Command
- Atomic Operations

Atomic Operations

- Modifier operations
- "Update if Current"
  - The ABA Nuance
- "Insert if Not Present"
- Find and Modify (or Remove)
- Applying to Multiple Objects At Once

MongoDB supports atomic operations *on single documents*. MongoDB does not support traditional locking and complex transactions for a number of reasons:

- First, in sharded environments, distributed locks could be expensive and slow. Mongo DB's goal is to be lightweight and fast.
- We dislike the concept of deadlocks. We want the system to be simple and predictable without these sort of surprises.
- We want Mongo DB to work well for realtime problems. If an operation may execute which locks large amounts of data, it might stop some small light queries for an extended period of time. (We don't claim Mongo DB is perfect yet in regards to being "real-time", but we certainly think locking would make it even harder.)

MongoDB does support several methods of manipulating single documents atomically, which are detailed below.

Modifier operations

The Mongo DB update command supports several modifiers, all of which atomically update an element in a document. They include:

- $set - set a particular value
- $unset - delete a particular field (v1.3+)
- $inc - increment a particular value by a certain amount
- $push - append a value to an array
- $pushAll - append several values to an array
- $pull - remove a value(s) from an existing array
- $pullAll - remove several value(s) from an existing array
- $bit - bitwise operations

These modifiers are convenient ways to perform certain operations atomically.

"Update if Current"

Another strategy for atomic updates is "Update if Current". This is what an OS person would call Compare and Swap. For this we

1. Fetch the object.
2. Modify the object locally.
3. Send an update request that says "update the object to this new value if it still matches its old value".

Should the operation fail, we might then want to try again from step 1.

For example, suppose we wish to fetch one object from inventory. We want to see that an object is available, and if it is, deduct it from the inventory. The following code demonstrates this using mongo shell syntax (similar functions may be done in any language):

```
> t=db.inventory
> s = t.findOne({sku:'abc'})
{"_id" : "49df4d3c9664d32c73ea865a" , "sku" : "abc" , "qty" : 30}
> qty_old = s.qty;
> --s.qty;
> t.update({_id:s._id, qty:qty_old}, s); db.$cmd.findOne({getlasterror:1});
{"err" :  , "updatedExisting" : true , "n" : 1 , "ok" : 1} // it worked
```

For the above example, we likely don't care the exact sku quantity as long as it is as least as great as the number to deduct. Thus the following code is better, although less general -- we can get away with this as we are using a predefined modifier operation ($inc). For more general updates, the "update if current" approach shown above is recommended.

```
> t.update({sku:"abc",qty:{$gt:0}}, { $inc : { qty : -1 } } ) ; db.$cmd.findOne({getlasterror:1})
{"err" : , "updatedExisting" : true , "n" : 1 , "ok" : 1} // it worked
> t.update({sku:"abcz",qty:{$gt:0}}, { $inc : { qty : -1 } } ) ; db.$cmd.findOne({getlasterror:1})
{"err" : , "updatedExisting" : false , "n" : 0 , "ok" : 1} // did not work
```

The ABA Nuance

In the first of the examples above, we basically did "update object if *qty* is unchanged". However, what if since our read, *sku* had been modified? We would then overwrite that change and lose it!

There are several ways to avoid this problem ; it's mainly just a matter of being aware of the nuance.

1. Use the entire object in the update's query expression, instead of just the _id and qty field.
2. Use $set to set the field we care about. If other fields have changed, they won't be affected.
3. Put a version variable in the object, and increment it on each update.
4. When possible, use a $ operator instead of an update-if-current sequence of operations.

"Insert if Not Present"

Another optimistic concurrency scenario involves inserting a value when not already there. When we have a unique index constraint for the criteria, we can do this. See the How to Make an Auto Incrementing Field page for an example.

Find and Modify (or Remove)

See the findandmodify Command documentation for more information.

Applying to Multiple Objects At Once

You can use multi-update to apply the same modifier to every relevant object. By default a multi-update will allow some other operations (which could be writes) to interleave. Thus, this will only be pseudo-atomic (pseudo-isolated). To make it fullly isolated you can use the $atomic motifier:

not isolated:

```
db.foo.update( { x : 1 } , { $inc : { y : 1 } } , false , true );
```

isolated:

```
db.foo.update( { x : 1 , $atomic : 1 } , { $inc : { y : 1 } } , false , true );
```

Isolated is not atomic. Atomic implies that there is an all-or-nothing semantic to the update; this is not possible with more than one document. Isolated just means than you are the only one writing when the update is done; this means each update is done without any interference from any other.

Atomic operation examples

A key goal of MongoDB is to handle a good breadth of use cases, and to handle in a way that is easy for the developer. We found that a good number of use cases require atomic operations / strong consistency; thus that is a feature of the product. Below are some examples (in mongo shell syntax).

```
// register a new user (atomically)
db.users.insert( { _id : 'joe123' } )
if( db.getLastErrorObj().err )
  print("name is use try another")
else
  print("you are registered")


// decrement y if y > 0 (atomically)
db.stats.update(
  { _id : ‘myid’, y : { $gt : 0 } },
  { $inc : { y : -1 } }
)


// assure everyone’s email address is unique
db.users.ensureIndex( {email:1} , {unique:true} )


// if joe hasn’t already voted, let him vote (without races)
db.posts.update(
  { _id : ‘some_post_of_interest’ , voters : { $ne : ‘joe’ } },
  { votes : { $inc : 1 } }
)
```

How to Make an Auto Incrementing Field

Generally in MongoDB, one does not use an auto-increment pattern for _id's (or other fields), as this does not scale up well on large database clusters. Instead one typically uses Object IDs.

Side counter method

One can keep a counter of the current _id in a side document, in a collection dedicated to counters.
Then use FindAndModify to atomically obtain an id and increment the counter.

```
> db.counters.insert({_id: "userId", c: 0});

> var o = db.counters.findAndModify(
...        {query: {_id: "userId"}, update: {$inc: {c: 1}}});
{ "_id" : "userId", "c" : 0 }
> db.mycollection.insert({_id:o.c, stuff:"abc"});

> o = db.counters.findAndModify(
...        {query: {_id: "userId"}, update: {$inc: {c: 1}}});
{ "_id" : "userId", "c" : 1 }
> db.mycollection.insert({_id:o.c, stuff:"another one"});
```

Once you obtain the next id in the client, you can use it and be sure no other client has it.

Optimistic loop method

One can do it with an optimistic concurrency "insert if not present" loop. The following example, in Mongo shell Javascript syntax, demonstrates.

```
// insert incrementing _id values into a collection
function insertObject(o) {
    x = db.myCollection;
    while( 1 ) {
        // determine next _id value to try
var c = x.find({},{_id:1}).sort({_id:-1}).limit(1);
        var i = c.hasNext() ? c.next()._id + 1 : 1;
        o._id = i;
        x.insert(o);
        var err = db.getLastErrorObj();
        if( err && err.code ) {
            if( err.code == 11000 /* dup key */ )
                continue;
            else
                print("unexpected error inserting data: " + tojson(err));
        }
        break;
    }
}
```

The above should work well unless there is an extremely high concurrent insert rate on the collection. In that case, there would be a lot of looping potentially.

See Also

- Atomic Operations

findAndModify Command

Find and Modify (or Remove)

v1.3.0 and higher

MongoDB 1.3+ supports a "find, modify, and return" command. This command can be used to atomically **modify a document** (at most one) and return it. Note that, by default, the document returned will not include the modifications made on the update.

If you don't need to return the document, you can use Update (which can affect multiple documents, as well).

The general form is

```
db.runCommand( { findAndModify : <collection>,
                 <options> } )
```

The MongoDB shell includes a helper method, findAndModify(), for executing the command. Some drivers provide helpers also.

At least one of the *update* or *remove* parameters is required; the other arguments are optional.

Argument	Description	Default
query	a filter for the query	{}
sort	if multiple docs match, choose the first one in the specified sort order as the object to manipulate	{}
remove	set to a true to remove the object before returning	N/A
update	a modifier object	N/A
new	set to true if you want to return the modified object rather than the original. Ignored for remove.	false
fields	see Retrieving a Subset of Fields (1.5.0+)	All fields
upsert	create object if it doesn't exist; a query must be supplied! examples (1.5.4+)	false

The *sort* option is useful when storing queue-like data. Let's take the example of fetching the highest priority job that hasn't been grabbed yet and atomically marking it as grabbed:

```
> db.jobs.save( {
     name: "Next promo",
     inprogress: false, priority:0,
     tasks : [ "select product", "add inventory", "do placement"]
} );

> db.jobs.save( {
     name: "Biz report",
     inprogress: false, priority:1,
     tasks : [ "run sales report", "email report" ]
} );

> db.jobs.save( {
     name: "Biz report",
     inprogress: false, priority:2,
     tasks : [ "run marketing report", "email report" ]
} );
```

```
> job = db.jobs.findAndModify({
     query: {inprogress: false, name: "Biz report"},
     sort : {priority:-1},
     update: {$set: {inprogress: true, started: new Date()}},
     new: true
});

{
 "_id" : ...,
 "inprogress" : true,
 "name" : "Biz report",
 "priority" : 2,
 "started" : "Mon Oct 25 2010 11:15:07 GMT-0700 (PDT)",
 "tasks" : [
  "run marketing report",
  "email report"
 ]
}
```

You can pop an element from an array for processing and update in a single atomic operation:

```
> task = db.jobs.findAndModify({
     query: {inprogress: false, name: "Next promo"},
     update : {$pop : { tasks:-1}}, fields: {tasks:1},
     new: false } )
{
 "_id" : ...,
 "tasks" : [
  "select product",
  "add inventory",
  "do placement"
 ]
}

> db.jobs.find( { name : "Next promo"} )
{
        "_id" : ...,
        "inprogress" : false,
        "name" : "Next promo",
        "priority" : 0,
        "tasks" : [ "add inventory", "do placement" ]
}
```

You can also simply remove the object to be returned.

```
> job = db.jobs.findAndModify( {sort:{priority:-1}, remove:true} );
{
 "_id" : ...,
 "inprogress" : true,
 "name" : "Biz report",
 "priority" : 2,
 "started" : "Mon Oct 25 2010 10:44:15 GMT-0700 (PDT)",
 "tasks" : [
  "run marketing report",
  "email report"
 ]
}

> db.jobs.find()
{
       "_id" : ...,
       "inprogress" : false,
       "name" : "Next promo",
       "priority" : 0,
       "tasks" : [ "add inventory", "do placement" ]
}
{
       "_id" : ...,
       "name" : "Biz report",
       "inprogress" : false,
       "priority" : 1,
       "tasks" : [ "run sales report", "email report" ]
}
```

If the client crashes before processing the job or task in the above examples, the data will be lost forever.

See the tests for more examples.

If your driver doesn't provide a helper function for this command, run the command directly with something like this:

```
job = db.runCommand({ findAndModify : "jobs",
                      sort : { priority : -1 },
                      remove : true
                    }).value;
```

Sharding limitations

findandmodify will behave the same when called through a *mongos* as long as the collection it is modifying is unsharded. If the collection is sharded, then the query must contain the shard key.

See Also

- Atomic Operations

Padding Factor

- Overview
- No padding after imports, repairs, compactions and initial replica syncs
- Shrinking Documents
- Manual Padding
- See Also

Overview

When you update a document in MongoDB, the update occurs in-place if the document has not grown in size. If the document did grow in size, however, then it might need to be relocated on disk to find a new disk location with enough contiguous space to fit the new larger document. This can lead to problems for write performance if the collection has many indexes since a move will require updating all the indexes for the document.

Mongo adaptively learns if documents in a collection tend to grow, and if they do, it adds some padding so that the document has room to grow. This helps to prevent excessive movements on subsequent writes. This statistic is tracked separately for each collection.

You can check the collection's current *padding factor* by running the collStats command helper in the shell. The padding factor indicates what the padding will be for new record allocations(for inserts, or on updates that grow a document and cause it to move to a new location).

```
> db.coll.stats()
{
        "ns" : "...", ...,
        "paddingFactor" : 1, ...,
        "ok" : 1
}
```

As each document is written at a different point in time the padding for each document will not be the same. Also, as the padding factor is relative to the size of each document you cannot calculate the exact amount of padding for a collection based on the average document size and padding factor.

The paddingfactor is 1.0 if there is no padding. 1.5 would indicate 50% padding on a new insert/moves.

No padding after imports, repairs, compactions and initial replica syncs

After compaction, repair and import operations, there is (generally) no padding as they were inserted and there were no updates (which would cause the paddingFactor to change). Thus you may see slower update performance after these cases, but the size required for storage will be lower.

Shrinking Documents

If a document gets smaller (e.g., because of an $unset or $pop), the document does not move but stays at its current location. It thus effectively has more padding. Thus space is never reclaimed if documents shrink by large amounts and never grow again. To reclaim that space run a compact operation (or repair).

Manual Padding

Padding in MongoDB is automatic. You should not have to do so manually. In exceptional cases one can do this though. The strategy is to add a faux field that assures allocation of a larger slot size for the document. The faux field is then removed; at this point there is extra room for expansion on a future update. Example below.

```
> t = db.zcollection;
>
> db.setProfilingLevel(2);
>
> // b is a very long string
>
> t.insert({q:1})
> t.update({q:1},{$set:{yy:b}})
>
> // note the "moved:true" in the output below -- indicating not enough
> // padding to avoid moving a document.
> db.system.profile.find()
...
{ "ts" : ISODate("2011-10-13T02:45:23.062Z"), "op" : "insert", "ns" : "test.zcollection",
  "millis" : 0, "client" : "127.0.0.1", "user" : "" }
{ "ts" : ISODate("2011-10-13T02:45:33.560Z"), "op" : "update", "ns" : "test.zcollection",
  "query" : { "q" : 1 }, "updateobj" : { "$set" : { "yy" : "aaaaaaaaa..." } },
  "nscanned" : 1, "moved" : true, "millis" : 3, "client" : "127.0.0.1", "user" : "" }
>
> // not important what value of 'padding' is, only its length:
> t.insert({q:2,padding:ourpaddingstring})
> t.update({q:2},{$unset:{padding:1}})
> t.update({q:2},{$set:{yy:b}})
>
> // no "moved:true" below (which is good).  Note however that the automatic adjustment
> // of a collection's padding factor normally achieves this regardless.  Manually padding
> // is rarely necessary.
> db.system.profile.find()
...
{ "ts" : ISODate("2011-10-13T02:46:34.920Z"), "op" : "insert", "ns" : "test.zcollection",
  "millis" : 1, "client" : "127.0.0.1", "user" : "" }
{ "ts" : ISODate("2011-10-13T02:46:42.775Z"), "op" : "update", "ns" : "test.zcollection",
  "query" : { "q" : 2 }, "updateobj" : { "$unset" : { "padding" : 1 } }, "nscanned" : 1,
  "millis" : 2, "client" : "127.0.0.1", "user" : "" }
{ "ts" : ISODate("2011-10-13T02:46:55.831Z"), "op" : "update", "ns" : "test.zcollection",
  "query" : { "q" : 2 }, "updateobj" : { "$set" : { "yy" : "aaaaaaaaa..." } },
  "nscanned" : 1, "millis" : 2, "client" : "127.0.0.1", "user" : "" }
```

See Also

- http://blog.mongodb.org/post/248614779/fast-updates-with-mongodb-update-in-place

two-phase commit

Redirection Notice
This page should redirect to http://cookbook.mongodb.org/patterns/perform-two-phase-commits/.

A common problem with non-relational database is that it is not possible to do transactions across several documents. When executing a transaction composed of several sequential operations, some issues arise:

- Atomicity: it is difficult to rollback changes by previous operations if one fails.
- Isolation: changes to a single document are seen by concurrent processes, which may have an inconsistent view of the data during the transaction execution.
- Consistency: In case of a major failure (network, hardware) it is possible that the data will be left inconsistent and difficult to repair.

MongoDB provides atomicity for an operation on a single document. Since documents can be fairly complex, this actually covers many more cases than with a traditional DB. Still there are cases where transactions across documents are needed, and that is when a two-phase commit can be used. The two-phase commit is made possible by the fact that documents are complex and can represent pending data and states. This process makes sure that the data is eventually consistent, which is usually what matters most to the system.

Account transfer example

Problem overview

The most common example of transaction is to transfer funds from account A to B in a reliable way. With a traditional RDBMS, funds are substracted from A and added to B within an atomic transaction. With MongoDB, a viable solution is to use a two-phase commit.

Let's have one collection holding accounts:

```
foo:PRIMARY> db.accounts.save({name: "A", balance: 1000, pendingTransactions: []})
foo:PRIMARY> db.accounts.save({name: "B", balance: 1000, pendingTransactions: []})
foo:PRIMARY> db.accounts.find()
{ "_id" : ObjectId("4d7bc66cb8a04f512696151f"), "name" : "A", "balance" : 1000, "pendingTransactions"
: [ ] }
{ "_id" : ObjectId("4d7bc67bb8a04f5126961520"), "name" : "B", "balance" : 1000, "pendingTransactions"
: [ ] }
```

And we need one collection representing transactions:

```
foo:PRIMARY> db.transactions.save({source: "A", destination: "B", value: 100, state: "initial"})
foo:PRIMARY> db.transactions.find()
{ "_id" : ObjectId("4d7bc7a8b8a04f5126961522"), "source" : "A", "destination" : "B", "value" : 100,
"state" : "initial" }
```

Transaction description

Step 1: the transaction state is switched to, "pending":

```
foo:PRIMARY> t = db.transactions.findOne({state: "initial"})
{
 "_id" : ObjectId("4d7bc7a8b8a04f5126961522"),
 "source" : "A",
 "destination" : "B",
 "value" : 100,
 "state" : "initial"
}
foo:PRIMARY> db.transactions.update({_id: t._id}, {$set: {state: "pending"}})
foo:PRIMARY> db.transactions.find()
{ "_id" : ObjectId("4d7bc7a8b8a04f5126961522"), "source" : "A", "destination" : "B", "value" : 100,
"state" : "pending" }
```

Step 2: apply the transaction to both accounts, and make sure the transaction is not already pending:

```
foo:PRIMARY> db.accounts.update({name: t.source, pendingTransactions: {$ne: t._id}}, {$inc: {balance:
-t.value}, $push: {pendingTransactions: t._id}})
foo:PRIMARY> db.accounts.update({name: t.destination, pendingTransactions: {$ne: t._id}}, {$inc:
{balance: t.value}, $push: {pendingTransactions: t._id}})
foo:PRIMARY> db.accounts.find()
{ "_id" : ObjectId("4d7bc97fb8a04f5126961523"), "balance" : 900, "name" : "A", "pendingTransactions" :
[ ObjectId("4d7bc7a8b8a04f5126961522") ] }
{ "_id" : ObjectId("4d7bc984b8a04f5126961524"), "balance" : 1100, "name" : "B", "pendingTransactions"
: [ ObjectId("4d7bc7a8b8a04f5126961522") ] }
```

Step 3: set the transaction's state to "committed":

```
foo:PRIMARY> db.transactions.update({_id: t._id}, {$set: {state: "committed"}})
foo:PRIMARY> db.transactions.find()
{ "_id" : ObjectId("4d7bc7a8b8a04f5126961522"), "destination" : "B", "source" : "A", "state" :
"committed", "value" : 100 }
```

Step 4: remove the pending transaction from accounts:

```
foo:PRIMARY> db.accounts.update({name: t.source}, {$pull: {pendingTransactions: ObjectId(
"4d7bc7a8b8a04f5126961522")}})
foo:PRIMARY> db.accounts.update({name: t.destination}, {$pull: {pendingTransactions: ObjectId(
"4d7bc7a8b8a04f5126961522")}})
foo:PRIMARY> db.accounts.find()
{ "_id" : ObjectId("4d7bc97fb8a04f5126961523"), "balance" : 900, "name" : "A", "pendingTransactions" :
[ ] }
{ "_id" : ObjectId("4d7bc984b8a04f5126961524"), "balance" : 1100, "name" : "B", "pendingTransactions"
: [ ] }
```

Step 5: set transaction's state to "done":

```
foo:PRIMARY> db.transactions.update({_id: t._id}, {$set: {state: "done"}})
foo:PRIMARY> db.transactions.find()
{ "_id" : ObjectId("4d7bc7a8b8a04f5126961522"), "destination" : "B", "source" : "A", "state" : "done",
"value" : 100 }
```

Failure scenarios

Now let's look at the failure scenarios and how to deal with them. For example, a failure can be that the application making the sequential operations suddenly dies, and is restarted.

Cases to cover:

- any failure between after step 1 and before step 3: Application should get a list of transactions in state "pending" and resume from step 2.
- any failure after step 3 and before step 5: Application should get a list of transactions in state "applied" and resume from step 4.

Application is thus always able to resume the transaction and eventually get to a consistent state.
These "repair" jobs should be run at application startup and possibly at regular interval to catch any unfinished transaction.
The time it takes to get to a consistent state may vary depending on how long it takes to resume a failed transaction.

Rollback

A common need may be to rollback a transaction, either because it has been cancelled or because it can never succeed (e.g. account B is closed).

Two cases:

- after step 3, the transaction is considered committed and should not be rolled back. Instead, to undo the transaction, a new transaction can be created with an opposite source and destination.
- after step 1 and before step 3: the process below should be applied.

Step 1: set the transaction's state to "canceling":

```
foo:PRIMARY> db.transactions.update({_id: t._id}, {$set: {state: "canceling"}})
```

Step 2: undo the transaction from accounts:

```
foo:PRIMARY> db.accounts.update({name: t.source, pendingTransactions: t._id}, {$inc: {balance:
t.value}, $pull: {pendingTransactions: t._id}})
foo:PRIMARY> db.accounts.update({name: t.destination, pendingTransactions: t._id}, {$inc: {balance:
-t.value}, $pull: {pendingTransactions: t._id}})
foo:PRIMARY> db.accounts.find()
{ "_id" : ObjectId("4d7bc97fb8a04f5126961523"), "balance" : 1000, "name" : "A", "pendingTransactions"
: [ ] }
{ "_id" : ObjectId("4d7bc984b8a04f5126961524"), "balance" : 1000, "name" : "B", "pendingTransactions"
: [ ] }
```

Step 3: set the transaction's state to "cancelled":

```
foo:PRIMARY> db.transactions.update({_id: t._id}, {$set: {state: "cancelled"}})
```

Multiple applications

A common issue that exists with any DBs is how to make it safe for several applications to run transactions.
It is important that only 1 application handles a given transaction at one point in time, because otherwise conflicts can happen.

One example is:

- application A1 and A2 both grab transaction T1 which is in "initial" state.
- A1 applies the whole transaction before A2 starts
- A2 applies transaction a 2nd time because it does not appear as pending in the accounts

To handle multiple applications, there should be a marker at the transaction level that the transaction is being handled.
One can use findAndModify:

```
foo:PRIMARY> t = db.transactions.findAndModify({query: {state: "initial", application: {$exists: 0}},
update: {$set: {state: "pending", application: "A1"}}, new: true})
{
 "_id" : ObjectId("4d7be8af2c10315c0847fc85"),
 "application" : "A1",
 "destination" : "B",
 "source" : "A",
 "state" : "pending",
 "value" : 150
}
```

The only remaining issue is if application A1 dies during transaction execution.
The resume processes described in "Failure scenarios" can be applied, but application should make sure it owns the transactions.
For example to resume pending jobs, query should be:

```
foo:PRIMARY> db.transactions.find({application: "A1", state: "pending"})
{ "_id" : ObjectId("4d7be8af2c10315c0847fc85"), "application" : "A1", "destination" : "B", "source" :
"A", "state" : "pending", "value" : 150 }
```

Proper two-phase commit

This implementation tries to be simple on purpose, it assumes that:

- an account operation can always be rolled back
- the account balance can go negative

A proper real world implementation would probably differ:

- accounts have both a current balance, pending credits, pending debits.
- during step 2, application makes sure accounts has sufficient funds for transaction, modifies credits/debits and adds transaction as pending, all in one update.
- during step 4, application actually applies the transaction on balance, modifies credits/debits and removes transaction from pending, all in one update.

Additional notes:

In the context of important transactions, you will probably want to use:

- reasonable "getLastError" to check that operations are actually written to the DB (see "getLastError" or "write concern" for your drivers).
- durability so that operations are consistently saved on disk when an operation returns successfully.

Updating Data in Mongo

- Updating a Document in the mongo Shell with `save()`
- Embedding Documents Directly in Documents
- Database References

Updating a Document in the mongo Shell with save()

As shown in the previous section, the `save()` method may be used to save a new document to a collection. We can also use `save()` to update an existing document in a collection.

Continuing with the *example* database from the last section, lets add new information to the document `{name:"mongo"}` that already is in the collection.

```
> var mongo = db.things.findOne({name:"mongo"});
> print(tojson(mongo));
{"_id" : "497dab624ee47b3a675d2d9c" , "name" : "mongo"}
> mongo.type = "database";
database
> db.things.save(mongo);
> db.things.findOne({name:"mongo"});
{"_id" : "497dab624ee47b3a675d2d9c" , "name" : "mongo" , "type" : "database"}
>
```

This was a simple example, adding a string valued element to the existing document. When we called `save()`, the method saw that the document already had an "_id" field, so it simply performed an update on the document.

In the next two sections, we'll show how to embed documents within documents (there are actually two different ways), as well as show how to query for documents based on values of embedded documents.

Embedding Documents Directly in Documents

As another example of updating an existing document, lets embed a document within an existing document in the collection. We'll keep working with the original {`name:"mongo"`} document for simplicity.

```
> var mongo = db.things.findOne({name:"mongo"});
> print(tojson(mongo));
{"_id" : "497da93d4ee47b3a675d2d9b" , "name" : "mongo", "type" : "database"}
> mongo.data = { a:1, b:2};
{"a" : 1 , "b" : 2}
> db.things.save(mongo);
> db.things.findOne({name:"mongo"});
{"_id" : "497da93d4ee47b3a675d2d9b" , "name" : "mongo" , "type" : "database", "data" : {"a" : 1 , "b"
: 2}}
>
```

As you can see, we added new data to the mongo document, adding {`a:1, b:2`} under the key "data".

Note that the value of "data" is a document itself - it is embedded in the parent mongo document. With BSON, you may nest and embed documents to any level. You can also query on embedded document fields, as shown here:

```
> db.things.findOne({ "data.a" : 1});
{"_id" : "497da93d4ee47b3a675d2d9b" , "name" : "mongo" , "data" : {"a" : 1 , "b" : 2}}
> db.things.findOne({ "data.a" : 2});
>
```

Note that the second `findOne()` doesn't return anything, because there are no documents that match.

Database References

Alternatively, a document can reference other documents which are not embedded via a *database reference*, which is analogous to a foreign key in a relational database. A database reference (or "DBRef" for short), is a reference implemented according to the Database References. Most drivers support helpers for creating DBRefs. Some also support additional functionality, like dereference helpers and auto-referencing. See specific driver documentation for examples / more information

Lets repeat the above example, but create a document and place in a different collection, say *otherthings*, and embed that as a reference in our favorite "mongo" object under the key "otherdata":

```
// first, save a new doc in the 'otherthings' collection

> var other = { s : "other thing", n : 1};
> db.otherthings.save(other);
> db.otherthings.find();
{"_id" : "497dbcb36b27d59a708e89a4" , "s" : "other thing" , "n" : 1}

// now get our mongo object, and add the 'other' doc as 'otherthings'

> var mongo = db.things.findOne();
> print(tojson(mongo));
{"_id" : "497dab624ee47b3a675d2d9c" , "name" : "mongo" , "type" : "database" , "data" : {"a" : 1 , "b"
: 2}}
> mongo.otherthings = new DBRef( 'otherthings' , other._id );
{"s" : "other thing" , "n" : 1 , "_id" : "497dbcb36b27d59a708e89a4"}
> db.things.save(mongo);
> db.things.findOne().otherthings.fetch();
{"_id" : "497dab624ee47b3a675d2d9c" , "name" : "mongo" , "type" : "database" , "data" : {"a" : 1 , "b"
: 2} , "otherthings" : {"_id" : "497dbcb36b27d59a708e89a4" , "s" : "other thing" , "n" : 1}}

// now, lets modify our 'other' document, save it again, and see that when the dbshell
// gets our mongo object and prints it, if follows the dbref and we have the new value

> other.n = 2;
2
> db.otherthings.save(other);
> db.otherthings.find();
{"_id" : "497dbcb36b27d59a708e89a4" , "s" : "other thing" , "n" : 2}
> db.things.findOne().otherthings.fetch();
{"_id" : "497dab624ee47b3a675d2d9c" , "name" : "mongo" , "type" : "database" , "data" : {"a" : 1 , "b"
: 2} , "otherthings" : {"_id" : "497dbcb36b27d59a708e89a4" , "s" : "other thing" , "n" : 2}}
>
```

MapReduce

Map/reduce in MongoDB is useful for batch processing of data and aggregation operations. It is similar in spirit to using something like Hadoop with all input coming from a collection and output going to a collection. Often, in a situation where you would have used GROUP BY in SQL, map/reduce is the right tool in MongoDB.

Overview

`map/reduce` is invoked via a database command. Typically the database creates a collection to hold output of the operation. `map` and `reduce` functions are written in JavaScript and execute on the server.

Command syntax:

```
db.runCommand(
 { mapreduce : <collection>,
   map : <mapfunction>,
   reduce : <reducefunction>
   [, query : <query filter object>]
   [, sort : <sorts the input objects using this key. Useful for optimization, like sorting by the
emit key for fewer reduces>]
   [, limit : <number of objects to return from collection, not supported with sharding>]
   [, out : <see output options below>]
   [, keeptemp: <true|false>]
   [, finalize : <finalizefunction>]
   [, scope : <object where fields go into javascript global scope >]
   [, jsMode : true]
   [, verbose : true]
 }
);
```

* `finalize` - function to apply to all the results when finished

- `keeptemp` - if true, the generated collection is not treated as temporary. Defaults to false. When `out` is specified, the collection is automatically made permanent. (MongoDB <=1.6)
- `scope` - can pass in variables that can be access from map/reduce/finalize. Note that updates to scope variables' values are not shared among shard members, so in a sharded cluster you should treat scope variables as global constants.
  example mr5
- `verbose` - provide statistics on job execution time

Incremental Map-reduce

If the data set over which you'd like to perform map-reduce aggregations is constantly growing, then you may want to take advantage of incremental map-reduce. The prevents you from having to aggregate over the entire data set each time you want to see your results.

To perform incremental map-reduce, take the following steps:

1. First, run a map-reduce job over an existing collection, and output the data to it's own output collection.

2. When you have more data to process, run a second map-reduce job, but use the `query` option to the filter the documents to include only new documents.

3. Use the `reduce` output option. This will use your reduce function to merge the new data into the existing output collection.

Output options

pre-v1.8: If you do not specify a value for `out`, then the results will be placed into a temporary collection whose name will be given in command's output (see below). Otherwise, you can specify the name of a collection for the `out` option and the results will be placed there.

v1.8+: the output options have changed. Map-reduce no longer generates temporary collections (thus, `keepTemp` has been removed). Now, you must always supply a value for `out`. The `out` directives are:

- "collectionName" - By default the output will by of type "replace".
- { **replace** : "collectionName" } - the output will be inserted into a collection which will atomically replace any existing collection with the same name.
- { **merge** : "collectionName" } - This option will merge new data into the old output collection. In other words, if the same key exists in both the result set and the old collection, the new key will overwrite the old one.
- { **reduce** : "collectionName" } - If documents exists for a given key in the result set and in the old collection, then a reduce operation (using the specified reduce function) will be performed on the two values and the result will be written to the output collection. If a finalize function was provided, this will be run after the reduce as well.
- { **inline** : 1} - With this option, no collection will be created, and the whole map-reduce operation will happen in RAM. Also, the results of the map-reduce will be returned within the result object. Note that this option is possible only when the result set fits within the 16MB limit of a single document. In **v2.0**, this is your only available option on a replica set secondary.

For example:

```
db.users.mapReduce(map, reduce, {out: { inline : 1}});
```

Additional options within `out` objects are:

* "db" - the db name to output to.

```
out : {replace : "collectionName", db : "otherDB"}
```

* { **sharded** : 1} - **MongoDB 1.9+** If true and combined with an output mode that writes to a collection, the output collection will be sharded using the _id field. See details in the sharding section.

Note: the order of the objects in the out parameter matter.

Result object

```
{
  [results : <document_array>,]
  [result : <collection_name> | {db: <db>, collection: <collection_name>},]
  timeMillis : <job_time>,
  counts : {
       input :  <number of objects scanned>,
       emit  : <number of times emit was called>,
       output : <number of items in output collection>
  } ,
  ok : <1_if_ok>
  [, err : <errmsg_if_error>]
}
```

Either the `result` or the `results` field will be present depending on your output type. The `results` element is only present if the inline output option was used. The value of the `results` element is an array of embedded documents containing the results. If you chose any other output type the `result` field will be a string with the name of the collection holding the results, or an embedded document containing the db and collection if you chose to output to another db.

A command helper is available in the MongoDB shell :

```
db.collection.mapReduce(mapfunction,reducefunction[,options]);
```

`map`, `reduce`, and `finalize` functions are written in JavaScript.

Map Function

The `map` function references the variable `this` to inspect the current object under consideration. A map function calls `emit(key,value)` any number of times to feed data to the reducer. In most cases you will emit once per input document, but in some cases such as counting tags, a given document may have one, many, or even zero tags. Each emit is limited to 50% of the maximum document size (e.g. 4MB for 1.6.x and 8MB for 1.8.x).

```
function map(void) -> void
```

Reduce Function

When you run a map/reduce, the `reduce` function will receive an array of emitted values and reduce them to a single value. Because the reduce function might be invoked more than once for the same key, the structure of the object returned by the reduce function must be identical to the structure of the `map` function's emitted value. We can clarify this with a simple example.

Suppose we're iterating over a collection of documents that represent user comments. A sample document might look like this:

```
{ username: "jones",
    likes: 20,
    text: "Hello world!"
  }
```

We want to use map/reduce to count the total number of comments per user and aggregate the total number of "likes" received across all of a user's comments. To do this, we'd first write a `map` function like this one:

```
function() {
    emit( this.username, {count: 1, likes: this.likes} );
  }
```

This essentially says that we'll be grouping by `username` and aggregating using an object with fields for `count` and `likes`.

When map/reduce is actually run, an array of values for each username will be sent to the `reduce` function. That's why the `reduce` function is always written to process an array of values. Here's the appropriate function for this example:

```
function(key, values) {
    var result = {count: 0, likes: 0};

    values.forEach(function(value) {
      result.count += value.count;
      result.likes += value.likes;
    });

    return result;
  }
```

Notice that the result document has the same structure as the documents emitted by the `map` function. This is important because, when the `reduce` function is run against a given key, it's not guaranteed to process every single value for that key (or username). In fact, the `reduce` function may have to run more than once. For example, while processing the comments collection, the `map` function might encounter ten comments from the user "jones." It then sends those comments' data to be reduced, and this results in the following aggregate object:

```
{ count: 10, likes: 247 }
```

Later, the `map` function encounters one more comment document by "jones." When this happens, the values in the extra comment must be reduced against the already existing aggregate value. If the new emitted document looks like this:

```
{ count: 1, likes: 5 }
```

Then the `reduce` function will be invoked in this way:

```
reduce("jones", [ {count: 10, likes: 247}, { count: 1, likes: 5} ] )
```

And the resulting document will be a simple combination (or reduction) of those values:

```
{ count: 11, likes: 252 }
```

So long as you understand that the `reduce` function might be invoked more than once for the same key, it's easy to see why the this function must return a value whose structure matches the `map` function's emitted value.

A more technical explanation

```
function reduce(key, array_of_value) -> value
```

OR

```
function reduce(key_obj, [value_obj, value_obj, ...]) -> value_obj
```

The map/reduce engine may invoke reduce functions iteratively; thus, these functions must be idempotent. That is, the following must hold for your reduce function:

```
for all k,vals : reduce( k, [reduce(k,vals)] ) == reduce(k,vals)
```

This also means the following is true:

```
reduce( k, [A, B] ) == reduce( k, [B, A] )
```

If you need to perform an operation only once, use a finalize function.

> ⚠ The output of the map function's emit (the second argument) and the value returned by reduce should be the same format to make iterative reduce possible. If not, there will be weird bugs that are hard to debug.

> ⚠ Currently, the return value from a reduce function cannot be an array (it's typically an object or a number).

Finalize Function

A `finalize` function may be run after reduction. Such a function is optional and is not necessary for many map/reduce cases. The finalize function takes a key and a value, and returns a finalized value.

```
function finalize(key, value) -> final_value
```

Your reduce function may be called multiple times for the same object. Use finalize when something should only be done a single time at the end; for example calculating an average.

jsMode flag

v2.0+
Normally, map/reduce execution follows the steps:

- convert from BSON to JS, execute map, convert from JS to BSON
- convert from BSON to JS, execute reduce, convert from JS to BSON

Thus it requires several translations but it can handle very large datasets during mapping by using a temporary collection.
It is possible to make the execution stay in JS by using {jsMode: true} which performs the following steps:

- convert from BSON to JS, execute map
- execute reduce, convert from JS to BSON

The execution time may be significantly reduced. Note that this mode is limited by either

> ⚠ `jsMode` is limited by the JS heap size and a maximum of 500k unique keys. Consequently it is not suitable for large jobs in which case mongo may revert to regular mode.

Sharded Environments

There are 2 aspects of sharding with Map/Reduce, input and output.

Sharded input

If the input collection is sharded, MongoS will automatically dispatch the map/reduce job to each of the shard, to be executed in parallel. There is no special option required. MongoS will then wait for jobs on all shards to finish.

Sharded output

By default the output collection will not be sharded. The process is:

- MongoS dispatches a map/reduce finish job to the shard that will store the target collection.
- that mongod will pull results from all other shards, run a final reduce/finalize, and write to the output.

If using the "sharded" option in the "out" object, the output will be sharded using "_id" as the shard key.
The process is:

- MongoS pulls the results from each shard, doing a merge sort to get them ordered.
- on the fly, it does reduce/finalize as needed. Then writes the result to the output collection in sharded mode.

Notes about sharded output:

- though MongoS does some processing, only a small amount of memory is required even for large datasets.
- there is currently a limitation in that shard chunks do not get automatically split and migrated during insertion. Some manual commands may be required until the chunks are granular and balanced.
- the `limit` option is not supported.

Examples

Shell Example 1

The following example assumes we have an `events` collection with objects of the form:

```
{ time : <time>, user_id : <userid>, type : <type>, ... }
```

We then use MapReduce to extract all users who have had at least one event of type "sale":

```
> m = function() { emit(this.user_id, 1); }
> r = function(k,vals) { return 1; }
> res = db.events.mapReduce(m, r, { query : {type:'sale'} });
> // or in v1.8+:
> // res = db.events.mapReduce(m, r, { query : {type:'sale'}, out : 'example1' });
> db[res.result].find().limit(2)
{ "_id" : 8321073716060 , "value" : 1 }
{ "_id" : 7921232311289 , "value" : 1 }
```

If we also wanted to output the number of times the user had experienced the event in question, we could modify the reduce function like so:

```
> r = function(k,vals) {
...     var sum=0;
...     for(var i in vals) sum += vals[i];
...     return sum;
... }
```

Note, here, that we cannot simply return `vals.length`, as the reduce may be called multiple times.

Shell Example 2

```
$ ./mongo
> db.things.insert( { _id : 1, tags : ['dog', 'cat'] } );
> db.things.insert( { _id : 2, tags : ['cat'] } );
> db.things.insert( { _id : 3, tags : ['mouse', 'cat', 'dog'] } );
> db.things.insert( { _id : 4, tags : []  } );

> // map function
> m = function(){
...    this.tags.forEach(
...        function(z){
...            emit( z , { count : 1 } );
...        }
...    );
...};

> // reduce function
> r = function( key , values ){
...    var total = 0;
...    for ( var i=0; i<values.length; i++ )
...        total += values[i].count;
...    return { count : total };
...};

> res = db.things.mapReduce(m, r, { out : "myoutput" } );
> res
{
 "result" : "myoutput",
 "timeMillis" : 12,
 "counts" : {
  "input" : 4,
  "emit" : 6,
  "output" : 3
 },
 "ok" : 1,
}
> db.myoutput.find()
{"_id" : "cat" , "value" : {"count" : 3}}
{"_id" : "dog" , "value" : {"count" : 2}}
{"_id" : "mouse" , "value" : {"count" : 1}}

> db.myoutput.drop()
```

* gist

Mongo Shell Script with Incremental Map-Reduce and Finalize

This example is a JavaScript script file. The map-reduce can be run repeatedly on different dates to incrementally augment the result. The finalize option computes averages.

The output of commands and the queries themselves are saved to variables so that they can be examined after the sample script is run via the load() command in the shell.

```
// work in the map-reduce example db
db = db.getSiblingDB("mrex");

// clean out from previous runs of this sample -- you wouldn't do this in production
db.session.drop();
db.session_stat.drop();

// simulate saving records that log the lengths of user sessions in seconds
db.session.save({userid:"a", ts: ISODate('2011-11-03 14:17:00'), length: 95});
db.session.save({userid:"b", ts: ISODate('2011-11-03 14:23:00'), length: 110});
db.session.save({userid:"c", ts: ISODate('2011-11-03 15:02:00'), length: 120});
db.session.save({userid:"d", ts: ISODate('2011-11-03 16:45:00'), length: 45});

db.session.save({userid:"a", ts: ISODate('2011-11-04 11:05:00'), length: 105});
```

```
db.session.save({userid:"b", ts: ISODate('2011-11-04 13:14:00'), length: 120});
db.session.save({userid:"c", ts: ISODate('2011-11-04 17:00:00'), length: 130});
db.session.save({userid:"d", ts: ISODate('2011-11-04 15:37:00'), length: 65});

/*
  For each user, count up the number of sessions, and figure out the average
  session length.

  Note that to be able to find the average session length, we need to keep a
  total of the all the session lengths, and then divide at the end.

  We're also going to set this up so that we can repeat the process to get
  incremental results over time.
*/

function mapf()
{
    emit(this.userid,
         {userid:this.userid, total_time:this.length, count:1, avg_time:0});
}

function reducef(key, values)
{
    var r = {userid:key, total_time:0, count:0, avg_time:0};
    values.forEach(function(v)
     {
         r.total_time += v.total_time;
         r.count += v.count;
     });
    return r;
}

function finalizef(key, value)
{
    if (value.count > 0)
 value.avg_time = value.total_time / value.count;

    return value;
}

/*
  Here's the initial run.

  The query isn't technically necessary, but is included here to demonstrate
  how this is the same map-reduce command that will be issued later to do
  incremental adjustment of the computed values.  The query is assumed to run
  once a day at midnight.
 */
var mrcom1 = db.runCommand( { mapreduce:"session",
         map:mapf,
         reduce:reducef,
         query: {ts: {$gt:ISODate('2011-11-03 00:00:00')}},
         out: { reduce: "session_stat" },
         finalize:finalizef
       });

function saveresults(a)
{
    /* append everything from the cursor to the argument array */
    var statcurs = db.session_stat.find();
    while(statcurs.hasNext())
 a.push(statcurs.next());
}

/* save the results into mrres1 */
var mrres1 = [];
saveresults(mrres1);

/* add more session records (the next day) */
db.session.save({userid:"a", ts: ISODate('2011-11-05 14:17:00'), length: 100});
```

```
db.session.save({userid:"b", ts: ISODate('2011-11-05 14:23:00'), length: 115});
db.session.save({userid:"c", ts: ISODate('2011-11-05 15:02:00'), length: 125});
db.session.save({userid:"d", ts: ISODate('2011-11-05 16:45:00'), length: 55});

/*
  Run map reduce again.

  This time, the query date is the next midnight, simulating a daily job that
  is used to update the values in session_stat.  This can be repeated daily
  (or on other periods, with suitable adjustments to the time).
 */
var mrcom2 = db.runCommand( { mapreduce:"session",
         map:mapf,
         reduce:reducef,
         query: {ts: {$gt:ISODate('2011-11-05 00:00:00')}},
         out: { reduce: "session_stat" },
         finalize:finalizef
       });

/* save the results into mrres2 */
```

```
var mrres2 = [];
saveresults(mrres2);
```

More Examples

- example mr1
- Finalize example: example mr2

Note on Permanent Collections

Even when a permanent collection name is specified, a temporary collection name will be used during processing. At map/reduce completion, the temporary collection will be renamed to the permanent name atomically. Thus, one can perform a map/reduce job periodically with the same target collection name without worrying about a temporary state of incomplete data. This is very useful when generating statistical output collections on a regular basis.

Parallelism

See info on Concurrency

Presentations

Map/reduce, geospatial indexing, and other cool features - Kristina Chodorow at MongoSF (April 2010)

Troubleshooting

- See Troubleshooting MapReduce

See Also

- Aggregation
- Kyle's Map/Reduce basics
- Blog post - walkthrough a mongodb map reduce job

Troubleshooting MapReduce

Tips on troubleshooting map/reduce.

Troubleshooting the `map` function

We can troubleshoot the map function in the shell by defining a test `emit` function in the shell and having it print out trace information.

For example suppose we have some data:

```
> db.articles.find()
{ "_id" : 123, "author" : "joe", "text" : "hello", "votes" : [
        {
                "who" : "john",
                "vote" : 1
        },
        {
                "who" : "jane",
                "vote" : 1
        },
        {
                "who" : "vince",
                "vote" : -1
        }
] }
{ "_id" : 127, "author" : "sri", "text" : "It was...", "votes" : [
        { "who" : "jane",
          "vote" : 2
        }
] }
```

And we have written a map function:

```
function map() {
    this.votes.forEach( function(x){emit(x.who,1);} );
}
```

It would be nice to visualize the output of this function. We can do this in the shell by defining a client side debug version of emit():

```
function emit(k, v) {
    print("emit");
    print("  k:" + k + " v:" + tojson(v));
}
```

For example, we could run the emit on a single document from the collection:

```
> x = db.articles.findOne(); // grab an object
> map.apply(x); // call our map function, client side, with x as 'this'
emit
  k:john v:1
emit
  k:jane v:1
emit
  k:vince v:1
```

Additionally we could apply the map on several objects:

```
> for( var c = db.articles.find(); c.hasNext(); ) {
...     var doc = c.next();
...     print("document _id=" + tojson(doc._id));
...     map.apply( doc );
...     print();
... }
document _id=123
emit
  k:john v:1
emit
  k:jane v:1
emit
  k:vince v:1

document _id=127
emit
  k:jane v:1
```

After verifying the emits from map are as expected, we write a reduce function and run the real job:

```
> function reduce(k, vals) {
...     var sum = 0;
...     for (var i in vals) {
...         sum += vals[i];
...     }
...     return sum;
...}
>
> db.articles.mapReduce(map,reduce,"out");
{
        "result" : "out",
        "timeMillis" : 62,
        "counts" : {
                "input" : 2,
                "emit" : 4,
                "output" : 3
        },
        "ok" : 1,
}
>
> db.out.find()
{ "_id" : "jane", "value" : 2 }
{ "_id" : "john", "value" : 1 }
{ "_id" : "vince", "value" : 1 }
```

Troubleshooting the `reduce` function

When troubleshooting the reduce function, problems usually crop up in two places:

1. `emit()` outputting different values than `reduce`
2. `reduce( k, [A, B] ) != reduce( k, [B, A] )`

Fortunately, it is easy to test for both of these cases directly from the shell.

When performing a reduce, there is no guarantee on the order of incoming values.

#1 - Test value format

Run a reduce on a sample key / value from `emit`. Wrap the value in an array construct. The output of the reduce should have the same format at the input. In most cases, it should actually *be* the same.

```
> reduce( { name : 'joe' }, [ { votes : 1 } ] )
{ votes : 1 }
```

The same can also be tested with two values. The format should still be the same.

```
> reduce( { name : 'joe' }, [ { votes : 1 }, { votes : 3 } ] )
{ votes : 4 }
```

#2 - Test Commutativity / Idempotence

Again, two simple tests that should work.

Order of the objects should not matter:

```
> reduce( { name : 'joe' }, [ { votes : 1 }, { votes : 3 } ] )
{ votes : 4 }
> reduce( { name : 'joe' }, [ { votes : 3 }, { votes : 1 } ] )
{ votes : 4 }
```

Reduce output can be re-reduced:

```
> reduce( { name : 'joe' }, [
    { votes : 1 },
    reduce ( { name : 'joe' }, [ { votes : 3 } ] )
  ] )
{ votes : 4 }
```

Data Processing Manual

By "data processing", we generally mean operations performed on large sets of data, rather than small interactive operations.

Import

One can always write a program to load data of course, but the mongoimport utility also works for some situations. mongoimport supports importing from json, csv, and tsv formats.

A common usage pattern would be to use mongoimport to load data in a relatively raw format and then use a server-side script (db.eval() or map/reduce) to reduce the data to a more clean format.

See Also

- Import/Export Tools
- Aggregation Framework
- Map/Reduce

mongo - The Interactive Shell

- MongoDB Manual Shell Reference
- More Information
- Presentations

The MongoDB distribution includes `bin/mongo`, the MongoDB interactive shell. This utility is a JavaScript shell that allows you to issue commands to MongoDB from the command line. (*it is basically an extended SpiderMonkey shell*)

The shell is useful for:

- inspecting a database's contents
- testing queries
- creating indices
- maintenance scripts
- other administrative functions

When you see sample code in this wiki and it looks like JavaScript, assume it is a shell example. See the driver syntax table for a chart that can be used to convert those examples to any language.

MongoDB Manual Shell Reference

- MongoDB Manual Reference Section
- mongo Manual Page
- JavaScript Interface
- MongoDB Database Commands

More Information

- Shell Overview
- Shell Scripts
- Shell Reference
- Shell APIDocs

Presentations

- Hacking the Shell - MongoSF (May 2011)
- CRUD and the JavaScript Shell - MongoSF (April 2010)

Scripting the shell

The MongoDB shell is not just an interactive shell, it can also be scripted using JS files. In addition to specifying a Javascript file (*.js) you can also use `--eval` with a snippet of JS.

Using the shell this way allows for tasks to be performed without the need for any additional drivers or language support; it can be used in cron, or automated administrative tasks. Please be aware there are data format issues in javascript so you should be careful how much you do in Javascript.

Common uses for the scripted shell includes:

- backups
- scheduled Map-Reduce commands
- offline reports
- administration

Running a Script

```
./mongo server:27017/dbname --quiet my_commands.js
```

The syntax stems from the interactive shell. This command will execute the `my_commands.js` as if it had been entered into the shell directly, with some exceptions.

- `./mongo`: command to start the interactive shell, may vary on your shell of choice
- `server:27017/dbname`: basic connection information
- `--quiet`: this is a flag for the `mongo` command. This switch removes some header information that is not typically necessary when building unattended scripts.
- `my_commands.js`: a file containing a series of shell commands to execute

`--quiet`

This option will remove the header printed at the top when you run the shell normally:

```
$ mongo
MongoDB shell version: 2.1.0
connecting to: test
> ^C
bye
$ mongo --quiet
> ^C
```

`--eval`

In addition to using a full Javascript file you can also pass in a Javascript fragment:

```
bash-3.2$ ./mongo test --eval "printjson(db.getCollectionNames())"
MongoDB shell version: 1.8.0
connecting to: test
[
        "system.indexes",
        "t1",
        "t2",
        "test.fam",
        "test1",
        "test11",
        "testBinary",
        "testarray"
]
```

Differences between scripted and interactive

Printing

When using the shell interactively, the shell will print returned values and format where possible. This is done as a general convenience from within the shell. However, when building a script, the printing needs to be defined explicitly.

There are two functions commonly used for this:

1. `print()`: works as normal javascript
2. `printjson()`: prints a nicely formatted JSON representation of the given object

***Example**: print JSON for the first 10 objects from a `find`*

```
db.foo.find({x:1}).forEach(printjson)
```

`use dbname`

This command does not work in scripted mode. Instead you will need to explicitly define the database in the connection (`/dbname` in the example above).

Alternately, you can also create a connection within the script:

```
> help connect // for more help
> var x = new Mongo('host[:port]');
> var mydb = x.getDB('mydb');
> // or
> var mydb = connect('host[:port]/mydb');
```

`it`

The iterator command `it` does not work outside of the interactive scripting environment.

getLastError

When running an `update/insert` command from the shell, the shell automatically awaits a reply (*i.e. runs a get last error*).

The same is not true when running from a script file. To wait for the status of an operation (such as a write), run the getLastError function after `update/insert`.

```
db.getLastErrorObj()
// or
db.getLastError()
```

Overview - The MongoDB Interactive Shell

Running the Shell

The interactive shell is included in the standard MongoDB distribution. To start the shell, go into the root directory of the distribution and type

```
./bin/mongo
```

It might be useful to add *mongo_distribution_root*/bin to your PATH so you can just type mongo from anywhere.

If you start with no parameters, it connects to a database named "test" running on your local machine on the default port (27017). You can see the db to which you are connecting by typing db:

```
./mongo
type "help" for help
> db
test
```

You can pass mongo an optional argument specifying the address, port and even the database to initially connect to:

./mongo *foo*	connects to the *foo* database on your local machine
./mongo 192.168.13.7/*foo*	connects to the *foo* database on 192.168.13.7
./mongo dbserver.mydomain.com/*foo*	connects to the *foo* database on dbserver.mydomain.com
./mongo 192.168.13.7:9999/*foo*	connects to the *foo* database on 192.168.13.7 on port 9999

.mongorc.js

1.9.1+

When the shell is launched, it checks the user's home directory for a javascript file named .mongorc.js. If this file is found, its contents are interpreted and run by the shell prior to displaying the prompt for the first time. This allows the user to define variables, customize the prompt, or update information that they would like updated every time they launch a shell. This functionality can be overridden with the --norc flag. It should be noted that if a file is specified to be executed by the shell, the rc file will not be run until after that file has completed.

Operations

Help

```
> help                            // top level help
> db.help()                       // help on db-specific methods
> db.mycollection.help()          // help on collection methods
> db.mycollection.find().help()   // cursor help
```

Select Database

The following are three basic commands that provide information about the available databases, and collections in a given database.

show dbs	displays all the databases on the server you are connected to
use *db_name*	switches to *db_name* on the same server
show collections	displays a list of all the collections in the current database

Querying

mongo uses a JavaScript API to interact with the database. Because mongo is also a complete JavaScript shell, db is the variable that is the current database connection.

To query a collection, you simply specify the collection name as a property of the db object, and then call the find() method. For example:

```
db.foo.find();
```

This will display the first 10 objects from the `foo` collection. Typing ***`it`*** after a find() will display the next 10 subsequent objects.

> By setting the shellBatchSize you can change this:
>
> ```
> DBQuery.shellBatchSize = #
> ```

> If the shell does not accept the collection name (for example if it starts with a number, contains a space etc), use
>
> ```
> db['foo'].find()
> ```
>
> instead.

Inserting

In order to insert data into the database, you can simply create a JavaScript object, and call the `save()` method. For example, to save an object { `name: "sara"`} in a collection called *foo*, type:

```
db.foo.save({ name : "sara"});
```

Note that MongoDB will implicitly create any collection that doesn't already exist.

Updating

Let's say you want to change someone's address. You can do this using the following `mongo` commands:

```
person = db.people.findOne( { name : "sara" } );
person.city = "New York";
db.people.save( person );
```

Deleting

`db.`*`foo`*`.drop()`	drop the entire *`foo`* collection
`db.`*`foo`*`.remove()`	remove all objects from the collection
db.*foo*.remove({ name : "sara" })	remove objects from the collection where `name` is `sara`

Indexes

`db.`*`foo`*`.getIndexKeys()`	get all fields that have indexes on them
`db.`*`foo`*`.ensureIndex({ _field_ : 1 })`	create an index on *`field`* if it doesn't exist

Open Additional Connections

You can use the following commands to open additional connections (normally you don't need to do this, but might from a script):

```
conn = new Mongo(host);
db = conn.getDB(dbname);
db.auth(username,password);
```

where `host` is a string that contains either the name or address of the machine you want to connect to (e.g. "192.168.13.7") or the machine and port (e.g. "192.168.13.7:9999"). Note that host in an optional argument, and can be omitted if you want to connect to the database instance running on your local machine. (e.g. `conn = new Mongo()`)

Alternatively you can use the `connect` helper method:

```
> db = connect("localhost:27020/mytestdb"); // example with a nonstandard port #
```

Working from the Prompt

Line Continuation

If a line contains open '(' or '{' characters, the shell will request more input before evaluating:

```
> function f() {
... x = 1;
... }
>
```

You can press Ctrl-C to escape from "..." mode and terminate line entry.

Key Shortcuts

- up/down array for command history
- in v1.9+ some basic emacs keystrokes work
- ctrl-l to clear the screen
- tab for auto-complete (newer versions only)
- ctrl-c to exit, or to break out of line continuation mode

Custom Prompt

1.9.1+

The shell's prompt can be customized by creating variable 'prompt' in the shell. It can be any arbitrary javascript, including a function that returns a string. This flexibility allows for additional information to be displayed in the prompt. For example, to have a prompt that contains the number of commands issued, type:

```
> cmdCount = 1;
> prompt = function() {
... return (cmdCount++) + "> ";
... }
1> command
2> anothercommand
3>
```

To make the prompt look a bit more familiar, we can make it database@host$:

```
> host = db.serverStatus().host; \\ since host should not change
> prompt = function() {
... return db+"@"+host+"$ ";
... }
admin@mylaptop.local$ use monkeys
switched to db monkeys
monkeys@mylaptop.local$
```

You could use the prompt to do a bit of database monitoring as well:

```
> prompt = function() {
... return "Uptime:"+db.serverStatus().uptime+" Files:"+db.stats().objects+" > ";
... }
Uptime:5897 Files:6 > db.monkeys.save({name : "James"});
Uptime:5948 Files:7 >
```

Using a real editor

> ⚠ 2.1.0+

We've added a feature to allow you edit larger values including functions using your editor of choice. Just run `edit nameOfVariableOrFunction` and we will open whatever editor you have defined in your `$EDITOR` environment variable. Make sure that you save the file when editing. If you wish to discard your changes, you can either not save or make your editor exit with an error (`:cq` in Vim or `(kill-emacs 1)` in Emacs).

```
$ EDITOR=vim mongo --nodb
MongoDB shell version: 2.1.0-pre-
> function f() {}
> edit f
> f
function f() {
    print("this really works");
}
> f()
this really works
> o = {}
{ }
> edit o
> o
{ "soDoes" : "this" }
>
```

It is possible that the code in functions will be slightly modified by the JavaScript compiler when you try to edit it again. For example it may convert `1+1` in to 2 and strip out comments. The actual changes will vary based on the version of JavaScript used, but should not effect the semantics of the code, only its appearance.

Some Notes on Datatypes in the Shell

Numbers

By default, the shell treats all numbers as floating-point values. You have the option to work with 64 bit integers by using a class built into the shell called NumberLong() If you have long/integer BSON data from the database you may see something like this:

```
"bytes" : {
   "floatApprox" : 575175
}
```

or something like this for larger numbers (in 1.6+):

```
{..., "bytes" : NumberLong("5284376243087482000") ,...}
```

Note that prior to 1.6 long numbers might be displayed like this:

```
"bytes" : {
   "floatApprox" : 5284376243087482000,
   "top" : 1230364721,
   "bottom" : 4240317554
}
```

In addition, setting/incrementing any number from javascript will (most likely) change the data type to a floating point value.

Here is an example of creating a document with a `long` field:

```
doc = { field: new NumberLong("123212313")}
```

Dates

The Date() function returns a string and a "**new** Date()" will return an object (which is what you should use to store values).

```
> Date()
Sun May 02 2010 19:07:40 GMT-0700 (Pacific Daylight Time)
> new Date()
"Sun May 02 2010 19:07:43 GMT-0700 (Pacific Daylight Time)"
> typeof(new Date())
object
> typeof(Date())
string
```

newer (1.7+) versions print this

```
> new Date()
ISODate("2010-11-29T19:41:46.730Z")
> ISODate("2010-11-29T19:41:46.730Z")
ISODate("2010-11-29T19:41:46.730Z")
```

As you can see, ISODate is a thin wrapper around the Date constructor to fix some of it's shortcomings. It returns a normal Date object with all of the normal methods that javascript Date methods support. We have also changed the way that Date objects print to make sure that they don't look like strings and that if you copy and paste the output you get the same object.

BinData

The BSON BinData datatype is represented via class BinData in the shell. Run `help misc` for more information.

```
> new BinData(2, "1234")
BinData(2,"1234")
```

See Also

- MongoDB Shell Reference

dbshell Reference

- Command Line
- Special Command Helpers
- Basic Shell Javascript Operations
- Queries
- Error Checking
- Administrative Command Helpers
- Opening Additional Connections
- Miscellaneous
- Examples

Command Line

`--help`	Show command line options
`--nodb`	Start without a db, you can connect later with `new Mongo()` or `connect()`
`--shell`	After running a .js file from the command line, stay in the shell rather than terminating

Special Command Helpers

Non-javascript convenience macros:

`help`	Show help
`db.help()`	Show help on db methods
`db.myColl.help()`	Show help on collection methods

`show dbs`	Print a list of all databases on this server
`use` *`dbname`*	Set the db variable to represent usage of *dbname* on the server
`show collections`	Print a list of all collections for current database
`show users`	Print a list of users for current database
`show profile`	Print most recent profiling operations that took >= 1ms

Basic Shell Javascript Operations

`db`	The variable that references the current database object / connection. Already defined for you in your instance.
`db.auth(user,pass)`	Authenticate with the database (if running in secure mode).
`coll = db.`*`collection`*	Access a specific *collection* within the database.
`cursor = coll.find();`	Find all objects in the collection. See queries.
`coll.remove(`*`objpattern`*`);`	Remove matching objects from the collection. *`objpattern`* is an object specifying fields to match. E.g.: `coll.remove( { name: "Joe" } );`
`coll.save(`*`object`*`)`	Save an object in the collection, or update if already there. If your object has a presave method, that method will be called before the object is saved to the db (before both updates and inserts)
`coll.insert(object)`	Insert object in collection. No check is made (i.e., no upsert) that the object is not already present in the collection.
`coll.update(...)`	Update an object in a collection. See the Updating documentation; update() has many options.
`coll.ensureIndex( {`*`name`*` : 1 } )`	Creates an index on `tab.`*`name`*. Does nothing if index already exists.
`coll.update(...)`	
`coll.drop()`	Drops the collection `coll`
`db.getSisterDB(name)`	Return a reference to another database using this same connection. This allows for cross database queries. Usage example: `db.getSisterDB('production').getCollectionNames()`

Queries

`coll.find()`	Find all.
`it`	Continue iterating the last cursor returned from find().
`coll.find(`*`criteria`*` );`	Find objects matching *`criteria`* in the collection. E.g.: `coll.find( { name: "Joe" } );`
`coll.findOne(`*`criteria`*` );`	Find and return a single object. Returns null if not found. If you want only one object returned, this is more efficient than just `find()` as `limit(1)` is implied. You may use regular expressions if the element type is a string, number, or date: `coll.find( { name: /joe/i } );`
`coll.find( criteria, fields );`	Get just specific fields from the object. E.g.: `coll.find( {}, {name:true} );`
`coll.find().sort( {`*`field`*`:1[,`*`field`*` :1] });`	Return results in the specified order (field ASC). Use -1 for DESC.
`coll.find(`*`criteria`*` ).sort( {`*`field`*` : 1 } )`	Return the objects matching *`criteria`*, sorted by *`field`*.
`coll.find( ... ).limit(`*`n`*`)`	Limit result to *n* rows. Highly recommended if you need only a certain number of rows for best performance.

`coll.find( ... ).skip(n )`	Skip *n* results.
`coll.count()`	Returns total number of objects in the collection.
`coll.find( ... ).count()`	Returns the total number of objects that match the query. Note that the number ignores limit and skip; for example if 100 records match but the limit is 10, `count()` will return 100. This will be faster than iterating yourself, but still take time.

More information: see queries.

Error Checking

`db.getLastError()`	Returns error from the last operation.
`db.getPrevError()`	Returns error from previous operations.
`db.resetError()`	Clear error memory.

Administrative Command Helpers

`db.cloneDatabase(fromhost)`	Clone the current database from the other host specified. fromhost database must be in noauth mode.
`db.copyDatabase(fromdb, todb, fromhost)`	Copy fromhost/fromdb to todb on this server. fromhost must be in noauth mode.
`db.fromColl .renameCollection(toColl)`	Rename collection from fromColl to toColl.
`db.repairDatabase()`	Repair and compact the current database. This operation can be very slow on large databases.
`db.addUser(user,pwd)`	Add user to current database.
`db.getCollectionNames()`	get list of all collections.
`db.dropDatabase()`	Drops the current database.

Opening Additional Connections

`db = connect("<host>:<port>/<dbname>")`	Open a new database connection. One may have multiple connections within a single shell, however, automatic getLastError reporting by the shell is done for the 'db' variable only.
`conn = new Mongo("hostname")`	Open a connection to a new server. Use `getDB()` to select a database thereafter.
`db = conn.getDB("dbname")`	Select a specific database for a connection

Miscellaneous

`Object.bsonsize(db.foo.findOne())`	prints the bson size of a db object (mongo version 1.3 and greater)
`db.foo.findOne().bsonsize()`	prints the bson size of a db object (mongo versions predating 1.3)

For a full list of functions, see the shell API.

Examples

The MongoDB source code includes a jstests/ directory with many mongo shell scripts.

Developer FAQ

- What's a "namespace"?
- How do I copy all objects from one database collection to another?
- If you remove an object attribute is it deleted from the store?
- Are null values allowed?
- Does an update fsync to disk immediately?

- How do I do transactions/locking?
- How do I do equivalent of SELECT count * and GROUP BY?
- What are so many "Connection Accepted" messages logged?
- Can I run on Amazon EBS? Any issues?
- Why are my data files so large?

- Do I Have to Worry About SQL Injection
- How does concurrency work
- SQL to Mongo Mapping Chart
- What is the Compare Order for BSON Types

See also:

- Intro FAQ
- Sharding FAQ
- Replica Set FAQ
- Indexing Advice and FAQ
- Markus Gattol's excellent FAQ on his website

What's a "namespace"?

MongoDB stores BSON objects in *collections*. The concatenation of the database name and the collection name (with a period in between) is called a *namespace*.

For example, `acme.users` is a namespace, where `acme` is the database name, and `users` is the collection name. Note that periods can occur in collection names, so a name such as `acme.blog.posts` is legal too (in that case `blog.posts` is the collection name.

How do I copy all objects from one database collection to another?

See below. The code below may be ran server-side for high performance with the eval() method.

```
db.myoriginal.find().forEach( function(x){db.mycopy.insert(x)} );
```

Data Type Fidelity
All of your data types must be supported in Javascript or you will get data conversion or write errors. It is better to use mongodump/restore for data type fidelity.

If you remove an object attribute is it deleted from the store?

Yes, you remove the attribute and then re-`save()` the object.

Are null values allowed?

For members of an object, yes. You cannot add null to a database collection though as null isn't an object. You can add {}, though.

Does an update fsync to disk immediately?

No, writes to disk are lazy by default. A write may hit disk a couple of seconds later. For example, if the database receives a thousand increments to an object within one second, it will only be flushed to disk once. (Note fsync options are available though both at the command line and via getLastError_old.)

How do I do transactions/locking?

MongoDB does not use traditional locking or complex transactions with rollback, as it is designed to be lightweight and fast and predictable in its performance. It can be thought of as analogous to the MySQL MyISAM autocommit model. By keeping transaction support extremely simple, performance is enhanced, especially in a system that may run across many servers.

The system provides alternative models for atomically making updates that are sufficient for many common use cases. See the wiki page Atomics Operations for detailed information.

How do I do equivalent of SELECT count * and GROUP BY?

See aggregation.

What are so many "Connection Accepted" messages logged?

If you see a tremendous number of connection accepted messages in the mongod log, that means clients are repeatedly connecting and disconnected. This works, but is inefficient.

With CGI this is normal. If you find the speed acceptable for your purposes, run mongod with --quiet to suppress these messages in the log. If you need better performance, switch to a solution where connections are pooled -- such as an Apache module.

Can I run on Amazon EBS? Any issues?

Works fine in our experience; more information here.

Why are my data files so large?

MongoDB does aggressive preallocation of reserved space to avoid file system fragmentation. This is configurable. More info here.

Do I Have to Worry About SQL Injection

See link at bottom for driver specific security notes; specifically one for PHP.

Instructions for the MongoDB server cluster are represented in two forms:

1. BSON
2. Javascript

With BSON being the main mechanism.

BSON

As a client program assembles a query in MongoDB, it builds a BSON object, not a string. Thus traditional SQL Injection attacks are not a problem. More details and some nuances are covered below.

MongoDB queries are represented as BSON objects. Typically the programming language gives a convenient way to build these objects that is injection free. For example in C++ one would write:

```
BSONObj my_query = BSON( "name" << a_name );
auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons", my_query);
```

my_query then will have a value such as { name : "Joe" }. If my_query contained special characters such as ", :, {, etc., nothing bad happens, they are just part of the string.

Javascript

Some care is appropriate when using server-side Javascript. For example when using the $where statement in a query, do not concatenate user supplied data to build Javascript code; this would be analogous to a SQL injection vulnerability. Fortunately, most queries in MongoDB can be expressed without Javascript. Also, we can mix the two modes. It's a good idea to make all the user-supplied fields go straight to a BSON field, and have your Javascript code be static and passed in the $where field.

If you need to pass user-supplied values into a $where clause, a good approach is to escape them using the CodeWScope mechanism. By setting the user values as variables in the scope document you will avoid the need to have them eval'ed on the server-side.

If you need to use db.eval() with user supplied values, you can either use a CodeWScope or you can supply extra arguments to your function. Something like: db.eval(function(userVal){...}, user_value); This will ensure that user_value gets sent as data rather than code.

Dollar Sign Operator Escaping

Field names in MongoDB query language have semantic meaning. The dollar sign is a reserved character and is used to represent operators, such as $inc. Some care must be taken to assure users can not delivery $operators without the application's expectation.

For example, sometimes it is useful to build a BSON object where the key is user-provided. In these situations, keys will need to have substitutions for the reserved $ and . characters. If you are unsure what characters to use, the Unicode full width equivalents aren't a bad choice: U+FF04 () and U+FFOE ()

For example:

```
BSONObj my_object = BSON( a_key << a_name );
```

The user may have supplied a $ value within a_key. my_object could be { $where : "things" }. Here we can look at a few cases:

- Inserting. Inserting into the the database will do no harm. We are not executing this object as a query, we are inserting the data in the database.
  Note: properly written MongoDB client drivers check for reserved characters in keys on inserts.
- Update. update(query, obj) allows $ operators in the obj field. $where is not supported in update. Some operators are possible that manipulate the single document only -- thus, the keys should be escaped as mentioned above if reserved characters are possible.
- Querying. Generally this is not a problem as for { x : user_obj }, dollar signs are not top level and have no effect. In theory one might let the user build a query completely themself and provide it to the database. In that case checking for $ characters in keynames is important. That however would be a highly unusual case.

One way to handle user-generated keys is to always put them in sub-objects. Then they are never at top level (where $operators live) anyway.

See Also

- Do I have to worry about SQL injection? (mailing list thread)

- Driver-specific security concerns:

How does concurrency work

- mongos
- mongod
- Viewing operations in progress
- Administrative commands and locking
- Operations
- Javascript
  - Group Command
  - MapReduce

The documentation below covers for MongoDB v2.0. v2.0 typically achieves significantly more concurrency in disk activity than v1.8 did. v2.2 will include substantial enhancements with concurrency work being by far the #1 item on the v2.2 road map.

mongos

For sharded environments, each `mongos` process can perform any number of operations concurrently. This results in downstream operations to mongod instances. Execution of operations at each mongod is independent; that is, one mongod does not block another.

mongod

The `mongod` process uses a modified reader/writer lock with dynamic yielding on page faults and long operations. Any number of concurrent read operations are allowed, but a write operation can block all other operations.

`mongod` threads *yield* their lock (read or write) in two classes of situations:

- yield-on-page-fault – v2.0 implements a yield-on-page-fault feature which results in much more concurrency than one would achieve with a pure reader/writer lock. For common operational cases, file system page faults are detected in advanced and handled outside of any lock, then the lock is resumed. Not all fault situations yield, but many do. This results in v2.0 having much better concurrency in practice than v1.8.
- yield-on-long-operation – `mongod` also yields periodically on common operations that are extremely long running. The goal here is to allow interleaving so that other operations which are quick-running can execute soon. Operations which yield include the following:
  - queries
  - multi document updates
  - multi document removes/deletes
  - bulk inserts

Write lock acquisition is *greedy*: a pending write lock acquisition will prevent further read lock acquisitions until fulfilled. Thus yielding by reads can be important.

Collection level locking is under development. SERVER-1240.

Viewing operations in progress

Use `db.currentOp()` to view operations in progress, and `db.killOp()` to terminate an operation.

Administrative commands and locking

Certain administrative commands can exclusively lock the `mongod` process for extended periods of time. Specifically, these commands block for long periods:

- foreground index creation
- reindex
- compact command

- repair database
- creating a very large (many gigabytes) capped collection
- validate collection command
- lock and fsync command

On a small collection, the lock time may only be a few seconds. On very large collections, take the `mongod` instance offline so that clients are not effected. For example if the server is part of a replica set, let other members service load while maintenance is in progress.

The following commands are fast and will not block the system excessively:

- dropIndex
- getLastError
- isMaster
- replSetGetStatus
- serverStatus
- auth
- addUser

Operations

Operation	Lock type	Notes
query	read lock	yields (periodically)
get more from a cursor	read lock	yields
insert	write lock	Inserts are normally fast and short-lived operations; exception is if the collection has many indexes and they do not fit in RAM, or if freelists are extremely long.
remove	write lock	yields (both periodically and when faults would occur)
update	write lock	yields (both periodically and when faults would occur)
map/reduce	at times locked	Allows substantial concurrent operation but exclusive to other javascript execution.
create index	See notes	Default "foreground mode" building blocks for extended periods of time.
db.eval()	write-lock/none	Substantial blocking without "nolock" option
getLastError command	non-blocking	
serverStatus command	non-blocking	
aggregate command	read lock	until $out/$tee is implemented only reads

Javascript

Only one thread in the `mongod` process executes Javascript at a time (other database operations are often possible concurrent with this). Note ticket https://jira.mongodb.org/browse/SERVER-4258 will allow multi-threading.

Group Command

The group command takes a read lock and does not allow any other threads to execute JavaScript while it is running.

MapReduce

The mapreduce operation is composed of many small events: reads from the input collection, executions of map(), executions of reduce(), writes to the output collection, etc.

There is a javascript lock so that only one thread can execute JS code at one point in time. But most JS steps of the MR (e.g. a single map()) are very short and consequently the lock is yielded very often. Note ticket https://jira.mongodb.org/browse/SERVER-4258 will allow multi-threading.

There are also several non-JavaScript operations that MapReduce performs that take locks:

- It reads from a collection (read lock yielded every 100 documents)
- It inserts documents into the temporary collection (write lock for a single write)
- It creates a final collection or merges into or replaces an existing collection (write lock)

The result is that while single-threaded, several MR jobs can be interleaved and appear to run in parallel.

Note the problematic lock can be the final write lock during post-processing, which is used to make results appear atomically. This lock can take a long time in "merge" or "reduce" output mode. A flag to disable atomicity has been added as per https://jira.mongodb.org/browse/SERVER-2581 (v2.1.1).

SQL to Mongo Mapping Chart

MySQL executable	Oracle executable	Mongo executable
`mysqld`	`oracle`	`mongod`
`mysql`	`sqlplus`	`mongo`

MySQL term	Mongo term/concept
database	database
table	collection
index	index
row	BSON document
column	BSON field
join	embedding and linking
primary key	_id field
group by	aggregation

MongoDB queries are expressed as JSON (BSON) objects. The following chart shows examples as both SQL and in Mongo Query Language syntax.

The query expression in MongoDB (and other things, such as index key patterns) is represented as JSON (BSON). However, the actual verb (e.g. "find") is done in one's regular programming language; thus the exact forms of these verbs vary by language. The examples below are Javascript and can be executed from the mongo shell.

SQL Statement	Mongo Statement
`CREATE TABLE USERS (a Number, b Number)`	implicit; can also be done explicitly with `db.createCollection("mycoll")`
`ALTER TABLE users ADD ...`	implicit
`INSERT INTO USERS VALUES(3,5)`	`db.users.insert({a:3,b:5})`
`SELECT a,b FROM users`	`db.users.find({}, {a:1,b:1})`
`SELECT * FROM users`	`db.users.find()`

`SELECT * FROM users WHERE age=33`	`db.users.find({age:33})`
`SELECT a,b FROM users WHERE age=33`	`db.users.find({age:33}, {a:1,b:1})`
`SELECT * FROM users WHERE age=33 ORDER BY name`	`db.users.find({age:33}).sort({name:1})`
`SELECT * FROM users WHERE age>33`	`db.users.find({age:{$gt:33}})`
`SELECT * FROM users WHERE age!=33`	`db.users.find({age:{$ne:33}})`
`SELECT * FROM users WHERE name LIKE "%Joe%"`	`db.users.find({name:/Joe/})`
`SELECT * FROM users WHERE name LIKE "Joe%"`	`db.users.find({name:/^Joe/})`
`SELECT * FROM users WHERE age>33 AND age<=40`	`db.users.find({'age':{$gt:33,$lte:40}})`
`SELECT * FROM users ORDER BY name DESC`	`db.users.find().sort({name:-1})`
`SELECT * FROM users WHERE a=1 and b='q'`	`db.users.find({a:1,b:'q'})`
`SELECT * FROM users LIMIT 10 SKIP 20`	`db.users.find().limit(10).skip(20)`
`SELECT * FROM users WHERE a=1 or b=2`	`db.users.find( { $or : [ { a : 1 } , { b : 2 } ] } )`
`SELECT * FROM users LIMIT 1`	`db.users.findOne()`

`SELECT order_id FROM orders o, order_line_items li WHERE li.order_id=o.order_id AND li.sku=12345`	`db.orders.find({"items.sku" :12345},{_id:1})`
`SELECT customer.name FROM customers,orders WHERE orders.id="q179" AND orders.custid=customer.id`	`var o = db.orders.findOne({_id:"q179" }); var name = db.customers.findOne({_id:o.custid})`
`SELECT DISTINCT last_name FROM users`	`db.users.distinct('last_name')`
`SELECT COUNT(*y) FROM users`	`db.users.count()`
`SELECT COUNT(*y) FROM users where AGE > 30`	`db.users.find({age: {'$gt': 30}}).count()`
`SELECT COUNT(AGE) from users`	`db.users.find({age: {'$exists': true }}).count()`
`CREATE INDEX myindexname ON users(name)`	`db.users.ensureIndex({name:1})`
`CREATE INDEX myindexname ON users(name,ts DESC)`	`db.users.ensureIndex({name:1,ts:-1})`
`EXPLAIN SELECT * FROM users WHERE z=3`	`db.users.find({z:3}).explain()`
`UPDATE users SET a=1 WHERE b='q'`	`db.users.update({b:'q'}, {$set:{a:1}}, false, true)`

`UPDATE users SET a=a+2 WHERE b='q'`	`db.users.update({b:'q'}, {$inc:{a:2}}, false, true)`
`DELETE FROM users WHERE z="abc"`	`db.users.remove({z:'abc'});`

More examples, specifically aggregation examples, here

See Also

- The MongoDB Manual Pages are a good place to learn more.
- SQL to Shell to C++

SQL to Shell to C++

MongoDB queries are expressed as JSON (BSON) objects. This quick reference chart shows examples as SQL, Mongo shell syntax, and Mongo C++ driver syntax.

A query expression in MongoDB (and other things, such as an index key pattern) is represented as BSON. In C++ you can use BSONObjBuilder (aka bson::bob) to build BSON objects, or the BSON() macro. The examples below assume a connection c already established:

```
using namespace bson;
DBClientConnection c;
c.connect("somehost");
```

Several of the C++ driver methods throw mongo::DBException, so you will want a try/catch statement as some level in your program.

SQL	Mongo Shell	C++ Driver

`INSERT INTO USERS VALUES(1,1)`	`db.users.insert({a:1,b:1})`	`// GENOID is optional. if not done by client, server will add an _id`
		`c.insert("mydb.users", BSON(GENOID<<"a"<<1<<"b"<<1));`
		`// then optionally:`
		`string err = c.getLastError();`

`SELECT a,b FROM users`	`db.users.find({}, {a:1,b:1})`	`auto_ptr<DBClientCursor> cursor =` `  c.query("mydb.users", Query(), 0, 0, BSON("a"<<1<<"b"<<1));`

SELECT * FROM users	db.users.find()	auto_ptr<DBClientCursor> cursor = c.query("mydb.users", Query());

SELECT *
FROM users
WHERE age=33
	db.users.find({age:33})	auto_ptr<DBClientCursor>
		cursor =
		  c.query("mydb.users",
		QUERY("age"<<33))
		// or:
		auto_ptr<DBClientCursor>
		cursor =
		  c.query("mydb.users",
		BSON("age"<<33))

SELECT * FROM users WHERE age=33 ORDER BY name	db.users.find({age:33}).sort({name:1})	auto_ptr<DBClientCursor> cursor = c.query("mydb.users", QUERY("age"<<33).sort("name"));
SELECT * FROM users WHERE age>33 AND age<=40	db.users.find({'age':{$gt:33},{$lte:40}})	auto_ptr<DBClientCursor> cursor = c.query("mydb.users", QUERY("age" <<GT<<33<<LTE<<40));
CREATE INDEX myindexname ON users(name)	db.users.ensureIndex({name:1})	c.ensureIndex("mydb.users", BSON("name"<<1));
SELECT * FROM users LIMIT 10 SKIP 20	db.users.find().limit(10).skip(20)	auto_ptr<DBClientCursor> cursor = c.query("mydb.users", Query(), 10, 20);
SELECT * FROM users LIMIT 1	db.users.findOne()	bo obj = c.findOne("mydb.users", Query());

`SELECT`
`DISTINCT`
`last_name`
`FROM users`
`WHERE x=1`
	`db.users.distinct('last_name',{x:1})`	`// no helper for`
		`distinct yet in c++`
		`driver, so send command`
		`manually`
		`bo cmdResult;`
		`bool ok = c.runCommand(`
		`  "mydb", BSON(`
		`"distinct" << "users"`
		`       << "key" <<`
		`"last_name"`
		`       << "query" <<`
		`BSON("x"<<1)),`
		`  cmdResult);`
		`list<bo> results;`
		`cmdResult["values"`
		`].Obj().Vals(results);`

`SELECT`
`COUNT(*y)`
`FROM users`
`where AGE >`
`30`
	`db.users.find({age: {'$gt': 30}}).count()`	`unsigned long long n =`
		`c.count("mydb.users",`
		`QUERY("age:"<<GT<<30));`

`UPDATE users`
`SET a=a+2`
`WHERE b='q'`
	`db.users.update({b:'q'}, {$inc:{a:2}},`
	`false, true)`
		`c.update("mydb.users",`
		`QUERY("b"<<"q"), BSON(`
		`"$inc"<<BSON("a"<<2)),`
		`false, true);`
		`// then optionally:`
		`string err =`
		`c.getLastError();`
		`bool ok = err.empty();`

`DELETE FROM`
`users WHERE`
`z="abc"`
	`db.users.remove({z:'abc'});`	`c.remove("mydb.users",`
		`QUERY("z"<<"abc"));`
		`// then optionally:`
		`string err =`
		`c.getLastError();`

See Also

- Several more examples (in shell syntax) are on the SQL to Mongo Mapping Chart page.
- C++ Language Center

What is the Compare Order for BSON Types

MongoDB allows objects in the same collection which have values which may differ in type. When comparing values from different types, a convention is utilized as to which value is less than the other. This (somewhat arbitary but well defined) ordering is listed below.

Note that some types are treated as equivalent for comparison purposes -- specifically numeric types which undergo conversion before comparison.

See also the BSON specification.

- Null

* Numbers (ints, longs, doubles)
* Symbol, String
* Object
* Array
* BinData
* ObjectID
* Boolean
* Date, Timestamp
* Regular Expression

Example (using the mongo shell):

```
> t = db.mycoll;
> t.insert({x:3});
> t.insert( {x : 2.9} );
> t.insert( {x : new Date()} );
> t.insert( {x : true } )
> t.find().sort({x:1})
{ "_id" : ObjectId("4b03155dce8de6586fb002c7"), "x" : 2.9 }
{ "_id" : ObjectId("4b03154cce8de6586fb002c6"), "x" : 3 }
{ "_id" : ObjectId("4b031566ce8de6586fb002c9"), "x" : true }
{ "_id" : ObjectId("4b031563ce8de6586fb002c8"), "x" : "Tue Nov 17 2009 16:28:03 GMT-0500 (EST)" }
```

MinKey and MaxKey

In addition to the above types MongoDB internally uses a special type for MinKey and MaxKey which are less than, and greater than all other possible BSON element values, respectively.

From the mongo Javascript Shell

For example we can continue our example from above adding two objects which have x key values of MinKey and MaxKey respectively:

```
> t.insert( { x : MaxKey } )
> t.insert( { x : MinKey } )
> t.find().sort({x:1})
{ "_id" : ObjectId("4b04094b7c65b846e2090112"), "x" : { $minKey : 1 } }
{ "_id" : ObjectId("4b03155dce8de6586fb002c7"), "x" : 2.9 }
{ "_id" : ObjectId("4b03154cce8de6586fb002c6"), "x" : 3 }
{ "_id" : ObjectId("4b031566ce8de6586fb002c9"), "x" : true }
{ "_id" : ObjectId("4b031563ce8de6586fb002c8"), "x" : "Tue Nov 17 2009 16:28:03 GMT-0500 (EST)" }
{ "_id" : ObjectId("4b0409487c65b846e2090111"), "x" : { $maxKey : 1 } }
```

From C++

See also the Tailable Cursors page for an example of using MinKey from C++. See also minKey and maxKey definitions in jsobj.h.

BSON Type Comparison in Queries

Comparing two distinct BSON types in a query always returns false.

```
> t.insert( { x : 1 } )
> t.insert( { x: 'abc' } )
> t.find().sort( { x : 1 } )
{ "_id" : ObjectId("4e51589a029fa59b62174306"), "x" : 10 }
{ "_id" : ObjectId("4e24c0ddb497b6f1d53beca1"), "x" : "abc" }
> db.t.find({x: {$gt:''}})
{ "_id" : ObjectId("4e24c0ddb497b6f1d53beca1"), "x" : "abc" }
> db.t.find({x: {$lt:''}})
```

Admin Zone

* Components
* Journaling

- MongoDB Monitoring Service
- The Database and Caching
- Production Notes
- Replication
- Sharding
- Hosting Center
- Monitoring and Diagnostics
- Backups
- Durability and Repair
- Security and Authentication
- Admin UIs
- Starting and Stopping Mongo
- GridFS Tools
- DBA Operations from the Shell
- Architecture and Components
- Windows
- Troubleshooting

Community Admin-Related Articles

- boxedice.com - notes from a production deployment
- Survey of Admin UIs for MongoDB
- MongoDB Nagios Check
- MongoDB Cacti Graphs

See Also

- Commands in Developer Zone

Components

Redirection Notice
This page should redirect to Architecture and Components.

Journaling

MongoDB v1.7.5+ supports write-ahead journaling of operations to facilitate fast crash recovery and durability in the storage engine.

Disabling/Enabling

In version 1.9.2+, journaling is enabled by default for 64-bit platforms. You can disable journaling with the mongod `--nojournal` command line option. For versions < 1.9.2 or 32-bit platforms, you can enable journaling with the `--journal` command line option.

It is OK to disable journaling after running with journaling by simply shutting down `mongod` cleanly and restarting with `--nojournal`. The reverse is also OK; shutdown cleanly and restart without `--nojournal`.

MongoDB may determine that it is faster to preallocate journal files than to create them as needed. If MongoDB decides to preallocate the files, it will not start listening on port 27017 until this process completes, which can take a few minutes. This means that your applications and the shell will not be able to connect to the database immediately on initial startup. Check the logs to see if MongoDB is busy preallocating. It will print the standard "waiting for connections on port *whatever*" when it has finished.

Journal Files

With journaling enabled, journal files will be created in a `journal/` subdirectory under your chosen db path. These files are write-ahead redo logs. In addition, a last sequence number file, `journal/lsn`, will be created. A clean shutdown removes all files under `journal/`.

The Mongo data files (*database*.ns, *database*.0, *database*.1, ...) have the same format as in previous releases. Thus, the upgrade process is seamless, and a rollback would be seamless too. (If you roll back to a pre v1.7.5 release, try to shut down cleanly first. Regardless, remove the journal/ directory before starting the pre v1.7.5 version of mongod.)

Recovery

On a restart after a crash, journal files in `journal/` will be replayed before the server goes online. This will be indicated in the log output. You do not need to run a repair.

The journal Subdirectory

You may wish, before starting `mongod` to symlink the `journal/` directory to a dedicated hard drive to speed the frequent (fsynced) sequential writes which occur to the current journal file.

Group Commits

MongoDB performs *group commits* (batch commits) when journaling. This means that a series of operations over many milliseconds are committed all at once. This is done to achieve high performance.

Group commits are performed approximately every 100ms by default. In version 1.9.2+, you can set this interval yourself using the `--journalCommitInterval` command line option. The allowed range is 2 to 300 milliseconds.

Commit Acknowledgement

You can wait for group commit acknowledgement with the getLastError Command. In versions before 1.9.0 using getLastError + fsync would do this, in newer versions the "j" option has been specifically created for this purpose.

In version 1.9.2+ the group commit delay is shortened when a commit acknowledgement (getLastError + j) is pending; this can be as little as 1/3 of the normal group commit interval.

FAQ

If I am using replication, can some members use journaling and others not?

Yes.

How's performance?

Read performance should be the same. Write performance should be very good but there is some overhead over the non-durable version as the journal files must be written. If you find a case where there is a large difference in performance between running with and without journaling, please let us know so we can tune it. Additionally, some performance tuning enhancements in this area are already queued for v1.8.1+.

Can I use the journaling feature to perform safe hot backups?

Yes, see Backups with Journaling Enabled.

32 bit nuances?

There is extra memory mapped file activity with journaling. This will further constrain the limited db size of 32 bit builds. Thus, for now journaling by default is disabled on 32 bit systems.

When did the --journal option change from --dur?

In 1.8 the option was renamed to `--journal`, but the old name is still accepted for backwards compatibility; please change to `--journal` if you are using the old option.

Will the journal replay have problems if entries are incomplete (like the failure happened in the middle of one)?

Each journal (group) write is consistent and won't be replayed during recovery unless it is complete.

How many times is data written to disk when replication and journaling are both on?

In v1.8, for an insert, four times. The object is written to the main collection, and also the oplog collection (so that is twice). Both of those writes are journaled as a single mini-transaction in the journal file (the files in /data/db/journal). Thus 4 times total.

There is an open item in to reduce this by having the journal be compressed. This will reduce from 4x to probably ~2.5x.

The above applies to collection data and inserts which is the worst case scenario. Index updates are written to the index and the journal, but not the oplog, so they should be 2X today not 4X. Likewise updates with things like $set, $addToSet, $inc, etc. are compactly logged all around so those are generally small.

See Also

- Journaling Administration Notes
- getLastError command for controlling writes per operation
- Durability Internals
- Presentations
  - Recorded Webinar on Journaling (April 2011)
  - Journaling and the Storage Engine (May 2011)

Journaling Administration Notes

- Journal Files (*e.g.* journal/j._0)
- Prealloc Files (*e.g.* journal/prealloc.0)
- serverStatus command
- journalLatencyTest Command

Journal Files (e.g. journal/j._0)

Journal files are append-only and are written to the `journal/` directory under the dbpath directory (which is `/data/db/` by default).

Journal files are named `j._0`, `j._1`, etc. When a journal file reached 1GB in size, a new file is created. Old files which are no longer needed are rotated out (automatically deleted). Unless your write bytes/second rate is extremely high, you should have only two or three journal files.

Note: in more recent versions, the journal files are smaller when using the `--smallfiles` command line option.

Prealloc Files (e.g. journal/prealloc.0)

`mongod` will create `prealloc` files in the journal directory under some circumstances to minimize journal write latency. On some filesystems, appending to a file and making it larger can be slower than writing to a file of a predefined size. `mongod` checks this at startup and if it finds this to be the case will use preallocated journal files. If found to be helpful, a small pool of prealloc files will be created in the journal directory before startup begins. This is a one time initiation and does not occur with future invocations. Approximately 3GB of files will be preallocated (and truly prewritten, not sparse allocated) - thus in this situation, expect roughly a 3 minute delay on the first startup to preallocate these files.

If you don't want to wait three minutes on startup, you can preallocate the files using another instance of `mongod` and then move them to your normal dbpath before starting with journaling. For example, if you had an instance of `mongod` running on port 27017 with a dbpath of */data/db* (the defaults), you could preallocate journal files for it with:

```
$ mkdir ~/tmpDbpath
$ mongod --port 10000 --dbpath ~/tmpDbpath --journal
# startup messages
# .
# .
# .
# wait for prealloc to finish
Thu Mar 17 10:02:52 [initandlisten] preallocating a journal file ~/tmpDbpath/journal/prealloc.0
Thu Mar 17 10:03:03 [initandlisten] preallocating a journal file ~/tmpDbpath/journal/prealloc.1
Thu Mar 17 10:03:14 [initandlisten] preallocating a journal file ~/tmpDbpath/journal/prealloc.2
Thu Mar 17 10:03:25 [initandlisten] flushing directory ~/tmpDbpath/journal
Thu Mar 17 10:03:25 [initandlisten] flushing directory ~/tmpDbpath/journal
Thu Mar 17 10:03:25 [initandlisten] waiting for connections on port 10000
Thu Mar 17 10:03:25 [websvr] web admin interface listening on port 11000
# then Ctrl-C to kill this instance
^C
$ mv ~/tmpDbpath/journal /data/db/
$ # restart mongod on port 27017 with --journal
```

prealloc files do not contain data, but are rather simply preallocated files that are ready to use that are truly preallocated by the file system (i.e. they are not "sparse"). It is thus safe to remove them, but if you restart mongod with journaling, it will create them again if they are missing.

serverStatus command

The serverStatus command now includes some statistics regarding journaling.

journalLatencyTest Command

You can use the `journalLatencyTest` command to measure how long it takes on your volume to write to the disk (including fsyncing the data) in an append-only fashion.

```
> use admin
> db.runCommand("journalLatencyTest")
```

You can run this command on an idle system to get a baseline sync time for journaling. In addition, it is safe to run this command on a busy system to see the sync time on a busy system (which may be higher if the journal directory is on the same volume as the data files).

In version 1.9.2+ you can set the group commit interval, using `--journalCommitInterval` command-line option, to between 2 and 300 milliseconds (default is 100ms). The actual interval will be the maximum of this setting and your disk latency as measured above.

journalLatencyTest is also a good way to check if your disk drive is buffering writes in its local cache. If the number is very low (e.g., less than 2ms) and the drive is non-ssd, the drive is probably buffering writes. In that case, you will want to enable cache write-through for the device in your operating system. (Unless you have a disk controller card with battery backed ram, then this is a good thing.)

MongoDB Monitoring Service

MongoDB Monitoring Service is a free SaaS solution for proactive monitoring of your MongoDB cluster(s). MMS's web interface features charts, custom dashboards, and automated alerting; and since it runs in the cloud, MMS requires minimal setup and configuration. Within minutes your devops and systems administration teams can manage and optimize your MongoDB deployment, and derive valuable insights from key operational metrics.

Getting Started

MMS is free and available to everyone in the MongoDB community. To get started with MMS now, visit the MMS setup page.

- Create an account at mms.10gen.com.
- Download and install the MMS agent on your cluster.
- Within minutes, data will be visible on mms.10gen.com.

Docs

Documentation for MMS is available at:

- https://mms.10gen.com/help/

Blog Posts

- Announcing the MongoDB Monitoring Service - 10gen Blog (Sept 2011)
- Getting Started with MMS - Kristina Chodorow's Blog (Sept 2011)
- Getting Started with Mongo Monitoring Service (MMS) on OpenShift - OpenShift Blog (Dec 2011)
- Monitoring mongoDB with MMS (from 10gen) - Dec 2011

The Database and Caching

With relational databases, object caching is usually a separate facility (such as memcached), which makes sense as even a RAM page cache hit is a fairly expensive operation with a relational database (joins may be required, and the data must be transformed into an object representation). Further, memcached type solutions are more scaleable than a relational database.

Mongo eliminates the need (in some cases) for a separate object caching layer. Queries that result in file system RAM cache hits are very fast as the object's representation in the database is very close to its representation in application memory. Also, the MongoDB can scale to (almost) any level and provides an object cache and database integrated together, which is very helpful as there is no risk of retrieving stale data from the cache. In addition, the complex queries a full DBMS provides are also possible.

Production Notes

Backups

- Backups Doc Page
- Import Export Tools

TCP Port Numbers

Default TCP port numbers for MongoDB processes:

- Standalone `mongod` : 27017
- `mongos` : 27017
- shard server (`mongod --shardsvr`) : 27018
- config server (`mongod --configsvr`) : 27019
- web stats page for `mongod` : add 1000 to port number (28017, by default)

Firewall rules for these ports and connections are available here.

Linux

General Unix Notes

- Turn off *atime* for the data volume
- Set file descriptor limit and user process limit to 4k+ (see *etc/limits* and *ulimit*)
- **Do not** use large VM pages with Linux (more info)
- Use `dmesg` to see if box is behaving strangely
- Try to disable NUMA in your BIOS. If that is not possible see NUMA
- Minimize clock skew between your hosts by using ntp; linux distros usually include this by default, but check and install the ntpd package if it isn't already installed

Linux Kernel Versions

Some have reported skepticism on behavior of Linux 2.6.33-31 and 2.6.32 kernel. 2.6.36 is given a thumbs up by the community.

For those running databases on ext4 filesystems, a 2.6.23 kernel is required for efficient filesystem preallocation, 2.6.25 is required for XFS support of the same feature. High filesystem I/O following the allocation of new database files is one symptom of this problem.

Checking Disk IO

`iostat -txm 2`

If the timestamp feature is not available, `iostat -xm 2`

Checking Network IO

- atop - Holistic system top

- Munin
- ethtool eth0 - check network port speed
- bwm-ng
- iptraf

Storage

Solid State Disks (SSDs)

- See SSD page

RAID

Typically we recommend using RAID-10.

RAID-5 and RAID-6 can be slow.

See also the ec2 page for comments on EBS striping.

Linux File Systems

MongoDB uses large files for storing data, and preallocates these. These filesystems seem to work well:

- ext4 (kernel version >= 2.6.23)
- xfs (kernel version >= 2.6.25)

In addition to the file systems above you might also want to (explicitly) disable file/directory modification times by using these mount options:

- noatime (also enables nodiratime)

We have found **`ext3`** to be very slow in allocating files (or removing them) as well as access within large files is also poor.

Remote mounts

We have found that some versions of NFS perform very poorly and do not recommend using NFS. See the NFS page for more information.

Amazon elastic block store (EBS) seems to work well up to its intrinsic performance characteristics, when configured well.

Swap

It is useful for the linux kernel to have swap space to use in emergencies. Because of the way MongoDB memory maps the database files none of this data will ever end up in swap; this means that on a healthy system the swap space will rarely be used on a system only running MongoDB. Having swap can keep the kernel from killing MongoDB when physical memory limits are reached.

You may also want to look at using something which compresses swap/memory like compcache.

What Hardware?

MongoDB tends to run well on virtually all hardware. In fact it was designed specifically with commodity hardware in mind (to facilitate cloud computing); that said it works well on very large servers too. That said if you are about to buy hardware here are a few suggestions:

- Fast CPU clock speed is helpful.
- Many cores helps but does not provide a high level of marginal return, so don't spend money on them. (This is both a consequence of the design of the program and also that memory bandwidth can be a limiter; there isn't necessarily a lot of computation happening inside a database).
- NUMA is not very helpful as memory access is not very localized in a database. Thus non-NUMA is recommended; or configure NUMA as detailed elsewhere in this document.
- RAM is good.
- SSD is good. We have had good results and have seen good price/performance with SATA SSDs; the (typically) more upscale PCI SSDs work fine too.
- Commodity (SATA) spinning drives are often a good option as the speed increase for random I/O for more expensive drives is not that dramatic (only on the order of 2x) – spending that money on SSDs or RAM may be more effective.

Tips

- Handling Halted Replication
- Starting and Stopping the Database

iostat

On Linux, use the iostat command to check if disk I/O is a bottleneck for your database.

We generally find the form:

```
iostat -xm 2
```

to work well. (Use a number of seconds with iostat, otherwise it will display stats since server boot, which is not very useful.)

Use the `mount` command to see what device your `/data/db` directory resides on.

Fields

- %util - this is the most useful field for a quick check, it indicates what percent of the time the device/drive is in use. If the number if near 100%, your server may be physical disk I/O bound. (There are some volume situations where this statistic overstates, but most often it is correct.)
- r/s - reads per second.
- w/s - writes per second
- rMB/s - read megabytes per second
- wMB/s - write megabytes per second
- avgrq-sz - average request size. The smaller this number, the more random your IO operations are. This is in sectors : typically sectors are 512 bytes, so multiply by 0.5 to see average request size in kilobytes.

On Windows Server use the performance monitor utility.

NFS

We have found that some versions of NFS perform very poorly, or simply don't work, and do not suggest using NFS. (We'd love to hear from you if you are using NFS and what results you are getting, either great or not great.)

NUMA

Linux, NUMA and MongoDB tend not to work well together. If you are running MongoDB on numa hardware, we recommend turning it off (running with an interleave memory policy). Problems will manifest in strange ways, such as massive slow downs for periods of time or high system cpu time.

numactl

To turn off NUMA, start mongod with

```
numactl --interleave=all ${MONGODB_HOME}/bin/mongod --config conf/mongodb.conf
```

proc settings

```
echo 0 > /proc/sys/vm/zone_reclaim_mode
```

You can change `zone_reclaim_mode` without restarting `mongod`. For more information on this setting see http://www.kernel.org/doc/Documentation/sysctl/vm.txt.

Testing

On Linux, `mongod` v2.0+ checks these settings on startup and prints a warning if they do not match the recommendations.

References

The MySQL “swap insanity” problem and the effects of the NUMA architecture describes the effects of NUMA on databases. This blog post was aimed at problems NUMA created for MySQL, but the issues are the same. The posting describes the NUMA architecture and goals, and how these are incompatible with the working of databases.

SSD

> ⚠ We are not experts on solid state drives, but tried to provide some information here that would be helpful. Comments very welcome.

Multiple MongoDB users have reported good success running MongoDB databases on solid state drives.

Write Endurance

Write endurance with solid state drives vary. SLC drives have higher endurance but newer generation MLC (and eMLC) drives are getting better.

As an example, the MLC Intel 320 drives specify endurance of 20GB/day of writes for five years. If you are doing small or medium size random reads and writes this is sufficient. The Intel 710 series is the enterprise-class models and have higher endurance.

If you intend to write a full drive's worth of data writing per day (and every day for a long time), this level of endurance would be insufficient. For large sequential operations (for example very large map/reduces), one could write far more than 20GB/day. Traditional hard drives are quite good at sequential I/O and thus may be better for that use case.

- Blog post on SSD lifespan

Reserve some unpartitioned space

Some users report good results when leaving 20% of their drives completely unpartitioned. In this situation the drive knows it can use that space as working space. Note formatted but empty space may or may not be available to the drive depending on TRIM support which is often lacking.

smartctl

On some devices, "smartctl -A" will show you the Media_Wearout_Indicator.

```
$ sudo smartctl -A /dev/sda | grep Wearout
233 Media_Wearout_Indicator 0x0032   099   099   000    Old_age   Always       -       0
```

Speed

A paper in ACM Transactions on Storage (Sep2010) listed the following results for measured 4KB peak random direct IO for some popular devices:

Device	Read IOPS	Write IOPS
Intel X25-E	33,400	3,120
FusionIO ioDrive	98,800	75,100

Intel's larger drives seem to have higher write IOPS than the smaller ones (up to 23,000 claimed for the 320 series). More info here.

Real-world results should be lower, but the numbers are still impressive.

Reliability

Some manufacturers specify relability stats indicating failure rates of approximately 0.6% per year. This is better than traditional drives (2% per year failure rate or higher), but still quite high and thus mirroring will be important. (And of course manufacture specs could be optimistic.)

Random reads vs. random writes

Random access I/O is the sweet spot for SSD. Historically random reads on SSD drives have been much faster than random writes. That said, random writes are still an order of magnitude faster than spinning disks.

Recently new drives have released that have much higher random write performance. For example the Intel 320 series, particular the larger capacity drives, has much higher random write performance than the older Intel X25 series drives.

PCI vs. SATA

SSD is available both as PCI cards and SATA drives. PCI is oriented towards the high end of products on the market.

Some SATA SSD drives now support 6Gbps sata transfer rates, yet at the time of this writing many controllers shipped with servers are 3Gbps. For random IO oriented applications this is likely sufficient, but worth considering regardless.

RAM vs. SSD

Even though SSDs are fast, RAM is still faster. Thus for the highest performance possible, having enough RAM to contain the working set of data from the database is optimal. However, it is common to have a request rate that is easily met by the speed of random IO's with SSDs, and SSD cost per byte is lower than RAM (and persistent too).

A system with less RAM and SSDs will likely outperform a system with more RAM and spinning disks. For example a system with SSD drives and 64GB RAM will often outperform a system with 128GB RAM and spinning disks. (Results will vary by use case of course.)

One helpful characteristic of SSDs is they can facilitate fast "preheat" of RAM on a hardware restart. On a restart a system's RAM file system cache must be repopulated. On a box with 64GB RAM or more, this can take a considerable amount of time – for example six minutes at 100MB/sec, and much longer when the requests are random IO to spinning disks.

FlashCache

FlashCache is a write back block cache for Linux. It was created by Facebook. Installation is a bit of work as you have to build and install a kernel module. Sep2011: If you use this please report results in the mongo forum as it's new and everyone will be curious how well it works.

- http://www.facebook.com/note.php?note_id=388112370932

OS scheduler

One user reports good results with the noop IO scheduler under certain configurations of their system. As always caution is recommended on nonstandard configurations as such configurations never get as much testing...

Run mongoperf

mongoperf is a disk performance stress utility. It is not part of the mongo database, simply a disk exercising program. We recommend testing your SSD setup with mongoperf. Note that the random writes it are a worst case scenario, and in many cases MongoDB can do writes that are much larger.

Helpful links

- Benchmarks of SSD drives from various manufacturers
  - http://techreport.com/articles.x/20653/5
  - http://www.anandtech.com/show/4244/intel-ssd-320-review/3
- Intel SSD Models Comparison (scroll down)
- Intel 710 and 720 series info

Virtualization

Generally MongoDB works very well in virtualized environments, with the exception of OpenVZ.

EC2

Compatible. No special configuration requirements.

VMWare

Some suggest not using overcommit as they may cause issues. Otherwise compatible.

Cloning a VM is possible. For example you might use this to spin up a new virtual host that will be added as a member of a replica set. If Journaling is enabled, the clone snapshot will be consistent. If not using journaling, stop mongod, clone, and then restart.

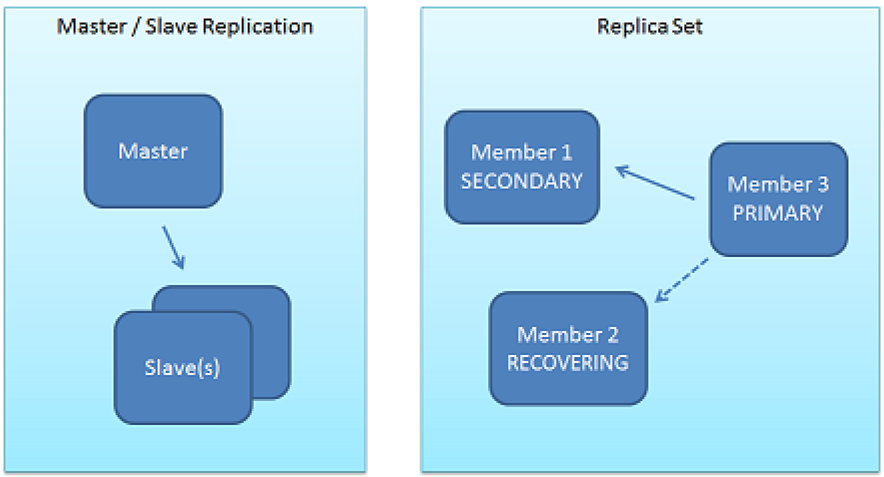
OpenVZ

Issues have been reported here.

Replication

MongoDB supports asynchronous replication of data between servers for failover and redundancy. Only one server (in the set/shard) is active for writes (the primary, or master) at a given time – this is to allow strong consistent (atomic) operations. One can optionally send read operations to the secondaries when eventual consistency semantics are acceptable.

Two forms of replication are available, Replica Sets and Master-Slave. Use Replica Sets – replica sets are a functional superset of master/slave and are handled by much newer, more robust code.



Each shard of a Mongo cluster is a replica set.

- Replica Set Documentation

Verifying propagation of writes with `getlasterror`

A client can block until a write operation has been replicated to N servers -- read more here .

See Also

- Master Slave Replication

Documentation of replica sets is available in the MongoDB Manual:

- Replication Documentation Index
- Replication Fundamentals
- Replication Architectures
- Replication Internals
- Replication Tutorials
- Replication Reference

Video

A MongoDB Replication Primer: Replica Sets in Practice - Presentation at MongoSV (Dec 2011)

About the local database

`mongod` reserves the database `local` for special functionality. It is special in that its contents are never replicated.

When authentication is used, authenticating against the `local` database is equivalent to authenticating against the `admin` database: it gives you permissions across all databases, not just `local`.

Replica Sets

Replica sets use the following collections in `local`:

- `local.system.replset` the replica set's configuration object is stored here. (View via the `rs.conf()` helper in the shell – or query it directly.)
- `local.oplog.rs` is a capped collection that is the oplog. You can use the `--oplogSize` command line parameter to set the size of this collection.
- `local.replset.minvalid` sometimes contains an object used internally by replica sets to track sync status

Master/Slave Replication

- Master
  - `local.oplog.$main` the "oplog"
  - `local.slaves`
- Slave
  - `local.sources`
- Other
  - `local.me`
  - `local.pair.*` (replica pairs, which are deprecated)

Verifying Propagation of Writes with getLastError

Please read the getLastError Command page first.

A client can await a write operation's replication to N servers (v1.6+). Use the getLastError Command with the parameter w:

```
// examples:

db.runCommand( { getlasterror : 1 , w : 2 } )

db.runCommand( { getlasterror : 1 , w : "majority" } )

db.runCommand( { getlasterror : 1 , w : "majority", j:true } )

db.runCommand( { getlasterror : 1 , w : "majority", wtimeout : 5000 } )
```

If `w` is not set, or equals one, the command may return almost immediately, implying the data is on one server (itself). If w is 2, then the data is on the current server and 1 other server (a secondary).

v2.0+ supports a "majority" value for `w` which indicates "await the data reaching a majority of members". This can be quite useful as a "cluster wide commit" of the write has occurred once the write has reached a majority of the (non-arbiter) members of a set.

The optional `wtimeout` parameter allows one to time out after a certain number of milliseconds (perhaps then returning an error to a user).

```
> db.runCommand({getlasterror : 1, w : 40, wtimeout : 3000})
{
        "err" : null,
        "n" : 0,
        "wtimeout" : true,
        "waited" : 3006,
        "errmsg" : "timed out waiting for slaves",
        "ok" : 0
}
```

The current implementation returns when the data has been delivered to `w` servers. The `j` option ensure that the operation has propagated to the journal, which means that the write operation will appear in the database even if the `mongod` experiences an unclean shutdown.

See also replica set configuration for information on how to change the getlasterror default parameters.

See Also

- getLastError Command
- Replica Set Design Concepts

Replica Sets

Overview

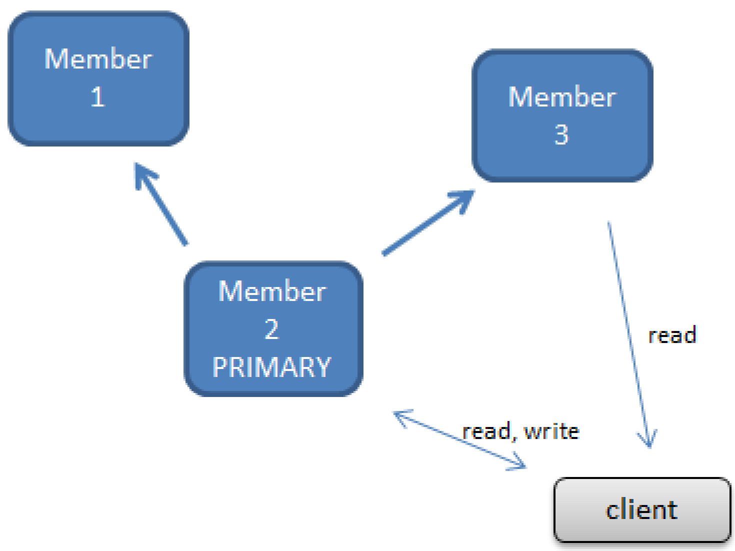
Replica sets are a form of asynchronous master/slave replication, adding automatic failover and automatic recovery of member nodes.

- A replica set consists of two or more nodes that are copies of each other. (*i.e.: replicas*)
- The replica set automatically elects a *primary* (master). No one member is intrinsically primary; that is, this is a share-nothing design.
- Drivers (and mongos) can automatically detect when a replica set primary changes and will begin sending writes to the new primary. (The mongos sharding process does this too.)

Replica sets have several common uses:

- Data Redundancy
- Automated Failover / High Availability
- Distributing read load
- Simplify maintenance (*compared to "normal" master-slave*)
- Disaster recovery

Getting started



- Why use replica sets?
- The basics
- Upgrading your client code
  - Reading from secondary (`slaveOkay`)
- How does replication work?
  - Oplog
  - Voting
  - Priorities
- Limitations
- **Tutorial**: single server replica set (*dev-only*)

Operation

- Configuration
- Administrative Commands

Advanced and More

- Data center awareness and tagging
- Authentication
- The http admin UI
- Cluster wide commit concepts

How-tos

- Sample Replica Set Config Session (PDF)
- Migrating to replica sets from master/slave replication
- Adding a New Set Member
- Adding an Arbiter
- Forcing a Member to be Primary
- Moving or Replacing a Member (also the same procedure for restoring a member)
- Reconfiguring when members are up

Troubleshooting

- Troubleshooting
- Resyncing a Very Stale Replica Set Member
- Reconfiguring when members are down

See Also

- Practical Replication Video
- Replica Sets Slides
- Webcast Demo of Replica Sets
- Replica set internals

Replica Sets - Basics

For production use you will want a minimum of three nodes in the replica set.

Either:

- 2 full nodes and 1 arbiter
- 3 full nodes

To avoid a single point of failure, these nodes must be on different computers.

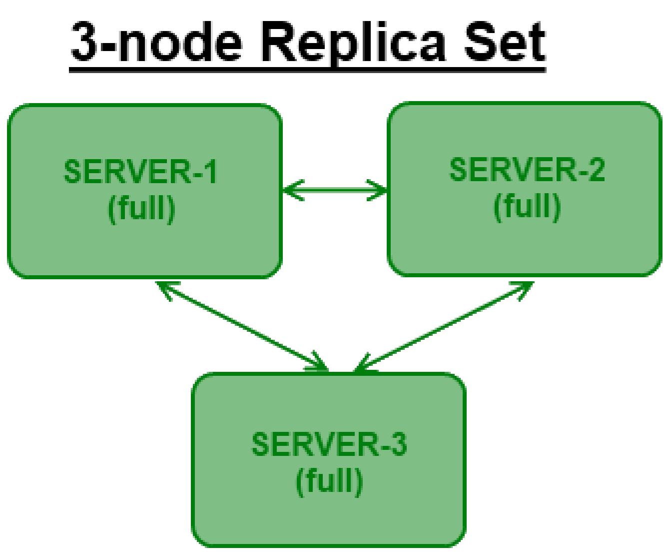
> ⚠ It is standard to have at least 2 nodes equipped to handle primary duties.

Basic Configuration

- Replica sets typically operate with 2 to 7 full nodes and possibly an arbiter.
- Within a given set, there should be an odd number of total nodes.
  - If full nodes are only available in even numbers, then an arbiter should be added to provide an odd number.
  - The **arbiter** is a lightweight `mongod` process whose purpose is to break ties when electing a primary.
  - The arbiter does not typically require a dedicated machine.

Example: 3 full servers

In this example, three servers are connected together in a single cluster. If the primary fails any of the secondary nodes can take over.



Getting Started – A sample session

The following is a simple configuration session for replica sets.

For this example assume a 3-node Replica set with 2 full nodes and one arbiter node. These servers will be named `sf1`, `sf2` and `sf3`.

Step 1: Start mongod with --replSet

On each of the servers start an instance of the mongo daemon:

```
sf1$ mongod --rest --replSet myset
sf2$ mongod --rest --replSet myset
sf3$ mongod --rest --replSet myset
```

> the `--replSet` parameter has the same value myset on all three instances.

Step 1a: Check the Replication UI (optional)

Visit http://sf1:28017/_replSet, this will give you an idea on the current status of the replica set. As you proceed through the remaining steps, refreshing this dashboard to see changes. See the docs here for more details.

Step 2: Initiate the replica set

Connect to mongo on sf1.

```
$ mongo --host sf1
> rs.initiate()
{
    "info2" : "no configuration explicitly specified -- making one",
    "info" : "Config now saved locally.  Should come online in about a minute.",
    "ok" : 1
}
>
```

> Initializing the replica set this way will cause the replica set to use the hostname of the current server in the replica set configuration. If your hostnames are not known to all mongo and application servers, you may need to initialize the hosts explicitly - see Replica Set Configuration for more details.

Step 3: Add nodes to the replica set

```
$ mongo --host sf1
> rs.add(“sf2”)
{ “ok” : 1 }
> rs.addArb(“sf3”)
{ “ok” : 1 }
```

Any operations that change data will now be replicated from `sf1` to `sf2`.

If `sf1` is shut down, you will see `sf2` take over as primary

Changing Client Code

- To leverage replica sets from your client code, you will need to modify your client connection code.
- The details for doing this will vary with each driver (*language*).

Replica Sets slaveDelay

Overview

A replica set member can be configured with a `slaveDelay:<secs>` option.

When `slaveDelay` is specified, the member will intentionally remain behind current in its application of replicated writes. This is typically used to keep a "rolling backup" that is a certain amount of time in the past. On a fat finger this can be quite useful to recover data from not too long in the past.

The value of `slaveDelay` is the number of seconds to trail the primary of the set. This effectively provides a copy of the data that lives some number of seconds in the past.

Example:
Assume that the current time is 09:52 and a secondary is configured with `slaveDelay:3600`. That secondary will only apply operations up to 08:52. (3600 seconds behind)

Notes

- When using `slaveDelay` feature, both the oplog and the data will be intentionally delayed.
- Generally, it does not make sense for a delayed member to become primary. For this reason, `slaveDelay` requires the server to be `priority:0`.
- Typically we would not want a delayed member to be used for `slaveOkay` reads. Thus, setting `slaveDelay` also hides the member from your application as if you also set `hidden:true`.
- Like other members of the set, delayed slaves get to vote in elections.

Configuration Example

```
PRIMARY> use admin
PRIMARY> db.runCommand('ismaster')
{
        "setName" : "repl0",
        "ismaster" : true,
        "secondary" : false,
        "hosts" : [
                "localhost:27017",
                "localhost:27019",
                "localhost:27018"
        ],
        "primary" : "localhost:27017",
        "me" : "localhost:27017",
        "maxBsonObjectSize" : 16777216,
        "ok" : 1
}

PRIMARY> var conf = rs.conf()
PRIMARY> conf
{
        "_id" : "repl0",
        "version" : 3,
        "members" : [
                {
                        "_id" : 0,
                        "host" : "localhost:27017"
                },
                {
                        "_id" : 1,
                        "host" : "localhost:27018"
                },
                {
                        "_id" : 2,
                        "host" : "localhost:27019"
                }
        ]
}
PRIMARY> conf.members[2].priority = 0
0
PRIMARY> conf.members[2].slaveDelay = 300
300
PRIMARY> conf
{
        "_id" : "repl0",
        "version" : 3,
        "members" : [
                {
                        "_id" : 0,
                        "host" : "localhost:27017"
                },
                {
                        "_id" : 1,
```

```
                        "host" : "localhost:27018"
                },
                {
                        "_id" : 2,
                        "host" : "localhost:27019",
                        "priority" : 0,
                        "slaveDelay" : 300
                }
        ]
}
PRIMARY> rs.reconfig(conf)
// rs.reconfig will cause the primary to close all connections, causing the shell to reconnect. This
is expected.
// After reconfig the delayed member is hidden from client drivers.
PRIMARY> db.runCommand('ismaster')
{
        "setName" : "repl0",
        "ismaster" : true,
        "secondary" : false,
        "hosts" : [
                "localhost:27017",
                "localhost:27018"
        ],
        "primary" : "localhost:27017",
        "me" : "localhost:27017",
        "maxBsonObjectSize" : 16777216,
```

```
        "ok" : 1
}
```

Replica Sets - Oplog

Replication of data between nodes is done using a special collection known as the `oplog`.

See also : Replication Oplog Length.

The basic replication process

1. All write operations are sent to the server (Insert, Update, Remove, DB/Collection/Index creation/deletion, etc.)
2. That operation is written to the database.
   - That operation is also written to the oplog.
3. Replicas (slaves) listen to the oplog for changes (known as “tailing the oplog”).
4. Each secondary copies the (idempotent) operation to their own oplog and applies the operations to their data.
5. This read + apply step is repeated

The oplog collection

- The oplog is a special collection type known as a capped collection. The oplog is a collection of fixed size containing information about the operation and a timestamp for that operation. The timestamps are in UTC, regardless of the host's default time zone.
- Because the oplog has a fixed size, it over-writes old data to make room for new data. At any given time, the oplog only contains a finite history of operations.

Falling Behind

- Each secondary keeps track of which oplog items have been copied, and applied locally. This allows the secondary to have a copy of the primary's oplog, which is consistent across the replicaset.
- If a secondary falls behind for a short period of time, it will make a best effort to “catch-up”.

Example:

- A secondary needs 5 minutes of downtime to be rebooted.
- When this computer comes online, the mongod process will compare its oplog to that of the Master.
- `mongod` will identify that it is 5 minutes behind.
- `mongod` will begin processing the primary’s oplog sequentially until it is “caught up”.

This is the ideal situation. The oplog has a finite length, so it can only contain a limited amount of history.

Becoming Stale

- If a secondary falls too far behind the primary’s oplog that node will become **stale**.
- A stale member will stop replication since it can no longer catch up through the oplog.

Example:

- an oplog contains 20 hours of data
- a secondary is offline for 21 hours
- that secondary will become stale, it will stop replicating

If you have a stale replica, see the documents for resyncing.

Preventing a Stale Replica

- The oplog should be large enough to allow for unplanned downtime, replication lag (due to network or machine load issues), and planned maintanance.
- The size of the oplog is configured at startup using the `--oplogSize` command-line parameter. This value is used when you initialize the set, which is the time when the oplog is created (1.7.2+). If you change the `--oplogSize` parameter later, it has no effect on your existing oplog.
- There is no easy formula for deciding on an oplog size. The size of each write operation in the oplog is not fixed.

    - Running your system for a while is the best way to estimate the space required. The size and amount of time in your oplog is related to the types and frequency of your writes/updates.
- Recovering a stale replica is similar to adding a new replica.

> A completely re-sync can often take a long time, especially with large datasets. To be able to bring up a new replica “from scratch” ensure that you have a large enough oplog to cover the time to re-sync.

See Also

- About the local database

Replica Sets - Priority

By default, all full nodes in a replica set have equal priority. If a primary steps down, all other members are electable as the new primary. That is, each node by default has `priority:1`.

- Arbiters have no data, and are never a primary (or even a read-only secondary). Thus priority has no meaning on arbiters.

- Members with the `hidden:true` property must have `priority:0`

MongoDB v2.0+ provides more fine-grained control over priorities.

Priority 0 nodes

A node with `priority:0` can never become primary. Typical uses:

- a less powerful server might be suitable to be a secondary but not primary
- a server in a secondary data center where one wants only manual fail-over (common for disaster recovery)
- a server used for taking backups
- a time-delayed secondary (slaveDelay property)

v1.6-1.8

In v1.6-1.8, the priority of a node can be zero or one only.

v2.0+ : specific priorities

As of v2.0, members can have more granular priorities (values between zero and one thousand, e.g., 673.2).

When an election is triggered, the highest priority amongst the most up-to-date nodes will be elected.

- "up-to-date" implies within 10 seconds of the primary. This is not configurable
- **Note** - if a node with a higher priority than the primary "catches up", this will trigger a new election (*which the higher priority node will win*)

Example: if B and C are candidates in an election, B having a higher priority but C being the most up to date:

1. C will be elected primary
2. Once B catches up a re-election should be triggered and B (the higher priority node) should win the election between B and C
3. Alternatively, suppose that once B is within 12 seconds of sync with C, C goes down.
    - B will be elected primary.
    - When C comes back up, those 12 seconds of unsynced writes will be written to a file in the *rollback* directory of your data directory (*rollback* is created when needed).
    - You can manually apply the rolled-back data; see Replica Sets - Rollbacks.

See Also

- Blog post on replica set priorities

Replica Sets - Rollbacks

Overview

The classic example of a rollback occurs when you have two replicas (a primary-A and secondary-B) and the secondary (B) is not up to date (replication is behind). If the primary (A) fails (or is shutdown) before B is up-to-date and B becomes primary, then there is data which B does not have but A does.

When this happens, MongoDB cannot automatically merge the old data to the new/current primary, but don't worry, *the data is not lost*. The rollback operation will ensure that all data is consistent based on the current primary's data.

In order to roll back the data on the original primary (A), the oplog is traversed from the point in time that the server B took over as primary. For each delete or update in the oplog on server A the modified documents are re-fetched from B any new documents after that point are deleted from server A.

This rolled back data, which only A had) is stored in the `rollback` directory. The data is stored in a BSON file, the name is date-encoded like this `foo.bar.2011-05-09T18-10-04.0.bson`. In newer versions of mongodb (>2.1) that data may be stored in the local database as well so that restoring it will be easier.

TODO: Add diagram.

Rolled back data

- To view the contents of this file in human-readable format, use the bsondump utility.
- To restore this information to the DB, use the mongorestore utility.

One common strategy for reconciling these conflicts is restore the rollback data to a new collection. Then reconcile between that rollback collection and the master with a custom program/script.

Rollback Limitations

MongoDB will not rollback more than 300MB of data. In this situation, the rollback will halt with the log message: `[replica set sync] replSet syncThread: 13410 replSet too much data to roll back`

To recover data from the member that has failed to rollback, you will need to manually intervene. If you do not care about this data, remove the data directory and fully resync (or restore from a backup) to resume normal operation.

Replica Sets - Voting

Each replica sets contains only one primary node. This is the only node in the set that can accept write commands (**insert/update/delete**).

The primary node is elected by a consensus vote of all reachable nodes.

Consensus Vote

For a node to be elected primary, it must receive a **majority** of votes. This is a majority of all votes in the set: if you have a 5-member set and 4 members are down, a majority of the set is still 3 members (*floor(5/2)+1*). Each member of the set receives a single vote and knows the total number of available votes.

If no node can reach a majority, then no primary can be elected and no data can be written to that replica set (although reads to secondaries are still possible).

Arbiters

An arbiter is a member which votes but has no data. An arbiter cannot be a primary or a secondary, as it has no data.

It is solely used for breaking ties in elections, so at most one arbiter is ever needed.

Reachable Node

Replicas in a set are in regular communication with each other. They do this via a "heartbeat" that is communicated to all nodes in the set.

If node A fails to receive a heartbeat from node B, A will assume that B is unreachable (it will continue to try to re-establish contact, but it will take that into consideration when determining whether a majority is reachable).

Triggering an Election

An election is triggered when the following is true:

- a node sees that the primary is not reachable
- that node is not an arbiter
- that node has priority greater than or equal to other eligible nodes in the set

This means that an election is triggered if the primary node is turned off (*`mongod` stopped, computer shutdown, port blocked,...*). An election can also be triggered if the primary stops responding due to a network issue (*DNS, internet connectivity,...*)

Changing votes

Do not change the number of votes.

- Do not change vote weights in an attempt to create a "preferred master" – this will not work. Instead use priorities to achieve this.
- In a two node replica set, it is far better to have an arbiter than to give one of the two members an extra vote.

By default each machine in a replica set receives one vote. The vote field can be set to any non-negative integer, however it is highly suggested that this number be either 0 or 1.

The primary purpose for changing the voting weight is to allow for larger replica sets. Each replica set is limited to 12 total nodes and 7 voting nodes.

The number of votes can be modified in the replica set configuration. You should never change the number of votes per member unless your set has more than seven members.

Why Replica Sets

Replica sets provide five distinct benefits over the use of a single node.

A system requiring one or more of the following features should consider using replica sets.

- Data Redundancy
- Automated Failover
- Read Scaling
- Maintenance
- Disaster Recovery

Data Redundancy

- Replica sets provide an automated method for storing multiple copies of your data.
- Supported drivers allow for the control of "write concerns". This allows for writes to be confirmed by multiple nodes before returning a success message to the client.

Automated Failover

- Replica sets will coordinate to have a single primary in a given set.
- Supported drivers will recognize the change of a primary within a replica set.
    - In most cases, this means that the failure of a primary can be handled by the client without any configuration changes.
- A correctly configured replica set basically provides a “hot backup”. Recovering from backups is typically very time consuming and can result in data loss. Having an active replica set is generally much faster than working with backups.

Read Scaling

- The primary means to scale in MongoDB is sharding. Replicas are foremost for data safety, high availability, and redundancy. That said, it is possible to perform queries on secondaries.
- By default, the primary node of a replica set is accessed for all reads and writes.
- Most drivers provide a `slaveOkay` method for identifying that a specific operation can be run on a secondary node. When using `slaveOkay`, a system can share the read load amongst several nodes.
- Unless there is an explicit need, we recommend simply querying the primaries. This keeps things simples and assures immediate consistency for the data read.

Maintenance

- When performing tasks such as upgrades, backups and compaction, it is typically required to remove a node from service.
- Replica sets allow for these maintenance tasks to be performed while operating a production system. As long as the production system can withstand the removal of a single node, then it’s possible to perform a “rolling” upgrade on such things.

Disaster Recovery

- Replica sets allows for a “delayed secondary” node.
- This node can provide a window for recovering from disastrous events such as:
    - bad deployments
    - dropped tables and collections

Next : Setup Basics

Moving or Replacing a Member

In the shell you can use the rs.remove() helper to remove a node from a replica set.

Then use rs.add() to add the new member to the set. See rs.help() in the shell for more information.

See the Adding a New Set Member for details on how to pre-copy all the datafiles from the old member to the new member if that is desired. (The new member will automatically sync all data on its own; copying the files would only be done if the administrator wants to get the new member up and online more quickly.)

Renaming or renumbering a member's host

If you need to change a hostname or IP address that a replica set uses to refer to one of the member processes, the easiest way to do so is to run rs.reconfig() with a configuration document that preserves the member's _id number, but changes the `host` field. For example, if the hostname changes for the first member of the set, you might do this in the shell:

```
> cfg = rs.conf()
> cfg.members[0].host = newname
> rs.reconfig(cfg)
```

Replica Set Versions and Compatibility

Features

Feature	Version
Slave delay	v1.6.3+
Hidden	v1.7+
replSetFreeze and replSetStepDown	v1.7.3+
Replicated ops in `mongostat`	v1.7.3+
Syncing from secondaries	v1.8.0
Authentication	v1.8.0
Replication from nearest server (by ping time)	v2.0.0

Syncing

1.8.x slaves **can** replicate from 1.6.x masters.

1.6.x slaves **cannot** replicate from 1.8.x masters.

See the upgrade documentation for advice on upgrading.

Replica Set Design Concepts

A replica set has at most one primary at a given time. If a majority of the set is up, the most up-to-date secondary will be elected primary. If a majority of the set is not up or reachable, no member will be elected primary.

There is no way to tell (from the set's point of view) the difference between a network partition and nodes going down, so members left in a minority will not attempt to become master (to prevent a set from ending up with masters on either side of a partition).

This means that, if there is no majority on either side of a network partition, the set will be read only (thus, we suggest an odd number of servers: e.g., two servers in one data center and one in another). The upshot of this strategy is that data is consistent: there are no multi-master conflicts to resolve.

There are several important concepts concerning data integrity with replica sets that you should be aware of:

1. A write is (cluster-wide) committed once it has replicated to a majority of members of the set.

For important writes, the client should request acknowledgement of this with a `getLastError({w:...})` call. (If you do not call getLastError, the servers do exactly the same thing; the getlasterror call is simply to get confirmation that committing is finished.)

2. Queries in MongoDB and replica sets have "READ UNCOMMITTED" semantics.

Writes which are committed at the primary of the set may be visible before the cluster-wide commit completes.

The read uncommitted semantics (an option on many databases) are more relaxed and make theoretically achievable performance and availability higher (for example we never have an object locked in the server where the locking is dependent on network performance).

3. On a failover, if there are writes which have not replicated from the primary, the writes are rolled back. Thus we use `getlasterror` as in #1 above when we need to confirm a cluster-wide commit.

The data is backed up to files in the *rollback* directory, although the assumption is that in most cases this data is never recovered as that would require operator intervention. However, it is not "lost," it can be manually applied at any time with mongorestore.

Rationale

Merging back old operations later, after another node has accepted writes, is a hard problem. One then has multi-master replication, with potential for conflicting writes. Typically that is handled in other products by manual version reconciliation code by developers. We think that is too much work : we want MongoDB usage to be less developer work, not more. Multi-master also can make atomic operation semantics problematic.

It is possible (as mentioned above) to manually recover these events, via manual DBA effort, but we believe in large system with many, many nodes that such efforts become impractical.

Comments

Some drivers support 'safe' write modes for critical writes. For example via setWriteConcern in the Java driver.

Additionally, defaults for { w : ... } parameter to getLastError can be set in the replica set's configuration.

Note a call to getLastError will cause the client to have to wait for a response from the server. This can slow the client's throughput on writes if large numbers are made because of the client/server network turnaround times. Thus for "non-critical" writes it often makes sense to make no getLastError check at all, or only a single check after many writes.

See Also

- Replica Set FAQ

Replica Set Tutorial

v1.6+

This tutorial will guide you through a basic replica set initial setup. Given the tutorial is an example and should be easy to try, it runs several mongod processes on a single machine (in the real world one would use several machines). In the real world each member is on a separate server (or VM). When ready for production be sure to read the rest of the replica set documentation.

Introduction

A replica set is group of *n* `mongod` nodes (members) that work together. The goal is that each member of the set has a complete copy (replica) of the data form the other nodes.

Setting up a replica set is a two-step process that requires starting each mongod process and then formally initiating the set. Here, we'll be configuring a set of three nodes, which is standard.

Once the `mongod` processes are started, we will issue a command to initialize the set. After a few seconds, one node will be elected master, and you can begin writing to and querying the set.

Starting the nodes

First, create a separate data directory for each of the nodes in the set. In a real environment with multiple servers we could use the default `/data/db` directory (if we wanted to), but on a single machine we have to specify non-defaults:

```
$ mkdir -p /data/r0
$ mkdir -p /data/r1
$ mkdir -p /data/r2
```

Next, start each `mongod` process with the `--replSet` parameter. The parameter requires that you specify a logical name for our replica set. Let's call our replica set "foo". We'll launch our first node like so:

```
$ mongod --replSet foo --port 27017 --dbpath /data/r0
```

Let's now start the second and third nodes:

```
$ # no need for special port numbers if on separate machines, but we need them to
$ # demo on a single server
$ mongod --replSet foo --port 27018 --dbpath /data/r1
$ mongod --replSet foo --port 27019 --dbpath /data/r2
```

You should now have three nodes running. At this point, each node should be printing the following warning:

```
Mon Aug  2 11:30:19 [startReplSets] replSet can't get local.system.replset config from self or any
seed (EMPTYCONFIG)
```

We can't use the replica set until we've *initiated* it, which we'll do next.

Initiating the set

We *initiate* the replica set by connecting to one of the members and running the `replSetInitiate` command (that is, `rs.initiate()` in the mongo shell). This command expects a json object which contains the set's configuration details.

The replSetInitiate command may be sent to any member of an yet-to-be initiated set. However, only the member performing the initiation may have any existing data. This data becomes the initial data for the set. The other members will begin synchronizing and receiving that data (if present; starting empty is fine too). This is called the "initial sync". Secondaries will not be online for reads (in state 2, "SECONDARY") until their initial sync completes.

Note: the replication oplog (in the local database) is allocated at initiation time. The oplog can be quite large, thus initiation may take some time.

```
$ mongo localhost:27017
MongoDB shell version: 1.5.7
connecting to: localhost:27017/test
> rs.help(); // if you are curious run this (optional)
>
> config = {_id: 'foo', members: [
                          {_id: 0, host: 'localhost:27017'},
                          {_id: 1, host: 'localhost:27018'},
                          {_id: 2, host: 'localhost:27019'}]
           }
> rs.initiate(config);
{
   "info" : "Config now saved locally.  Should come online in about a minute.",
   "ok" : 1
}
```

We specify the config object and then pass it to `rs.initiate()`. Then, if everything is in order, we get a response saying that the replica set will be online in a minute. During this time, one of the nodes will be elected primary (master).

To check the status of the set, run `rs.status()`:

```
> rs.status()
{
 "set" : "foo",
 "date" : "Mon Aug 02 2010 11:39:08 GMT-0400 (EDT)",
 "myState" : 1,
 "members" : [
  {
   "name" : "localhost:27017",
   "self" : true,
  },
  {
   "name" : "localhost:27018",
   "health" : 1,
   "uptime" : 101,
   "lastHeartbeat" : "Mon Aug 02 2010 11:39:07 GMT-0400",
  },
  {
   "name" : "localhost:27019",
   "health" : 1,
   "uptime" : 107,
   "lastHeartbeat" : "Mon Aug 02 2010 11:39:07 GMT-0400",
  }
 ],
 "ok" : 1
}
```

You'll see that the other members of the set are up. You may also notice that the `myState` value is 1, indicating that we're connected to the member which is currently primary; a value of 2 indicates a secondary.

You can also check the set's status in the HTTP Admin UI.

Replication

Go ahead and write something to the primary:

```
> db.messages.insert({name: "ReplSet Tutorial"});
```

If you look at the logs on the secondary nodes, you'll see the write replicates.

Failover

The purpose of a replica set is to provide automated failover. This means that, if the primary node goes down, a secondary node can take over. When this occurs the set members which are up perform an election to select a new primary. To see how this works in practice, go ahead and kill the master node with Control-C (^C) (or if running with --journal, kill -9 would be ok too):

```
^CMon Aug  2 11:50:16 got kill or ctrl c or hup signal 2 (Interrupt), will terminate after current cmd
ends
Mon Aug  2 11:50:16 [interruptThread] now exiting
Mon Aug  2 11:50:16  dbexit:
```

If you look at the logs on the secondaries, you'll see a series of messages indicating fail-over, for example:

```
Mon Aug  2 11:50:16 [ReplSetHealthPollTask] replSet info localhost:27017 is now down (or slow to
respond)
Mon Aug  2 11:50:17 [conn1] replSet info voting yea for 2
Mon Aug  2 11:50:17 [rs Manager] replSet not trying to elect self as responded yea to someone else
recently
Mon Aug  2 11:50:27 [rs_sync] replSet SECONDARY
```

And on the next member, this:

```
Mon Aug  2 11:50:17 [ReplSetHealthPollTask] replSet info localhost:27017 is now down (or slow to
respond)
Mon Aug  2 11:50:17 [rs Manager] replSet info electSelf 2
Mon Aug  2 11:50:17 [rs Manager] replSet PRIMARY
Mon Aug  2 11:50:27 [initandlisten] connection accepted from 127.0.0.1:61263 #5
```

Both nodes notice that the old primary is done, as a result, a new primary node is elected. In this case, the node at port 27019 is promoted. If we bring the failed node on 27017 back online, it will come back up as a secondary.

Changing the replica set configuration

There are times when you'll want to change the replica set configuration. Suppose, for instance, that you want to make a member have attribute `priority:0`, indicating the member should never be primary. To do this, pass a new configuration object to the database's `replSetReconfig` command. The shell `rs.reconfig()` helper makes this easier.

One note: the reconfig command must be sent to the current primary of the set. This implies that you need a majority of the set up to perform a reconfiguration. (There are ways to reconfigure without a majority for emergencies, see the rest of the replica set documentation.)

```
> // we should be primary here.  can be checked with rs.status() or with:
> rs.isMaster();
>
> // view existing config
> var c = rs.conf();
{_id: 'foo', members: [
                       {_id: 0, host: 'localhost:27017'},
                       {_id: 1, host: 'localhost:27018'},
                       {_id: 2, host: 'localhost:27019'}]
}
>
> // reconfig
> c.members[2].priority = 0;
> c
{_id: 'foo', members: [
                       {_id: 0, host: 'localhost:27017'},
                       {_id: 1, host: 'localhost:27018'},
                       {_id: 2, host: 'localhost:27019', priority: 0}]
}
> rs.reconfig(c);
> // done. to see new config,and new status:
> rs.conf()
> rs.status()
```

Running with two nodes

Suppose we want to run replica sets with just two database servers (that is, have a replication factor of two). This is possible, but as replica sets perform elections, here a majority would be 2 out of 2 which is not helpful. Thus in this situation one normally also runs an *arbiter* on a separate server. An arbiter is a set member which has no data but gets to vote in elections. In the case here, the arbiter is the tie breaker in elections. Arbiters are very lightweight and can be ran anywhere – say, on an app server or a micro vm. With an arbiter in place, the replica set will behave appropriately, recovering automatically during both network partitions and node failures.

You start up an arbiter just as you would a standard replica set node, as a mongod process with the `--replSet` option. However, when initiating, you need to include the `arbiterOnly` option in the config document.

With an arbiter, the configuration presented above would look like this:

```
config = {_id: 'foo', members: [
                          {_id: 0, host: 'localhost:27017'},
                          {_id: 1, host: 'localhost:27018'},
                          {_id: 2, host: 'localhost:27019', arbiterOnly: true}]
           }
```

Drivers

Most of the MongoDB drivers are replica set aware. The driver when connecting takes a list of seed hosts from the replica set and can then discover which host is primary and which are secondary (the isMaster command is used internally by the driver for this). The driver also attempts to discover new members of the set which were not specified on the command line.

The driver automatically finds the new primary if the current one fails. See your driver's documentation for specific details.

If you happen to be using the Ruby driver, you may want to check out Replica Sets in Ruby.

getLastError

For important writes, use the getLastError to ensure cluster-wide commits of critical writes.

See Also

- Videos available here

Replica Set Configuration

- Command Line
- Initial Setup
- The Replica Set Config Object
  - Minimum Config - Required Arguments
  - Advanced Config - Optional Arguments
    - Member options
    - Set options
- Shell Example 1
- Shell Example 2
- See Also

Command Line

Each `mongod` participating in the set should have a `--replSet` parameter on its command line. The syntax is

```
mongod --replSet setname --rest
```

setname is the logical name of the set.

> Use the `--rest` command line parameter when using replica sets, as the web admin interface of *mongod* (normally at port 28017) shows status information on the set. See Replica Set Admin UI for more information.

Initial Setup

We use the `replSetInitiate` command for initial configuration of a replica set. Send the initiate command to a single server to christen the set. The member being initiated may have initial data; the other servers in the set should be empty.

```
> db.runCommand( { replSetInitiate : <config_object> } )
```

A shorthand way to type the above is via a helper method in the shell:

```
> rs.initiate(<config_object>)
```

A quick way to initiate a set is to leave out the config object parameter. The initial set will then consist of the member to which the shell is communicating, along with all the seeds that member knows of. However, see the configuration object details below for more options.

```
> rs.initiate()
```

The Replica Set Config Object

The `local.system.replset` collection holds a singleton object which contains the replica set configuration. The config object automatically propagates among members of the set. The object is not directly manipulated, but rather changed via commands (such as `replSetInitiate`).

Minimum Config - Required Arguments

At its simplest, the config object contains the name of the replica set and a list of its members:

```
{
    _id : <setname>,
    members : [
        {_id : 0, host : <host0>},
        {_id : 1, host : <host1>},
        ...
    ]
}
```

Every replica set configuration must contain an `_id` field and a `members` field with one or more hosts listed.

Setting	Description
`_id`	The set name. This must match command line setting. Set names are usually alphanumeric and, in particular, cannot contain the '/' character.
`members`	An array of servers in the set. For simpler configs, one can often simply set `_id` and `host` fields only – all the rest are optional. • `_id` - Each member has an _id ordinal, typically beginning with zero and numbered in increasing order. When a node is retired (removed from the config), its `_id` should not be reused. • `host` - Host name and optionally the port for the member

Advanced Config - Optional Arguments

There are many optional settings that can also be configured using the config object. The full set is:

```
{
  _id : <setname>,
  members: [
    {
      _id : <ordinal>,
      host : <hostname[:port]>
      [, arbiterOnly : true]
      [, buildIndexes : <bool>]
      [, hidden : true]
      [, priority: <priority>]
      [, tags: {loc1 : desc1, loc2 : desc2, ..., locN : descN}]
      [, slaveDelay : <n>]
      [, votes : <n>]
    }
    , ...
  ],
  [settings: {
    [getLastErrorDefaults: <lasterrdefaults>]
    [, getLastErrorModes : <modes>]
  }]
}
```

Member options

Each member can be configured to have any of the following options.

Command	Default	Description	Min Version
`arbiterOnly`	`false`	If `true`, this member will participate in vote but receive no data.	1.6

`buildIndexes`	`true`	When `false`, prevent secondary indexes from being created on this member. This is typically used on machines that are pure "backup" machines that are never queried. By not having the secondary indexes, the member performs less works on writes and requires less ram. Note the _id index is still created. Can only be set to `false` if `priority:0`. It is rare to use this option.	1.6
`hidden`	`false`	If `true`, do not advertise the member's existence to clients in `isMaster` command responses. Hidden replicas makes sense for replicas of data which have very different use patterns (reporting, integration, backup, etc.) than the main set of replicas; this option allows you to keep from sending normal non-primary queries to the node.	1.7
`priority`	1.0	Priority of the server for elections. Higher priority servers will be preferred as primary. (*more information*)	1.6, 1.9
`tags`	{}	An document representing the location of this server. Tags can be used for location-aware write guarantees and read locality, see Data Center Awareness	1.9.1
`slaveDelay`	0	Number of seconds to remain behind the primary. A value of 0 implies "as up-to-date as possible". Used to recover from human errors (*e.g.: accidentally dropping a database*). Can only be set on members with priority 0. Slave delay members are a great way to keep a rolling backup from a certain amount of time in the past.	1.6.3
`votes`	1	Number of votes this member has in an election. Generally you should not change this. (*more information*)	1.6

Set options

The final optional argument, `settings`, can be used to set options on the set as a whole. Often one can leave out `settings` completely from the config as the defaults are reasonable.

Setting	Description	Min Version
`getLastErrorDefaults`	Specifies defaults for the `getlasterror` command. If the client calls `getLastError` with **no parameters** then the defaults specified here are used.	1.6.2
`getLastErrorModes`	Define and name combinations of tags that can be used by the application to guarantee writes to certain servers, racks, data centers, etc. See Data Center Awareness.	1.9.1

Shell Example 1

```
> // all at once method
> cfg = {
... _id : "acme_a",
... members : [
... { _id : 0, host : "sf1.acme.com" },
... { _id : 1, host : "sf2.acme.com" },
... { _id : 2, host : "sf3.acme.com" } ] }
> rs.initiate(cfg)
> rs.status()
```

Shell Example 2

```
$ # incremental configuration method
$ mongo sf1.acme.com/admin
> rs.initiate();
> rs.add("sf2.acme.com");
> rs.add("sf3.acme.com");
> rs.status();
```

See Also

- Adding a New Set Member
- Reconfiguring when Members are Up
- Reconfiguring a replica set when members are down

Adding an Arbiter

Arbiters are nodes in a replica set that only participate in elections: they don't have a copy of the data and will never become the primary node (or even a readable secondary). They are mainly useful for breaking ties during elections (e.g. if a set only has two members).

You should only add an arbiter if you have an even number of full members to bring the set to an odd number of voters. As a corollary to this, having two (or more) arbiters is **not** (usually) safer than just one.

To add an arbiter, bring up a new node as a replica set member (`--replSet` on the command line) - just like when Adding a New Set Member.

Pre-v1.8: it is best to specify `--oplogSize 1` on the arbiter's (mongod) command line so that 5% of available disk space isn't allocated to the oplog (arbiters do not need an oplog). This is handled automatically in v1.8 and beyond.

To start as an arbiter, we'll use `rs.addArb()` instead of `rs.add()`. While connected to the current primary:

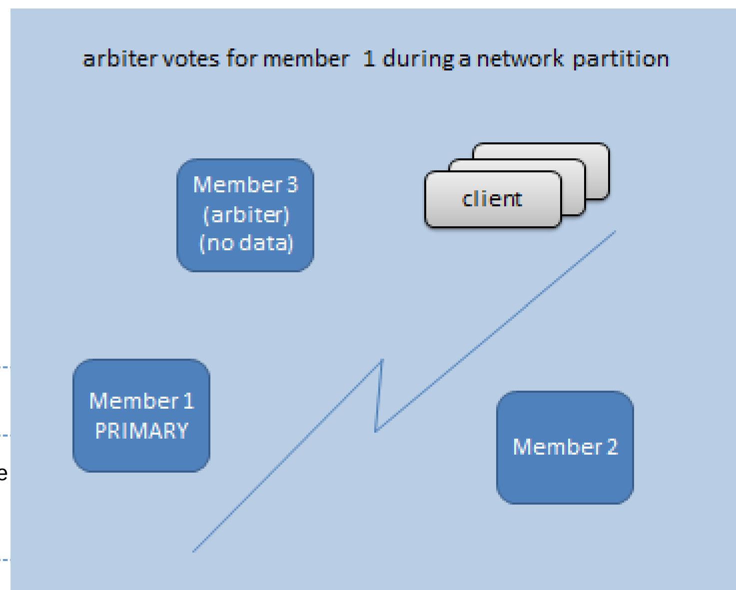
```
> rs.addArb("broadway:27017");
{ "ok" : 1 }
```

When to add an arbiter

- Two members with data : add an arbiter to have three voters. 2 out of 3 votes for a member establishes it as primary.
- Three members with data : no need to add an arbiter. In fact having 4 voters is worse as 3 of 4 needed to elect a primary instead of 2 of 3. In theory one might add two arbiters thus making number of votes five, and 3 of 5 would be ok; however this is uncommon and generally not recommended.
- Four members with data : add one arbiter.

See Also

- Adding a New Set Member



Adding a New Set Member

- Adding a former member
- Starting with an existing copy of the dataset
- See Also

Adding a new member to an existing replica set is easy. The new member should either have an empty data directory or a recent copy of the data from another set member. When starting `mongod` on the new server, provide the replica set name:

```
$ mongod --replSet foo
```

After bringing up the new server (we'll call it `broadway:27017`) we need to add it to the set – we connect to our primary server using the shell:

```
$ mongo --host our_primary_host
MongoDB shell version: ...
connecting to: test
PRIMARY> rs.add("broadway:27017");
{ "ok" : 1 }
```

After adding the node it will synchronize (doing a full initial sync if starting empty, see below) and then come online as a secondary.

You can also specify any member configuration options using `rs.add()`. Some examples:

```
> // add an arbiter
> rs.add({_id: 3, host: "broadway:27017", arbiterOnly: true})
>
> // add a hidden member
> rs.add({_id: 3, host: "broadway:27017", priority: 0, hidden: true})
>
> // add a member with tags
> rs.add({_id: 3, host: "broadway:27017", tags: {dc : "nyc", rack: "rack1"}})
```

Adding a former member

A member can be removed from a set and re-added later. If the removed member's data is still relatively fresh, it can recover and catch up from its old data set. See the rs.add() and rs.remove() helpers.

If there is any trouble re-adding the member, restart its mongod process. You may also need to add it back with its former _id (although probably not if mongod were restarted).

Starting with an existing copy of the dataset

If you have a backup or snapshot of an existing member, you can move the data files to a new machine and use them to quickly add a new member. These files must be:

- **clean:** the existing dataset must be from a consistent snapshot / backup of the database from a member of the same replica set. See the backups page for more information on copying and snapshotting databases.
- **recent:** the snapshot/backup must have been taken more recently than the oldest operation in the primary's oplog (so that it can catch up by just copying from the oplog)

You do not have to make any modifications to the files before starting up a member.

In versions before v2.0, you must start the new member before running `rs.add()`. In v2.0+ you can do it in either order, although a majority of the new set must be up for the reconfig to work. Two examples:

- If you have two members and you're adding a third, you can add it to the set before it's live and then bring up the third member. Or you can bring up the third member and then add it to the set. Either way will work.
- If you have a one-member set and you're adding a second member, you must have the second member up before you add it to the set. Otherwise, after the reconfig you'll have one out of two members up (**not** a majority) and the primary will have to step down.

Additionally with versions before v2.0 you need to restart mongod on the newly added node after the rs.add().

See Also

- Adding an Arbiter

Reconfiguring when Members are Up

v1.8+

Use the rs.reconfig() helper in the shell. (Run "rs.reconfig" in the shell with no parenthesis to see what it does.)

```
$ mongo
> // example : give 1st member priority 2
> cfg = rs.conf()
> cfg.members[0].priority = 2
> rs.reconfig(cfg)
```

Earlier versions (1.6)

You can reconfigure a set from any other language/driver/version using the `replSetReconfig` command directly.

```
$ mongo
> // shell v1.6:
> // example : give 1st set member 2 votes
> cfg = rs.conf()
> cfg.members[0].votes = 2
> cfg.version++
> use admin
> db.runCommand( { replSetReconfig : cfg } )
```

Requirements

- You must connect to the admin db of the current primary.
- A majority of members of the set must be up.

Notes

- You may experience a short downtime period while the set renegotiates master after a reconfiguration. This typically is 10-20 seconds. As always, it is best to do admin work during planned maintenance windows regardless just to be safe.
- In certain circumstances, the primary steps down (perhaps transiently) on a reconfiguration. On a step-down, the primary closes sockets from clients to assure the clients know quickly that the server is no longer primary. Thus, your shell session may experience a disconnect on a reconfig command.

See Also

- Reconfiguring when members are down

Recommended Configurations

Replica sets support quite a few options, and it can be confusing determining the best configuration. Here following a few suggestions.

One data center

If you have just one data center, then the most economical setup is a three-node replica set, one of which is designated as an arbiter. The standard nodes each get their own box, and the arbiter lives on an application server.

Two data centers

With two data centers, you'll want to designate one data center as primary and the other as a backup for the very rare case where the first data center fails entirely.

From here, a three-node replica set is sufficient. You'll keep two replica set nodes in the primary data center and one node in the backup data center. You'll also want to give the node in the backup DC a priority of 0. As long as the first data center continues to function, your primary node will always live there. If you lose any one node, the replica set will still be able to elect a primary in the main data center.

With this relatively economical setup, you're protected against the failure of any one node and against the failure of any one data center. Of course, if it's the primary data center that fails, then you'll need to manually promote the node in the secondary data center to primary. But if you use write concern carefully, then you won't lose any data, the manual failover won't lead to much downtime.

See Also

- Data Center Awareness

Data Center Awareness

Examples

One primary data center, one disaster recovery site

Multiple set members can be primary at the main data center. Have a member at a remote site that is never primary (at least, not without human intervention).

```
{ _id: 'myset',
  members: [
   { _id:0, host:'sf1', priority:1 },
   { _id:1, host:'sf2', priority:1 },
   { _id:2, host:'ny1', priority:0 }
  ]
}
```

Multi-site with local reads

The following example shows one set member in each of three data centers. At election time, any healthy update to date node, arbitrarily, can become primary. The others are then secondaries and can service queries locally if the client uses slaveOk mode.

```
{ _id: 'myset',
  members: [
   { _id:0, host:'sf1', priority:1 },
   { _id:1, host:'ny1', priority:1 },
   { _id:2, host:'uk1', priority:1 }
  ]
}
```

Refer to your driver's documentation for more information about read routing.

Confirming propagation of writes with getLastError

Calling [getLastError] (called "write concern" in some drivers) with `w:"majority"` (v2.0+) assures the write reaches a majority of the set before acknowledgement. For example, if you had a three-member replica set, calling `db.runCommand({getLastError : 1, w : "majority"})` would make sure the last write was propagated to at least 2 servers.

Once a write reaches a majority of the set members, the cluster wide commit has occurred (see Replica Set Design Concepts).

Replicating from nearby members

In v2.0+, secondaries automatically sync data from members which are nearby. You can see the latencies that the `mongod` process is observing to its peers in the replSetGetStatus command's output. If nearby members are not healthy, more distant members will be used for syncing.

Example output, highlighting new ping time and sync target fields:

```
> rs.status()
{
        ...
        "syncingTo" : "B:27017",
        "members" : [
                {
                        "_id" : 0,
                        "name" : "A:27017",
                        ...
                        "self" : true
                },
                {
                        "_id" : 1,
                        "name" : "B:27017",
                        ...
                        "pingMs" : 14
                },
                {
                        "_id" : 2,
                        "name" : "C:27017",
                        ...
                        "pingMs" : 271
                }
        ],
        "ok" : 1
}
```

Tagging (version 2.0+)

Tagging gives you fine-grained control over where data is written. It is:

- Customizable: you can express your architecture in terms of machines, racks, data centers, PDUs, continents, etc. (in any combination or level that is important to your application).
- Developer/DBA-friendly: developers do not need to know about where servers are or changes in architecture.

Each member of a replica set can be tagged with one or more physical or logical locations, e.g., `{"dc" : "ny", "rack" : "rk1", "ip" : "192.168", "server" : "192.168.4.11"}`. *Modes* can be defined that combine these tags into targets for `getLastError`'s *w* option.

For example, suppose we have 5 servers, *A*, *B*, *C*, *D*, and *E*. *A* and *B* are in New York, *C* and *D* are in San Francisco, and *E* is in the cloud somewhere.

Our replica set configuration might look like:

```
{
    _id : "someSet",
    members : [
        {_id : 0, host : "A", tags : {"dc": "ny"}},
        {_id : 1, host : "B", tags : {"dc": "ny"}},
        {_id : 2, host : "C", tags : {"dc": "sf"}},
        {_id : 3, host : "D", tags : {"dc": "sf"}},
        {_id : 4, host : "E", tags : {"dc": "cloud"}}
    ]
    settings : {
        getLastErrorModes : {
            veryImportant : {"dc" : 3},
            sortOfImportant : {"dc" : 2}
        }
    }
}
```

Now, when a developer calls `getLastError`, they can use any of the modes declared to ensure writes are propagated to the desired locations, e.g.:

```
> db.foo.insert({x:1})
> db.runCommand({getLastError : 1, w : "veryImportant"})
```

`"veryImportant"` makes sure that the write has made it to at least 3 tagged "regions", in this case, "ny", "sf", and "cloud". Once the write has been replicated to these regions, getLastError will return success. (For example, if the write was present on *A*, *D*, and *E*, that would be a success condition).

If we used `"sortOfImportant"` instead, `getLastError` would return success once the write had made it to two out of the three possible regions. Thus, *A* and *C* having the write or *D* and *E* having the write would both be "success." If *C* and *D* had the write, `getLastError` would continue waiting until a server in another region also had the write.

Below are some common examples and how you'd specify tags and `w` modes for them.

Server *X* should have a copy.

Suppose you want to be able to specify that your backup server (B) should have a copy of a write. Then you'd use the following tags:

Server	Tags
B	`{"backup" : “B”}`

To define a mode for "server *B* should have a copy," create the mode:

```
backedUp : {“backup” : 1}
```

You want one server with a "backup" tag to have the write.

So, your config would look like:

```
{
   _id : replSetName,
   members : [
       {
           “_id” : 0,
           “host” : B,
           “tags” : {"backup" : "B"}
       },
       ...
   ],
   settings : {
       getLastErrorModes : {
           backedUp : {backup : 1}
       }
   }
}
```

To use this mode in your application, you'd call `getLastError` with `w` set to `backedUp`:

```
> db.runCommand({getLastError : 1, w : "backedUp"})
```

In the following examples, we will skip the configuration and the usage for brevity. Tags are always added to a member's configuration, modes are always added to `getLastErrorModes`.

Make *n* backups

Suppose you have three backup servers (*B1*, *B2*, *B3*) and you want at least two of them to have a copy. Then you'd give each of them a unique "backup" tag:

Server	Tags
B1	`{"backup" : "B1"}`
B2	`{"backup" : "B2"}`

B3	`{"backup" : "B3"}`

Then you would create the mode:

```
backedUp : {"backup" : 2}
```

Make sure there are at least three copies of the data and it is present on at least two continents.

All of the rules up until now have only had one condition, but you can include as many and-conditions as you want. Suppose we have the following:

Server	Tags
S1	`{"continent" : "nAmerica", "copies" : "S1"}`
S2	`{"continent" : "nAmerica", "copies" : "S2"}`
S3	`{"continent" : "nAmerica", "copies" : "S3"}`
S4	`{"continent" : "Africa", "copies" : "S4"}`
S5	`{"continent" : "Asia", "copies" : "S5"}`

Then create a mode like:

```
level : {copies : 3, continent : 2}
```

Note that modes can contain as many clauses as you need.

Make sure at least two servers across at least two racks in `nyc` have it.

This is a complication of our original example. The key concept here is that not all tags need to be present on all servers. For example, some servers below are tagged with "nyc", others are not.

Server	Tags
S1	`{"nycRack" : "rk1", "nyc" : "S1"}`
S2	`{"nycRack" : "rk2", "nyc" : "S2"}`
S3	`{"nycRack" : "rk2", "nyc" : "S3"}`
S4	`{"sfRack" : "rk1", "sf" : "S4”}`
S5	`{"sfRack" : "rk2", "sf" : S5"}`

Now our rule would look like:

```
customerData : {“nycRack” : 2}
```

Notes

The examples above generally use hostnames (e.g., `"nyc" : "S1"`). This isn't required, it's just and convenient way to specify a server-unique tag. You could just as well use "foo", "bar", "baz" or "1", "2", "3", or any other identifiers.

Do not use "*" or "$" in tags, these characters are reserved for future use.

Replica Set Authentication

Authentication was added in 1.7.5

Replica set authentication works a little differently from single-server authentication, so that each member can automatically authenticate itself to the other members of the set. See the main docs on authentication for details.

Example

If we had a two-member replica set with members *a* and *b*, we could start them up with authentication enabled by running:

```
a$ echo "this is my super secret key" > mykey
a$ chmod 600 mykey
a$ mongod --keyFile mykey # other options...

b$ echo "this is my super secret key" > mykey
b$ chmod 600 mykey
b$ mongod --keyFile mykey # other options...
```

Then run `rs.initiate()` and so on.

Using the Database with Replica Set Authentication On

From the client's perspective, authentication works the same way with replica sets as it does with single servers.

For example, suppose you create a new replica set and start the members with `--keyFile`. Connect to the master locally to add users:

```
master$ mongo
MongoDB shell version: x.y.z
connecting to: test
> db.addUser("foo", "bar")
```

Clients should authenticate as usual when they make connections.

```
any-member$ mongo -u foo -p
MongoDB shell version: x.y.z
Enter password: <bar>
```

Upgrading to Replica Sets

Upgrading From a Single Server

If you're running MongoDB on a single server, upgrading to replica sets is trivial (and a good idea!). First, we'll initiate a new replica set with a single node. We need a name for the replica set - in this case we're using `foo`. Start by shutting down the server and restarting with the `--replSet` option, and our set name:

```
$ ./mongod --replSet foo
```

Add the --rest option too (just be sure that port is secured): the <host>:28017/_replSet diagnostics page is incredibly useful.

The server will allocate new *local* data files before starting back up. Consider pre-allocating those files if you need to minimize downtime.

Next we'll connect to the server from the shell and initiate the replica set:

```
$ ./mongo
MongoDB shell version: ...
connecting to: test
> rs.initiate();
{
    "info2" : "no configuration explicitly specified -- making one",
    "info" : "Config now saved locally.  Should come online in about a minute.",
    "ok" : 1
}
```

The server should now be operational again, this time as the primary in a replica set consisting of just a single node. The next step is to add some additional nodes to the set.

Upgrading From Replica Pairs or Master/Slave

The best way to upgrade is to simply restart the current master as a single server replica set, and then add any slaves after wiping their data directory. To find the master in a replica pair, run `db.isMaster()`.

```
s> db.isMaster()
{
        "ismaster" : 0,
        "remote" : "localhost:27018",
        "info" : "direct negotiation",
        "maxBsonObjectSize" : 16777216,
        "ok" : 1
}
```

Once you know the master, shut down the *mongod* processes on the master and slave.

```
m$ killall mongod
s$ killall mongod
```

Backup your */data/db* directories, just in case.

```
m$ cp /data/db/* /to_somewhere_backup/
s$ cp /data/db/* /to_slave_backup/
```

Now, start up the master with the *--replSet* option, and initialize a one-member replica set.

```
m$ mongod --rest --replSet mysetname
m$ mongo
m> rs.initiate()
```

Now there are two paths we can take: either resyncing the slaves from scratch or using their existing data. Resyncing takes longer. Using the existing data is only possible if the slave was up-to-date before the replica pair was shut down **and** you add it to the replica set before the master has handled "too many" new writes (the size of the oplog determines what "too many" is).

Resyncing the Slaves

To resync, clear the data directory:

```
s$ rm -r /data/db/* # if you're using a non-default dbpath, this may be somewhere else
s$ # /data/db is now empty
```

Then start up the slave with the *--replSet* option.

```
s$ mongod --rest --replSet mysetname
```

In the database shell, add the slave as a new member in the replica set.

```
m> // still in the mongo shell on the master
m> rs.add("s") // "s" is your slave host name
m> rs.status(); // see also http://localhost:28017/_replSet
```

Adding An Arbiter

If there are an even number of replica set members, we should add an arbiter to break ties on elections and know who is up in a network partition. An arbiter is very lightweight and can run on virtually any server (including 32 bit servers). We use different directories and ports here so that the server is still available as a "normal" mongod server if that is desired and also to avoid confusion. The */data/arb* directory will be very small in content size.

```
arb$ mkdir /data/arb
arb$ mongod --rest --replSet mysetname --dbpath /data/arb --port 30000
```

Then add the arbiter to your replica set:

```
m> rs.addArb("arb:30000"); // replace 'arb' with your arb host name
m> rs.status()
```

Upgrading Drivers

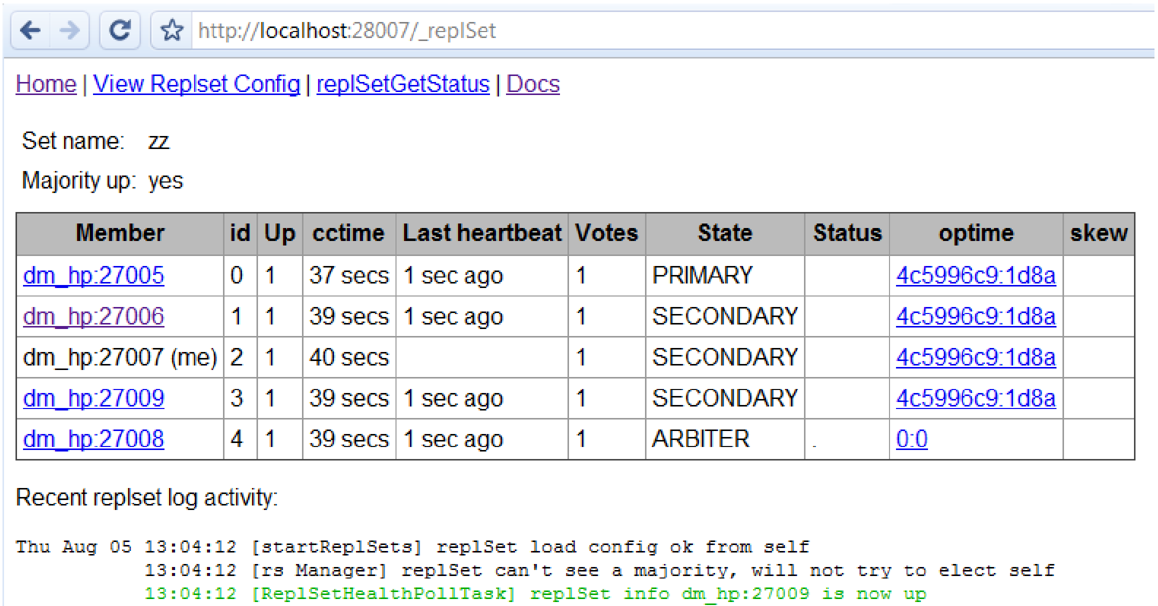
There are new versions of most MongoDB Drivers which support replica sets elegantly. See the documentation pages for the specific driver of interest.

Replica Set Admin UI

The `mongod` process includes a simple administrative UI for checking the status of a replica set.

To use, first enable `--rest` from the `mongod` command line. The rest port is the db port plus 1000 (thus, the default is 28017). Be sure this port is secure before enabling this.

Then you can navigate to `http://<hostname>:28017/` in your web browser. Once there, click Replica Set Status (/_replSet) to move to the Replica Set Status page.



Member	id	Up	cctime	Last heartbeat	Votes	State	Status	optime	skew
dm_hp:27005	0	1	37 secs	1 sec ago	1	PRIMARY		4c5996c9:1d8a	
dm_hp:27006	1	1	39 secs	1 sec ago	1	SECONDARY		4c5996c9:1d8a	
dm_hp:27007 (me)	2	1	40 secs		1	SECONDARY		4c5996c9:1d8a	
dm_hp:27009	3	1	39 secs	1 sec ago	1	SECONDARY		4c5996c9:1d8a	
dm_hp:27008	4	1	39 secs	1 sec ago	1	ARBITER	.	0:0	

See Also

- Http Interface

Replica Set Commands

- Shell Helpers
- Commands
    - isMaster
    - replSetGetStatus
        - state
        - health
        - pingMs
        - optime, optimeDate
        - errmsg
    - replSetInitiate
    - replSetReconfig
        - `force` option
    - replSetStepDown
    - replSetFreeze

Shell Helpers

```
rs.help()                      show help
rs.status()                    { replSetGetStatus : 1 }
rs.initiate()                  { replSetInitiate : null } initiate
                                   with default settings
rs.initiate(cfg)               { replSetInitiate : cfg }
rs.add(hostportstr)            add a new member to the set
rs.add(membercfgobj)           add a new member to the set
rs.addArb(hostportstr)         add a new member which is arbiterOnly:true
rs.remove(hostportstr)         remove a member (primary, secondary, or arbiter) from the set
rs.stepDown()                  { replSetStepDown : true }
rs.conf()                      return configuration from local.system.replset
db.isMaster()                  check who is primary
```

rs.conf() can be particularly useful when using commands that manipulate a replica set, such as replSetReconfig, because it can be used to get a copy of the current configuration which can be modified and then put back. For an example of this, see Reconfiguring when Members are Up.

Commands

isMaster

Checks if the node to which we are connecting is currently primary. Most drivers do this check automatically and then send queries to the current primary.

Returns an object that looks like:

```
> db.adminCommand( { isMaster : 1 } )
{
    "setName" : "florble",
    "ismaster" : false,
    "secondary" : true,
    "hosts" : [
        "sf1.example.com",
        "sf4.example.com",
        "ny3.example.com"
    ],
    "passives" : [
        "sf3.example.com",
        "sf2.example.com",
        "ny2.example.com",
    ],
    "arbiters" : [
        "ny1.example.com",
    ]
    "primary" : "sf4.example.com",
    "me" : "ny3.example.com",
    "maxBsonObjectSize" : 16777216,
    "ok" : 1
}
```

The *hosts* array lists primary and secondary servers, the *passives* array lists passive servers, and the *arbiters* array lists arbiters.

If the "`ismaster`" field is `false`, there will be a "`primary`" field that indicates which server is primary.

replSetGetStatus

Get status on the replica set from this node's point of view. `rs.status()` is the mongo shell helper for this command. The output looks like:

```
> db.adminCommand( { replSetGetStatus : 1 } )
{
        "set" : "foo",
        "date" : ISODate("2012-01-06T00:30:47Z"),
        "myState" : 1,
        "members" : [
                {
                        "_id" : 0,
                        "name" : "localhost:27018",
                        "health" : 1,
                        "state" : 1,
                        "stateStr" : "PRIMARY",
                        "optime" : {
                                "t" : 1325809572000,
                                "i" : 1
                        },
                        "optimeDate" : ISODate("2012-01-06T00:26:12Z"),
                        "self" : true
                },
                {
                        "_id" : 1,
                        "name" : "localhost:27019",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 274,
                        "optime" : {
                                "t" : 1325809572000,
                                "i" : 1
                        },
                        "optimeDate" : ISODate("2012-01-06T00:26:12Z"),
                        "lastHeartbeat" : ISODate("2012-01-06T00:30:47Z"),
                        "pingMs" : 0
                },
                {
                        "_id" : 2,
                        "name" : "localhost:27020",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 274,
                        "optime" : {
                                "t" : 1325809572000,
                                "i" : 1
                        },
                        "optimeDate" : ISODate("2012-01-06T00:26:12Z"),
                        "lastHeartbeat" : ISODate("2012-01-06T00:30:47Z"),
                        "pingMs" : 1
                }
        ],
        "ok" : 1
}
```

state

The *state*, *myState*, and *stateStr* fields indicates the state of this server. Possible state values are:

0	Starting up, phase 1 (parsing configuration)
1	Primary
2	Secondary
3	Recovering (initial syncing, post-rollback, stale members)
4	Fatal error
5	Starting up, phase 2 (forking threads)

6	Unknown state (member has never been reached)
7	Arbiter
8	Down
9	Rollback
10	Removed

health

The *health* field indicates the health of the server/member. Typical values are zero if a server is down (from this server's POV) and 1 if it is up.

pingMs

The *pingMs* field (v2.0+) reports how long heartbeat commands take from this member to the other member. When members are performing well the pingMs value should approach the round trip ping times of the network between the nodes. If a server is overloaded, this value could be significant larger (which would indicate a problem). Also if the network has problems this value will intermittently report a high value. The metric is an ongoing average of the last several ping requests (represenative of average ping time for roughly say, the last minute).

optime, optimeDate

The *optime*, *optimeDate* fields specify the timestamp of the last operation on the node. This value could be a bit behind realtime as heartbeat requests between servers only occur every few seconds. Thus the real replication lag might be very low even if this statistic shows a few seconds.

errmsg

The *errmsg* field contains informational messages, as shown above.

replSetInitiate

```
> db.adminCommand( { replSetInitiate : <config> } )
```

Initiate a replica set. Run this command at one node only, to initiate the set. Whatever data is on the initiating node becomes the initial data for the set. This is a one time operation done at cluster creation. `rs.initiate(<cfg>)` is the mongo shell helper for this command. See also Configuration.

replSetReconfig

Adjust configuration of a replica set (just like initialize).

```
db._adminCommand({replSetReconfig: cfg })
```

Note: `db._adminCommand` is short-hand for `db.getSisterDB("admin").runCommand();`

Note: as of v1.7.2, `replSetReconfig` closes all connections, meaning there will be no database response to this command.

`force` option

As of 2.0 you can force a reconfigure when you have less than a majority of the replica set up using the {force:true} option.

replSetStepDown

```
db.adminCommand( { replSetStepDown : <seconds> } )
```

Manually tell a member to step down as primary. Node will become eligible to be primary again after the specified number of seconds. (Presumably, another node will take over by then if it were eligible.)

If the primary cannot see anyone who has synced to within 10 seconds of its latest op, the primary will reject the step down request. You can force the request by passing a `force : true` option.

For example:

```
> db.adminCommand({replSetStepDown : 1})
{
        "closest" : 1302040824,
        "difference" : 25,
        "errmsg" : "no secondaries within 10 seconds of my optime",
        "ok" : 0
}
> db.adminCommand({replSetStepDown : 1, force : true})
// works
```

v1.7.2+: `replSetStepDown` closes all connections, meaning there will be no database response to this command.

v1.7.3+: the seconds parameter above can be specified. In older versions, the step down was always for one minute only.

v1.9.0+: Added ability for primary to reject step down and added force option.

replSetFreeze

```
db.adminCommand( { replSetFreeze : <seconds> } )
```

v1.7.3+

'Freeze' state of this member to the extent we can do that. What this really means is that this node will not attempt to become primary until the time period specified expires.

You can call again with `{replSetFreeze:0}` to unfreeze sooner. A process restart unfreezes the member also.

If the node is already primary, you need to use `replSetStepdown` instead.

Forcing a Member to be Primary

Replica sets automatically negotiate which member of the set is primary and which are secondaries. If you want a certain member to be primary, there are a couple ways to force this.

v2.0+

In v2.0+, you can set the priority of the preferred primary to be higher than the priorities of the other nodes.

For example, if we had members A, B, and C and A is the current primary and we want B to be primary, we could give B a higher priority like so:

```
> config = rs.conf()
{
    "_id" : "foo",
    "version" : 1,
    "members" : [
        {
            "_id" : 0,
            "host" : "A",
        },
        {
            "_id" : 1,
            "host" : "B",
        },
        {
            "_id" : 2,
            "host" : "C",
        }
    ]
}
> config.version++
> // the default priority is 1
> config.members[1].priority = 2
> rs.reconfig(config)
```

Assuming B is synced to within 10 seconds of A, A will step down, B (and C) will catch up to where A is, and B will be elected primary.

If B is far behind A, A will not step down until B is within 10 seconds of its optime. This minimizes the amount of time there will be no primary on failover. If you do not care about how long the set is primary-less, you can force A to step down by running:

```
> db.adminCommand({replSetStepDown:1000000, force:1})
```

B will sync until it is caught up with A and then become primary.

Older versions

If you want to force a node to be primary at a given point in time, use the `replSetFreeze` and `replSetStepdown` commands (v1.8+). If we have members A, B, and C, and A is current primary, and we want B to become primary, we would send freeze to C so that it does not attempt to become primary, and then stepDown to A.

See the Commands page for more information.

```
$ mongo --host C
> // first check that everyone is healthy and in the states we expect:
> rs.status()
> // C : not eligible to be primary for 120 seconds
> rs.freeze(120)
> exit

$ mongo --host A
> // A : step down as primary and ineligible to be primary for 120 seconds
> rs.stepDown(120)
> // B will now become primary. for this to work B must be up to date.
```

Note that during transitions of primary, there is a short window when no node is primary.

Command replSetStepDown

```
> db.adminCommand( { replSetStepDown : <seconds> } )
```

Step down as primary. Will not try to reelect self for the specified time period (1 minute if no numeric secs value specified). (If another member with same priority takes over in the meantime, that member will stay primary.)

Note: seconds parameter is new to v1.8. Old versions default to 60 seconds regardless of param value.

Command replSetFreeze

v1.8+

```
> db.adminCommand( { replSetFreeze : <seconds> } )
```

'freeze' state of member to the extent we can do that. What this really means is that this node will not attempt to become primary until the time period specified expires. You can call again with {replSetFreeze:0} to unfreeze sooner. A process restart unfreezes the member also.

Connecting to Replica Sets from Clients

Most drivers have been updated to provide ways to connect to a replica set. In general, this is very similar to how the drivers support connecting to a replica pair.

Instead of taking a pair of hostnames, the drivers will typically take a comma separated list of host[:port] names. This is a *seed host list*; it need not be every member of the set. The driver then looks for the primary from the seeds. The seed members will report back other members of the set that the client is not aware of yet. Thus we can add members to a replica set without changing client code.

With Sharding

With sharding, the client connects to a `mongos` process. The `mongos` process will then automatically find the right member(s) of the set.

See Also

- Driver authors should review Connecting Drivers to Replica Sets.

Replica Set FAQ

See Also:

- Intro FAQ
- Developer FAQ
- Sharding FAQ

How long does replica set failover take?

It may take 10-30 seconds for the primary to be declared down by the other members and a new primary elected. During this window of time, the cluster is down for "primary" operations – that is, writes and strong consistent reads. However, you may execute eventually consistent queries to secondaries at any time (in slaveOk mode), including during this window.

What's a master or primary?

This is a node/member which is currently the primary and processes all writes for the replica set. In a replica set, on a failover event, a different member can become primary.

By default all reads and writes go to the primary. To read from a secondary use the slaveOk option.

What's a secondary or slave?

A secondary is a node/member which applies operations from the current primary. This is done by tailing the replication oplog (local.oplog.rs). Replication from primary to secondary is asynchronous, however the secondary will try to stay as close to current as possible (often this is just a few milliseconds on a LAN).

Can I replicate over a WAN? The internet? What if the connection is noisy?

This typically works well as the replication is asynchronous; for example some MongoDB users replicate from the U.S. to Europe over the Internet. If the TCP connection between secondary and primary breaks, the secondary will try reconnecting until it succeeds. Thus network flaps do not require administrator intervention. Of course, if the network is very slow, it may not be possible for the secondary to keep up.

Should I use master/slave replication, or replica sets?

v1.8+, replica sets are preferred. (Most r&d at this point is done on replica sets.)

Should I use replica sets or replica pairs?

v1.6+, use Replica Sets. Replica pairs are deprecated.

Why is journaling recommended with replica sets given that replica set members already have redundant copies of the data?

We recommend using journaling with replica sets. A good way to start is to turn it on for a single member, then you can see if there is any noticable performance difference between that member and the others.

Journaling facilitates fast crash recovery and eliminates the need for repairDatabase or a full resync from another member. Also if you are working with only one data center it is helpful if all machines lose power simultaneously.

Additionally this makes nodes going down and up fully automated with no sys admin intervention (at least for the database layer of the stack).

Note there is some write overhead from journaling; reads are the same speed. Journaling defaults to on in v2.0+.

Do I have to call getLastError to make a write durable?

No. If you don't call getLastError (aka "Safe Mode") the server does exactly the same behavior as if you had. The getLastError call simply lets one get confirmation that the write operation was successfully committed. Of course, often you will want that confirmation, but the safety of the write and its durability is independent.

What happens if I accidentally delete the local.* files on a node?

Please post to support forums for help.

How many arbiters should I have?

Two members with data and one arbiter is a common configuration. A majority is needed to elect a primary; adding the primary achieves three voters and thus 2 out of 3 votes yields a majority.

A set with three members which have data does not need an arbiter as it has three voting members.

If the members with data are in two data centers, it is good practice to put an arbiter elsewhere so that the system can tell which data center is up / visible to the world.

Replica Sets Troubleshooting

can't get local.system.replset config from self or any seed (EMPTYCONFIG)

Set needs to be initiated. Run `rs.initiate()` from the shell.

If the set is already initiated and this is a new node, verify it is present in the replica set's configuration and there are no typos in the host names:

```
> // send to a working node in the set:
> rs.conf()
```

Replication halts with "objects in a capped ns cannot grow" (assertion 10003)

Generally this happens if you have a capped collection without an `_id` index and you are using a custom `_id`.

To fix, make sure that any capped collections you are using have a unique index on the `_id` field and resync the halted slave.

"couldn't initiate : can't find self in the replset config my port: 27017" under Mac OS X

Generally this happens because your hostname and computer name do not match. The safest solution is to supply a config object to rs.initiate. Another solution is to:

Open `System Preferences`, select the `Sharing` page, and set your `Computer Name` at the top. Then open `Terminal` and run

```
$ sudo hostname (name chosen above including the .local at the end)
```

It will prompt you for your password. Once the command has finished, rs.initiate should work.

"not electing self, not all members up and we have been up less than 5 minutes"

The idea here is that if a bunch of nodes bounce all at once, we don't want to drop data if we don't have to – we'd rather be offline and wait a little longer instead. Once a node has been up for five minutes, it is eligible to be primary as long as it can achieve a majority. In addition, if all members are up, a member can become primary immediately.

Reconfiguring a replica set when members are down

One can easily reconfigure a replica set when some members are down as long as *a majority is still up*. In that case, simply send the reconfig command to the current primary.

If there is no majority (i.e. a majority of nodes are not up or reachable and will not be anytime soon), you must manually intervene to change the set configuration. Reconfiguring a minority partition can be dangerous, as two sides of a network partition won't both be aware of the reconfiguration. Thus, this should not be done in scripts, but only by an operator after careful consideration.

In v2.0+, the `force : true` option was added for reconfiguring the set when a majority is down. `force` is for **manually** reconfiguring the set when there is a serious problem, such as a disaster recovery failover. Do not use it every time you reconfigure, or put it into a script that automatically runs, or use it when there is still a primary.

> If you use `force` and look at your replica set config, you will notice that your version number has jumped a lot (by tens or hundreds of thousands). This is normal and designed to prevent set version collisions if network partitioning ends.

Suggestions on how to deal with majority outages are outlined below.

v2.0+

In newer versions we are able to recover when a majority of nodes are down or on the wrong side of a network partition. We will connect to a surviving member (any one is fine) and "force" a reconfiguration of the set without the down hosts.

1. Consider doing a backup on a surviving member, both for safety and to bring up more new set members more easily.
2. Save the current config somewhere so that you can switch back to the old configuration once the down/unreachable members are healed.

```
> config = rs.config()
> printjson(config) # store this somewhere
```

3. Remove the down members from the `config.members` array.

```
> // remove the unreachable members of the set. this is just an example, look
> // at config to determine which members to keep
> config.members = [config.members[1], config.members[3], config.members[4]]
```

4. Reconfigure the set using the `force` option on the surviving member.

```
> rs.reconfig(config, {force : true})
```

You should now have a new primary, most likely the node you are connected to.

If the failure or partition was only temporary, when healed the nodes will detect that they have been removed from the set and enter a special state where they are up but refuse to answer requests as they are no longer syncing changes. You can now re-add them to the config object and do a reconfig (without `force`). **Be sure that each host has the same `_id` it had before.** If it does not, it will not become an active member of the set and complain verbosely in the logs until you restart it (at which point it will be fine).

Once you add the removed hosts back into the set, they will will detect that they have been added and synchronize to the current state of the set. Be aware that if the original master was one of the removed nodes, these members may need to [rollback].

Pre v2.0:

Versions before 1.9.1 do not support the `force` flag to reconfig. This means that we cannot cleanly reconfigure an existing set to remove down nodes unless we can reach the primary, even if no primary exists. Instead, we need to abandon the old replica set. There are two options for moving forward and using the data on surviving nodes.

Option 1: Turn off replication

One option is to make a surviving mongod a standalone server and not a set member:

1. stop a surviving `mongod`
2. consider doing a backup...
3. delete the `local.*` datafiles in the data directory. This will prevent potential future confusion if it is ever restarted with --replSet in the future.
4. restart `mongod` without the `--replSet` parameter.

We are now back online with a single node that is not a replica set member. Clients can use it for both reads and writes.

Option 2: "Break the mirror"

This option will will use the "break the mirror" technique. One of the surviving members will be selected to be the new master and be the "seed" for a new replica set. Unfortunately all slaves will need to completely resync from this new master.

1. stop the surviving `mongod`(s)
2. consider doing a backup...
3. delete the `local.*` datafiles on the new master
4. delete (ideally just move to a backup location) all the datafiles from any other surviving members
5. restart all `mongod`(s) with the new replica set name either on the command line or in a config file
6. initiate this new set on the new master
7. then, add each of the slaves to this new master

See Also

- Reconfiguring when Members are Up

Resyncing a Very Stale Replica Set Member

Error RS102

MongoDB writes operations to an oplog. For replica sets this data is stored in collection local.oplog.rs. This is a capped collection and wraps when full "RRD"-style. Thus, it is important that the oplog collection is large enough to buffer a good amount of writes when some members of a replica set are down. If too many writes occur, the down nodes, when they resume, cannot catch up. In that case, a full resync would be required.

In v1.8+, you can run db.printReplicationInfo() to see the status of the oplog on both the current primary and the overly stale member. This should show you their times, and if their logs have an overlapping time range. If the time ranges don't overlap, there is no way for the stale secondary to recover and catch up (except for a full resync).

There is also a MMS graph of the oplog time length.

```
> db.printReplicationInfo()
configured oplog size:   47.6837158203125MB
log length start to end: 132secs (0.04hrs)
oplog first event time:  Wed Apr 13 2011 02:58:08 GMT-0400
oplog last event time:   Wed Apr 13 2011 03:00:20 GMT-0400
now:                     Wed Apr 13 2011 14:09:08 GMT-0400
```

Sizing the oplog

The command line `--oplogSize` parameter sets the oplog size. A good rule of thumb is 5 to 10% of total disk space. On 64 bit builds, the default is large and similar to this percentage. You can check your existing oplog sizes from the `mongo` shell :

```
> use local
> db.oplog.rs.stats()
```

What to do on a RS102 sync error

If one of your members has been offline and is now too far behind to catch up, you will need to resync. There are a number of ways to do this.

- **Perform a full resync**. If you stop the failed `mongod`, delete all data in the dbpath (including subdirectories), and restart it, it will automatically resynchronize itself. *Obviously it would be better/safer to back up the data first. If disk space is adequate, simply move it to a backup location on the machine if appropriate.* Resyncing may take a long time if the database is huge or the network slow – even idealized one terabyte of data would require three hours to transmit over gigabit ethernet.

or

- **Copy data from another member**: You can copy all the data files from another member of the set IF you have a snapshot of that member's data file's. This can be done in a number of ways. The simplest is to stop `mongod` on the source member, copy all its files, and then restart `mongod` on both nodes. The Mongo fsync and lock feature is another way to achieve this if you are using EBS or a SAN. On a slow network, snapshotting all the datafiles from another (inactive) member to a gziped tarball is a good solution. Also similar strategies work well when using SANs and services such as Amazon Elastic Block Service snapshots.

or

- **Find a member with older data**: *Note: this is only possible (and occurs automatically) in v1.8+*. If another member of the replica set has a large enough oplog or is far enough behind that the stale member can sync from it, the stale member can bootstrap itself from this member.

See Also

- Adding a New Set Member
- Moving or Replacing a Member

Replica Sets Limits

- A set can contain
  - A maximum of 12 members
  - A maximum of 7 members that can vote

- Typically the set configuration can be changed only when a majority can be established. Limits on config changes to sets at first.

Especially when a lot of set members are down.

v1.6

- Authentication is supported only in v1.8+.
- Map/reduce writes new collections to the server. Because of this, for now it may only be used on the primary. This will be enhanced later.

Replica Set Internals

> ⚠ This page contains notes on the original MongoDB replica set design. While the concepts still apply, this page is not kept perfectly up-to-date; consider this page historical rather than definitive.

Design Concepts

Check out the Replica Set Design Concepts for some of the core concepts underlying MongoDB Replica Sets.

Configuration

Command Line

We specify --replSet *set_name*/*seed_hostname_list* on the command line. *seed_hostname_list* is a (partial) list of some members of the set. The system then fetches full configuration information from the collection local.system.replset. *set_name* is specified to help the system catch misconfigurations. In current versions of MongoDB (1.8+) *seed_hostname_list* is not required; --replSet *set_name* will suffice.

Node Types

Conceptually, we have some different types of nodes:

- Standard - a standard node as described above. Can transition to and from being a *primary* or a *secondary* over time. There is only one primary (master) server at any point in time.
- Passive - a server can participate as if it were a member of the replica set, but be specified to never be Primary.
- Arbiter - member of the cluster for consensus purposes, but receives no data. Arbiters cannot be seed hosts.

Each node in the set has a *priority* setting. On a resync (see below), the rule is: choose as master the node with highest priority that is healthy. If multiple nodes have the same priority, pick the node with the freshest data. For example, we might use 1.0 priority for Normal members, 0.0 for passive (0 indicates cannot be primary no matter what), and 0.5 for a server in a less desirable data center.

local.system.replset

This collection has one document storing the replica set's configuration. See the configuration page for details.

Set Initiation (Initial Setup)

For a new cluster, on negotiation the max OpOrdinal is zero everywhere. We then know we have a new replica set with no data yet. A special command

```
{replSetInitiate:1}
```

is sent to a (single) server to begin things.

Design

Server States

- *Primary* - Can be thought of as "master" although which server is primary can vary over time. Only 1 server is primary at a given point in time.
- *Secondary* - Can be thought of as a slave in the cluster; varies over time.
- *Recovering* - getting back in sync before entering Secondary mode.

Applying Operations

Secondaries apply operations from the Primary. Each applied operation is also written to the secondary's local oplog. We need only apply from the current primary (and be prepared to switch if that changes).

OpOrdinal

We use a monotonically increasing ordinal to represent each operation.

These values appear in the oplog (local.oplog.$main). maxLocalOpOrdinal() returns the largest value logged. This value represents how up-to-date we are. The first operation is logged with ordinal 1.

Note two servers in the set could in theory generate different operations with the same ordinal under some race conditions. Thus for full uniqueness we must look at the combination of server id and op ordinal.

Picking Primary

We use a consensus protocol to pick a primary. Exact details will be spared here but that basic process is:

1. get maxLocalOpOrdinal from each server.
2. if a majority of servers are not up (from this server's POV), remain in Secondary mode and stop.
3. if the last op time seems very old, stop and await human intervention.
4. else, using a consensus protocol, pick the server with the highest maxLocalOpOrdinal as the Primary.

Any server in the replica set, when it fails to reach master, attempts a new election process.

Heartbeat Monitoring

All nodes monitor all other nodes in the set via heartbeats. If the current primary cannot see half of the nodes in the set (including itself), it will fall back to secondary mode. This monitoring is a way to check for network partitions. Otherwise in a network partition, a server might think it is still primary when it is not.

Heartbeats requests are sent out every couple of seconds and can either receive a response, get an error, or time out (after ~20 seconds).

Assumption of Primary

When a server becomes primary, we assume it has the latest data. Any data newer than the new primary's will be discarded. Any discarded data is backed up to a flat file as raw BSON, to allow for the possibility of manual recovery (see this case for some details). In general, manual recovery will not be needed - if data must be guaranteed to be committed it should be written to a majority of the nodes in the set.

Failover

We renegotiate when the primary is unavailable, see Picking Primary.

Resync (Connecting to a New Primary)

When a secondary connects to a new primary, it must resynchronize its position. It is possible the secondary has operations that were never committed at the primary. In this case, we roll those operations back. Additionally we may have new operations from a previous primary that never replicated elsewhere. The method is basically:

- for each operation in our oplog that does not exist at the primary, (1) remove from oplog and (2) resync the document in question by a query to the primary for that object. update the object, deleting if it does not exist at the primary.

We can work our way back in time until we find a few operations that are consistent with the new primary, and then stop.

Any data that is removed during the rollback is stored offline (see Assumption of Primary, so one can manually recover it. It can't be done automatically because there may be conflicts.

Reminder: you can use w= to ensure writes make it to a majority of slaves before returning to the user, to ensure no writes need to be rolled back.

Consensus

Fancier methods would converge faster but the current method is a good baseline. Typically only ~2 nodes will be jockeying for primary status at any given time so there isn't be much contention:

- query all others for their maxappliedoptime
- try to elect self if we have the highest time and can see a majority of nodes
  - if a tie on highest time, delay a short random amount first
  - elect (selfid,maxoptime) msg -> others
- if we get a msg and our time is higher, we send back NO
- we must get back a majority of YES
- if a YES is sent, we respond NO to all others for 1 minute. Electing ourself counts as a YES.
- repeat as necessary after a random sleep

Increasing Durability

We can trade off durability versus availability in a replica set. When a primary fails, a secondary will assume primary status with whatever data it has. Thus, we have some desire to see that things replicate quickly. Durability is guaranteed once a majority of servers in the replica set have an operation.

To improve durability clients can call getlasterror and wait for acknowledgement until replication of a an operation has occurred. The client can then selectively call for a blocking, somewhat more synchronous operation.

Reading from Secondaries and Staleness

Secondaries can report via a command how far behind the primary they are. Then, a read-only client can decide if the server's data is too stale or close enough for usage.

Example

server-a: secondary oplog: ()
server-b: secondary oplog: ()
server-c: secondary oplog: ()
…
server-a: primary oplog: (a1,a2,a3,a4,a5)
server-b: secondary oplog: ()
server-c: secondary oplog: ()
…
server-a: primary oplog: (a1,a2,a3,a4,a5)
server-b: secondary oplog: (a1)
server-c: secondary oplog: (a1,a2,a3)
…
// server-a goes down
…
server-b: secondary oplog: (a1)
server-c: secondary oplog: (a1,a2,a3)
...
server-b: secondary oplog: (a1)
server-c: primary oplog: (a1,a2,a3) // c has highest ord and becomes primary
...
server-b: secondary oplog: (a1,a2,a3)
server-c: primary oplog: (a1,a2,a3,c4)
...
server-a resumes
...
server-a: recovering oplog: (a1,a2,a3,a4,a5)
server-b: secondary oplog: (a1,a2,a3)
server-c: primary oplog: (a1,a2,a3,c4)
…
server-a: recovering oplog: (a1,a2,a3,c4)
server-b: secondary oplog: (a1,a2,a3,c4)
server-c: primary oplog: (a1,a2,a3,c4)
…
server-a: secondary oplog: (a1,a2,a3,c4)
server-b: secondary oplog: (a1,a2,a3,c4)
server-c: primary oplog: (a1,a2,a3,c4,c5,c6,c7,c8)
…

server-a: secondary oplog: (a1,a2,a3,c4,c5,c6,c7,c8)
server-b: secondary oplog: (a1,a2,a3,c4,c5,c6,c7,c8)
server-c: primary oplog: (a1,a2,a3,c4,c5,c6,c7,c8)

In the above example, server-c becomes primary after server-a fails. Operations (a4,a5) are lost. c4 and c5 are new operations with the same ordinals.

Administration

See the Replica Set Commands page for full info.

Commands:

- { replSetFreeze : <bool> } "freeze" or unfreeze a set. When frozen, new nodes cannot be elected master. Used when doing administration. Details TBD.
- { replSetGetStatus : 1 } get status of the set, from this node's POV
- { replSetInitiate : 1 }
- { ismaster : 1 } check if this node is master

Future Versions

- add support for replication trees / hierarchies
- replicating to a slave that is not a member of the set (perhaps we do not need this given we have the Passive set member type)

See Also

- About the local database

Master Slave

> Use Replica Sets rather than this – replica sets are a functional superset of master/slave, and newer, more robust code.

Configuration and Setup

To configure an instance of Mongo to be a master database in a master-slave configuration, you'll need to start two instances of the database, one in *master* mode, and the other in *slave* mode.

> **Data Storage**
> The following examples explicitly specify the location of the data files on the command line. This is unnecessary if you are running the master and slave on separate machines, but in the interest of the readers who are going try this setup on a single node, they are supplied in the interest of safety.

```
$ bin/mongod --master [--dbpath /data/masterdb/]
```

As a result, the master server process will create a `local.oplog.$main` collection. This is the "transaction log" which queues operations which will be applied at the slave.

To configure an instance of Mongo to be a slave database in a master-slave configuration:

```
$ bin/mongod --slave --source <masterhostname>[:<port>]  [--dbpath /data/slavedb/]
```

Details of the source server are then stored in the slave's `local.sources` collection. Instead of specifying the `--source` parameter, one can add an object to `local.sources` which specifies information about the master server:

```
$ bin/mongo <slavehostname>/local
> db.sources.find(); // confirms the collection is empty.  then:
> db.sources.insert( { host: <masterhostname> } );
```

- *`host: masterhostname`* is the IP address or FQDN of the master database machine. Append `:`*`port`* to the server hostname if you wish to run on a nonstandard port number.
- *`only: databasename`* (optional) if specified, indicates that only the specified database should replicate. NOTE: A bug with `only` is fixed in v1.2.4+.

A slave may become out of sync with a master if it falls far behind the data updates available from that master, or if the slave is terminated and then restarted some time later when relevant updates are no longer available from the master. If a slave becomes out of sync, replication will terminate and operator intervention is required by default if replication is to be restarted. An operator may restart replication using the `{resync:1}` command. Alternatively, the command line option `--autoresync` causes a slave to restart replication automatically (after ten second pause) if it becomes out of sync. If the `--autoresync` option is specified, the slave will not attempt an automatic resync more than once in a ten minute periond.

The `--oplogSize` command line option may be specified (along with `--master`) to configure the amount of disk space in megabytes which will be allocated for storing updates to be made available to slave nodes. If the `--oplogSize` option is not specified, the amount of disk space for storing updates will be 5% of available disk space (with a minimum of 1GB) for 64bit machines, or 50MB for 32bit machines.

Command Line Options

Master

```
--master              master mode
  --oplogSize arg       size limit (in MB) for op log
```

Slave

```
--slave               slave mode
  --source arg          arg specifies master as <server:port>
  --only arg            arg specifies a single database to replicate
  --slavedelay arg      arg specifies delay (in seconds) to be used
                        when applying master ops to slave
  --autoresync          automatically resync if slave data is stale
```

--slavedelay

Sometimes its beneficial to have a slave that is purposefully many hours behind to prevent human error. In MongoDB 1.3.3+, you can specify this with the --slavedelay mongod command line option. Specify the delay in seconds to be used when applying master operations to the slave.

Specify this option at the slave. Example command line:

```
mongod --slave --source mymaster.foo.com --slavedelay 7200
```

Diagnostics

Check master status from the `mongo` shell with:

```
// inspects contents of local.oplog.$main on master and reports status:
db.printReplicationInfo()
```

Check slave status from the `mongo` shell with:

```
// inspects contents of local.sources on the slave and reports status:
db.printSlaveReplicationInfo()
```

(Note you can evaluate the above functions without the parenthesis above to see their javascript source and a bit on the internals.)

As of 1.3.2, you can do this on the slave

```
db._adminCommand( { serverStatus : 1 , repl : N } )
```

`N` is the level of diagnostic information and can have the following values:

- 0 - none
- 1 - local (doesn't have to connect to other server)
- 2 - remote (has to check with the master)

Security

When security is enabled, one must configure a user account for the local database that exists on both servers.

The slave-side of a replication connection first looks for a user repl in local.system.users. If present, that user is used to authenticate against the local database on the source side of the connection. If repl user does not exist, the first user object in local.system.users is tried.

The local database works like the admin database: an account for local has access to the entire server.

Example security configuration when security is enabled:

```
$ mongo <slavehostname>/admin -u <existingadminusername> -p<adminpassword>
> use local
> db.addUser('repl', <replpassword>);
^c
$ mongo <masterhostname>/admin -u <existingadminusername> -p<adminpassword>
> use local
> db.addUser('repl', <replpassword>);
```

Master Slave vs. Replica Sets

Master/slave and replica sets are alternative ways to achieve replication with MongoDB.

Replica sets are newer (v1.6+) and more flexible, although a little more work to set up and learn at first.

The following replica set configuration is equivalent to a two node master/slave setup with hosts M (master) and S (slave):

```
$ # run mongod instances with "--replSet mysetname" parameter
$ # then in the shell:
$ mongo --host M
> cfg = {
>   _id : 'mysetname',
>   members : [
>     { _id : 0, host : 'M', priority : 1 },
>     { _id : 1, host : 'S', priority : 0, votes : 0 }
>   ]
> };
> rs.initiate(cfg);
```

Administrative Tasks

Failing over to a Slave (Promotion)

To permanently fail over from a down master (*A*) to a slave (*B*):

- shut down *A*
- stop `mongod` on B
- backup or delete local.* datafiles on B

- restart `mongod` on B with the `--master` option

Note that is a one time cutover and the "mirror" is broken. A cannot be brought back in sync with B without a full resync.

Inverting Master and Slave

If you have a master (*A*) and a slave (*B*) and you would like to reverse their roles, this is the recommended sequence of steps. Note the following assumes *A* is healthy, up-to-date and up.

1. Halt writes on *A* (using the fsync command)
2. Make sure *B* is caught up
3. Shut down *B*
4. Wipe *local.** on B to remove old local.sources
5. Start up *B* with the `--master` option
6. Do a write on *B* (primes the oplog to provide a new sync start point).
7. Shut down *B*. *B* will now have a new set of local.* files.
8. Shut down *A* and replace *A*'s *local.** files with a copy of *B*'s new *local.** files. Remember to compress the files before/while copying them – they can be quite large.
9. Start *B* with the `--master` option
10. Start *A* with all the usual slave options plus `--fastsync`

If *A* is **not** healthy but the hardware is okay (power outage, server crash, etc.):

- Skip the first two steps
- Replace **all** of *A*'s files with *B*'s files in step 8.

If the hardware is not okay, replace *A* with a new machine and then follow the instructions in the previous paragraph.

Creating a slave from an existing master's disk image

If you can stop write operations to the master for an indefinite period, you can copy the data files from the master to the new slave, and then start the slave with `--fastsync`.

> Be careful with --fastsync. If the data is not perfectly in sync, a discrepancy will exist forever.

`--fastsync` is a way to start a slave starting with an existing master disk image/backup. This option declares that the adminstrator guarantees the image is correct and completely up to date with that of the master. If you have a full and complete copy of data from a master you can use this option to avoid a full synchronization upon starting the slave.

Creating a slave from an existing slave's disk image

You can just copy the other slave's data file snapshot without any special options. Note data snapshots should only be taken when a `mongod` process is down or in fsync-and-lock state.

Resyncing a slave that is too stale to recover

Slaves asynchronously apply write operations from the master. These operations are stored in the master's oplog. The oplog is finite in length. If a slave is too far behind, a full resync will be necessary. See the Halted Replication page.

Slave chaining

Slaves cannot be "chained", they must all connect to the master directly. If a slave is chained to another slave you may see the following in the logs: `assertion 13051 tailable cursor requested on non capped collection ns:local.oplog.$main`

Correcting a slave's source

If you accidentally type the wrong host for the slave's source or wish to change it, you can do so by manually modifying the slave's *local.sources* collection. For example, say you start the slave with:

```
$ mongod --slave --source prod.missisippi
```

Restart the slave without the *--slave* and *--source* arguments.

```
$ mongod
```

Now start the shell and update the *local.sources* collection.

```
> use local
switched to db local
>  db.sources.update({host : "prod.missisippi"}, {$set : {host : "prod.mississippi"}})
```

Restart the slave with the correct command line arguments or no *--source* argument (once *local.sources* is set, no *--source* is necessary).

```
$ ./mongod --slave --source prod.mississippi
$ # or
$ ./mongod --slave
```

Now your slave will be pointing at the correct master.

See Also

- Replica Sets

Halted Replication

These instructions are for master/slave replication. For replica sets, see Resyncing a Very Stale Replica Set Member instead.

If you're running mongod with master-slave replication, there are certain scenarios where the slave will halt replication because it hasn't kept up with the master's oplog.

The first is when a slave is prevented from replicating for an extended period of time, due perhaps to a network partition or the killing of the slave process itself. The best solution in this case is to resyc the slave. To do this, open the mongo shell and point it at the slave:

```
$ mongo <slave_host_and_port>
```

Then run the resync command:

```
> use admin
> db.runCommand({resync: 1})
```

This will force a full resync of all data (which will be very slow on a large database). The same effect can be achieved by stopping `mongod` on the slave, delete all slave datafiles, and restarting it.

Increasing the OpLog Size

Since the oplog is a capped collection, it's allocated to a fixed size; this means that as more data is entered, the collection will loop around and overwrite itself instead of growing beyond its pre-allocated size. If the slave can't keep up with this process, then replication will be halted. The solution is to increase the size of the master's oplog.

Resync warning
At present, the only way to increase the oplog's size is to delete the oplog and create a new one; this has the side-effect that the new oplog's oldest entry will be newer than any slaves' last replication timestamp, and so slaves will need to be resynced after allocating the new oplog. (This warning does not apply to replica sets.)

There are a couple of ways to do this, depending on how big your oplog will be and how much downtime you can stand. But first you need to figure out how big an oplog you need. If the current oplog size is wrong, how do you figure out what's right? The goal is not to let the oplog age out in the time it takes to clone the database. The first step is to print the replication info. On the master node, run this command:

```
> db.printReplicationInfo();
```

You'll see output like this:

```
configured oplog size:   1048.576MB
log length start to end: 7200secs (2hrs)
oplog first event time:  Wed Mar 03 2010 16:20:39 GMT-0500 (EST)
oplog last event time:   Wed Mar 03 2010 18:20:39 GMT-0500 (EST)
now:                     Wed Mar 03 2010 18:40:34 GMT-0500 (EST)
```

This indicates that you're adding data to the database at a rate of 524MB/hr. If an initial clone takes 10 hours, then the oplog should be at least 5240MB, so something closer to 8GB would make for a safe bet.

The standard way of changing the oplog size involves stopping the `mongod` master, deleting the local.* datafiles, and then restarting with the oplog size you need, measured in MB:

```
$ # Stop mongod - killall mongod or kill -2 or ctrl-c) - then:
$ rm /data/db/local.*
$ mongod --oplogSize=8038 --master
```

Once you've changed the oplog size, restart with slave with --autoresync:

```
mongod --slave --autoresync
```

This method of oplog creation might pose a problem if you need a large oplog (say, > 10GB), since the time it takes `mongod` to pre-allocate the oplog files may mean too much downtime. If this is the case, read on.

Manually Allocating OpLog Files

An alternative approach is to create the oplog files manually before shutting down `mongod`. Suppose you need an 20GB oplog; here's how you'd go about creating the files:

1. Create a temporary directory, /tmp/local.
2. You can either create the files yourself or let MongoDB allocate them. If you'd like to create them yourself, here's a shell script for doing just that:

```
cd /tmp/local
  for i in {0..9}
      do
      echo $i
      head -c 2146435072 /dev/zero > local.$i
      done
```

Note that the datafiles aren't exactly 2GB due MongoDB's max int size.

If you'd like MongoDB to preallocate them for you, you can do:

```
$ mongod --dbpath /tmp/local --port 27099 --master --oplogSize=20000
```

Set the port to be something that is different than the other `mongod` running on the machine. Once this instance has finished allocating oplog files (watch the log), shut it down. If you are allocating these files for a replica set, remove the *local.ns* file:

```
$ rm /tmp/local/local.ns
```

3. Shut down the `mongod` master (kill -2) and then replace the oplog files:

```
$ mv /data/db/local.* /safe/place
$ mv /tmp/local/* /data/db/
```

4. Restart the master with the new oplog size:

```
$ mongod --master --oplogSize=20000
```

5. Finally, resync the slave. This can be done by shutting down the slave, deleting all its datafiles, and restarting it.

Replica Pairs

Replica pairs will be removed for 1.8

Replica pairs should be migrated to their replacement, Replica Sets.

Setup of Replica Pairs

Mongo supports a concept of *replica pairs*. These databases automatically coordinate which is the master and which is the slave at a given point in time.

At startup, the databases will negotiate which is master and which is slave. Upon an outage of one database server, the other will automatically take over and become master from that point on. In the event of another failure in the future, master status would transfer back to the other server. The databases manage this themselves internally.

Note: Generally, start with empty /data/db directories for each pair member when creating and running the pair for the first time. See section on Existing Databases below for more information.

To start a pair of databases in this mode, run each as follows:

```
$ ./mongod --pairwith <remoteserver> --arbiter <arbiterserver>
```

where

- *`remoteserver`* is the hostname of the other server in the pair. Append `:`*`port`* to the server hostname if you wish to run on a nonstandard port number.
- *`arbiterserver`* is the hostname (and optional port number) of an *arbiter*. An arbiter is a Mongo database server that helps negotiate which member of the pair is master at a given point in time. Run the arbiter on a third machine; it is a "tie-breaker" effectively in determining which server is master when the members of the pair cannot contact each other. You may also run with no arbiter by not including the --arbiter option. In that case, both servers will assume master status if the network partitions.

One can manually check which database is currently the master:

```
$ ./mongo
> db.$cmd.findOne({ismaster:1});
{ "ismaster" : 0.0 , "remote" : "192.168.58.1:30001" , "ok" : 1.0  }
```

(Note: When security is on, `remote` is only returned if the connection is authenticated for the `admin` database.)

However, Mongo drivers with replica pair support normally manage this process for you.

Consistency

Members of a pair are only eventually consistent on a failover. If machine L of the pair was master and fails, its last couple seconds of operations may not have made it to R - R will not have those operations applied to its dataset until L recovers later.

Security

Example security configuration when security is enabled:

```
$ ./mongo <lefthost>/admin -u <adminusername> -p<adminpassword>
> use local
> db.addUser('repl', <replpassword>);
^c
$ ./mongo <righthost>/admin -u <adminusername> -p<adminpassword>
> use local
> db.addUser('repl', <replpassword>);
```

Replacing a Replica Pair Server

When one of the servers in a Mongo replica pair set fails, should it come back online, the system recovers automatically. However, should a machine completely fail, it will need to be replaced, and its replacement will begin with no data. The following procedure explains how to replace one of the machines in a pair.

Let's assume nodes (`n1`, `n2`) is the old pair and that `n2` dies. We want to switch to (`n1`,`n3`).

1. If possible, assure the dead `n2` is offline and will not come back online: otherwise it may try communicating with its old pair partner.
2. We need to tell `n1` to pair with `n3` instead of `n2`. We do this with a `replacepeer` command. Be sure to check for a successful return value from this operation.

   ```
   n1> ./mongo n1/admin
   > db.$cmd.findOne({replacepeer:1});
   {
       "info" : "adjust local.sources hostname; db restart now required" ,
       "ok" : 1.0
   }
   ```

   At this point, `n1` is still running but is reset to not be confused when it begins talking to `n3` in the future. The server is still up although replication is now disabled.

3. Restart `n1` with the right command line to talk to `n3`

   ```
   n1> ./mongod --pairwith n3 --arbiter <arbiterserver>
   ```

4. Start `n3` paired with `n1`.

   ```
   n3> ./mongod --pairwith n1 --arbiter <arbiterserver>
   ```

   Note that `n3` will not accept any operations as "master" until fully synced with `n1`, and that this may take some time if there is a substantial amount of data on `n1`.

Querying the slave

You can query the slave if you set the slave ok flag. In the shell:

```
db.getMongo().setSlaveOk()
```

What is and when should you use an arbiter?

The arbiter is used in some situations to determine which side of a pair is master. In the event of a network partition (left and right are both up, but can't communicate) whoever can talk to the arbiter becomes master.

If your left and right server are on the same switch, an arbiter isn't necessary. If you're running on the same ec2 availability zone, probably not needed as well. But if you've got left and right on different ec2 availability zones, then an arbiter should be used.

Working with an existing (non-paired) database

Care must be taken when enabling a pair for the first time if you have existing datafiles you wish to use that were created from a singleton database. Follow the following procedure to start the pair. Below, we call the two servers "left" and "right".

- assure no mongod processes are running on both servers
- we assume the data files to be kept are on server left. Check that there is no local.* datafiles in left's /data/db (--dbpath) directory. If there

are, remove them.
- check that there are no datafiles at all on right's /data/db directory
- start the left process with the appropriate command line including --pairwith argument
- start the right process with the appropriate paired command line

If both left and right servers have datafiles in their dbpath directories at pair initiation, errors will occur. Further, you do not want a local database (which contains replication metadata) during initiation of a new pair.

Replication Oplog Length

Replication uses an operation log ("oplog") to store write operations. These operations replay asynchronously on other nodes.

The length of the oplog is important if a secondary is down. The larger the log, the longer the secondary can be down and still recover. Once the oplog has exceeded the downtime of the secondary, there is no way for the secondary to apply the operations; it will then have to do a full synchronization of the data from the primary.

By default, on 64 bit builds, the oplog is allocated to 5% of disk space. Generally this is a reasonable setting.

The oplog is a capped collection, and fixed size once allocated. Once it is created it is not easy to change without losing the existing data. This will be addressed in future versions so that it can be extended.

The `mongod --oplogSize` command line parameter sets the size of the oplog. Changing this parameter after the oplog is created does not change the size of your oplog.

This collection is named:

- `local.oplog.$main` for **master/slave** replication
- `local.oplog.rs` **for replica sets**

See also

- The Halted Replication page
- Resyncing a Very Stale Replica Set Member
- Replica Sets - Oplog

Sharding

MongoDB scales horizontally via an auto-sharding (partitioning) architecture. MongoDB sharding provides:

- Automatic balancing for changes in load and data distribution
- Easy addition of new machines without down time
- Scaling to one thousand nodes
- No single points of failure
- Automatic failover

Getting Started

- Introduction. Philosophy, use cases, and its core components.
- Simple Initial Sharding Architecture
- Configuration. Setting up your cluster.
- Administration

Additional Info

- Failover How failover/HA works.
- Sharding Internals Implementation details.
- Restrictions and Limitations
- FAQ Frequently asked questions.
- HOWTO
    - Changing Config Servers

Presentations and Further Materials

- How Sharding Works - O'Reilly Webcast (February 2011)
- How queries work with sharding (PDF)
- Illustration of chunks and migration (PDF) http://www.10gen.com/video/mongosf2011/sharding
- Scaling MongoDB - O'Reilly ebook
- Schema design at scale (video)
- Mongo Sharding Architecture, Implementation, Internals (video)

See Also

- Replication

Video - How and When to Scale MongoDB with Sharding

Presentation at MongoSV (December 2011)

Changing Config Servers

Sections:

- Upgrading from one config server to three
- Moving your config servers - same host name or virtual ip
- Renaming a config server - different host name
- Replacing a dead config server

> The config server data is the most important data in your entire cluster. Back it up before doing config server maintenance.

Adding and changing config servers is a bit tricky right now. This will be improved in a future release, see http://jira.mongodb.org/browse/SERVER-1658.

Note that config servers are **not** a replica set: instead they use a two phase commit protocol to keep their data synchronous. Thus each config server (if you have 3) has exactly the same data. When one or more config servers are down, the others are available for reading, but not for writing. During that window of time, sharding metadata will be static (which is fine for a while).

If you do not have a full set of config servers then you may wish to read about Sharding and Failover to understand things a bit more.

Upgrading from one config server to three

Unfortunately you will need to shutdown the entire system.

1. Shutdown all processes (mongod, mongos, config server).
2. Copy the data subdirectories (dbpath tree) from the config server to the new config servers.
3. Start the config servers.
4. Restart mongos processes with the new --configdb parameter.
5. Restart mongod processes.

Moving your config servers - same host name or virtual ip

If you are using hostnames or virtual ips for your config, this is pretty simple.

1. Shutdown config server you want to move
2. Change dns entry to new machine
3. Move data to new machine
4. Start new config server

Renaming a config server - different host name

If you wish to use a different name or ip address in the *--configdb* option then this applies to you.

1. Shutdown config server you want to move
2. Move data to new machine
3. Start new config server
4. Shutdown all processes (mongod, mongos, config server).
5. Restart mongod processes.
6. Restart mongos processes with the new --configdb parameter.

Replacing a dead config server

Let's assume we have been running with mongos commands lines of:

```
mongos --configdb hosta,hostb,hostc
```

and that hostb has died. We want to replace it.

You cannot change the name/ip used in the `mongos --configdb` line; if you wish to do this you must follow the directions above for renaming or moving your servers.

1. Provision a new machine. Give it the same hostname (hostb) that the host to be replaced had.
2. Shut down (only) one of the other config server processes – say, hostc. Then copy its data files to hostb. (We shut down so that we know we have a consistent image of the datafiles. See also the backups page for alternatives.)
3. Restart the config server on hostc.
4. Start the config server for the first time on hostb.

That's it. The key above is that we reused the logical (DNS) name of the host. That way we do not have to tell the other (many) processes in the system where the config servers are. Check the length of your DNS ttl using the dig command.

flushRouterConfig command

flushRouterConfig

This command will clear the current cluster information that a mongos process has cached and load the latest settings from the config db. This can be used to force an update when the config db and the data cached in mongos are out of sync. It was added in v1.8.2.

Warning: do <u>not</u> change the config db's content except in ways that are explicitly documented as being acceptable. The config database is not intended to be write manipulated manually.

Example

```
$ mongo mongos-server.local
> db.adminCommand("flushRouterConfig")
{... "flushed":true }
```

Simple Initial Sharding Architecture

Overview

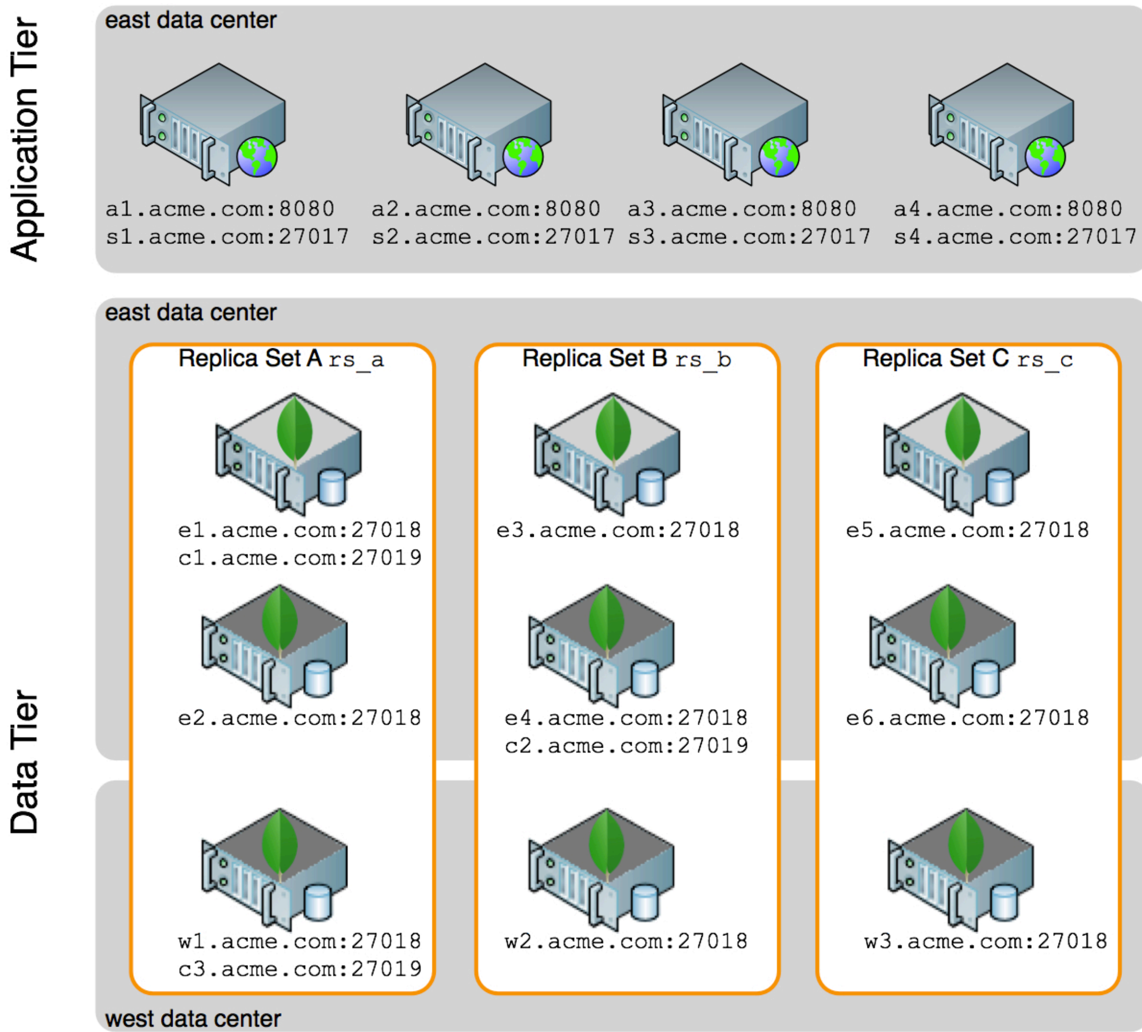
This is a sample sharding architecture that is a good starting point for building your cluster.

Goal

- Two datacenters (East=primary, West=backup/DR)
- Data Tier (MongoDB)
  - 3 shards
  - 3 nodes per shard
  - 9 hosts total
- Application Tier
  - 4 application servers

Machines Locations

- e1-e6 are in 1 datacenter (East)
- w1-w3 are in another datacenter (West)



Datacenter Roles

We'll use datacenter East as the primary, and data center West as disaster recovery.

Replica sets

The replica set nodes in West will be priority 0 so they don't become master automatically.

The first thing we need to do is setup the 3 replica sets

- Replica Set A:
  - e1.acme.com : priority=1
  - e2.acme.com : priority=1
  - w1.acme.com : priority=0

Start the mongod process on each node, e.g.

```
e1.acme.com# mongod --shardsvr --replSet rs_a
e2.acme.com# mongod --shardsvr --replSet rs_a
w1.acme.com# mongod --shardsvr --replSet rs_a
```

Note: --shardsvr will default the port to 27018

In the mongo shell, create the replica set

```
> cfg = {
    _id : "rs_a",
    members : [
        {_id : 0, host : "e1.acme.com:27018", priority : 1},
        {_id : 1, host : "e2.acme.com:27018", priority : 1},
        {_id : 2, host : "w1.acme.com:27018", priority : 0}
    ]
}

> rs.initiate(cfg)
```

Repeat for each replica set as follows

- Replica Set B:
  - e3.acme.com : priority=1
  - e4.acme.com : priority=1
  - w2.acme.com : priority=0
- Replica Set C:
  - e5.acme.com : priority=1
  - e6.acme.com : priority=1
  - w3.acme.com : priority=0

Config Servers

The next thing we need is to choose 3 config server locations.

We'll pick 2 random nodes in E (primary) and 1 in W (backup/DR). Since we have multiple MongoDB process on a node we need to ensure that different ports are being used, so we will use 27019 for the Config Servers.

- Config Servers
  - e1 - c1.acme.com:27019
  - e4 - c2.acme.com:27019
  - w1 - c3.acme.com:27019

> **DNS Aliases**
> We should make dns aliases for these so its easy to change later

Start up the config servers, e.g.

```
e1.acme.com> mongod --configsvr
e4.acme.com> mongod --configsvr
w1.acme.com> mongod --configsvr
```

Note: --configsvr will default the port to 27019

MongoS (routers)

The last question is where to put the mongos.

Suggested

The suggested configuration is to run an instance on **each app-server** (as shown by a1-a4 in diagram above).

Alternative

These are the other common options:

- On each server (all 9)
- Create a (sticky) load-balanced cluster (independent of client/shards)

Startup Options

When we start the mongos, we'll use

```
> mongos --configdb c1.acme.com:27019,c2.acme.com:27019,c3.acme.com:27019
```

Note: be sure to use DNS names for yoru configdb names, not ip addresses. Otherwise moving a config server later will be quite difficult.

Then we'll need to add the 3 replica sets as shards

```
> db.adminCommand( { addShard : "rs_a/e1.acme.com:27018,e2.acme.com:27018,w1.acme.com:27018" } )
> db.adminCommand( { addShard : "rs_b/e3.acme.com:27018,e4.acme.com:27018,w2.acme.com:27018" } )
> db.adminCommand( { addShard : "rs_c/e5.acme.com:27018,e6.acme.com:27018,w3.acme.com:27018" } )
```

Everything is running

At this point your basic architecture is ready to go. You've got 3 shards for scalability, and 3 copies of each piece of data (with one ready for DR). There are obviously many ways to configure this, but this is a pretty simple way to get started.

Your next step is to enable sharding for any database you would like to use sharded, and to then enable sharding for any collection you want sharded. Databases and collections by default will be "unsharded" and simply reside in the first shard.

To save yourself time, don't bother sharding tiny collections, just do the big ones.

Notes

- Names (DNS) should be used everywhere, and consistently
- All client writes/reads will be isolated to the Primary Datacenter (East)
- SlaveOk can be used to allow stale/eventually-consistent reads
    - If this is done, hidden should be used with the West nodes e.g.

```
> cfg = {
    _id : "rs_a",
    members : [
        {_id : 0, host : "e1.acme.com:27018", priority : 1},
        {_id : 1, host : "e2.acme.com:27018", priority : 1},
        {_id : 2, host : "w1.acme.com:27018", priority : 0, hidden : true}
    ]
}

> rs.initiate(cfg)
```

Sharding Introduction

MongoDB supports an automated sharding/partitioning architecture, enabling horizontal scaling across multiple nodes. For applications that outgrow the resources of a single database server, MongoDB can convert to a sharded cluster, automatically managing failover and balancing of nodes, with few or no changes to the original application code.

This document explains MongoDB's auto-sharding approach to scalability in detail and provides an architectural overview of the various components that enable it.

Be sure to acquaint yourself with the current limitations.

MongoDB's Auto-Sharding

Sharding in a Nutshell

Sharding is the partitioning of data among multiple machines in an order-preserving manner. To take an example, let's imagine sharding a collection of users by their state of residence. In a simplistic view, if we designate three machines as our shard servers, the users might be divided up by machine as follows:

Machine 1	Machine 2	Machine 3
Alabama Arizona	Colorado Florida	Arkansas California
Indiana Kansas	Idaho Illinois	Georgia Hawaii
Maryland Michigan	Kentucky Maine	Minnesota Missouri
Montana Montana	Nebraska New Jersey	Ohio Pennsylvania
New Mexico North Dakota	Rhode Island South Dakota	Tennessee Utah
	Vermont West Virgina	Wisconsin Wyoming

Note that each machine stores multiple "chunks" of data, based on state of residence. MongoDB distributes these chunks evenly across all of the machines available.

The chunking mechanism would only kick in if the amount of data you were storing reached the threshold where sharding is advantageous.

Our application connects to the sharded cluster through a `mongos` process, which routes operations to the appropriate shard(s). In this way, the sharded MongoDB cluster looks like a single logical server to our application. If our `users` collection receives heavy writes, those writes are now distributed across three shard servers. Most queries continue to be efficient, as well, because they too are distributed. And since the documents are organized in an order-preserving manner, any operations specifying the state of residence will be routed only to those nodes containing that state.

Sharding is performed on a per-collection basis. Small collections need not be sharded. For instance, if we were building a service like Twitter, our collection of tweets would likely be several orders of magnitude larger than the next biggest collection. The size and throughput demands of such a collection would be prime for sharding, whereas smaller collections would still live on a single server. Non-sharded collections reside on just one shard.

Balancing

Balancing is necessary when the load on any one shard node grows out of proportion with the remaining nodes. In this situation, the data must be redistributed to equalize load across shards.

Failover

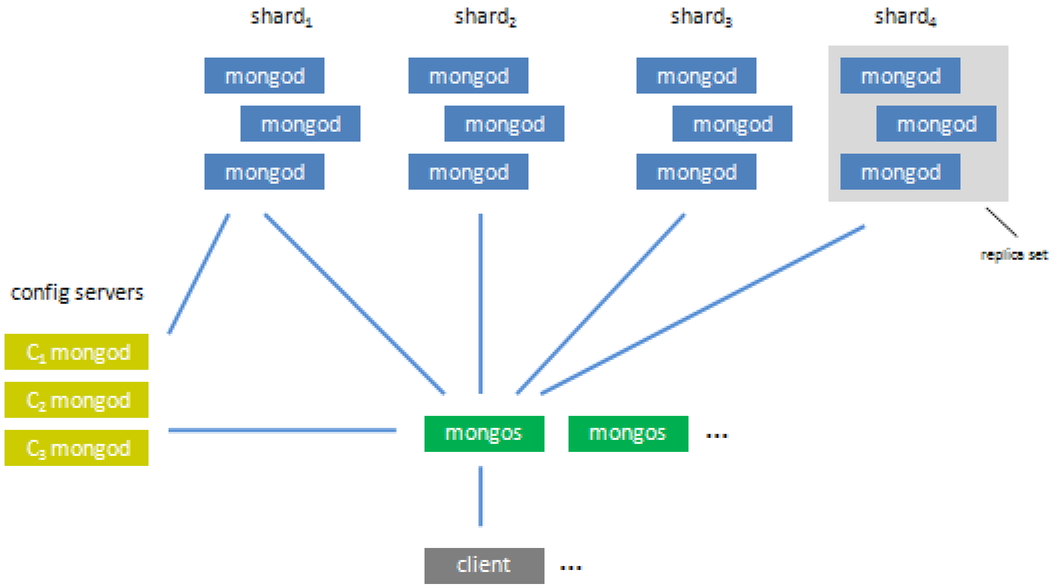
Failover is also quite important since proper system functioning requires that each logical shard be always online. In practice, this means that each shard consists of more than one machine in a configuration known as a replica set. A replica set is a set of *n* servers, typically two or three, each of which contains a replica of the entire data set for the given shard. One of the *n* servers in a replica set will always be primary. If the primary fails, the remaining replicas are capable of electing a new master.

Scaling Model

MongoDB's auto-sharding scaling model shares many similarities with Yahoo's PNUTS and Google's BigTable. Readers interested in detailed discussions of distributed databases using order-preserving partitioning are encouraged to look at the PNUTS and BigTable white papers.

Architectural Overview

A MongoDB shard cluster consists of two or more shards, one or more config servers, and any number of routing processes to which the application servers connect. Each of these components is described below in detail.



Shards

Each shard consists of one or more servers and stores data using `mongod` processes (`mongod` being the core MongoDB database process). In a production situation, each shard will consist of multiple servers to ensure availability and automated failover. The set of servers/`mongod` process within the shard comprise a replica set.

In MongoDB, *sharding is the tool for scaling a system, and replication is the tool for data safety, high availability, and disaster recovery*. The two work in tandem yet are orthogonal concepts in the design.

For testing, you can use sharding with a single `mongod` instance per shard. Production databases typically need redundancy, so they use replica sets.

Shard Keys

To partition a collection, we specify a shard key pattern. This pattern is similar to the key pattern used to define an index; it names one or more fields to define the key upon which we distribute data. Some example shard key patterns include the following:

```
{ state : 1 }
{ name : 1 }
{ _id : 1 }
{ lastname : 1, firstname : 1 }
{ tag : 1, timestamp : -1 }
```

MongoDB's sharding is order-preserving; adjacent data by shard key tend to be on the same server. The config database stores all the metadata indicating the location of data by range:

collection	minkey	maxkey	location
users	{ name : 'Miller' }	{ name : 'Nessman' }	shard$_2$
users	{ name : 'Nessman' }	{ name : 'Ogden' }	shard$_4$
...			

Chunks

A chunk is a contiguous range of data from a particular collection. Chunks are described as a triple of `collection`, `minKey`, and `maxKey`. Thus, the shard key K of a given document assigns that document to the chunk where minKey <= K < maxKey.

Chunks grow to a maximum size, usually 64MB. Once a chunk has reached that approximate size, the chunk *splits* into two new chunks. When a particular shard has excess data, chunks will then *migrate* to other shards in the system. The addition of a new shard will also influence the migration of chunks.

When choosing a shard key, the values must be of high enough cardinality (granular enough) that data can be broken into many chunks, and thus distribute-able. For instance, in the above example, where we are sharding on `name`, if a huge number of users had the same name, that could be problematic as all documents involving that name would be in a single undivided chunk. With names that typically does not happen and thus `name` is a reasonable choice as a shard key.

If it is possible that a single value within the shard key range might grow exceptionally large, it is best to use a compound shard key instead so that further discrimination of the values will be possible.

Config DB Processes

The config servers store the cluster's metadata, which includes basic information on each shard server and the chunks contained therein.

Chunk information is the main data stored by the config servers. Each config server has a complete copy of all chunk information. A two-phase commit is used to ensure the consistency of the configuration data among the config servers. Note that config server use their own replication model; they are not run in as a replica set.

If any of the config servers is down, the cluster's meta-data goes read only. However, even in such a failure state, the MongoDB cluster can still be read from and written to.

Routing Processes (mongos)

The `mongos` process can be thought of as a routing and coordination process that makes the various components of the cluster look like a single system. When receiving client requests, the `mongos` process routes the request to the appropriate server(s) and merges any results to be sent back to the client.

`mongos` processes have no persistent state; rather, they pull their state from the config server on startup. Any changes that occur on the the config servers are propagated to each `mongos` process.

`mongos` processes can run on any server desired. They may be run on the shard servers themselves, but are lightweight enough to exist on each application server. There are no limits on the number of `mongos` processes that can be run simultaneously since these processes do not coordinate between one another.

Operation Types

Operations on a sharded system fall into one of two categories: *global* and *targeted*.

For targeted operations, `mongos` communicates with a very small number of shards -- often a single shard. Such targeted operations are quite efficient.

Global operations involve the `mongos` process reaching out to all (or most) shards in the system.

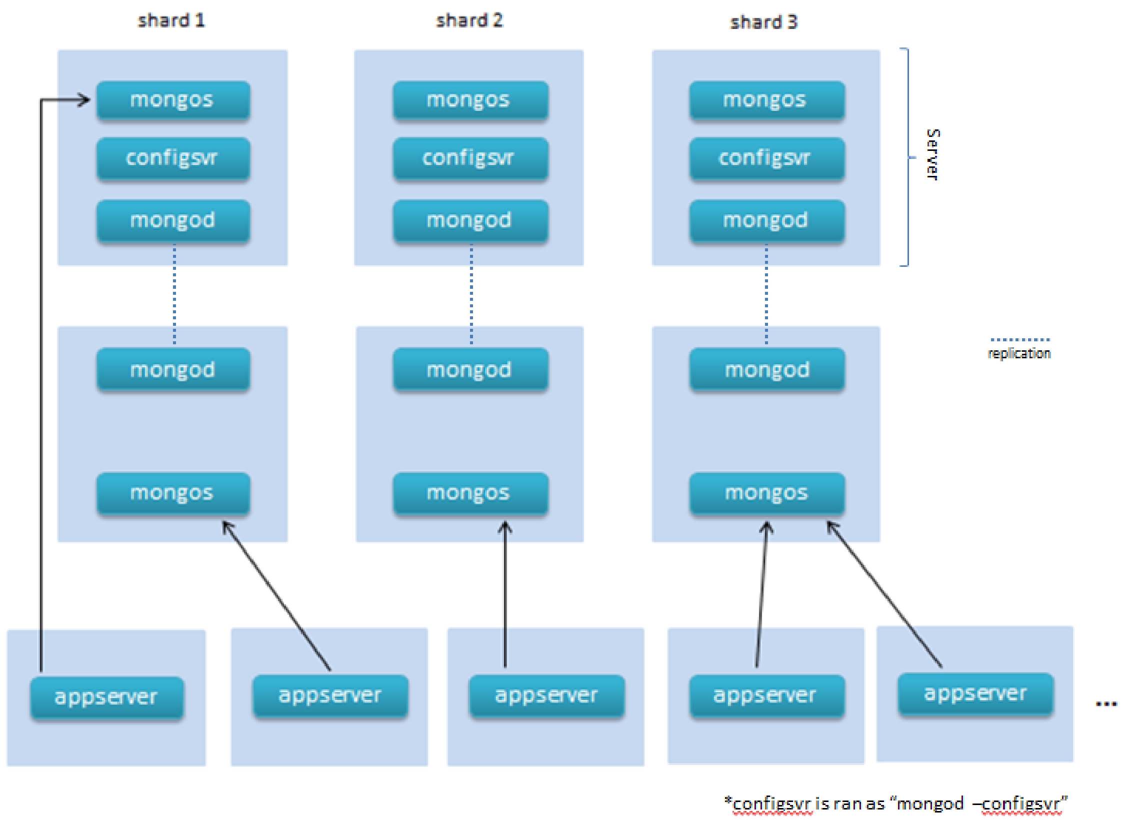
The following table shows various operations and their type. For the examples below, assume a shard key of { `x` : 1 }.

Operation	Type	Comments
db.foo.find({ x : 300 })	Targeted	Queries a single shard.
db.foo.find({ x : 300, age : 40 })	Targeted	Queries a single shard.
db.foo.find({ age : 40 })	Global	Queries all shards.
db.foo.find()	Global	sequential
db.foo.find(...).count()	Variable	Same as the corresponding find() operation
db.foo.find(...).sort({ age : 1 })	Global	parallel
db.foo.find(...).sort({ x : 1 })	Global	sequential
db.foo.count()	Global	parallel
db.foo.insert(<object>)	Targeted	
db.foo.update({ x : 100 }, <object>) db.foo.remove({ x : 100 })	Targeted	
db.foo.update({ age : 40 }, <object>, false, true) db.foo.remove({ age : 40 })	Global	Updates without the shard key must be multi-updates

db.getLastError()		
db.foo.ensureIndex(...)	Global	

Server Layout

Machines may be organized in a variety of fashions. For instance, it's possible to have separate machines for each config server process, `mongos` process, and `mongod` process. However, this can be overkill since the load is almost certainly low on the config servers. Here is an example where some sharing of physical machines is used to lay out a cluster. The outer boxes are machines (or VMs) and the inner boxes are processes.



In the picture about a given connection to the database simply connects to a random mongos. mongos is generally very fast so perfect balancing of those connections is not essential. Additionally the implementation of a driver could be intelligent about balancing these connections (but most are not at the time of this writing).

Yet more configurations are imaginable, especially when it comes to `mongos`. Alternatively, as suggested earlier, the `mongos` processes can exists on each application server. There is some potential benefit to this configuration, as the communications between app server and `mongos` then can occur over the localhost interface.

Exactly three config server processes are used in almost all sharded mongo clusters. This provides sufficient data safety; more instances would increase coordination cost among the config servers.

Next : Configuring Sharding

Configuring Sharding

This document describes the steps involved in setting up a basic sharding cluster. A sharding cluster has three components:

1. One to 1000 shards. Shards are partitions of data. Each shard consists of one or more `mongod` processes which store the data for that shard. When multiple `mongod`'s are in a single shard, they are each storing the same data – that is, they are replicating to each other.
2. Either one or three config server processes. For production systems use three.
3. One or more `mongos` routing processes.

For testing purposes, it's possible to start all the required processes on a single server, whereas in a production situation, a number of server configurations are possible.

Once the shards (`mongod`'s), config servers, and `mongos` processes are running, configuration is simply a matter of issuing a series of commands to establish the various shards as being part of the cluster. Once the cluster has been established, you can begin sharding individual collections.

This document is fairly detailed; for a terse, code-only explanation, see the sample shard configuration. If you'd like a quick script to set up a test cluster on a single machine, we have a python sharding script that can do the trick.

Sharding Components

First, start the individual shards (`mongod`'s), config servers, and `mongos` processes.

Shard Servers

A shard server consists of a `mongod` process or a replica set of mongod processes. For production, use a replica set for each shard for data safety and automatic failover. To get started with a simple test, we can run a single `mongod` process per shard, as a test configuration doesn't demand automated failover.

Config Servers

Run a `mongod --configsvr` process for each config server. If you're only testing, you can use only one config server. For production, use three.

Note: Replicating data to each config server is managed by the router (mongos); they have a synchronous replication protocol optimized for three machines, if you were wondering why that number. Do not run any of the config servers with --replSet; replication between them is automatic.

Note: As the metadata of a MongoDB cluster is fairly small, it is possible to run the config server processes on boxes also used for other purposes.

`mongos` Router

Run `mongos` on the servers of your choice. Specify the --configdb parameter to indicate location of the config database(s). Note: use dns names, *not ip addresses*, for the --configdb parameter's value. Otherwise moving config servers later is difficult.

Note that each `mongos` will read from the first config server in the list provided. If you're running config servers across more than one data center, you should put the closest config servers early in the list.

Configuring the Shard Cluster

Once the shard components are running, issue the sharding commands. You may want to automate or record your steps below in a .js file for replay in the shell when needed.

Start by connecting to one of the `mongos` processes, and then switch to the `admin` database before issuing any commands.

The mongos will route commands to the right machine(s) in the cluster and, if commands change metadata, the mongos will update that on the config servers. So, regardless of the number of `mongos` processes you've launched, you'll only need run these commands on one of those processes.

You can connect to the admin database via `mongos` like so:

```
./mongo <mongos-hostname>:<mongos-port>/admin
> db
admin
```

Adding shards

Each shard can consist of more than one server (see replica sets); however, for testing, only a single server with one `mongod` instance need be used.

You must explicitly add each shard to the cluster's configuration using the `addshard` command:

```
> db.runCommand( { addshard : "<serverhostname>[:<port>]" } );
{"ok" : 1 , "added" : ...}
```

Run this command once for each shard in the cluster.

If the individual shards consist of replica sets, they can be added by specifying *replicaSetName* /<serverhostname>[:port][,serverhostname2[:port],...], where at least one server in the replica set is given.

```
> db.runCommand( { addshard : "foo/<serverhostname>[:<port>]" } );
{"ok" : 1 , "added" : "foo"}
```

Any databases and collections that existed already in the mongod/replica set will be incorporated to the cluster. The databases will have as the "primary" host that mongod/replica set and the collections will not be sharded (but you can do so later by issuing a shardCollection command).

Optional Parameters

name
Each shard has a name, which can be specified using the `name` option. If no name is given, one will be assigned automatically.

maxSize
The `addshard` command accepts an optional `maxSize` parameter. This parameter lets you tell the system a maximum amount of disk space in megabytes to use on the specified shard. If unspecified, the system will use the entire disk. maxSize is useful when you have machines with different disk capacities or when you want to prevent storage of too much data on a particular shard.

As an example:

```
> db.runCommand( { addshard : "sf103", maxSize:100000/*MB*/ } );
```

Listing shards

To see current set of configured shards, run the `listshards` command:

```
> db.runCommand( { listshards : 1 } );
```

This way, you can verify that all the shard have been committed to the system.

Removing a shard

See the removeshard command.

Enabling Sharding on a Database

> In versions prior to v2.0, dropping a sharded database causes issues - see SERVER-2253 for workaround.

Once you've added one or more shards, you can enable sharding on a database. Unless enabled, all data in the database will be stored on the same shard. After enabling you then need to run shardCollection on the relevant collections (i.e., the big ones).

```
> db.runCommand( { enablesharding : "<dbname>" } );
```

Once enabled, `mongos` will place new collections on the primary shard for that database. Existing collections within the database will stay on the original shard. To enable partitioning of data, we have to shard an individual collection.

Sharding a Collection

> When sharding a collection, "pre-splitting", that is, setting a seed set of key ranges, is recommended. Without a seed set of ranges, sharding works, however the system must learn the key distribution and this will take some time; during this time performance is not as high. The presplits do not have to be particularly accurate; the system will adapt to the actual key distribution of the data regardless.

Use the `shardcollection` command to shard a collection. When you shard a collection, you must specify the shard key. If there is data in the collection, mongo will require an index to be created upfront (it speeds up the chunking process); otherwise, an index will be automatically created for you.

```
> db.runCommand( { shardcollection : "<namespace>",
                   key : <shardkeypatternobject> });
```

Running the "shardcollection" command will mark the collection as sharded with a specific key. Once called, there is currently no way to disable sharding or change the shard key, even if all the data is still contained within the same shard. It is assumed that the data may already be spread around the shards. If you need to "unshard" a collection, drop it (of course making a backup of data if needed), and recreate the collection (loading the backup data).

For example, let's assume we want to shard a GridFS `chunks` collection stored in the `test` database. We'd want to shard on the `files_id` key, so we'd invoke the `shardcollection` command like so:

```
> db.runCommand( { shardcollection : "test.fs.chunks", key : { files_id : 1 } } )
{ "collectionsharded" : "mydb.fs.chunks", "ok" : 1 }
```

You can use the {`unique: true`} option to ensure that the underlying index enforces uniqueness so long as the unique index is a prefix of the shard key. (note: prior to version 2.0 this worked only if the collection is empty).

```
db.runCommand( { shardcollection : "test.users" , key : { email : 1 } , unique : true } );
```

If the "unique: true" option is **not** used, the shard key does not have to be unique.

```
db.runCommand( { shardcollection : "test.products" , key : { category : 1, _id : 1 } } );
```

You can shard on multiple fields if you are using a compound index.

In the end, picking the right shard key for your needs is extremely important for successful sharding. Choosing a Shard Key.

Examples

- Sample configuration session
- The following example shows how to run a simple shard setup on a single machine for testing purposes: Sharding JS Test.

See Also

- Sharding Administration
- Notes on TCP Port Numbers

A Sample Configuration Session

The following example uses two shards (each with a single mongod process), one config db, and one *mongos* process, all running on a single test server. In addition to the script below, a python script for starting and configuring shard components on a single machine is available.

Creating the Shards

First, start up a couple *mongod*s to be your shards.

```
$ mkdir /data/db/a /data/db/b
$ ./mongod --shardsvr --dbpath /data/db/a --port 10000 > /tmp/sharda.log &
$ cat /tmp/sharda.log
$ ./mongod --shardsvr --dbpath /data/db/b --port 10001 > /tmp/shardb.log &
$ cat /tmp/shardb.log
```

Now you need a configuration server and *mongos*:

```
$ mkdir /data/db/config
$ ./mongod --configsvr --dbpath /data/db/config --port 20000 > /tmp/configdb.log &
$ cat /tmp/configdb.log
$ ./mongos --configdb localhost:20000 > /tmp/mongos.log &
$ cat /tmp/mongos.log
```

mongos does not require a data directory, it gets its information from the config server.

> In a real production setup, mongod's, mongos's and configs would live on different machines. The use of hostnames or IP addresses is mandatory in that case. 'localhost' appearance here is merely illustrative – but fully functional – and should be confined to single-machine, testing scenarios only.

You can toy with sharding by using a small `--chunkSize`, e.g. 1MB. This is more satisfying when you're playing around, as you won't have to insert 64MB of documents before you start seeing them moving around. It should not be used in production.

```
$ ./mongos --configdb localhost:20000 --chunkSize 1 > /tmp/mongos.log &
```

Setting up the Cluster

We need to run a few commands on the shell to hook everything up. Start the shell, connecting to the *mongos* process (at localhost:27017 if you followed the steps above).

To set up our cluster, we'll add the two shards (*a* and *b*).

```
$ ./mongo
MongoDB shell version: 1.6.0
connecting to: test
> use admin
switched to db admin
> db.runCommand( { addshard : "localhost:10000" } )
{ "shardadded" : "shard0000", "ok" : 1 }
> db.runCommand( { addshard : "localhost:10001" } )
{ "shardadded" : "shard0001", "ok" : 1 }
```

Now you need to tell the database that you want to spread out your data at a database and collection level. You have to give the collection a key (or keys) to partition by.
This is similar to creating an index on a collection.

```
> db.runCommand( { enablesharding : "test" } )
{"ok" : 1}
> db.runCommand( { shardcollection : "test.people", key : {name : 1} } )
{"ok" : 1}
```

Administration

To see what's going on in the cluster, use the *config* database.

```
> use config
switched to db config
> show collections
chunks
databases
lockpings
locks
mongos
settings
shards
system.indexes
version
```

These collections contain all of the sharding configuration information.

Choosing a Shard Key

It is important to choose the right shard key for a collection. If the collection is gigantic it is difficult to change the key later. When in doubt please ask for suggestions in the support forums or IRC.

- Cardinality
- Write scaling
- Query isolation
- Sorting
- Reliability
- Index optimization
- GridFS

The discussion below considers a structured event logging system. Documents in the events collection have the form:

```
{
   mon_node : "ny153.example.com" ,
   application : "apache" ,
   time : "2011-01-02T21:21:56.249Z" ,
   level : "ERROR" ,
   msg : "something is broken"
}
```

Cardinality

All data in a sharded collection is broken into *chunks*. A chunk is a range on the shard key. It is very important that the shard key is granular enough that data can be partitioned among many machines.

Using the logging example above, if you chose

```
{mon_node:1}
```

as the shard key, then all data for a single mon_node is in one chunk, and thus on one shard. One can easily imagine having a lot of data for a mon_node and wanting to split it among shards. In addition, chunks are approximately 64MB in size. If the chunk is unsplittable because it represents a single mon_node value, the balancer migrating that chunk from server to server would be slow.

If one were to shard on the key

```
{mon_node:1,time:1}
```

we can then split data for a single mon_node down to the millisecond. No chunk will ever be too large. (Note there will not be a chunk for every mon_node,time value – rather on ranges such as

```
{mon_node:"nyc_app_24",time:"2011-01-01"}..{mon_node:"nyc_app_27",time:"2011-07-12"}
```

or if a lot of data for one mon_node:

```
{mon_node:"nyc_app_24",time:"2011-01-01"}..{mon_node:"nyc_app_24",time:"2011-01-07"}
```

Keeping chunks to a reasonable size is very important so data can be balanced and moving chunks isn't too expensive.

Write scaling

One of the primary reasons for using sharding is to distribute writes. To do this, its important that writes hit as many chunks as possible.

Again using the example above, choosing

```
{ time : 1 }
```

as the shard key will cause all writes to go to the newest chunk. If the shard key were

```
{mon_node:1,application:1,time:1}
```

then each mon_node,application pair can potentially be written to different shards. Thus there were 100 mon_node,application pairs, and 10 shards, then each shard would get 1/10th the writes.

Note that because the most significant part of an ObjectId is time-based, using ObjectId as the shard key has the same issue as using time directly.

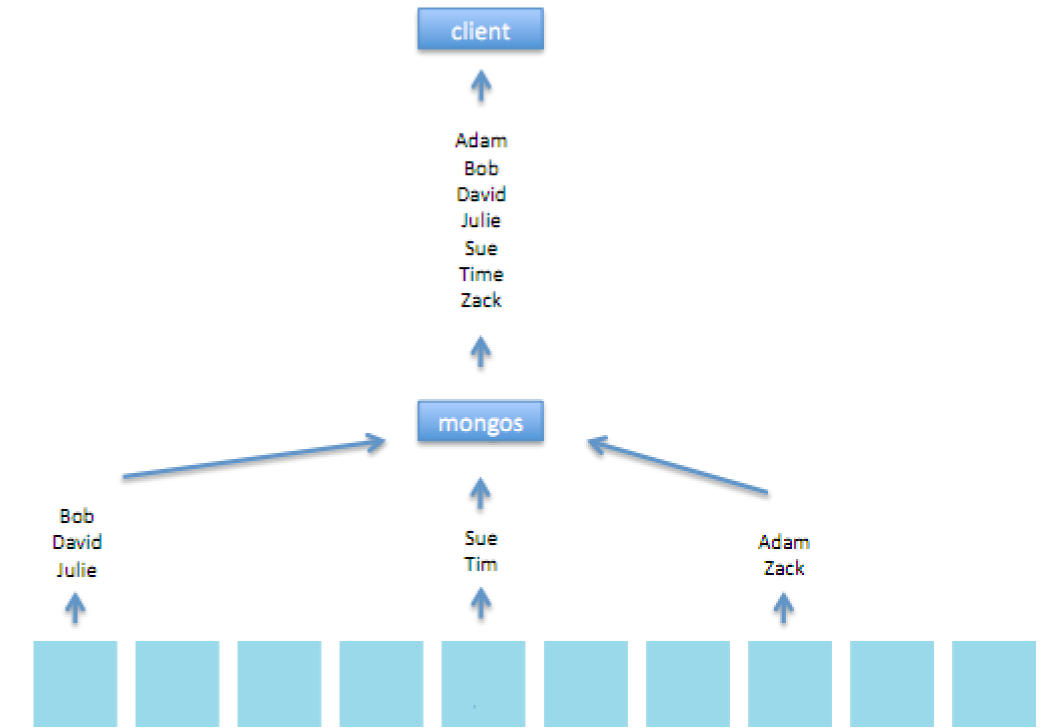
Query isolation

Another key consideration is how many shards any query has to hit. Ideally a query would go directly from `mongos` to the shard (`mongod`) that owns the document requested. Thus, if you know that most common queries include a condition on *mon_node*, then starting the shard key with mon_node would be efficient.

All queries work regardless of the shard key, but if `mongos` cannot determine which shard that owns the data, it will send the operation to all shards in parallel. This will increase the response time latency and increase the volume of network traffic and load.

Sorting

When a query includes a sort criteria, the query is sent to the same shards that would have received the equivalent query without the sort expression. Each shard queried performs a sort on its subset of the data locally (potentially utilizing an appropriate index locally if one is available). `mongos` merges the ordered results coming from each shard and returns the merged stream to the client. Thus the `mongos` process does very little work and requires little RAM for request of this nature.



Reliability

One important aspect of sharding is how much of the system will be affected in case an entire shard becomes unavailable (even though it is usually a reliable replica set).

Say you have a twitter-like system, with comment entries like:

```
{
   _id: ObjectId("4d084f78a4c8707815a601d7"),
   user_id : 42 ,
   time : "2011-01-02T21:21:56.249Z" ,
   comment : "I am happily using MongoDB",
}
```

One might use *id* or *user_id* as shard key. *id* will give you better granularity and spread, but in case a shard goes down, it will impact almost all your users (some data missing). If you use *user_id* instead, a lower percentage of users will be impacted (say 20% for a 5 shard setup), although these users will not see any of their data.

Index optimization

As mentioned in other sections about indexing, it is usually much better performance-wise if only a portion of the index is read/updated, so that this "active" portion can stay in RAM most of the time. Most shard keys described above result in an even spread over the shards but also within each mongod's index. Instead, it can be beneficial to factor in some kind of timestamp at the beginning of the shard key in order to limit the portion of the index that gets hit.
Say you have a photo storage system, with photo entries like:

```
{
   _id: ObjectId("4d084f78a4c8707815a601d7"),
   user_id : 42 ,
   title: "sunset at the beach",
   upload_time : "2011-01-02T21:21:56.249Z" ,
   data: ...,
}
```

You could instead build a custom *id* that includes the month of the upload time, and a unique identifier (ObjectId, MD5 of data, etc). Your entry now looks like:

```
{
   _id: "2011-01_4d084f78a4c8707815a601d7",
   user_id : 42 ,
   title: "sunset at the beach",
   upload_time : "2011-01-02T21:21:56.249Z" ,
   data: ...,
}
```

This custom id would be the key used for sharding, and also the id used to retrieve a document. It gives both a good spread across all servers, and it reduces the portion of the index hit on most queries.

Further notes:

- at the beginning of each month, only one shard would be hit for some time until the balancer starts splitting buckets. To avoid this potential slowdown and migration of data, it is advised to create ranges ahead of time (e.g. 5 or more ranges for each month if you have 5 servers).
- As a further improvement, you can incorporate the user id into the photo id. This way it will keep all documents belonging to same user within the same shard. For example: "2011-01_42_4d084f78a4c8707815a601d7"

GridFS

There are different ways that GridFS can be sharded, depending on the need. One common way to shard, based on pre-existing indexes, is:

- "files" collection is not sharded. All file records will live in 1 shard. It is highly recommended to make that shard very resilient (at least 3 node replica set)
- "chunks" collection should be sharded using the index "files_id: 1". The existing "files_id, n" index created by the drivers cannot be used to shard on "files_id" (this is a sharding limitation that will be fixed), so you have to create a separate index on just "files_id". The reason to use "files_id" is so that all chunks of a given file live on the same shard, which is safer and allows "filemd5" command to work (required by certain drivers).
  Run the commands:

  ```
  > db.fs.chunks.ensureIndex({files_id: 1});
  > db.runCommand({ shardcollection : "test.fs.chunks", key : { files_id : 1 }})
  { "collectionsharded" : "test.fs.chunks", "ok" : 1 }
  ```

As the default `files_id` is an ObjectId, `files_id` will be ascending and thus all GridFS chunks will be sent to a single sharding chunk. If your write load is too high for a single server to handle, you may want to shard on a different key or use a different value for the `_id` in the files collection.

Changing a Shard Key

There is no automatic support for changing the shard key for a collection. In addition this would fundamentally be a very expensive operation most of the data in the cluster for the collection likely moves from machine to machine.

Thus it is very important to choose the right shard key up front.

If you do need to change a shard key, an export and import is likely the best solution. Create a new pre-sharded collection, and then import the exported data to it. If desired use a dedicated mongos for the export and the import.

https://jira.mongodb.org/browse/SERVER-4000

removeshard command

Removing a Shard

The `removeshard` command will remove a shard from an existing cluster. It has two phases which are described below.

Starting

Before a shard can be removed, we have to make sure that all the chunks and databases that once lived there were relocated to other shards. The 'removeshard' command takes care of "draining" the chunks out of a shard for us. To start the shard removal, you can issue the command

> ⚠ The balancer must be running for this process to work. It does all the work of migrating chunks and removing the shard when that is done.

First run

```
> db.runCommand( { removeshard : "shard0000" } );
{ msg : "draining started successfully" , state: "started" , shard :"shard0000" , ok : 1
```

That will put the shard in "draining mode". Its chunks are going to be moved away slowly over time, so not to cause any disturbance to a running system. The command will return right away but the draining task will continue on the background. If you issue the command again during it, you'll get a progress report instead:

```
> db.runCommand( { removeshard : "shard0000" } );
{ msg: "draining ongoing" ,  state: "ongoing" , remaining : { chunks :23 , dbs : 1 }, ok : 1 }
```

Whereas the chunks will be removed automatically from that shard, the databases hosted there -- the 'dbs' counter attribute in the above output -- will need to be moved manually. (This has to do with a current limitation that will go away eventually). If you need to figure out which database the removeshard output refers to, you can use the printShardingStatus command. It will tell you what is the "primary" shard for each non-partitioned database. In versions 2.1.0 and higher, this is unnecessary because it will list the dbs in a "dbsToMove" field. You need to remove these with the following command:

Move the primary somewhere else

This is **only** necessary to do if there are databases listed by printShardingStatus whose primary shard is the shard which has been drained. MovePrimary should only be run after the shard has finished draining (i.e. all chunks have been removed, the status may not update), and only for those databases with primaries on the drained shard.

```
> db.runCommand( { movePrimary : "test", to : "shard0001" } );
{ "primary" : "shard0001", "ok" : 1 }
```

When the shard is empty, you could issue the 'removeshard' command again and that will clean up all metadata information:

> Important: You should only issue the movePrimary command after draining has completed - in general you should never use movePrimary if you still have undrained sharded collection data on the primary shard.

Second run

```
> db.runCommand( { removeshard : "shard0000" } );
{ msg: "remove shard completed succesfully" , stage: "completed", host: "shard0000", ok : 1 }
```

After the 'removeshard' command reported being done with that shard, you can take that shard down.

Upgrading from a Non-Sharded System

A `mongod` process can become part of a sharded cluster without any change to that process or downtime. If you haven't done so yet, feel free to have a look at the Sharding Introduction to familiarize yourself with the components of a sharded cluster and at the Sample Configuration Sesssion to get to know the basic commands involved.

Adding the `mongod` process to a cluster

If you haven't changed the `mongod` default port, it would be using port 27017. You care about this now because a mongo shell will always try to connect to it by default. But in a sharded environment, **you want your shell to connect to a mongos instead**.

If the port 27017 is taken by a `mongod` process, you'd need to bring up the `mongos` in a different port. Assuming that port is 30000 you can connect your shell to it by issuing:

```
$ mongo <mongos-host-address>:30000/admin
```

We're switching directly to the `admin` database on the `mongos` process. That's where we will be able to issue the following command

```
MongoDB shell version: 1.6.0
connecting to: <mongos-address>:30000/admin
> db.runCommand( { addshard : "192.168.25.203:27017" } )
> { "shardAdded" : "shard0000", "ok" : 1 }
```

The host address and port you see on the command are the original `mongod`'s. All the databases of that process were added to the cluster and are accessible now through `mongos`.

```
> db.runCommand( { listdatabases : 1 } )
{
        "databases" : [
                {
                        "name" : "mydb"
                        ...
                        "shards" : {
                                "shard0000" : <size-in-shard00000>
                        }
                },
                ...
```

Note that that doesn't mean that the database or any of its collections is sharded. They haven't moved (see next). All we did so far is to make them visible within the cluster environment.

You should stop accessing the former stand-alone `mongod` directly and should have all the clients connect to a `mongos` process, just as we've been doing here.

Sharding a collection

All the databases of your `mongod`-process-turned-shard can be chunked and balanced among the cluster's shards. The commands and examples to do so are listed at the
Configuring Sharding page. Note that a chunk size defaults to 64MB, so if you want to change that – for testing purposes, say – you would do so by starting the `mongos` process with the additional --chunkSize parameter.

Difference between upgrading and starting anew

You should pay attention to the host addresses and ports when upgrading.
Again, if you haven't changed the default ports of your `mongod` process, it would be listening on 27017, which is the port that `mongos` would try to bind by default, too.

Sharding Administration

Here we present a list of useful commands for obtaining information about a sharding cluster.

To set up a sharding cluster, see the docs on sharding configuration.

- Identifying a Shard Cluster
- List Existing Shards
- List Which Databases are Sharded
- View Sharding Details
- Balancing
- Balancer window
- Chunk Size Considerations

Identifying a Shard Cluster

```
// Test if we're speaking to a mongos process or straight to a mongod process.
// If connected to a mongod this will return a "no such cmd" message.
> db.runCommand({ isdbgrid : 1});

// If connected to mongos, this command returns { "ismaster": true,
// "msg": "isdbgrid", "maxBsonObjectSize": XXX, "ok": 1 }
> db.runCommand({ismaster:1});
```

List Existing Shards

```
> db.runCommand({ listShards : 1});
{"servers" :
 [{"_id" :  ObjectId( "4a9d40c981ba1487ccfaa634")  ,
   "host" : "localhost:10000"},
  {"_id" :  ObjectId( "4a9d40df81ba1487ccfaa635")  ,
   "host" : "localhost:10001"}
 ],
 "ok" : 1
}
```

List Which Databases are Sharded

Here we query the config database, albeit through `mongos`. The `getSisterDB` command is used to return the config database. This will list all databases in the cluster. Databases with `partitioned : true` have sharding enabled.

```
> config = db.getSisterDB("config")
> config.databases.find()
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "MyShardedDatabase", "partitioned" : true, "primary" : "localhost:30001" }
{ "_id" : "MyUnshardedDatabase", "partitioned" : false, "primary" : "localhost:30002" }
```

View Sharding Details

```
> use admin
> db.printShardingStatus();

// A very basic sharding configuration on localhost
sharding version: { "_id" : 1, "version" : 2 }
  shards:
      { "_id" : ObjectId("4bd9ae3e0a2e26420e556876"), "host" : "localhost:30001" }
      { "_id" : ObjectId("4bd9ae420a2e26420e556877"), "host" : "localhost:30002" }
      { "_id" : ObjectId("4bd9ae460a2e26420e556878"), "host" : "localhost:30003" }

  databases:
 { "name" : "admin", "partitioned" : false,
          "primary" : "localhost:20001",
          "_id" : ObjectId("4bd9add2c0302e394c6844b6") }
 my chunks

        { "name" : "foo", "partitioned" : true,
          "primary" : "localhost:30002",
          "sharded" : { "foo.foo" : { "key" : { "_id" : 1 }, "unique" : false } },
          "_id" : ObjectId("4bd9ae60c0302e394c6844b7") }
        my chunks
        foo.foo { "_id" : { $minKey : 1 } } -->> { "_id" : { $maxKey : 1 } }
                  on : localhost:30002 { "t" : 1272557259000, "i" : 1 }
```

Notice the output to the `printShardingStatus` command. First, we see the locations of the three shards comprising the cluster. Next, the various databases living on the cluster are displayed.

The first database shown is the admin database, which has not been partitioned. The **primary** field indicates the location of the database, which, in the case of the admin database, is on the config server running on port 20001.

The second database *is* partitioned, and it's easy to see the shard key and the location and ranges of chunks comprising the partition. Since there's no data in the `foo` database, only a single chunk exists. That single chunk includes the entire range of possible shard keys.

Balancing

The balancer is a background task that tries to keep the number of chunks even across all servers of the cluster. The activity of balancing is transparent to querying. Your application doesn't need to know or care that there is any data-moving activity ongoing.

To make that so, the balancer is careful about when and how much data it would transfer. Let's look at how much to transfer first. The unit of transfer is a chunk. With a steady state, the size of chunks should be roughly 64MB of data. This has shown to be the sweet spot of how much data to move at once. More than that, and the migration would take longer and the queries might perceive that in a wider difference in response times. Less than that, and the overhead of moving wouldn't pay off as highly.

Regarding when to transfer load, the balancer waits for a threshold of uneven chunk counts to occur before acting. In the field, having a difference of 8 chunks between the least and most loaded shards showed to be a good heuristic. (This is an arbitrary number, granted.) The concern here is not to incur overhead if -- exagerating to make a point -- there is a difference of one doc between shard A and shard B. It's just inefficient to monitor load differences at that fine of a grain.

Now, once the balancer "kicked in," it will redistribute chunks, one at a time -- in what we call rounds -- until that difference in chunks beween any two shards is down to 2 chunks.

A common source of questions is why a given collection is not being balanced. By far, the most probable cause is: it doesn't need to. If the chunk difference is small enough, redistributing chunks won't matter enough. The implicit assumption here is that you actually have a large enough collection and the overhead of the balancing machinery is little compared to the amount of data your app is handling. If you do the math, you'll find that you might not hit "balancing threshold" if you're doing an experiment on your laptop.

Another possibility is that the balancer is not making progress. The balancing task happens at an arbitrary mongos (query router) in your cluster. Since there can be several query routers, there is a mechanism they all use to decide which mongos will take the responsibility. The mongos acting as balancer takes a "lock" by inserting a document into the 'locks' collection of the config database. When a mongos is running the balancer the 'state' of that lock is 1 (taken).

To check the state of that lock

```
// connect to mongos
> use config
> db.locks.find( { _id : "balancer" } )
```

A typical output for that command would be

```
{ "_id" : "balancer", "process" : "guaruja:1292810611:1804289383", "state" : 1, "ts" : ObjectId(
"4d0f872630c42d1978be8a2e"), "when" : "Mon Dec 20 2010 11:41:10 GMT-0500 (EST)", "who" :
"guaruja:1292810611:1804289383:Balancer:846930886", "why" : "doing balance round" }
```

There are at least three points to note in that output. One, the ***state*** attribute is 1 (2 in v2.0), which means that lock is taken. We can assume the balancer is active. Two, that balancer has been running since Monday, December the 20th. That's what the attribute "when" tells us. And, the third thing, the balancer is running on a machine called "guaruja". The attribute "who" gives that away.

To check what the balancer is actually doing, we'd look at the mongos log on that machine. The balancer outputs rows to the log prefixed by "[Balancer]".

```
Mon Dec 20 11:53:00 [Balancer] chose [shard0001] to [shard0000] { _id: "test.foo-_id_52.0", lastmod:
Timestamp 2000|1, ns: "test.foo", min: { _id: 52.0 }, max: { _id: 105.0 }, shard: "shard0001" }
```

What this entry is saying is that the balancer decided to move the chunk _id:[52..105) from shard0001 to shard0000. Both mongod's log detailed entries of how that migrate is progressing.

In MongoDB versions 1.7 and earlier, the lock will always appear in db.locks.find even if the balancer is deactivated:

```
{ "_id" : "balancer", "process" : "guaruja:1292810611:1804289383", "state" : 1, "ts" :
ObjectId("4d0f872630c42d1978be8a2e"), "when" : "Mon Dec 20 2010 11:41:10 GMT-0500 (EST)"
, "who" : "guaruja:1292810611:1804289383:Balancer:846930886", "why" : "doing balance
round" }
```

To detect if the balancer is doing balancing rounds, you need to look for the Balancer entries in the log files. We will update this in a future release so that you can check if the balancer is active without looking in the logs. see jira 2632.
In newer versions of MongoDB the balancer does not create new lock entries when disabled. However, a pre-existing lock entry will continue to exist with its state set to 0.

If you want to pause the balancer temporarily for maintenance, you can by modifying the settings collection in the config db.

```
// connect to mongos
> use config
> db.settings.update( { _id: "balancer" }, { $set : { stopped: true } } , true );
```

As a result of that command, one should stop seeing "[Balancer]" entries in the mongos that was running the balancer. If, for curiosity, you're running that mongos in a more verbose level, you'd see an entry such as the following.

```
Mon Dec 20 11:57:35 "[Balancer]" skipping balancing round because balancing is disabled
```

You would just set `stopped` to false to turn on again.

For more information on chunk migration and commands, see: Moving Chunks

Balancer window

By default the balancer operates continuously, but it is also possible to set an active "window" of time each day for balancing chunks. This can often be useful when a cluster adds or removes only a few GB of data per day and traffic would be less disrupted if balancing occurred during low-traffic times. This time interval is also specified in the `config.settings` balancer document. For example, to balance chunks only from 9AM to 9PM:

```
// connect to mongos
> use config
> db.settings.update({ _id : "balancer" }, { $set : { activeWindow : { start : "9:00", stop : "21:00"
} } }, true )
```

To specify a time range spanning midnight, just swap the order of the times:

```
// connect to mongos
> use config
> db.settings.update({ _id : "balancer" }, { $set : { activeWindow : { start : "21:00", stop : "9:00"
} } }, true )
```

This enables balancing from 9PM to 9AM. If enabling the active window feature, it is important to check periodically that the amount of data inserted each day is not more than the balancer can handle in the limited window.

Currently only time of day is supported for automatic scheduling.

Chunk Size Considerations

mongoDB sharding is based on *range partitioning*. Chunks are split automatically when they reach a certain size threshold. The threshold varies, but the rule of thumb is, expect it to between a half and the maximum chunk size in the steady state. The default maximum chunk size is 64MB (sum of objects in the collection, not counting indices), though in older versions it was 200MB.

Chunk size has been intensely debated -- and much hacked. So let's understand the tradeoffs that that choice of size implies.

When you move data, there's some state resetting in mongos. Queries that used to hit a given shard for a migrated chunk, now need to hit a new shard. This state resetting isn't free, so we want to move chunks infrequently (pressure to move a lot of keys at once). However, the actual moving has a cost that's proportional to the number of keys you're moving (pressure to move few keys).

If you opt to change the default chunk size for a production site, you can do that by changing the value of the chunksize setting on the config database by running

```
> use config
> db.settings.save({_id:"chunksize", value:<new_chunk_size_in_mb>})
```

You will need to restart all mongoses for them to pick up the change. Note though that for an existing cluster, it may take some time for the collections to split to that size, if smaller than before, and currently autosplitting is only triggered if the collection gets new documents or updates.

For more information on chunk splitting and commands, see: Splitting Shard Chunks

Backing Up Sharded Cluster

See also Backups and Import Export Tools for more information on backups, particularly on the step of backing up each individual server. If you are on EC2 you should look at EC2 Backup & Restore for how to take advantage of EBS snapshots.

mongos->mongodump (small clusters only)

> You do not need to turn off the balancer when running mongodump against mongos.

If you have a small sharded cluster, you can use mongodump, connected to a `mongos`, to dump the entire cluster's data, or a subset of it.

This is easy, but only works if one machine can dump all the data in a reasonable period of time. The default dump, when you don't specify a database or collection to dump, will contain both the sharding config metadata (the data on the config servers) and the actual database content.

The mongodump `--oplog` cannot be used when dumping from a `mongos`. This option is only available when running directly against a replica member.

Backing up via shards

> The balancer **must** be turned off during a backup of the cluster (when done directly from shards)

In MongoDB, documents move from shard to shard as the system continually rebalances the data and load. Thus care is needed to assure getting a good backup of a large cluster. The diagram at the bottom of this page illustrates a bit how data is distributed and some of the nuances of a cluster backup.

Snapshotting

The procedures outlined below do not result in a cluster-wide snapshot. Rather, each shard is backed up somewhat independently. Thus new documents written during the backup may or may not be captured in the image (data preceding the backup start time will be present). The unit of isolation (and the rest of the letters in "ACID") in MongoDB is a document, not the entire database. Documents can be large and quite rich, thus for many use cases this is often sufficient.

Large cluster backup (v2.0+)

- Turn off the balancer

```
// connect to a mongos (not a config server!)
> use config
> db.settings.update( { _id: "balancer" }, { $set : { stopped: true } } , true );
>
> // wait for any migrations that were in progress to finish
> // "state" field is zero if no migrations in progress
> while( db.locks.findOne({_id: "balancer"}).state ) { print("waiting..."); sleep(1000); }
```

- `mongodump` the config db from one of the config servers to back up the cluster's metadata (either through mongos or by direct connection to the config server). You only need to back up one config server, as they all have replicas of the same information.

- Backup each shard. Use the standard practices for backing up a replica set (e.g. fsync+lock then snapshot, or mongodump). Shards can be backed up one at a time or in parallel.

- Turn the balancer back on

```
>use config
>db.settings.update( { _id: "balancer" }, { $set : { stopped: false } } , true );
```

Large cluster backup (v1.8 and earlier)

- Turn off the balancer

```
// connect to mongos (not a config server!)
> use config
> db.settings.update( { _id: "balancer" }, { $set : { stopped: true } } , true );
>
> // wait for any migrations that were in progress to finish
> // "state" field is zero if no migrations in progress
> while( db.locks.findOne({_id: "balancer"}).state ) { print("waiting..."); sleep(1000); }
```

v1.6: in this older release you must check the config.changelog collection rather than config.locks to see if migrates are still in-flight.

- stop one (and only one) config server (of your three config servers). This will make the configuration database read only. The db cluster is still fully readable and writable. Do not lock+fsync config servers; that can would block write operations on the cluster.

- Backup datafiles from the the stopped config server.

- Backup each shard. Use the standard practices to backup a replica sets (e.g. fsync+lock then snapshot, or mongodump). Shards can be backed up one at a time or in parallel.

- Restart the config server that was stopped.

- Turn the balancer back on

```
>use config
>db.settings.update( { _id: "balancer" }, { $set : { stopped: false } } , true );
```

Restoring the entire cluster

- Stop all processes. No server should be running mongod, mongos, or a config server mongod processes.
- Restore data files for each server in each shard and also for each config server. Normally each shard is comprised of a replica set. You must restore all the members of the replica set, or use the other standard approaches for restoring a replica set from backup.
- If shard IPs/hosts are different, you have to manually update config.shards to use the new servers:
    - 1. Start up the three config servers
      2. Load/restore the config database to each config server
      3. Start one mongos instance for the rest
      4. Update the config database collection named "shards" to reflect the new shards' ip addresses; the contents of config.shards are a replica set seed list, so use the replica set name style "replName/seedAddress1,..."
- Restart the mongos instances
- Restart the mongod instances (the replica set members)
- Connect to a mongos from the shell. Run "printShardingStatus()" and "show collections" to make sure all shards are correctly seen.

Restoring a single shard

Clusters are designed to be restored as a whole. Since the last backup, data (chunks) may have moved from shard to shard, making restoration of a single shard difficult.

That said, we can with sufficient work restore a single shard from backup.

- Restore the shard which was lost
- Chunks that were migrated away from this shard are fine. Those documents do not need to be deleted from this shard because they are automatically filtered out from queries by mongos . (See diagram below.)
- Chunks that were migrated to this shard since the last backup are now empty. Those documents must be recovered manually from the other shard backups. You can see a changelog of chunk migrations in the config.changelog collection.

For a "mature" cluster that has been running for a while, if desired we can mitigate the above scenario by:

- keeping the auto-balancer off for significant time periods. If the shards are well balanced this is not a problem. If you do this, occasionally check the balance between the shards via either MMS or the mongo shell.
- if there has been no migrations since the last backup, we can restore the lost shard in a straightforward fashion
- at set times, turn the balancer on let it run for a time to rebalance the cluster if any rebalancing is needed. afterwards, a new backup would be warranted.

The above mitigation is not necessarily recomended. With the balancer off the cluster could become unbalanced unless we are monitoring it. Keeping full cluster backups and using a good degree of replication (perhaps including slaveDelay replication) is often sufficient.

The following illustration shows a five shard cluster and what might happen on a single shard restoration from a backup.

	s1	s2	s3	s4	s5
at time of backup	[a,b) [b,c)	[e,j) [n,p)	[c,e)	[j,n)	[p,z)
just before failure	[a,b) [b,c)	[e,i) [n,p)	[c,e) [s,u)	[j,n) [u,z)	[p,s) [i,j)
s5 has been lost, and needs restoration from backup. the s5 backup image has the old s5 backup image contains [p,z) data. A full restore would be straightforward; however we decide to restore s5 only as that is faster and allows us to keep data from the other shards which is fresher than the backup.	[a,b) [b,c)	[e,i) [n,p)	[c,e) [s,u)	[j,n) [u,z)	
after restorating the s5 backup we have the [p,s) data. the older [s,z) data has also been restored to s5 but will be ignored. [i,j) is missing as it was not in the s5 backup image.	[a,b) [b,c)	[e,i) [n,p)	[c,e) [s,u)	[j,n) [u,z)	[p,s)
we manually find and restore the [i,j) data, which in this case was present in the s2 backup.	[a,b) [b,c)	[e,i) [n,p)	[c,e) [s,u)	[j,n) [u,z)	[p,s) [i,j)

Other Suggestions

Note that unsharded collections are always on shard a single shard (the first shard, or the one marked as the primary shard in the databases collection in the config db) and do not move. Thus you might want to back up that shard more frequently to have fresher bacukps of those small collections.

See Also

- Single server backups documentation

Sharding and Failover

A properly-configured MongoDB shard cluster will have no single point of failure.

This document describes the various potential failure scenarios of components within a shard cluster, and how failure is handled in each situation.

1. Failure of a `mongos` routing process.

One `mongos` routing process will be run on each application server, and that server will communicate to the cluster exclusively through the `mongos` process. `mongos` process aren't persistent; rather, they gather all necessary config data on startup from the config server.

This means that the failure of any one application server will have no effect on the shard cluster as a whole, and all other application servers will continue to function normally. Recovery is simply a matter starting up a new app server and `mongos` process.

2. Failure of a single `mongod` server within a shard.

Each shard will consist of a group of *n* servers in a configuration known as a replica set. If any one server in the replica set fails, read and write operations on the shard are still permitted. What's more, no data need be lost on the failure of a server because the replica allows an option on write that forces replication of the write before returning. This is similar to setting *W* to 2 on Amazon's Dynamo.

Replica sets have been available since MongoDB v1.6. Read more about replica set internals.

3. Failure of all `mongod` servers comprising a shard.

If all replicas within a shard are down, the data within that shard will be unavailable. However, operations that can be resolved at other shards will continue to work properly. See the documentation on global and targeted operations to see why this is so. One caveat - authentication data is stored on the primary shard, so new authenticated connections to a sharded collection require the primary shard be online.

If the shard is configured as a replica set, with at least one member of the set in another data center, then an outage of an entire shard is extremely unlikely. This will be the recommended configuration for maximum redundancy.

4. Failure of a config server.

A production shard cluster will have three config server processes, each existing on a separate machine. Writes to config servers use a two-phase commit to ensure an atomic and replicated transaction of the shard cluster's metadata.

On the failure of any one config server, the system's metadata becomes read-only. The system will continue to function, but chunks will be unable to split within a single shard or migrate across shards. For most use cases, this will present few problems, since changes to the chunk metadata will be infrequent.

That said, it will be important that the down config server be restored in a reasonable time period (say, a day) so that shards do not become unbalanced due to lack of migrates (again, for many production situations, this may not be an urgent matter).

Sharding Limits

Security

Authentication mode will be available with sharding as of v2.0. See SERVER-921 for details.

Differences from Unsharded Configurations

- Prior to v2.0, sharding must be run in trusted security mode, without explicit security.
- Shard keys are immutable in the current version.
- All (non-multi)updates, upserts, and inserts must include the current shard key. This may cause issues for anyone using a mapping library since you don't have full control of updates.

$where

$where works with sharding. However, do not reference the db object from the $where function (one normally does not do this anyway).

db.eval

db.eval() may not be used with sharded collections. However, you may use db.eval() if the evaluation function accesses unsharded collections within your database. Use map/reduce in sharded environments.

group

Currently, one must use MapReduce instead of group() on sharded collections.

getPrevError

getPrevError is unsupported for sharded databases, and may remain so in future releases (TBD). Let us know if this causes a problem for you.

Unique Indexes

For a sharded collection, you may (optionally) specify a unique constraint on the shard key. You also have the option to have other unique indices **if and only if** the shard key is a prefix of their attributes. In other words, MongoDB **does not** enforce uniqueness across shards. You may specify other secondary, non-unuique indexes (via a global operation), again, as long as no unique constraint is specified.

Scale Limits

Goal is support of systems of up to 1,000 shards. Testing so far has been limited to clusters with a modest number of shards (e.g., 100). More information will be reported here later on any scaling limitations which are encountered.

There is no hard-coded limit to the size of a collection -- but keep in mind the last paragraph. You can create a sharded collection and go about adding data for as long as you add the corresponding number of shards that your workload requires. And, of course, as long as your queries are targeted enough (more about that in a bit).

Query speed

Queries involving the shard key should be quite fast and comparable to the behavior of the query in an unsharded environment.

Queries not involving the shard key use a scatter/gather method which sends the query to all shards. This is fairly efficient if one has 10 shards, but would be fairly inefficient on 1000 shards (although still ok for infrequent queries).

Sharding an existing collection

It is possible to shard an existing collection, but there are some limitations. Put differently, if you have an existing single node (or single replica set) and you want to upgrade that data to a sharded configuration, this is possible.

The current limitations are **size** and **time**.

1. **Size**: In v1.6 we put a cap on the maximum size of the original collection of 25GB (increased to 256GB in v1.8). This limit is going to be pushed up and may eventually disappear. If you are above that limit and you shard an existing collection it will work, but all of your data will start out in one chunk, making initial distribution slower. In practice, if your collection contains many large documents, this limit may be slightly higher (due to the statistical way in which split points are calculated). One workaround is to increase the default chunk size in db.config.settings to a higher value (say 512MB or 1GB), which will enable the initial split and some migration. Then the large chunks will be naturally split over time as data is inserted.

1. **Time**: When sharding an existing collection, please be aware that this process will take some time. This will happen in the background, so operations are not significantly impacted typically. However, it will take quite a while for the data to migrate/balance on a large collection. For example, on a system with ten shards, 90% of the data for the collection will need to transfer to elsewhere to attain balance. Note that only large collections rebalance. If the collection is small (less than say, 400MB) we recommend not bothering to shard it.

Steps to shard an existing collection

1. Start your 3 config servers required for a sharded cluster - if your cluster will have large numbers of mongoses or shards (more than 10 of either, for example) moderately powerful machines will be needed for the config servers.

2. Start a mongos process pointing at these config servers to perform cluster administration.

3. Using this mongos process, add the existing database server or replica set as a shard to the cluster, as well as any empty servers or shards you would like to add to the cluster (more info here).

> Once the shards have been added, any application mongoses can be started and the cluster can start handling application reads and writes.

4. Choose a shard key for all existing collections you would like to shard, and create an index in the background over the shard key fields (more info here). Depending on the size of the collection and the index, this can take a long time. Note that background index creation becomes foreground index creation on secondaries, which will block other operations.

Choosing a shard key index is a complex topic, and is covered here

5. Using the administrative mongos, run the `enableSharding` and `shardCollection` commands on any existing collections you want distributed across the cluster (see link) with the shard key chosen above. At this point the collections are considered "sharded," but enabling sharding is just a metadata operation and no data has been touched. However, once chosen, your shard key cannot be changed without significant manual intervention.

All mongoses should be restarted at this stage for mongos versions < 2.1

6. If any of the existing collections to be sharded were greater than 256GB in size, you will need to manually generate split points for these collections. The procedure for doing so will vary depending on your data and shard key, but is documented here It is not necessary to do the splits in this case before the data is inserted because the data is already present.

7. Wait for the balancer operations from mongos to migrate excess data from the full shard to the empty shards. This can take significant time, longer if competing for resources with current traffic. Estimating the time migrations will take is difficult, as there are a number of factors which affect the speed:

- the relationship between shard key and organization on disk (disk seeks)
- the current requests to the chunks being moved
- the in-memory state of the documents in the chunks being moved (page faults)

8. Once most of the data is balanced, the cluster will be in steady state, and as more data is inserted chunks will be split and moved automatically. If the shard key was chosen appropriately, traffic should be balanced across all servers for the existing collection.

Sharding Internals

This section includes internal implementation details for MongoDB auto sharding. See also the main sharding documentation.

Note: some internals docs could be out of date -- if you see that let us know so we can fix.

Internals

- Moving Chunks
- Sharding Config Schema
- Sharding Design
- Sharding Use Cases
- Shard Ownership
- Splitting Shard Chunks

Unit Tests

```
./mongo --nodb jstests/sharding/*js
```

Moving Chunks

At any given time, a chunk is hosted at one **mongod** server. The sharding machinery routes all the requests to that server automatically, without the application needing to know which server that is. From times to time, the balancer may decide to move chunks around.

It is possible to issue a manual command to move a chunk, using the following command:

```
db.runCommand( { moveChunk : "test.blog.posts" ,
                 find : { author : "eliot" } ,
                 to : "shard1" } )
```

Parameters:

- `moveChunk`: a full collection namespace, including the database name
- `find`: a query expression that falls within the chunk to be moved; the command will find the FROM (donor) shard automatically
- `to`: shard id where the chunk will be moved

The command will return as soon as the TO and FROM shards agreed that it is now the TO's responsibility to handle the chunk.

Moving a chunk is a complex but under-the-covers operation. It involves two interconnected protocols. One, to clone the data of the actual chunk, including any changes made during the cloning process itself. The second protocol is a commit protocol that makes sure that all the migration participants – the TO-shard, the FROM-shard, and the config servers – agree that the migration has completed.

Example

Suppose we are sharding the *test.foo* collection using the *x* field as the shard key. Suppose that we want to move a chunk that looks like:

```
> db.chunks.find({ns : "test.foo", min : {x : 4}})
{ "_id" : "test.foo-x_17", "lastmod" : { "t" : 43000, "i" : 0 }, "ns" : "test.foo", "min" : { "x" : 4
}, "max" : { "x" : 9 }, "shard" : "bar" }
```

This chunk is currently on the shard called "bar" and we want to move it to the shard called "baz". We can do this by choosing some value *v* in the range `4 <= v < 9`, so we'll choose 5:

```
> db.adminCommand({moveChunk : "test.foo", find : {x : 5}, to : "baz"})
{ "millis" : 146880, "ok" : 1 }
```

MongoDB will find the chunk with `{x:5}` in its range and move it to shard baz. The command will return when the move is complete.

If someone is already doing something to that particular chunk, it will be locked and you will not be able to migrate it until the other operation is complete.

```
> db.adminCommand({moveChunk : "test.foo", find : {x : 5}, to : "baz"})
{
        "cause" : {
                "who" : {
                        "_id" : "test.foo",
                        "process" : "ip-10-114-74-220:1299610147:182607800",
                        "state" : 1,
                        "ts" : ObjectId("4d768c5d6bf858cd08a80ac8"),
                        "when" : ISODate("2011-03-08T20:06:53.687Z"),
                        "who" : "ip-10-114-74-220:1299610147:182607800:conn22:510531961",
                        "why" : "migrate-{ x: 5 }"
                },
                "errmsg" : "The collection's metadata lock is already taken.",
                "ok" : 0
        },
        "ok" : 0,
        "errmsg" : "move failed"
}
```

Troubleshooting

Under certain circumstances, a migrate can fail in a way where the shard cannot figure out what to do next. If this occurs, the shard will terminate (instead of serving possible inconsistent data) with a message such as:

```
Tue Dec 10 16:15:49 [conn123] ERROR: moveChunk commit failed: version is at 124|1 instead of 125|1
Tue Dec 10 16:15:49 [conn123] ERROR: TERMINATING
```

If this occurs, make sure:

- All of your config servers are reachable by the shards and mongos processes.
- All of the config servers have the same chunk information for that chunk that failed the migrate.

If the chunk is different, you can restart your mongos servers to reconcile the differences, then restart your shard.

Sharding Config Schema

Sharding configuration schema. This lives in the config servers.

Collections

version

This is a singleton that contains the current meta-data version number.

```
> db.getCollection("version").findOne()
{ "_id" : 1, "version" : 3 }
```

settings

Key/Value table for configurable options (chunkSize/balancer)

```
> db.settings.find()
{ "_id" : "chunksize", "value" : 64 }
{ "_id" : "balancer", "stopped" : false } //you can stop the balancer
```

shards

Stores information about the shards (possibly replicasets in rsname/list-of-host:port,...). The addshard command creates these documents, and mongos processes periodically update the seed lists as the replica sets evolve over time.

```
> db.shards.findOne()
{ "_id" : "shard0", "host" : "localhost:30001" }
{ "_id" : "shard1", "host" : "shard1/localhost:30002" } //shard1 is a replicaset (think of this as a
seed list)
```

databases

```
{
  "_id" : "admin",
  "partitioned" : false,
  "primary" : "shard0"
}
```

collections

Stores meta information about each sharded collection. One entry per sharded collection.

```
mongos> db.collections.find()
{
  "_id" : "test.foo",
  "lastmod" : ISODate("1970-01-16T04:42:21.124Z"),
  "dropped" : false,
  "key" : { "_id" : 1 },
  "unique" : false
}
```

chunks

```
{
 "_id" : "test.foo-x_MinKey",
 "lastmod" : {
  "t" : 1271946858000,
  "i" : 1
 },
 "ns" : "test.foo",
 "min" : {
  "x" : { $minKey : 1 }
 },
 "max" : {
  "x" : { $maxKey : 1 }
 },
 "shard" : "shard0"
}
```

mongos

Record of all mongos affiliated with this cluster. mongos will ping every 30 seconds so we know who is alive. This is not used by the system for anything other than reporting.

```
> db.mongos.findOne()
{
 "_id" : "erh-wd1:27017",
 "ping" : "Fri Apr 23 2010 11:08:39 GMT-0400 (EST)",
 "up" : 30
}
```

changelog

Human readable log of all meta-data changes. Capped collection that defaults to 10mb.

```
> db.changelog.findOne()
{
 "_id" : "erh-wd1-2010-3-21-17-24-0",
 "server" : "erh-wd1",
 "time" : "Wed Apr 21 2010 13:24:24 GMT-0400 (EST)",
 "what" : "split",
 "ns" : "test.foo",
 "details" : {
  "before" : {
   "min" : {
    "x" : { $minKey : 1 }
   },
   "max" : {
    "x" : { $maxKey : 1 }
   }
  },
  "left" : {
   "min" : {
    "x" : { $minKey : 1 }
   },
   "max" : {
    "x" : 5
   }
  },
  "right" : {
   "min" : {
    "x" : 5
   },
   "max" : {
    "x" : { $maxKey : 1 }
   }
  }
 }
}
```

Changes

2 (<= 1.5.0) -> 3 (1.5.1)

- shards : _id is now the name
- databases : _id is now the db name
- general : all references to a shard can be via name or host

Sharding Design

concepts

- *config database* - the top level database that stores information about servers and where things live.
- *shard*. this can be either a single server or a replica pair.
- *database* - one top level namespace. a database can be partitioned or not
- *chunk* - a region of data from a particular collection. A chunk can be though of as (*collectionname,fieldname,lowvalue,highvalue*). The range is inclusive on the low end and exclusive on the high end, *i.e.*, [lowvalue,highvalue).

components and database collections

- config database
- config.servers - this contains all of the servers that the system has. These are logical servers. So for a replica pair, the entry would be *192.168.0.10,192.168.0.11*
- config.databases - all of the databases known to the system. This contains the *primary* server for a database, and information about whether its partitioned or not.
    - config.shards - a list of all database *shards*. Each shard is a db pair, each of which runs a db process.
    - config.homes - specifies which shard is *home* for a given client db.
- shard databases
    - *client*.system.chunklocations - the home shard for a given client db contains a *client*.system.chunklocations collection. this

    collection lists where to find particular chunks; that is, it maps chunk->shard.
- mongos process
    - "routes" request to proper db's, and performs merges. can have a couple per system, or can have 1 per client server.
    - gets chunk locations from the client db's home shard. load lazily to avoid using too much mem.
        - chunk information is cached by mongos. This information can be stale at a mongos (it is always up to date at the owning shard; you cannot migrate an item if the owning shard is down). If so, the shard contacted will tell us so and we can then retry to the proper location.

db operations

- moveprimary - move a database's primary server
- migrate - migrate a chunk from one machine to another.
    - lock and migrate
    - shard db's coordinate with home shard to atomically pass over ownership of the chunk (two phase commit)
- split - split a chunk that is growing too large into pieces. as the two new chunks are on the same machine after the split, this is really just a metadata update and very fast.
- reconfiguration operations
    - add shard - dbgrid processes should lazy load information on a new (unknown) shard when encountered.
    - retire shard - in background gradually migrate all chunks off

minimizing lock time

If a chunk is migrating and is 50MB, that might take 5-10 seconds which is too long for the chunk to be locked.

We could perform the migrate much like Cloner works,where we copy the objects and then apply all operations that happened during copying. This way lock time is minimal.

Sharding Use Cases

What specific use cases do we want to address with db partioning (and other techniques) that are challenging to scale? List here for discussion.

- video site (e.g., youtube) (also, GridFS scale-up)
    - seems straightforward: partition by video
    - for related videos feature, see search below
- social networking (e.g., facebook)
    - this can be quite hard to partition, because it is difficult to cluster people.
- very high RPS sites with small datasets
    - N replicas, instead of partioning, might help here
        - replicas only work if the dataset is really small as we are using/wasting the same RAM on each replica. thus, partioning might help us with ram cache efficiency even if entire data set fits on one or two drives.
- twitter
- search & tagging

Log Processing

Use cases related to map-reduce like things.

- massive sort
- top N queries per day
- compare data from two nonadjacent time periods

Shard Ownership

By shard ownership we mean which server owns a particular key range.

Early draft/thoughts will change:

Contract

- the master copy of the ownership information is in the config database
- mongos instances have cached info on which server owns a shard. this information may be stale.
- mongod instances have definitive information on who owns a shard (atomic with the config db) when they know about a shards ownership

mongod

The mongod processes maintain a cache of shards the mongod instance owns:

```
map<ShardKey,state> ownership
```

State values are as follows:

- missing - no element in the map means no information available. In such a situation we should query the config database to get the state.
- 1 - this instance owns the shard
- 0 - this instance does not own the shard (indicates we queried the config database and found another owner, and remembered that fact)

Initial Assignment of a region to a node.

This is trivial: add the configuration to the config db. As the ShardKey is new, no nodes have any cached information.

Splitting a Key Range

The mongod instance A which owns the range R breaks it into R1,R2 which are still owned by it. It updates the config db. We take care to handle the config db crashing or being unreachable on the split:

```
lock(R) on A
update the config db -- ideally atomically perhaps with eval().  await return code.
ownership[R].erase
unlock(R) on A
```

After the above the cache has no information on the R,R1,R2 ownerships, and will requery configdb on the next request. If the config db crashed and failed to apply the operation, we are still consistent.

Migrate ownership of keyrange R from server A->B. We assume here that B is the coordinator of the job:

```
B copies range from A
lock(R) on A and B
  B copies any additional operations from A (fast)
  clear ownership maps for R on A and B.  B waits for a response from A on this operation.
  B then updates the ownership data in the config db. (Perhaps even fsyncing.) await return code.
  unlock(R) on B
  delete R on A (cleanup)
unlock (R) on A
```

We clear the ownership maps first. That way, if the config db update fails, nothing bad happens, IF mongos filters data upon receipt for being in the correct ranges (or in its query parameters).

R stays locked on A for the cleanup work, but as that shard no longer owns the range, this is not an issue even if slow. It stays locked for that operation in case the shard were to quickly migrate back.

Migrating Empty Shards

Typically we migrate a shard after a split. After certain split scenarios, a shard may be empty but we want to migrate it.

Splitting Shard Chunks

MongoDB uses two key operations to facilitate sharding - *split* and *migrate*. Migrate moves a *chunk* (the data associated with a key range) to another shard. This is done as needed to rebalance. Split splits a chunk into two ranges; this is done to assure no one chunk is unusually large. Split is an inexpensive metadata operation, while migrate is expensive as large amounts of data may be moving server to server.

Both splits and migrates are performed automatically. MongoDB has a sub-system called Balancer, which monitors shards loads and moves chunks around if it finds an imbalance. If you add a new shard to the system, some chunks will eventually be moved to that shard to spread out the load.

A recently split chunk may be moved immediately to a new shard if the system predicts that future insertions will benefit from that move.

Manually Splitting a Chunk

Typically there is no need to manually split a chunk.

The following command splits the chunk where the { _id : 99 }} resides (or would reside if present) in two. The key used as the split point is computed internally and is approximately the key which would divide the chunk in two equally sized new chunks.

```
> use admin
switched to db admin
> db.runCommand( { split : "test.foo" , find : { _id : 99 } } )
...
```

The Balancer treats all chunks the same way, regardless if they were generated by a manual or an automatic split.

Pre-splitting

There is a second version of the split command that takes the exact key you'd like to split on. Often this is most useful when you do not initially have data in a collection, but want to ensure that data is loaded in a distributed way to multiple shards.

In the example below the command splits the chunk where the _id 99 would reside using that key as the split point. Again note that a **key need not exist** for a chunk to use it in its range. The chunk may even be empty.

```
> use admin
switched to db admin
> db.runCommand( { split : "test.foo" , middle : { _id : 99 } } )
...
```

This version of the command allows one to do a **data presplitting** that is especially useful in a load. If the range and distribution of keys to be inserted are known in advance, the collection can be split proportionately to the number of servers using the command above, and the (empty) chunks could be migrated upfront using the moveChunk command.

Pre-Splitting Example #1

Lets say you have 5 shards, and want to insert 100M user profiles sharded by email address. What you should do before inserting is

```
for ( var x=97; x<97+26; x++ ){
  for( var y=97; y<97+26; y+=6 ) {
    var prefix = String.fromCharCode(x) + String.fromCharCode(y);
    db.runCommand( { split : <collection> , middle : { email : prefix } } );
  }
}
```

Then wait for the system to balance (should take about 5 minutes).

Pre-Splitting Example #2

Lets assume you have a sharded database setup that looks like this:

```
> db.printShardingStatus()
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
      { "_id" : "shard0000", "host" : "localhost:30000" }
      { "_id" : "shard0001", "host" : "localhost:30001" }
  databases:
 { "_id" : "admin", "partitioned" : false, "primary" : "config" }
```

and you would like to bulk insert 100M documents into a new sharded collection. Each document has a "hash" field which contains a (unrealistically small) 4-byte hexadecimal string hash value like "00aa". Ideally half the inserts will go to each shard, since there are two shards, and since the **range of the hash value is limited**, we are sure that all documents will fall between "0000" and "ffff". Using the hash value as a shard key will ensure our writes are distributed evenly between the shards, assuming chunks are also distributed evenly between shards.

The first step is to create a sharded collection to contain the data, which can be done in three steps:

```
> use admin
> db.runCommand({ enableSharding : "foo" })
```

Next, we add a unique index to the collection "foo.bar" which is required for the shard key.

```
> use foo
> db.bar.ensureIndex({ hash : 1 }, { unique : true })
```

Finally we shard the collection (which contains no data) using the hash value.

```
> use admin
> db.runCommand({ shardCollection : "foo.bar", key : { hash : 1 } })
> db.printShardingStatus()
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
      { "_id" : "shard0000", "host" : "localhost:30000" }
      { "_id" : "shard0001", "host" : "localhost:30001" }
  databases:
 { "_id" : "admin", "partitioned" : false, "primary" : "config" }
 { "_id" : "test", "partitioned" : false, "primary" : "shard0001" }
 { "_id" : "foo", "partitioned" : true, "primary" : "shard0001" }
  foo.bar chunks:
    shard0001 1
   { "hash" : { "$MinKey" : true } } -->> { "hash" : { "$MaxKey" : true } } on : shard0001 { "t" :
1000, "i" : 0 }
```

Note that one chunk exists on `shard0001`, and contains all values from `$MinKey` to `$MaxKey`. All inserts will initially go to this chunk, which is only on a single shard. To pre-split this chunk such that inserts go to two separate shards, we just need to choose a midpoint and move one of the chunks.

```
> use admin
> db.runCommand({ split : "foo.bar", middle : { hash : "8000" } })
> db.runCommand({ moveChunk : "foo.bar", find : { hash : "8000" }, to : "shard0000" })
> db.printShardingStatus()
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
      { "_id" : "shard0000", "host" : "localhost:30000" }
      { "_id" : "shard0001", "host" : "localhost:30001" }
  databases:
 { "_id" : "admin", "partitioned" : false, "primary" : "config" }
 { "_id" : "test", "partitioned" : false, "primary" : "shard0001" }
 { "_id" : "foo", "partitioned" : true, "primary" : "shard0001" }
  foo.bar chunks:
    shard0001 1
    shard0000 1
   { "hash" : { "$MinKey" : true } } -->> { "hash" : "8000" } on : shard0001 { "t" : 2000, "i" : 1 }
   { "hash" : "8000" } -->> { "hash" : { "$MaxKey" : true } } on : shard0000 { "t" : 2000, "i" : 0 }
```

Inserts will now go to both shards equally.

> Chunks will not split until the data reaches a certain minimum amount in size (hundreds of megabytes). Until this occurs balancing and migration will not take place. When the data volume is this small, distributing data between multiple servers is not required anyway. When pre-splitting manually, many chunks can exist even if very little data is present for each chunk initially.

Pre-Splitting Example #3 – UUID's

Suppose our shard key has UUID's for values. To presplit to 100 chunks we could predefine the following ranges:

["00", "01")
["01", "02")

...
["98","99")
["99",":")

The above example shows UUIDs as strings. Storing them as BinData is more efficient. For example, to generate the value corresponding to "98" above one can invoke:

```
> b = UUID("98000000000000000000000000000000")
> b.hex()
98000000000000000000000000000000
```

Sharding FAQ

- Should I start out with sharded or with a non-sharded MongoDB environment?
- How does sharding work with replication?
- Where do unsharded collections go if sharding is enabled for a database?
- When will data be on more than one shard?
- What happens if I try to update a document on a chunk that is being migrated?
- What if a shard is down or slow and I do a query?
- How do queries distribute across shards?
- How do queries involving sorting work?
- Now that I sharded my collection, how do I <...> (e.g. drop it)?
- If I don't shard on _id how is it kept unique?
- Why is all my data on one server?
- Can I remove old files in the `moveChunk` directory?
- How many connections does each mongos need?
- Why does `mongos` never seem to give up connections?
- How can I see the connections used by `mongos`?
- What is `writebacklisten` in my logs and currentOp()?
- If a moveChunk fails do I need to cleanup the partially moved docs?
- Can I move/rename my config servers, or go from one to three?
- When do the mongos servers pickup config server changes?
- I changed my replicaset configuration, how can I apply these quickly on my `mongos` servers?
- What does setting `maxConns` do on `mongos`?

See also:

- Intro FAQ
- Developer FAQ
- Replica Set FAQ

Should I start out with sharded or with a non-sharded MongoDB environment?

We suggest starting unsharded for simplicity and quick startup unless your initial data set will not fit on single servers. Upgrading to sharding from unsharded is easy and seamless, so there is not a lot of advantage to setting up sharding before your data set is large.

Whether with or without sharding, you should be using replication (replica sets) early on for high availability and disaster recovery.

How does sharding work with replication?

Each shard is a logical collection of partitioned data. The shard could consist of a single server or a cluster of replicas. We recommmend using a replica set for each shard.

Where do unsharded collections go if sharding is enabled for a database?

In the current version of MongoDB, unsharded data goes to the "primary" for the database specified (query config.databases to see details). Future versions will parcel out unsharded collections to different shards (that is, a collection could be on any shard, but will be on only a single shard if unsharded).

When will data be on more than one shard?

MongoDB sharding is range based. So all the objects in a collection get put into a chunk. Only when there is more than 1 chunk is there an option for multiple shards to get data. Right now, the default chunk size is 64mb, so you need at least 64mb for a migration to occur.

What happens if I try to update a document on a chunk that is being migrated?

The update will go through immediately on the old shard, and then the change will be replicated to the new shard before ownership transfers.

What if a shard is down or slow and I do a query?

If a shard is down, the query will return an error unless the "Partial" query options is set. If a shard is responding slowly, mongos will wait for it.

You won't get partial results by default.

How do queries distribute across shards?

There are a few different cases to consider, depending on the query keys and the sort keys. Suppose 3 distinct attributes, X, Y, and Z, where X is the shard key. A query that keys on X and sorts on X will translate straightforwardly to a series of queries against successive shards in X-order. This is faster than querying all shards in parallel because mongos can determine which shards contain the relevant chunks without waiting for all shards to return results. A query that keys on X and sorts on Y will execute in parallel on the appropriate shards, and perform a merge sort keyed on Y of the documents found. A query that keys on Y must run on all shards: if the query sorts by X, the query will run in parallel over shards; if the query sorts by Z, the query will parallelize over shards and perform a merge sort keyed on Z of the documents found.

How do queries involving sorting work?

Each shard pre-sorts its results and the mongos does a merge before sending to the client. See the How queries work with sharding PDF for more details.

Now that I sharded my collection, how do I <...> (e.g. drop it)?

Even if chunked, your data is still part of a collection and so all the collection commands apply.

If I don't shard on _id how is it kept unique?

If you don't use _id as the shard key then it is your responsibility to keep the _id unique. If you have duplicate _id values in your collection **bad things will happen** (as mstearn says).

Best practice on a collection not sharded by _id is to use an identifier that will always be unique, such as a BSON ObjectID, for the _id field.

Why is all my data on one server?

Be sure you declare a shard key for your large collections. Until that is done they will not partition.

MongoDB sharding breaks data into chunks. By default, the default for these chunks is 64MB (in older versions, the default was 200MB). Sharding will keep chunks balanced across shards. You need many chunks to trigger balancing, typically 2gb of data or so. db.printShardingStatus() will tell you how many chunks you have, typically need 10 to start balancing.

Can I remove old files in the `moveChunk` directory?

Yes, these files are made as backups during normal shard balancing operations. Once the operations are done then they can be deleted. The cleanup process is currently manual so please do take care of this to free up space.

How many connections does each mongos need?

In a sharded configuration `mongos` will have 1 incoming connection from the client but may need 1 outgoing connection to each shard (possibly times the number of nodes if the shard is backed by a replicaset).

This means that the possible number of open connections that a `mongos` server requires could be `(1 + (N*M) * C)` where N = number of shards, M = number of replicaset nodes, and C = number of client connections.

Why does `mongos` never seem to give up connections?

`mongos` uses a set of connection pools to communicate to each shard (or shard replicaset node). These pools of connections do not currently constrict when the number of clients decreases. This will lead to a possibly large number of connections being kept if you have even used the `mongos` instance before, even if it is currently not being used.

How can I see the connections used by `mongos`?

Run this command on each `mongos` instance:

```
db._adminCommand("connPoolStats");
```

What is `writebacklisten` in my logs and currentOp()?

Writeback listeners are part of the internal communications between shards and config dbs. If you are seeing these in the currentOp or in the slow logs on the server this is part of the normal operation. In particular, the writeback listener is performing long operations, so it can appear in the slow logs even during normal operation.

If a moveChunk fails do I need to cleanup the partially moved docs?

No, chunk moves are consistent and deterministic; the move will retry and when completed the data will only be on the new shard.

Can I move/rename my config servers, or go from one to three?

Yes, see Changing Config Servers

When do the mongos servers pickup config server changes?

The mongos servers have a cache of the config db for sharding metadata (like chunk placement on shards). Periodically, and during specific events, the cache is updated. There is not way to control this behavior from the client.

I changed my replicaset configuration, how can I apply these quickly on my `mongos` servers?

The `mongos` will pick these changes up over time, but it will be faster if you issue a flushRouterConfig command to each mongos directly.

What does setting `maxConns` do on `mongos`?

This limits the number of connections accepted by `mongos`. If you have a client (driver/application) which creates lots of connections but doesn't close them, letting them timeout, then it might make sense to set this value slightly higher than the maximum number of connections being created by the client, or the connection pool max size. Doing this will keep connection spikes from being sent down to the shards which could cause much worse problems, and memory allocation.

Hosting Center

Database-as-a-Service

- MongoOd.com
- MongoHQ
- MongoLab
- HostedMongo.com
- MongoGrid

Infrastructure-as-a-Service

- Amazon EC2
- Joyent

Platform-as-a-Service

- alwaysdata
- Windows Azure
- cloudControl offers a fully managed platform-as-a-service solution with MongoDB as one of their powerful add-ons. Read the blog post MongoDB Setup at cloudControl for more information.
- dotCloud
- Heroku has add-on connectors to allow you to use from MongoDB from Heroku applications
- RedHat OpenShift
- VMware CloudFoundry
- NodeGrid

Dedicated Servers

MongoDB runs well on both virtualized and non-virtualized servers.

- ServerBeach offers preconfigured, dedicated MongoDB servers. Blog

VPS

- (mt) Media Temple's (ve) server platform is an excellent choice for easy MongoDB deployment.
- A2 Hosting has a quick installer to add MongoDB to your VPS hosting account. Instructions for running the installer are on A2's wiki
- Dreamhost offers instant configuration and deployment of MongoDB
- LOCUM Hosting House is a project-oriented shared hosting and VDS. MongoDB is available for all customers as a part of their subscription plan.

More

- Linode
- Webfaction
- Presentations

Amazon EC2

MongoDB runs well on Amazon EC2. This page includes some notes in this regard.

Getting Started on EC2

This guide is intended to provide instructions on using the MongoDB AMI to set up production instances of MongoDB across Amazon's Web Services (AWS) EC2 infrastructure.

First, we'll step through deployment planning (instance specifications, deployment size, etc.) and then we'll set up a single production node. We'll use those setup steps to deploy a three node MongoDB replica set for production use. Finally, we'll briefly cover some advanced topics such as multi-region availability and data backups.

See the Amazon EC2 Quickstart guide for more information.

Backup, Restore, Verify

Depending upon the configuration of your EC2 instances, there are a number of ways to conduct regular backups of your data. For specific instructions on backing up, restoring and verifying refer to EC2 Backup & Restore.

Deployment Notes

MongoDB via AWS Marketplace

If you installed MongoDB via the AWS Marketplace refer to the MongoDB via AWS Marketplace guide to get a development instance up and running. If you are interested in creating a production deployment, refer to the Amazon EC2 Quickstart. Start with the section on Configuring Storage to set up a place for your data to be stored. After that refer to the Starting MongoDB section to get your MongoDB instance running. If you're interested in scaling your deployment, check out the sections on Replica Sets and Sharding.

Automating Deployment with CloudFormation

CloudFormation from Amazon Web Services provides an easy mechanism to create and manage a collection of AWS resources. To use CloudFormation you create and deploy a template which describes the resources in your stack via the AWS Management Console. We have created a series of reference templates that you can use as a starting point to build your own MongoDB deployments using CloudFormation. Check out Automating Deployment with CloudFormation for a walkthrough and the template files.

Instance Types

MongoDB works on most EC2 types including Linux and Windows. We recommend you use a 64 bit instance as this is required for all MongoDB databases of significant size. Additionally, we find that the larger instances tend to be on the freshest ec2 hardware.

Installing MongoDB

One can download a binary or build from source. Generally it is easier to download a binary. We can download and run the binary without being root. For example on 64 bit Linux:

```
[~]$ curl -O http://downloads.mongodb.org/linux/mongodb-linux-x86_64-1.0.1.tgz
[~]$ tar -xzf mongodb-linux-x86_64-1.0.1.tgz
[~]$ cd mongodb-linux-x86_64-1.0.1/bin
[bin]$ ./mongod --version
```

Before running the database one should decide where to put datafiles. Run df -h to see volumes. On some images /mnt will be the many locally attached storage volume. Alternatively you may want to use Elastic Block Store which will have a different mount point.

If you mount the file-system, ensure that you mount with the noatime and nodiratime atributes, for example

```
/dev/mapper/my_vol /var/lib/mongodb xfs noatime,noexec,nodiratime 0 0
```

Create the mongodb datafile directory in the desired location and then run the database:

```
mkdir /mnt/db
./mongod --fork --logpath ~/mongod.log --dbpath /mnt/db/
```

Operating System

Occasionally, due to the shared and virtualized nature of EC2, an instance can experience intermittent I/O problems and low responsiveness compared to other similar instances. Terminating the instance and bringing up a new one can in some cases result in better performance.

Some people have reported problems with ubuntu 10.04 on ec2.
Please read https://bugs.launchpad.net/ubuntu/+source/linux-ec2/+bug/614853 and https://bugzilla.kernel.org/show_bug.cgi?id=16991

Networking

Port Management

By default the database will now be listening on port 27017. The web administrative UI will be on port 28017.

Keepalive

Change the default TCP keepalive time to 300 seconds. See our troubleshooting page for details.

Storage Configuration

For production use, we recommend raid 10 across 4-8 ebs drives for best performance. Local ec2 drives may be faster but are generally not suitable as they are ephemeral. Multiple ebs drives increase the potential number of random IO's per second (iops), but not necessarily sequential read/write throughput much. In most database applications random iops are important.

For more information, refer to EBS info at Amazon Web Services.

EBS Snapshots

If your datafiles are on a single EBS volume, you can snapshot them for backups.

If you are using journaling, simply take a snapshot (including the journal/ directory).

If not using journaling, you need to use the lock+fsync command (v1.3.1+).
Use this command to lock the database to prevent writes. Then, snapshot the volume. Then use the unlock command to allow writes to the database again. See the full EC2 Backup, Verify & Recovery guide for more information. This method may also be used with slaves / secondaries.

Securing instances

Secure your instances from direct external access by the use of Security Groups. A common approach is to create a MongoDB security group that contains the nodes of your cluster (replica set members or sharded cluster members). Then create a separate group for your app servers or clients.

Create a Rule in your MongoDB security group with the Source field set to the Security Group name containing your app servers and the port set to 27017 (or whatever port you use for your MongoDB). This will ensure that only your app servers have permission to connect to your MongoDB instances.

Communication across regions

Every EC2 instance will have a private IP address that can be used to communicate within the EC2 network. It is also possible to assign a public "elastic" IP to communicate with the servers from another network. If using different EC2 regions, servers can only communicate via public IPs.

To set up a cluster of servers that spans multiple regions, it is recommended to cname the server hostname to the "public dns name" provided by EC2. This will ensure that servers from a different network use the public IP, while the local servers use the private IP, thereby saving costs. This is required since EC2 security groups are local to a region.

For example one solution is following, on each server:

- set the hostname of the server

```
sudo hostname server1
```

- install "bind", it will serve as local resolver
- add a zone for your domain, say "myorg.com", and add the CNAMEs for all your servers

```
...
server1          IN     CNAME   ec2-50-19-237-42.compute-1.amazonaws.com.
server2          IN     CNAME   ec2-50-18-157-87.us-west-1.compute.amazonaws.com.
```

- restart bind and modify /etc/resolv.conf to use the local bind

```

search myorg.conf
nameserver 127.0.0.1
```

Then:

- verify that you can properly resolve server1, server2, ... using a tool like dig.
- when running mongod, db.serverStatus() should show the correct hostname, e.g. "server1:27017".
- you can then set up replica sets or shards using the simple hostname. For example connect to server1 and run "rs.initiate()", then "rs.add('server2:27017')".

Presentations

MongoDB & AWS - Free Webinar on January 20, 2012

Presentation from MongoSV (December 2011)

More Presentations

- Running MongoDB in the Cloud - MongoSF (May 2011)
- MongoDB on Amazon EC2 - Webinar (March 2011)

AWS Marketplace

- Development Instance
    - Starting MongoDB
    - Security
- Production Deployments

If you installed MongoDB via the AWS Marketplace, refer to the sections below to get up and running with your EC2 instance. If you are interested in using the instance for development purposes, check out Development Instance below. If you are interested in creating a production setup, refer to the Amazon EC2 Quickstart guide for more information on security and storage configurations for multi-node deployments.

Development Instance

Starting MongoDB

To use your newly created instance as a platform to test out MongoDB, use the following steps to get up and running. First, connect to the instance via SSH, using the key pair selected when you created your instance. Next, create a data directory and set it's ownership to the MongoDB user:

```
$ sudo mkdir /data
$ sudo chown -R mongod:mongod /data
```

Then update the MongoDB configuration file and set the `dbpath` to the data directory:

```
$ sudo nano /etc/mongod.conf
...
dbpath=/data
...
```

Now set MongoDB to run at boot time and start the `mongod` daemon:

```
$ sudo chkconfig mongod on
$ sudo /etc/init.d/mongod start
Starting mongod:                                           [  OK  ]
[ec2-user@ip-10-114-209-219 ~]$ forked process: 1322
all output going to: /var/log/mongo/mongod.log
```

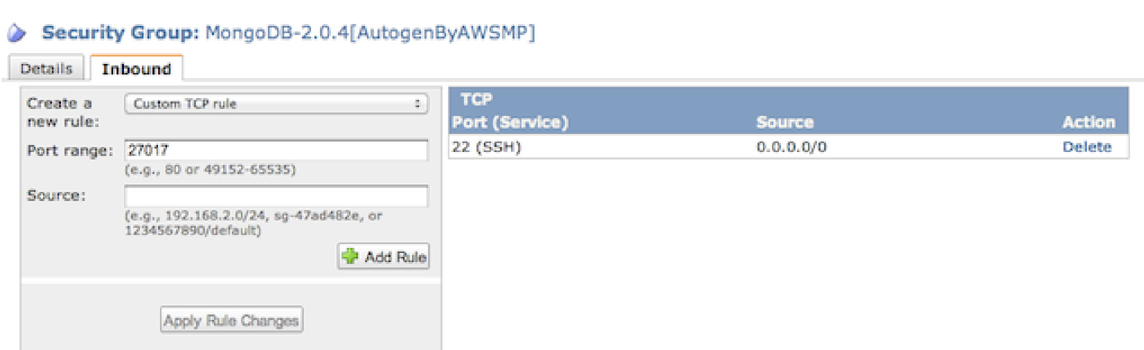
In a few moments MongoDB will finish starting up and then you can connect to MongoDB using the `mongo` client:

```
$ mongo
MongoDB shell version: 2.0.4
connecting to: test
>
```

If you have trouble connecting to the MongoDB daemon, check the log file (`/var/log/mongo/mongod.log`) for information about the `mongod` process.

Security

By default, the instance starts up with a newly created security group and that group specifies that only access via port 22 (SSH) is allowed. In order to make MongoDB accessible to other instances, you'll need to add an additional rule to your security group specifying access to port 27017 along with a source (this can be a specific IP address, instances from within another security group or from an IP). Start by logging in to the AWS EC2 Management Console and navigate to the Security Groups section. Next, find the group that was created when you created your instance and add an Inbound Rule:



After the rule is applied, you'll be able to access MongoDB running in your instance from the source you specified. Consult the AWS EC2 Security Groups documentation for more information about configuring access to your instance with security groups.

Production Deployments

If you are interested in using your instance as part of a production deployment for MongoDB, refer to the Amazon EC2 Quickstart guide for information on planning your deployment, configuring security and configuring storage for production-level instances.

Automating Deployment with CloudFormation

CloudFormation from Amazon Web Services provides an easy mechanism to create and manage a collection of AWS resources. To use CloudFormation you create and deploy a template which describes the resources in your stack via the AWS Management Console. CloudFormation templates are simple JSON formatted text files that can be placed under your normal source control mechanisms, stored in private or public locations such as Amazon S3.

We have created a series of reference templates (see attachments below) that you can use as a starting point to build your own MongoDB deployments using CloudFormation. The sample templates show how to build a single node MongoDB deployment and a MongoDB replica set deployment. Refer to the following sections for information on how these sample templates were developed and instructions on how to customize them for your own deployment onto AWS. The documentation provided below focuses on the sections specific to deploying MongoDB on AWS, for background on standard template sections (e.g. input parameters) refer to the AWS CloudFormation User Guide.

The sample templates we created are as follows (click to download):

- MongoDB_SingleNode.template is an example single-node MongoDB deployment with 4 EBS volumes
- MongoDB_ReplicaSetStack.template sets up a single node and references the...
- MongoDB_ReplicaSetMember.template twice to create the additional nodes (each node has 4 EBS volumes)

Template Walkthrough

Security

EC2 instances in AWS must have security groups designated during creation that specify firewall rules for each instance. In our templates, we create a simple security group `MongoSecurityGroup` that opens up port 22 (SSH) to the outside world and a separate rule (`AWS::EC2::SecurityGroupIngress`) that's used to open up the standard `mongod` port (27017) to other instances within that group:

```
"MongoSecurityGroup" : {
    "Type" : "AWS::EC2::SecurityGroup",
    "Properties" : {
        "GroupDescription" : "MongoDB security group",
        "SecurityGroupIngress" : [
            {
                "IpProtocol" : "tcp",
                "FromPort" : "22",
                "ToPort" : "22",
                "CidrIp" : "0.0.0.0/0"
            }
        ]
    }
},
"MongoSecurityGroupIngress" : {
    "Type" : "AWS::EC2::SecurityGroupIngress",
    "Properties" : {
        "GroupName" : { "Ref" : "MongoSecurityGroup" },
        "IpProtocol" : "tcp",
        "FromPort" : "27017",
        "ToPort" : "27017",
        "SourceSecurityGroupName" : { "Ref" : "MongoSecurityGroup" }
    }
},
```

Depending on the type of deployment you are creating, you may need to add additional security group ingress rules to open up additional ports, available to your instances or to the outside world (e.g. port 27018 for sharding).

Storage Configuration

When using MongoDB on AWS, we recommend using multiple EBS volumes configured as a single RAID10 storage device for your data.

Configuring storage via EBS volumes using CloudFormation requires multiple steps. First the EBS volume must be created, using the `VolumeSize` input value and the same availability zone that we'll use for our EC2 instance (see Instance Configuration).

```
"MongoVolume1" : {
    "Type" : "AWS::EC2::Volume",
    "Properties" : {
        "Size" : { "Ref" : "VolumeSize" },
        "AvailabilityZone" : { "Fn::GetAtt" : [ "MongoInstance", "AvailabilityZone" ]}
    }
},
```

The next step is to attach the volume to an EC2 instance. By referencing the instance name in "InstanceId" we ensure that the EBS volumes will be created after the instance is created.

```
"MongoVolumeMount1" : {
    "Type" : "AWS::EC2::VolumeAttachment",
    "Properties" : {
        "InstanceId" : { "Ref" : "MongoInstance" },
        "VolumeId" : { "Ref" : "MongoVolume1" },
        "Device" : "/dev/sdh1"
    }
},
```

In the attached sample templates we used 4 EBS volumes as the basis for our RAID10 configuration. If you are interested in increasing the number of EBS volumes, you will need to add additional `AWS::EC2::Volume` and `AWS::EC2::VolumeAttachment` resources inside of your template. Refer to the Customizing Storage section below for more information on the steps required.

Instance Configuration

The centerpiece of the CloudFormation template is the creation of the EC2 instances that will be used as the MongoDB server. There are two main sections to be configured for your instance, the instance metadata and instance properties. In the sample provided, the metadata contains information about the packages to be installed (mdadm and sysstat) and any files to be created within the instance. In our sample, the only file to be created is a `yum` repository entry to facilitate MongoDB being installed via yum after the instance has booted.

```
"MongoInstance" : {
    "Type" : "AWS::EC2::Instance",
    "Metadata" : {
        "AWS::CloudFormation::Init" : {
            "config" : {
                "packages" : {
                    "yum" : {
                        "mdadm" : [],
                        "sysstat" : []
                    }
                },
                "files" : {
                    "/etc/yum.repos.d/10gen.repo" : {
                        "content" : { "Fn::Join" : ["", [
                            "[10gen]\n",
                            "name=10gen Repository\n",
                            "baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64\n",
"gpgcheck=0\n"
                        ] ] },
                        "mode" : "000644",
                        "owner" : "root",
                        "group" : "root"
                    }
                }
            }
        }
    },
```

For more information about the possible options for the metadata section, refer to the CloudFormation documentation for the AWS::EC2::Instance resource. The properties section in the template is uses to specify things like the instance type, AMI, security groups and a script to be run after boot (found in the "UserData" section):

```
"Properties" : {
    "InstanceType" : { "Ref" : "InstanceType" },
    "ImageId" : { "Fn::FindInMap" : [ "RegionImageZone", { "Ref" : "AWS::Region" },
        { "Fn::FindInMap" : [ "InstanceTypeArch", { "Ref" : "InstanceType" }, "Arch" ] } ] },
    "SecurityGroups" : [ { "Ref" : "MongoSecurityGroup" } ],
    "KeyName" : { "Ref" : "KeyName" },
    "UserData" : { "Fn::Base64" : { "Fn::Join" : ["", [
        "#!/bin/bash\n",
        ...
```

The instance type is determined by the `InstanceType` input parameter. The "ImageId" specifies the AMI to use, which is determined by the chosen instance type (e.g. m1.large) and region in which the instance is launched (e.g. us-east-1). Refer to the "Mappings" section inside the sample templates for a list of the available AMIs.

The "UserData" section shown above contains a bash script that be executed once the instance is launched and available. The first step is install the `aws-cfn-bootstrap` tools which are used in the script to initialize the instance, signal when errors have occurred and when the the instance creation is complete:

```
"yum update -y aws-cfn-bootstrap\n",

"## Error reporting helper function\n",
"function error_exit\n",
"{\n",
"   /opt/aws/bin/cfn-signal -e 1 -r \"$1\" '", { "Ref" : "WaitHandleMongoInstance" }, "'\n",
"   exit 1\n",
"}\n",

"## Initialize CloudFormation bits\n",
"/opt/aws/bin/cfn-init -v -s ", { "Ref" : "AWS::StackName" }, " -r MongoInstance",
"   --access-key ",  { "Ref" : "HostKeys" },
"   --secret-key ", {"Fn::GetAtt": ["HostKeys", "SecretAccessKey"]},
"   --region ", { "Ref" : "AWS::Region" }, " > /tmp/cfn-init.log 2>&1 || error_exit
$(</tmp/cfn-init.log)\n",
```

Next we include a series of sleep conditions in case our EBS volumes are not yet available. If you plan to use more than 4 EBS volumes in your CloudFormation template, you should add additional sleep conditions here:

```
"## Waiting for EBS mounts to become available\n",
"while [ ! -e /dev/sdh1 ]; do echo waiting for /dev/sdh1 to attach; sleep 10; done\n",
"while [ ! -e /dev/sdh2 ]; do echo waiting for /dev/sdh2 to attach; sleep 10; done\n",
"while [ ! -e /dev/sdh3 ]; do echo waiting for /dev/sdh3 to attach; sleep 10; done\n",
"while [ ! -e /dev/sdh4 ]; do echo waiting for /dev/sdh4 to attach; sleep 10; done\n",
```

Then we install MongoDB and create the RAID10 device:

```
"yum -y install mongo-10gen-server > /tmp/yum-mongo.log 2>&1\n",

"## Create RAID10 and persist configuration\n",
"mdadm --verbose --create /dev/md0 --level=10 --chunk=256 --raid-devices=4 /dev/sdh1 /dev/sdh2
/dev/sdh3 /dev/sdh4 > /tmp/mdadm.log 2>&1\n",
"echo '`mdadm --detail --scan`' | tee -a /etc/mdadm.conf\n",
```

With the RAID10 created, we can set some block device attributes (read-ahead) for each storage device:

```
"## Set read-ahead on each device\n",
"blockdev --setra 128 /dev/md0\n",
"blockdev --setra 128 /dev/sdh1\n",
"blockdev --setra 128 /dev/sdh2\n",
"blockdev --setra 128 /dev/sdh3\n",
"blockdev --setra 128 /dev/sdh4\n",
```

Now we use LVM to create a series of logical volumes for data, journal and logs. The values used below for each volume are specified as percentages, those may need to be changed for your deployment. After creating the volumes, we create the filesystem, mount points and entries in the filesystem table. The last storage-related step is to set the user:group ownership of each mount point to `mongod`:`mongod`.

```
"## Create physical and logical volumes\n",
"dd if=/dev/zero of=/dev/md0 bs=512 count=1\n",
"pvcreate /dev/md0\n",
"vgcreate vg0 /dev/md0\n",
"lvcreate -l 90%vg -n data vg0\n",
"lvcreate -l 5%vg -n log vg0\n",
"lvcreate -l 5%vg -n journal vg0\n",

"## Create filesystems and mount point info\n",
"mke2fs -t ext4 -F /dev/vg0/data > /tmp/mke2fs1.log 2>&1\n",
"mke2fs -t ext4 -F /dev/vg0/log > /tmp/mke2fs2.log 2>&1\n",
"mke2fs -t ext4 -F /dev/vg0/journal > /tmp/mke2fs3.log 2>&1\n",

"mkdir /data\n",
"mkdir /log\n",
"mkdir /journal\n",

"echo '/dev/vg0/data /data ext4 defaults,auto,noatime,noexec 0 0' | tee -a /etc/fstab\n",
"echo '/dev/vg0/log /log ext4 defaults,auto,noatime,noexec 0 0' | tee -a /etc/fstab\n",
"echo '/dev/vg0/journal /journal ext4 defaults,auto,noatime,noexec 0 0' | tee -a /etc/fstab\n",

"mount /data > /tmp/mount1.log 2>&1\n",
"mount /log > /tmp/mount2.log 2>&1\n",
"mount /journal > /tmp/mount3.log 2>&1\n",

"ln -s /journal /data/journal\n",

"chown -R mongod:mongod /data > /tmp/chown1.log 2>&1\n",
"chown -R mongod:mongod /log > /tmp/chown2.log 2>&1\n",
"chown -R mongod:mongod /journal > /tmp/chown3.log 2>&1\n",
```

Next we proceed to creating a MongoDB configuration file, specifiying the logpath and data directory (among others), and start MongoDB:

```
"## Update mongod configuration\n",
"cat <<EOF > /etc/mongod.conf\n",
"logpath=/data/log/mongod.log\n",
"logappend=true\n",
"fork = true\n",
"dbpath=/data\n",
"rest=true\n",
"EOF\n",

"## Start mongod\n",
"/etc/init.d/mongod start > /tmp/mongod-start.log 2>&1\n",
```

The final step is to signal our previously created WaitCondition that setup is complete:

```
"## CloudFormation signal that setup is complete\n",
"/opt/aws/bin/cfn-signal -e 0 -r \"MongoInstance setup complete\" '", { "Ref" :
"WaitHandleMongoInstance" }, "'\n"
```

Once this script has completed executing, the instance and associated resources have been created and our CloudFormation stack is ready to go.

Replica Set Stack

The "ReplicaSetStack" sample template first creates two "ReplicaSetMember" instances (complete with storage configuration) and then creates the overall replica set. The "ReplicaSetMember" template is modeled very closely after the "SingleNode" template except it includes additional input parameters and adds additional commands to the instance setup script in "UserData" specific to creating replica set members (adding in the `replSet` parameter to the MongoDB configuration file).

```
"## Update mongod configuration\n",
"cat <<EOF > /etc/mongod.conf\n",
"logpath=/data/log/mongod.log\n",
"logappend=true\n",
"fork = true\n",
"dbpath=/data\n",
"rest=true\n",
"replSet=", {"Ref" : "ReplicaSetName"}, "\n",
"EOF\n",
```

The "ReplicaSetStack" template also closely follows the "SingleNode" template but adds the following replica set specific steps: (1) it creates a "replicaSetConfigInit.js" file containing the replica set configuration (with hostnames for the additional members) and (2) initiates the replica set. These steps are executed just prior to signaling that the instance setup has been completed:

```
"cat <<EOF > /tmp/replicaSetConfigInit.js\n",
"config = {_id: \"", { "Ref" : "ReplicaSetName" } ,"\", members : [",
    "{_id : 0, host:\"$HOSTNAME:27017\"},",
    "{_id : 1, host:\"", { "Fn::GetAtt" : ["ReplicaSetMember1", "Outputs.ReplicaSetMemberName"] },"
:27017\"},",
    "{_id : 2, host:\"", { "Fn::GetAtt" : ["ReplicaSetMember2", "Outputs.ReplicaSetMemberName"] },"
:27017\"}",
"]};\n",
"rs.initiate(config);\n",
"EOF\n",

"/usr/bin/mongo /tmp/replicaSetConfigInit.js > /tmp/replica-setup.log 2>&1\n",
```

The child "ReplicaSetMember" instances are created from within the "ReplicaSetStack" template using the following resource definition. The inputs for the "ReplicaSetMember" instances are taken from the "ReplicaSetStack" template:

```
"ReplicaSetMember1" : {
    "Type" : "AWS::CloudFormation::Stack",
    "Properties" : {
        "TemplateURL" : "http://S3URL/MongoDB_ReplicaSetMember.template",
"Parameters" : {
            "KeyName" : { "Ref" : "KeyName" },
            "InstanceType" : { "Ref" : "InstanceType" },
            "VolumeSize" : { "Ref" : "VolumeSize" },
            "AccessKeyId" : { "Ref" : "AccessKey" },
            "SecretAccessKey" : {"Fn::GetAtt": ["AccessKey", "SecretAccessKey"]},
            "ReplicaSetName" : { "Ref" : "ReplicaSetName" },
            "SecurityGroupName" : { "Ref" : "MongoSecurityGroup" },
            "InstanceZone" : { "Fn::FindInMap" : ["RegionZone", { "Ref" : "AWS::Region"}, "AZ1"] }
        }
    }
}
```

Customizing Templates

Storage

In the sample templates provided, we used 4 EBS volumes as the basis for our RAID10 configuration. If you are interested in using additional volumes you will need to update the following items for each new volume you want to add:

- Add an "AWS::EC2::Volume" resource
- Add an "AWS::EC2::VolumeAttachment" resource and mount point
- Add an additional sleep condition
- Update the call to `mdadm` to include your additional volumes

Instances

The sample "ReplicaSetStack" template creates three instances, one from the "stack" template and two additional "ReplicaSetMember" instances, via the "AWS::CloudFormation::Stack" resource in the "ReplicaSetStack" template. If you are interested in adding additional replica set members, you'll need to create an additional member instances and edit the `replicaSetConfigInit.js` found within the "ReplicaSetStack" template. Refer to the Replica Set Stack for information about the additional resources and config file to be updated. When creating the replica set the templates spread the created instances across multiple availability zones (e.g. `us-east-1a` or `us-east-1b`). When adding additional instances be sure to specify your desired Availability Zone for increase redundancy.

Sample Template Usage

If you are interesting in launching a single MongoDB node in AWS, download the MongoDB_SingleNode.template file and edit it for your specific deployment. Once you have a completed template, login to the AWS Management Console and navigate to the "AWS CloudFormation" and "Create New Stack". There you'll be prompted to upload your template and input the necessary parameters.

If instead you are interested in launching a multi-node replica set, download the MongoDB_ReplicaSetStack.template and MongoDB_ReplicaSetMember.template. In order for a parent template ("ReplicaSetStack") to refer to child templates ("ReplicaSetMember"), the child template must be uploaded to S3 and the S3 URL of the child template must be specified in the parent template. Once you've uploaded the child template to S3, update the "TemplateURL" parameter in each "ReplicaSetMember" resource in the "ReplicaSetStack" template:

```
"ReplicaSetMember1" : {
    "Type" : "AWS::CloudFormation::Stack",
    "Properties" : {
        "TemplateURL" : "http://S3URL/MongoDB_ReplicaSetMember.template",
"Parameters" : {
            "KeyName" : { "Ref" : "KeyName" },
            "InstanceType" : { "Ref" : "InstanceType" },
            "VolumeSize" : { "Ref" : "VolumeSize" },
            "AccessKeyId" : { "Ref" : "AccessKey" },
            "SecretAccessKey" : {"Fn::GetAtt": ["AccessKey", "SecretAccessKey"]},
            "ReplicaSetName" : { "Ref" : "ReplicaSetName" },
            "SecurityGroupName" : { "Ref" : "MongoSecurityGroup" },
            "InstanceZone" : { "Fn::FindInMap" : ["RegionZone", { "Ref" : "AWS::Region"}, "AZ1"] }
        }
    }
}
```

After updating the "TemplateURL" parameters, login to the AWS Management Console and navigate to the "AWS CloudFormation" and "Create New Stack". There you'll be prompted to upload your template and input the necessary parameters.

For more information on deploying MongoDB on AWS, refer to the Amazon EC2 page and the Amazon EC2 Quickstart guide.

Amazon EC2 Quickstart

This guide is intended to provide instructions on setting up production instances of MongoDB across Amazon's Web Services (AWS) EC2 infrastructure.

First, we'll step through deployment planning (instance specifications, deployment size, etc.) and then we'll set up a single production node. We'll use those setup steps to deploy a three node MongoDB replica set for production use. Finally, we'll briefly cover some advanced topics such as multi-region availability and data backups.

If you installed MongoDB via the AWS Marketplace this guide can be used to get your instance up and running quickly. Start with the section on Configuring Storage to set up a place for your data to be stored. After that refer to the Starting MongoDB section to get your instance started and allow you to get started using MongoDB. If you're interested in scaling your deployment, check out the sections on Replica Sets and Sharding below.

Prerequisites

Generally, there are two ways to work with EC2 - via the command line tools or the AWS Management Console. This guide will use the EC2 command line tools to manage security and storage and launch instances. Use the following steps to setup the EC2 tools on your system:

- Download the EC2 command line tools
- Next, refer to **Prerequisites** and **Setting Up the Tools** from Amazon's Getting Started with the Command Line Tools

Planning Your Deployment

Before starting up your EC2 instances, it's best to sketch out a few details of the planned deployment. Regardless of the configuration or the number of nodes in your deployment, we'll configure each one in roughly the same manner.

Instance Specifications

Amazon has several instance choices available ranging from low to high (based on CPU and memory) throughput. Each instance available serves a different purpose and plays a different role when planning your deployment. There are several roles to consider when deploying a MongoDB production cluster. Consider a situation where your deployment contains an even number of replicated data (`mongod`) instances, an arbiter participates in electing the primary but doesn't hold any data. Therefore a Small instance may be appropriate for the arbiter role but for data nodes you'll want to use 64-bit (standard Large or higher) instances, which have greater CPU and memory scaling. For the purposes of this guide we'll be focused on deploying `mongod` instances that use the standard Large instance. The AMI (ID: `ami-41814f28`) is the 64-bit base Amazon Linux, upon which we'll install MongoDB.

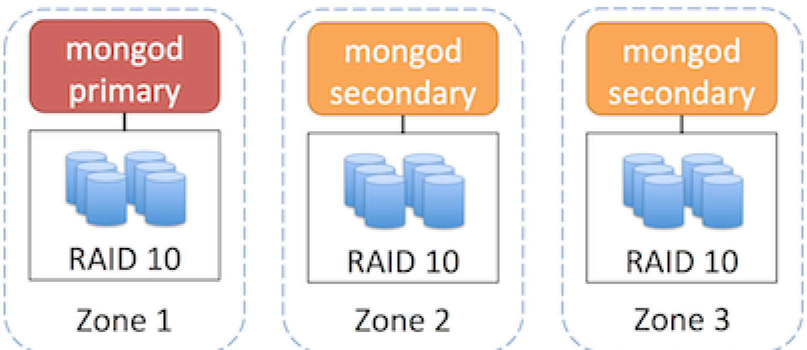
Storage Configuration

For storage we recommend using multiple EBS volumes (as opposed to instance-based storage which is ephemeral) in a RAID-based configuration. Specifically for production deployments you should use RAID 10 across 4-8 EBS volumes for the best performance. When deploying RAID 10, you'll need enough volume storage to be twice that of the desired available storage for MongoDB. Therefore for 8 GiB of available storage you'll need to have 16 GiB of allocated storage space across multiple EBS volumes.

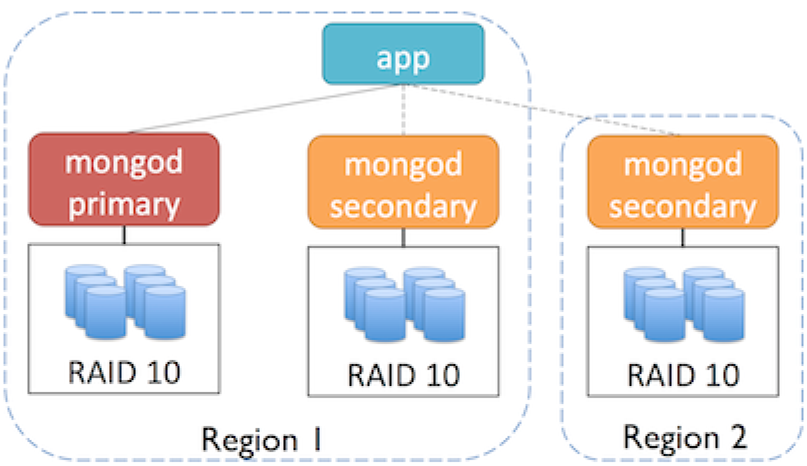
Topology

For the purposes of this guide, our topology will be somewhat simple: one to three EC2 instances, each with multiple EBS volumes attached, all located in the same availability zone (and by extension, within the same region). If you are interested in creating a deployment that spans availability zones or regions, it's best to do that planning up front and take into account security group designations (they cannot span regions) and hostname/DNS access (AWS internal IP addresses can only be used within a zone).

An example 3 node replica set with RAID 10 storage, spanning multiple availability zones would like similar to the following. Availability zones within EC2 are similar to different server racks, therefore it is recommended that you deploy your replica set across multiple zones.



For even greater redundancy and failover, you could also deploy your replica set across multiple regions (and go further with multiple zones in each region):

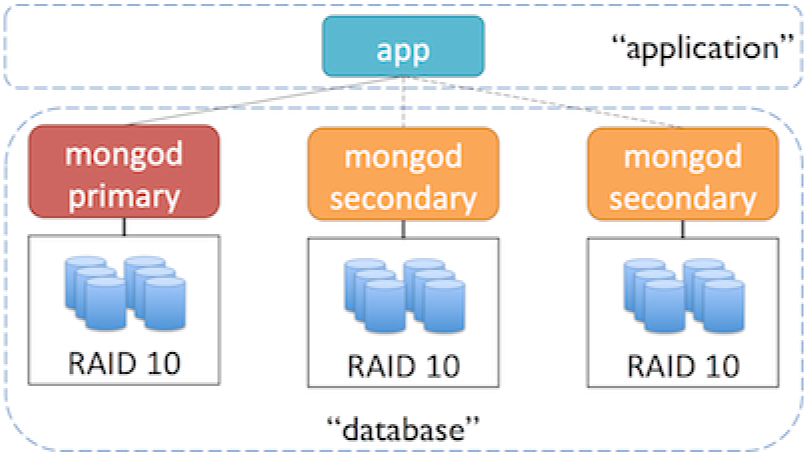


Refer to the AWS documentation on Using Regions and Availability Zones for more information.

Security

The recommended approach to securing your instances is to use multiple security groups for your MongoDB deployment, one for each type of interaction. For example, you could use one group to manage communication amongst the nodes in your cluster, another group that allows your application to communicate with the database and optionally, a group for tools and maintenance tasks.

An example setup with two security groups might look like this:



Securing Your Deployment

Before starting up instances we want to get the security groups created. As previously discussed, we recommend using multiple groups, one for each type of interaction. The following steps will show you how to create two groups (one for your app and another for your database) and provide the authorizations necessary for communication between them.

From the command line, create the `database` group and authorize SSH:

```
$ ec2-create-group database --description "security group for database"
GROUP   sg-0814f660 database    security group for database

$ ec2-authorize database -p 22
GROUP           database
PERMISSION      database    ALLOWS  tcp 22  22  FROM    CIDR    0.0.0.0/0   ingress
```

Authorize communication within the group of MongoDB instances by adding the group to itself. Note you'll need to provide the user account number (using the `-u` flag) when authorizing groups:

```
$ ec2-authorize database -o database -u [AWS_ACCOUNT_NUMBER]
GROUP           database
PERMISSION      database    ALLOWS  tcp 0   65535   FROM    USER    AWS_ACCOUNT_NUMBER  NAME database
ingress
PERMISSION      database    ALLOWS  udp 0   65535   FROM    USER    AWS_ACCOUNT_NUMBER  NAME database
ingress
PERMISSION      database    ALLOWS  icmp    -1  -1  FROM    USER    AWS_ACCOUNT_NUMBER  NAME database
ingress
```

Optionally, for testing you could also enable the port for the MongoDB web-based status interface (port `28017`):

```
$ ec2-authorize database -p 28017
GROUP           database
PERMISSION      database    ALLOWS  tcp 28017   28017   FROM    CIDR    0.0.0.0/0   ingress
```

Now create a group that will hold application servers, which will communicate with the database cluster:

```
$ ec2-create-group application --description "security group for application servers"
GROUP   sg-081bf960 application security group for application servers
```

Finally, authorize communication from the application servers (group `application`) to the MongoDB instances (group `database`):

```
$ ec2-authorize database -o application -u [AWS_ACCOUNT_NUMBER]
GROUP           database
PERMISSION      database    ALLOWS  tcp 0   65535   FROM    USER    AWS_ACCOUNT_NUMBER  NAME
application        ingress
PERMISSION      database    ALLOWS  udp 0   65535   FROM    USER    AWS_ACCOUNT_NUMBER  NAME
application        ingress
PERMISSION      database    ALLOWS  icmp    -1  -1  FROM    USER    AWS_ACCOUNT_NUMBER  NAME
application        ingress
```

Refer to the AWS guide Using Security Groups for more information on creating and managing security groups.

The next step is to generate an SSH key-pair that we'll use to connect to our running EC2 instances. Amazon's tools provide a mechanism to quickly generate a public-private key pair. Once generated, we'll need to save the private key so that we can use it to connect via SSH later (click here for more info on key pairs and AWS).

First, generate the key pair:

```
$ ec2-add-keypair cluster-keypair
KEYPAIR cluster-keypair 05:9b:b3:a6:2e:2f:54:6e:1d:a7:0a:89:56:36:b2:e8:44:37:cf:66
-----BEGIN RSA PRIVATE KEY-----
...
[private key contents]
...
-----END RSA PRIVATE KEY-----
```

Save the contents of the key to a file (including the `BEGIN` and `END` lines) and make sure that file is only readable by you:

```
$ chmod 600 private-key-file
```

Optionally, you can also the key to the SSH agent to ease connecting to our instances later:

```
$ ssh-add private-key-file
```

We're finished with the pre-deployment steps; we've covered the storage and security considerations that's necessary to setup and deploy our instances.

Deploying a Single Node

We'll start our deployment by setting up single node because later on we'll use the same steps to set up a larger cluster. The first step is to launch the instance and then setup the EBS-backed RAID 10 storage for the instance. Setting up the storage requires creating, attaching, configuring and formatting the volumes where our data will be stored.

Note: If you created a MongoDB instance via the AWS Marketplace, skip ahead to Configure Storage below.

Launch Instance

From the command line we can launch the instance. We'll need to supply an ID for an Amazon Machine Image (AMI) that we'll build our node from. We recommend using a 64-bit Amazon Linux AMI as the base of your MongoDB nodes. In this example, we are using `ami-e565ba8c` with the number of nodes (1), security group (`database`), authentication keypair (`cluster-keypair`), type of instance (`m1.large`) and availability zone (`us-east-1a`). Depending on the region you deploy into, a different AMI ID may be needed:

```
$ ec2-run-instances ami-e565ba8c -n 1 -g database -k cluster-keypair -t m1.large -z us-east-1a

RESERVATION r-f57f8094  711489243244    database
INSTANCE    i-a3323dc0  ami-e565ba8c            pending cluster-keypair 0       m1.large
2011-12-06T23:18:18+0000    us-east-1c  aki-825ea7eb            monitoring-disabled
ebs                 paravirtual xen     sg-0814f660, sg-1e15f776    default
```

Next, let's add some tags to the instance so we can identify it later. Tags are just metadata key-value pairs:

```
$ ec2-create-tags i-11eee072 --tag Name=QuickstartTestNode --tag Owner=Bob

TAG instance    i-11eee072  Name    QuickstartTestNode
TAG instance    i-11eee072  Owner   Bob
```

Now we can ascertain some status information about running instances at AWS (includes EBS volumes as well):

```
$ ec2-describe-instances

RESERVATION r-f57f8094  711489243244    database
INSTANCE    i-a3323dc0  ami-e565ba8c    ec2-107-20-68-113.compute-1.amazonaws.com
ip-10-2-182-67.ec2.internal running cluster-keypair 0       m1.large    2011-12-06T23:18:18+0000
us-east-1c  aki-825ea7eb            monitoring-disabled 107.20.68.113   10.2.182.67         ebs
paravirtual xen     sg-0814f660, sg-1e15f776    default
BLOCKDEVICE /dev/sda1   vol-2348cf4e    2011-12-06T23:18:43.000Z
```

Configure Storage

Now that the instance is running, let's create the EBS volumes we'll use for storing our data. In this guide we'll set up 4 volumes with 4 GiB of storage each (configured that's 16 GiB but because we're using a RAID 10 configuration that will become 8 GiB of available storage).

First off, create the EBS volumes supplying the size (4) and zone (`us-east-1a`) and save the results into a temporary file that we'll read from for the next command. Here's the command we'll use:

```
$ for x in {1..4}; do \
> ec2-create-volume -s 4 -z us-east-1a; \
> done > vols.txt
```

Here's the output of that command:

```
$ more vols.txt
VOLUME  vol-e796108a    4               us-east-1a      creating        2011-12-07T04:37:21+0000
VOLUME  vol-c39610ae    4               us-east-1a      creating        2011-12-07T04:37:30+0000
VOLUME  vol-a19610cc    4               us-east-1a      creating        2011-12-07T04:37:39+0000
VOLUME  vol-b19610dc    4               us-east-1a      creating        2011-12-07T04:37:47+0000
```

Now, let's attach those newly created volumes to our previously launched running instance from above. From the command line we'll start with the temp file (`vols.txt`), the running instance ID (`i-11eee072`), and a prefix for each attached device (`/dev/sdh`):

```
$ (i=0; \
> for vol in $(awk '{print $2}' vols.txt); do \
>   i=$((i+1)); \
>   ec2-attach-volume $vol -i i-11eee072 -d /dev/sdh${i}; \
> done)
```

Assuming the volumes attached successfully, you should see something like this:

```
ATTACHMENT  vol-e796108a    i-11eee072  /dev/sdh1   attaching   2011-12-07T04:48:22+0000
ATTACHMENT  vol-c39610ae    i-11eee072  /dev/sdh2   attaching   2011-12-07T04:48:29+0000
ATTACHMENT  vol-a19610cc    i-11eee072  /dev/sdh3   attaching   2011-12-07T04:48:37+0000
ATTACHMENT  vol-b19610dc    i-11eee072  /dev/sdh4   attaching   2011-12-07T04:48:44+0000
```

Now we'll need to connect to the running instance via SSH and configure those attached volumes as a RAID array. If you added the private key to your running SSH agent, you should be able to connect with something like (substituting your instance's hostname):

```
$ ssh ec2-user@ec2-a-b-c-d.amazonaws.com
```

And now create the RAID array using the built-in `mdadm`. You'll need the level (`10`), number of volumes (`4`), name of the new device (`/dev/md0`) and the attached device prefix (`/dev/sdh*`):

```
$ sudo mdadm --verbose --create /dev/md0 --level=10 --chunk=256 --raid-devices=4 /dev/sdh1 /dev/sdh2
/dev/sdh3 /dev/sdh4
$ echo 'DEVICE /dev/sdh1 /dev/sdh2 /dev/sdh3 /dev/sdh4' | sudo tee -a /etc/mdadm.conf
$ sudo mdadm --detail --scan | sudo tee -a /etc/mdadm.conf
```

Once `mdadm` is done and we've persisted the storage configuration, we'll need to tune the EBS volumes to achieve desired performance levels. This tuning is done by setting the "read-ahead" on each device. For more information refer to the blockdev man page.

```
$ sudo blockdev --setra 128 /dev/md0
$ sudo blockdev --setra 128 /dev/sdh1
$ sudo blockdev --setra 128 /dev/sdh2
$ sudo blockdev --setra 128 /dev/sdh3
$ sudo blockdev --setra 128 /dev/sdh4
```

With the RAID10 created we now turn to the Logical Volume Manager (`lvm`) which we'll use to create logical volumes for the data, log files and journal for MongoDB. The purpose of using `lvm` is to (1) safely partition differnt objects from each other and (2) provide a mechanism that we can use to grow our storage sizes later. First we start by zeroing out our RAID, creating a physical device designation and finally a volume group for that device.

```
$ sudo dd if=/dev/zero of=/dev/md0 bs=512 count=1
$ sudo pvcreate /dev/md0
$ sudo vgcreate vg0 /dev/md0
```

Once the volume group has been created, we now create logical volumes for the data, logs and journal. Depending upon the amount of available storage you may want to designate specific sizes vs. volume group percentages (as shown below). We recommend approximately 10GB for log storage and 10GB for journal storage.

```
$ sudo lvcreate -l 90%vg -n data vg0
$ sudo lvcreate -l 5%vg -n log vg0
$ sudo lvcreate -l 5%vg -n journal vg0
```

At this point we have three volumes to configure (`/dev/vg0/...`). For each volume we'll create a filesystem, mount point and an entry in the filesystem table. In the example below we used the ext4 filesystem however you could instead elect to use xfs, just be sure to edit the `mke2fs` and `sed` commands accordingly. The `/etc/fstab` entries require the partition (e.g. `/dev/vg0/data`), a mount point for the filesystem (`/data`), the filesystem type (`ext4` or `xfs`) and the mount parameters (`defaults,auto,noatime,noexec,nodiratime 0 0`, refer to the mount man page for more information on these parameters:

```
$ sudo mke2fs -t ext4 -F /dev/vg0/data
$ sudo mke2fs -t ext4 -F /dev/vg0/log
$ sudo mke2fs -t ext4 -F /dev/vg0/journal

$ sudo mkdir /data
$ sudo mkdir /log
$ sudo mkdir /journal

$ echo '/dev/vg0/data /data ext4 defaults,auto,noatime,noexec 0 0' | sudo tee -a /etc/fstab
$ echo '/dev/vg0/log /log ext4 defaults,auto,noatime,noexec 0 0' | sudo tee -a/etc/fstab
$ echo '/dev/vg0/journal /journal ext4 defaults,auto,noatime,noexec 0 0' | sudo tee -a /etc/fstab
```

Now mount all of the storage devices. By adding the entry to `/etc/fstab`, we've shortened the call to mount because it will look in that file for the extended command paramters.

```
$ sudo mount /data
$ sudo mount /log
$ sudo mount /journal
```

With the devices mounted we issue one last call to set the MongoDB journal files to be written to our new journal device, via a symbolic link to the new device:

```
$ sudo ln -s /journal /data/journal
```

Install and Configure MongoDB

Note: If you created a MongoDB instance via the AWS Marketplace, skip ahead to Starting MongoDB below.

Now that the storage has been configured, we need to install and configure MongoDB to use the storage we've set up, then set it to start up on boot automatically. First, add an entry to the local `yum` repository for MongoDB:

```
$ echo "[10gen]
name=10gen Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64
gpgcheck=0" | sudo tee -a /etc/yum.repos.d/10gen.repo
```

Next, install MongoDB and the `sysstat` diagnostic tools:

```
$ sudo yum -y install mongo-10gen-server
$ sudo yum -y install sysstat
```

Set the storage items (data, log, journal) to be owned by the user (`mongod`) and group (`mongod`) that MongoDB will be starting under:

```
$ sudo chown mongod:mongod /data
$ sudo chown mongod:mongod /log
$ sudo chown mongod:mongod /journal
```

Now edit the MongoDB configuration file and update the following parameters

```
$ sudo nano /etc/mongod.conf
...
logpath=/log/mongod.log
logappend=true
fork=true
dbpath=/data
...
```

Starting MongoDB

Set the MongoDB service to start at boot and activate it:

```
$ sudo chkconfig mongod on
$ sudo /etc/init.d/mongod start
```

When starting for the first time, it will take a couple of minutes for MongoDB to start, setup it's storage and become available. Once it is, you should be able to connect to it from within your instance:

```
$ mongo
MongoDB shell version: 2.0.4
connecting to: test
>
```

Just to confirm the system is working correctly, try creating a test database, test collection and save a document:

```
> use testdb
switched to db testdb
> db.createCollection("testCollection")
{ "ok" : 1 }
> db.testCollection.save({"name":"bob"})
> db.testCollection.find()
{ "_id" : ObjectId("4edfda1c86176ab8e27ee976"), "name" : "bob" }
```

Now that we've got a single node up and running with EBS backed RAID storage, let's move on and create a multi-node replica set.

Deploying a Multi-node Replica Set

Replica Set Background

Replica sets are a form of asynchronous master/slave replication, adding automatic failover and automatic recovery of member nodes. A replica set consists of two or more nodes that are copies of each other (i.e.: replicas). Refer to the MongoDB Replica Set documentation for more information.

Create and Configure Instances

For this guide, we'll set up a three node replica set. To set up each node use the instructions from Deploying a Single Node above. Once that's completed, we'll update the configurations for each node and get the replica set started.

First we'll need to edit the MongoDB configuration and update the `replSet` parameter:

```
$ sudo nano /etc/mongod.conf
...
replSet=exampleReplicaSetName
...
```

Save the configuration file and restart `mongod`:

```
$ sudo /etc/init.d/mongod restart
```

Configure Replica Set

Once MongoDB has started and is running on each node, we'll need to connect to the desired primary node, initiate the replica set and add the other nodes. First connect to the desired primary via SSH and then start mongo to initiate the set:

```
$ mongo
MongoDB shell version: 2.0.4
connecting to: test
> rs.initiate()
{
    "info2" : "no configuration explicitly specified -- making one",
    "me" : "ip-10-127-127-91:27017",
    "info" : "Config now saved locally.  Should come online in about a minute.",
    "ok" : 1
}
```

Next, add the other nodes to the replica set:

```
> rs.add("ec2-107-21-46-145.compute-1.amazonaws.com")
{ "ok" : 1 }
PRIMARY> rs.add("ec2-108-90-58-191.compute-1.amazonaws.com")
{ "ok" : 1 }
PRIMARY>
```

The 3 node replica set is now configured. You can confirm the setup by checking the health of the replica set:

```
PRIMARY> rs.status()
{
    "set" : "exampleReplicaSetName",
    "date" : "Tue Dec 06 2011 11:39:08 GMT-0500 (CDT)",
    "myState" : 1,
    "members" : [
        {
            "name" : "ip-10-127-127-91:27017",
            "self" : true,
        },
        {
            "name" : "ec2-107-21-46-145.compute-1.amazonaws.com:27017",
            "health" : 1,
            "uptime" : 101,
            "lastHeartbeat" : "Tue Dec 06 2011 11:39:07 GMT-0500",
        },
        {
            "name" : "ec2-108-90-58-191.compute-1.amazonaws.com:27017",
            "health" : 1,
            "uptime" : 107,
            "lastHeartbeat" : "Tue Dec 06 2011 11:39:07 GMT-0500",
        }
    ],
    "ok" : 1
}
```

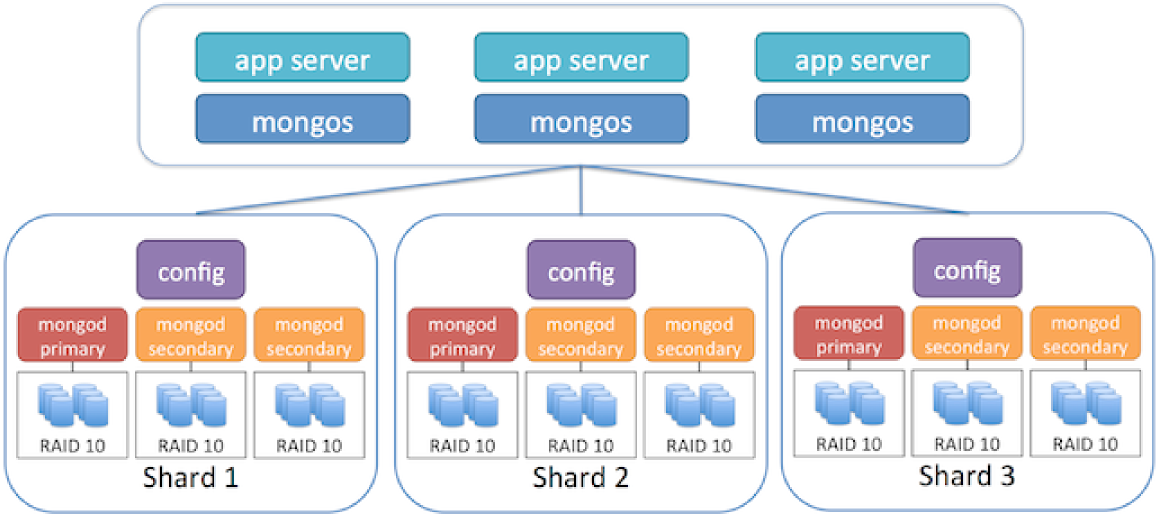
What we've completed here is a simple replica set; there are additional configurations out there that may make more sense for your deployment, refer to the MongoDB documentation for more information. If you intend to use your replica set to help scale read capacity, you'll also need to update your application's code and add the appropriate `slaveOk=true` where necessary so that read results can be returned from additional nodes more quickly.

Deploying a Sharded Configuration

MongoDB scales horizontally via a partitioned data approach known as sharding. MongoDB provides the ability to automatically balance and distribute data across multiple partitions to support write scalability. For more information, refer to the MongoDB sharding docs.

Simple Sharding Architecture

To build our simple sharded configuration, we'll be building upon the replica set steps we just worked on. To get started you'll need to create two additional replica set configurations, just the same from above. When configuring each server instance we'll set the `shardsvr` parameter inside the `mongod` configuration file. Next we'll take one node from each replica set and set it to run as a config server as well. The config server maintains metadata about the sharded data storage cluster. Finally, we'll add instances for the `mongos` router, which handles routing requests from your app to the correct shard. The recommended approach is to run this component on your application servers. The following image shows a recommended topology for use with sharding:



Use the instructions from Deploying a Single Node above to create the required nodes for each replica set (3 instances per set, for a total of 9 instances). Now start configuring the replica set but before saving `/etc/mongod.conf`, add this parameter:

```
shardsvr = true
```

Save the configuration file and restart `mongod`:

```
$ sudo /etc/init.d/mongod restart
```

Once `/etc/mongod.conf` has been updated, initiate the replica set and add the members as described in Configure Replica Set. Once that's complete, choose one instance from each replica set and start an additional `mongod` process those instances, this time as the config server component:

```
$ mongod --configsvr
```

Now that we've got N config servers running (where N is the number of running replica sets) we can set up the request router `mongos`. This process typically runs on your application servers and handles routing database requests from your app to the correct database shard. Assuming you already have your application configured and deployed, use `ssh` to connect to each instance and install MongoDB using the steps from Install and Configure MongoDB.

Before we continue, it is important to consider the role DNS plays in a sharding setup. Generally we recommend using DNS hostnames for configuring replica sets, which Amazon handles appropriately, as opposed to using specific IP addresses. Essentially, AWS knows the mapping between public and private addresses and hostnames and manages inter-region domain name resolution. Therefore, by using the public DNS name for our servers we can ensure that whether our servers are in a single region or across multiple regions, AWS will correctly route our requests. When it comes to setting up sharding, we recommend an additional step of using DNS aliases for the instances that will be acting as config servers. The routers must know the hostnames of the config servers so by using DNS aliases we gain additional flexibility if config servers ever need to change. All it takes is pointing the DNS alias to another instance and no additional update to the router configuration is needed. For more information on this topic, refer to docs on changing config servers.

Once the DNS settings have taken effect, we can proceed with the `mongos` setup. Go back to the command line on each of the instances you'll be using for `mongos` and start the service and point it to the instances running the config server using their DNS aliases (ex: `alias1.yourdomain.com`, `alias2.yourdomain.com` and `alias3.yourdomain.com`) along with the config server port `27019`:

```
$ mongos --configdb
alias1.yourdomain.com:27019,alias2.yourdomain.com:27019,alias3.yourdomain.com:27019
```

With the `mongos` routers running, we can now complete the setup for the sharding architecture. The last step is to add the previously created replica sets to the overall system. Choose one of the instances that is running `mongos`, start the `mongo` client using the hostname and port (`27017`) and connect to the `admin` database. You'll need to have the name for each replica set (ex: `replicaSetName1`) and the hostnames for each member of the set (e.g: `replicaSetHost1`)

```
$ mongo host-running-mongos:27017/admin
MongoDB shell version: 2.0.4
connecting to: admin
> db.adminCommand({addShard:
"replicaSetName1/replicaSetHost1:27018,replicaSetHost2:27018,replicaSetHost3:27018"})
```

The `addShard` command will need to be repeated for each replica set that is part of the sharded setup:

```
> db.adminCommand({addShard:
"replicaSetName2/replicaSetHost4:27018,replicaSetHost5:27018,replicaSetHost6:27018"})
> db.adminCommand({addShard:
"replicaSetName3/replicaSetHost7:27018,replicaSetHost8:27018,replicaSetHost9:27018"})
```

Once these steps have been completed, you'll have a simple sharded configuration. The architecture we used includes 3 database shards for write scalability and three replicas within each shard for read scalability and failover. This type of setup deployed across multiple regions (ex: one node from each replica located in `us-west-1`) would also provide some degree of disaster recovery as well.

In order to utilize this newly created configuration, you'll need to specify which databases and which collections are to be sharded. See Enabling Sharding on a Database and Sharding a Collection for more information.

Backup and Restore

There are several ways to backup your data when using AWS, refer to the EC2 Backup & Restore guide for more information.

dotCloud

Running MongoDB on dotCloud

MongoDB can run on dotCloud. It supports replica sets, and has alpha support for sharding.

The whole point of dotCloud is to run your apps and your databases in the same place, to optimize for latency and reliability. However, you can also deploy MongoDB on dotCloud and use it to power an app running anywhere else.

If you don't have a dotCloud account yet...

Well, what are you waiting for?

Go ahead and create one (it's free!) and install the CLI:

```
sudo easy_install pip ; sudo pip install dotcloud
```

If you need help to get the CLI running, check the dotCloud install docs and don't hesitate to ask for help.

With a dotCloud account

The following snippet will deploy MongoDB on dotCloud for you in no time:

```
mkdir mongodb-on-dotcloud
cat >mongodb-on-dotcloud/dotcloud.yml <<EOF
db:
  type: mongodb
EOF
dotcloud push mongorocks mongodb-on-dotcloud
dotcloud info mongorocks.db
```

The last command will show you the host, port, and credentials to be used to connect to your database.

Scaling

Assuming you followed the instructions of the previous section, if you want to get a replica sets of 3 servers:

```
dotcloud scale mongorocks db=3
```

Advanced use

If you want to have a closer look at your MongoDB server, nothing beats SSH access:

```
dotcloud ssh mongorocks.db
```

Moar docs

- dotCloud documentation for the MongoDB service
- generic introduction to dotCloud (in case you want to run not only MongoDB, but also Node.js, Python, Ruby, Perl, Java, RabbitMQ, Redis, MySQL, PostgreSQL, CouchDB, Riak, Erlang, or something else, on dotCloud)

Ready-to-use apps

All you need to do to run them is a `git clone` and a `dotcloud push`:

- Django setup using MongoDB to store objects
- MongoDB + Node.js sample app

Getting help

dotCloud has a Q&A site, and the dotCloud team can be reached through the FreeNode IRC network on #dotcloud.

Joyent

For the quickest start, you can use the Joyent SmartMachine for MongoDB Appliance

For installing MongoDB on a Joyent Node Smart Machine, see this article

The prebuilt MongoDB Solaris 64 binaries work with Joyent accelerators.

Some newer gcc libraries are required to run -- see sample setup session below.

```
$ # assuming a 64 bit accelerator
$ /usr/bin/isainfo -kv
64-bit amd64 kernel modules

$ # get mongodb
$ # note this is 'latest' you may want a different version
$ curl -O http://downloads.mongodb.org/sunos5/mongodb-sunos5-x86_64-latest.tgz
$ gzip -d mongodb-sunos5-x86_64-latest.tgz
$ tar -xf mongodb-sunos5-x86_64-latest.tar
$ mv "mongodb-sunos5-x86_64-2009-10-26" mongo

$ cd mongo

$ # get extra libraries we need (else you will get a libstdc++.so.6 dependency issue)
$ curl -O http://downloads.mongodb.org.s3.amazonaws.com/sunos5/mongo-extra-64.tgz
$ gzip -d mongo-extra-64.tgz
$ tar -xf mongo-extra-64.tar
$ # just as an example - you will really probably want to put these somewhere better:
$ export LD_LIBRARY_PATH=mongo-extra-64
$ bin/mongod --help
```

RedHat OpenShift

- Getting Started
- Sample MongoDB Apps
- Additional Information
- Documentation

OpenShift is a Platform as a Service (PaaS) offering from RedHat, which provides support for rapid deployment and automatic scalability support for web applications developed with Java EE, Node.js, Python, PHP, Perl and Ruby, and several databases including MongoDB.

Getting Started

To get started with OpenShift and MongoDB, check out the OpenShift Quickstart guide. The guide reviews the steps necessary to create, deploy and manage your MongoDB-backed apps. The guide also covers things like application snapshots and database backups.

Sample MongoDB Apps

- Python Twitter Clone on Github
- PHP Twitter Clone on Github

Additional Information

- Getting Started with MongoDb Shell on OpenShift (video)
- Getting Started with MongoDB Monitoring Service (MMS) on OpenShift (video)
- Deploying Python Apps in the Cloud with MongoDB & OpenShift (video)
- Deploying a PHP Twitter App in the Cloud with MongoDB & OpenShift (video)
- How to Manage MongoDB on OpenShift with Your Favorite Admin Tool (video)

Documentation

- OpenShift documentation
- OpenShift community

OpenShift Quickstart

- Cloning the App
- Deploying onto Express
- Testing the Deployment
- Advanced Functionality
  - App Administration
- Additional Information

OpenShift is a Platform as a Service (PaaS) offering from RedHat, which provides support for rapid deployment and automatic scalability support for web applications developed with Java EE, Node.js, Python, PHP, Perl and Ruby. OpenShift Express is a free shared-cloud solution for deploying your applications. With Express, you simply push your app using the command line tools to OpenShift and the platform takes care of the hosting infrastructure.

In this guide, we'll take a simple Java web application and deploy it onto OpenShift Express. We'll walk through the steps required to setup and administer MongoDB as well as backing up your data.

Cloning the App

We'll be deploying an existing Java web application onto OpenShift. The app is a simple REST-style checkin mechanism with a single endpoint. The app supports POST-ing new checkins (by supplying a comment and location) and GET-ing checkins near a given location (by supplying a pair of coordinates). The app was built using Java SE 6 and
Maven. Make sure to have those components installed, along with Git for your platform, before continuing.

First, start by cloning the repository for the web application::

```
$ git clone https://github.com/crcsmnky/openshift-checkins.git
$ cd openshift-checkins
```

Next, make sure you can build the app as-is using Maven::

```
$ mvn package
```

If that completes successfully, you're ready to move on. To prepare our app for deployment, we'll need to setup our OpenShift Express account.

Deploying onto Express

To deploy apps onto Express you'll need to create an OpenShift account from the
OpenShift sign up page and install the OpenShift command line tools.

First, follow the steps from the OpenShift Express getting started guide in the ***Install the client tools*** section for your platform. Once the tools have been installed we'll create a domain name, app name and then push the sample app to OpenShift Express (the URL for the app will be something like **http://appname-domainname.rhcloud.com**).

First, create the domain name::

```
$ rhc domain create -n your-domain -l your-openshift-login
```

This command will prompt you for your account password then for an SSH passphrase (it generates a public/private keypair to use for connecting to the OpenShift service). Next, create an entry for the app that we'll be deploying. Here you'll need to supply the app name (example, `expresscheckin`) and `jbossas-7.0` as the app type::

```
$ rhc app create -a expresscheckin -t jbossas-7
Password:
Creating application: expresscheckin
Now your new domain name is being propagated worldwide (this might take a minute)...
Warning: Permanently added 'expresscheckin-your-domain.rhcloud.com,50.16.164.248' (RSA) to the list of
known hosts.
Confirming application 'expresscheckin' is available:  Success!

expresscheckin published:  http://expresscheckin-your-domain.rhcloud.com/
git url:  ssh:
//429827960cbf4518b1785ed928db9be7@expresscheckin-your-domain.rhcloud.com/~/git/expresscheckin.git/
Successfully created application: expresscheckin
```

After the app is created the command will also clone the app's repository locally. Before continuing, we need to set up MongoDB on OpenShift and also update our app's MongoDB connection information::

```
$ rhc app cartridge add -a expresscheckin -c mongodb-2.0

RESULT:

MongoDB 2.0 database added.  Please make note of these credentials:

       Root User: admin
   Root Password: 1wbLGYAmPgDM
   Database Name: expresscheckin
  Connection URL: mongodb://127.6.85.129:27017/

You can manage your new MongoDB by also embedding rockmongo-1.1
```

Now that we've added support for MongoDB to our app (on OpenShift) we'll need to take the credentials returned and update our `openshift-checkins` app configuration. Go back to the `openshift-checkins` directory and edit `CheckinServlet.java` and add/update the following lines in the `init` function using the MongoDB details provided by OpenShift (don't forget to uncomment the `if` statement with the authentication statement)::

```
$ cd openshift-checkins
$ nano src/main/java/CheckinServlet.java
...
conn = new Mongo("127.6.85.129", 27017);
db = conn.getDB("expresscheckin");

if (db.authenticate("admin", "1wbLGYAmPgDM".toCharArray())) {
  throw new MongoException("unable to authenticate");
}
...
```

Then build the `openshift-checkins` WAR file::

```
$ mvn package
```

Now, return to the cloned repository, remove the sample code generated from the `expresscheckin` app repo that was cloned to our system and copy the `checkins.war` file into the `deployments` as `ROOT.war`::

```
$ cd ../expresscheckin
$ rm -rf pom.xml src
$ cp ../openshift-checkins/target/checkins.war deployments/ROOT.war
```

As part of the deployment process, we also need to flag our WAR file to be deployed (expanded and copied to the right places)::

```
$ touch deployments/ROOT.war.dodeploy
```

Now we can add it to the repository, commit and push::

```
$ git add -A
$ git commit -m "initial deployment of expresscheckin app onto OpenShift Express"
$ git push origin
```

After pushing the app, it will take a few minutes for the app to become available at http://expresscheckin-your-domain.rhcloud.com/. That's it, you've deployed a simple Java app that uses MongoDB to OpenShift Express. Refer to Testing the Deployment below for some notes on testing the app's functionality.

Testing the Deployment

Once you've deployed your app onto OpenShift Express, the URL for the app will be something of of the form http://expresscheckin-your-domain.rhcloud.com. We'll now use that URL to conduct some tests on our deployed app.

Since the app is a simple RESTish mechanism we can use `curl` to test it out. Let's start by posting a new comment and location to the URL (ex. http://appurl.rhcloud.com/checkin)::

```
$ curl -X POST -d "comment=hello&x=1&y=1" http://appurl.rhcloud.com/checkin
```

Now let's see if we can find the comment we just posted at that location (x = 1, y = 1)::

```
$ curl "http://appurl.rhcloud.com/checkin?x=1&y=1"
{ "_id" : { "$oid" : "4f0e068c3004bfd40822840b"} , "comment" : "hello" , "location" : [ 1.0 , 1.0]}
```

If these worked, then it looks like we've got a functional app deployed onto OpenShift.

Advanced Functionality

Once you've deployed your app, you'll probably need to connect to your server at some point so you can do things like reviewing app logs or query data in MongoDB. The following steps will cover how you can do that (and if available, set up some advanced functionality).

App Administration

Using the command line tools, we can connect to the server our app is deployed on. Run the following command to get information about the current running apps::

```
$ rhc domain show
Password:

User Info
=========
Namespace: checkins
  RHLogin: sandeep@clusterbeep.org


Application Info
================
expresscheckin
    Framework: jbossas-7
     Creation: 2012-04-14T14:04:54-04:00
         UUID: 485daa768043454d9cdcb9343018eb6e
      Git URL: ssh:
//485daa768043454d9cdcb9343018eb6e@expresscheckin-checkins.rhcloud.com/~/git/expresscheckin.git/
Public URL: http://expresscheckin-checkins.rhcloud.com/

 Embedded:
      mongodb-2.0 - Connection URL: mongodb://127.6.85.129:27017/
```

The `UDID` above shows the user ID we can use to SSH into our server::

```
$ ssh 429827960cbf4518b1785ed928db9be7@expresscheckin-your-domain.rhcloud.com

Welcome to OpenShift shell

This shell will assist you in managing openshift applications.

!!! IMPORTANT !!! IMPORTANT !!! IMPORTANT !!!
Shell access is quite powerful and it is possible for you to
accidentally damage your application.  Procede with care!
If worse comes to worse, destroy your application with rhc-ctl-app
and recreate it
!!! IMPORTANT !!! IMPORTANT !!! IMPORTANT !!!

type "help" for more info.
```

Now at the prompt type `help` to see the available commands (in addition to normal shell commands)::

```
[openshift]$ help
Help menu: The following commands are available to help control your openshift
application and environment.

ctl_app         control your application (start, stop, restart, etc)
ctl_all         control application and deps like mysql in one command
tail_all        tail all log files
export          list available environment variables
rm              remove files / directories
ls              list files / directories
ps              list running applications
kill            kill running applications
mongo           interactive MongoDB shell
```

To connect to MongoDB, we'll need to use the credentials provided to us from OpenShift above when we set it up initially. Once you have those in hand, here's how you should connect to the database::

```
[openshift]$ mongo expresscheckin -u admin -p 1wbLGYAmPgDM
MongoDB shell version: 2.0.2-rc1
connecting to: 127.1.13.1:27017/expresscheckin
>
```

From here you can query your data as needed for your application. Finally, to backup your app's data, follow the instructions found on the MongoDB Backup docs. Specifically you'll need to use the mongodump command to do a live backup of your data.

Backing up your app is a simple single step where we create a ***snapshot*** of the entirety of the application including data, which we can do locally on our development machine::

```
$ rhc app snapshot save -a expresscheckin
Password:
Pulling down a snapshot to expresscheckin.tar.gz

Running extra dump: mongodb_dump.sh
MongoDB already running
Stopping application...
Done
Creating and sending tar.gz
Running extra cleanup: mongodb_cleanup.sh
Starting application...
Done
```

Additional Information

For additional information, refer to the following:

- OpenShift Express User Guide or
- OpenShift Flex User Guide

VMware CloudFoundry

MongoDB is a supported service on VMware's Cloud Foundry.

Starting a MongoDB service

```
vmc create-service mongodb --name MyMongoDB
```

Once you create a MongoDB service, you can bind and use it inside of Cloud Foundry applications.

Developing applications with MongoDB and Cloud Foundry

- Developing applications with Java
- Developing applications with Ruby
- Developing applications with Node.js

See Also

- Getting started with VMware CloudFoundry, MongoDB and Rails
- Getting started with VMware Cloud Foundry, MongoDB and Node.js
- VMware Cloud Foundry with MongoDB webinar

Monitoring and Diagnostics

- mongostat
- mongotop
- Query Profiler
- Http Console
- mongo Shell Diagnostic Commands
- Trending/Monitoring Adaptors
- Hosted Monitoring
- Database Record/Replay (diagLogging command)
- Additional Resources

- Checking Server Memory Usage
- collStats Command
- Database Profiler
- mongoperf
- Munin configuration examples
- serverStatus Command
- Http Interface
- mongostat
- mongosniff

* Admin UIs

mongostat

mongostat is a great utility which exposes many internal MongoDB metrics. For any MongoDB related issues it is a good start for the analysis of performance issues.

mongotop

- http://docs.mongodb.org/manual/reference/mongotop/

Query Profiler

Use the Database Profiler to analyze slow queries.

db.currentOp() is another way to get a snapshot of what is currently happening.

Http Console

The mongod process includes a simple diagnostic screen at http://localhost:28017/. See the Http Interface docs for more information.

mongo Shell Diagnostic Commands

- db.serverStatus()
  - See the serverStatus Command page.
- db.stats()
  - Stats on the current database. Command takes some time to run, typically a few seconds unless the .ns file is very large (via use of --nssize). While running other operations may be blocked.
  - fileSize is the total size of all files allocated for the db.
- db.foo.find().explain()
  - explain plan
- help
  - db.help()
  - db.foo.help()

Trending/Monitoring Adaptors

- munin
  - Server stats: this will retrieve server stats (requires python; uses http interface)
  - Collection stats, this will display collection sizes, index sizes, and each (configured) collection count for one DB (requires python; uses driver to connect)
- Ganglia:
  - ganglia-gmond
  - mongodb-ganglia
- cacti
- Mikoomi provides a MongoDB plugin for Zabbix
- Nagios
- mtop - A top like utility for Mongo
- Mongo Live - A Chrome extension that provides a real-time server status view (uses the rest interface).

Chris Lea from (mt) Media Temple has made an easy to install Ubuntu package for the munin plugin.

Hosted Monitoring

- MongoDB Monitoring Service (MMS) is a free hosted monitoring tool for MongoDB provided by 10gen
- Server Density provides hosted monitoring for your hardware and software infrastructure, and supports a number of status checks for MongoDB.
- Cloudkick
- scout app slow queries
- AppFirst

Database Record/Replay (diagLogging command)

Recording database operations, and replaying them later, is sometimes a good way to reproduce certain problems in a controlled environment.

To enable logging:

```
db._adminCommand( { diagLogging : 1 } )
```

To disable:

```
db._adminCommand( { diagLogging : 0 } )
```

Values for diagLogging:

- 0 off. Also flushes any pending data to the file.
- 1 log writes
- 2 log reads
- 3 log both
  Note: if you log reads, it will record the findOnes above and if you replay them, that will have an effect!

Output is written to diaglog.bin_ in the /data/db/ directory (unless --dbpath is specified).

To replay the logged events:

```
nc ''database_server_ip'' 27017 < ''somelog.bin'' | hexdump -c
```

Additional Resources

- Monitoring MongoDB
- Server Status Reference
- Database Stats Reference
- Collection Stats Reference

Checking Server Memory Usage

- How Caching Works
- Memory Mapped Files
- Windows
- Virtual Memory Size
- Swap
  - Always Have Swap

How Caching Works

See Caching

Memory Mapped Files

Depending on the platform you may see the mapped files as memory in the process (see the Virtual Memory section below), but this is not strictly correct. If mapped files are counted as process memory, the `top` utility may show way more memory for `mongod` than is really appropriate for what is in physical memory.

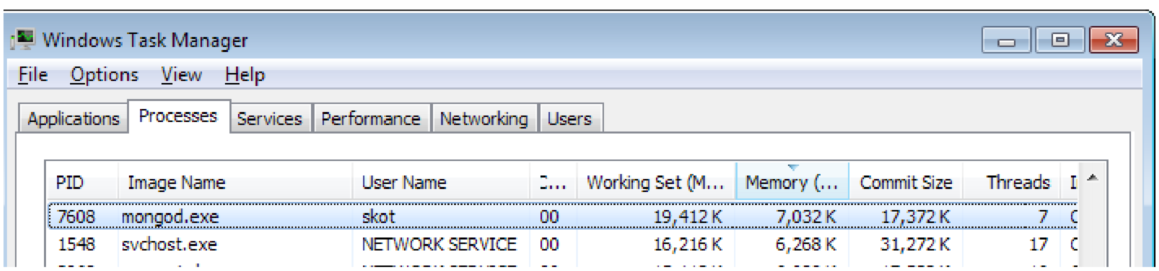
The operating system manages the memory where the memory mapped files reside (depending on the OS, this is usually done by the virtual memory manager). You can usually see mapped files using a program like `free -lmt`.

Memory mapped files are shown under "cached" memory:

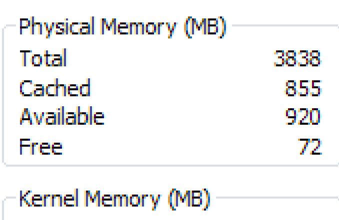
```
skot@stump:~$ free -tm
             total       used       free     shared    buffers     cached
Mem:          3962       3602        359          0        411       2652
-/+ buffers/cache:        538       3423
Swap:         1491         52       1439
Total:        5454       3655       1799
```

Windows

By bringing up the Task Manager you can see the process memory allocation for mongod.



In addition in the Performance tab the "cached" memory which represents the memory allocated by the memory mapped (data) files.



Virtual Memory Size

Virtual Memory includes the mapped memory files which can be much larger then the actual physical (or swap) available on your host. If journaling is enabled then the database files will be mapped twice leading to much higher virtual memory allocation than might be obvious.

It is important to remember that only a portion of the virtual memory number is really in physical memory. There are many utilities which will report memory usage as the virtual memory number which can be misleading.

Read below for more metrics about virtual memory allocation and how to detect related problems.

Swap

Since the database files are memory mapped and should constitute most of your mongodb memory usage it is very unlikely that mongod with ever use any swap space. The reason for this is because any of the memory mapped files can simply either be released from memory (without going to swap) to free memory for other programs or written back to the database files which means they never need to be "swapped" out to disk in the swap space (since they are already backed by files).

If you see continuous swap usage then it most likely means one of these things:

- Your system is extremely constrained by memory (see free -ltm "cached" to confirm)
- There is a memory leak (like the javascript engine, or lots of no_timeout cursors never being closed, or other internal data structures piling up)
- Some other program is stealing lots of memory

Always Have Swap

Because of these possible conditions it is always good to have some swap space available on your system. Think of the swap space as something like a steam release valve which allows excess pressure to release without blowing the system up.

Commands

The serverStatus() command provides memory usage information. Shell example:

```
> db.serverStatus()
> db.serverStatus().mem
> db.serverStatus().extra_info
```

One can verify there is no memory leak in the mongod process by comparing the mem.virtual and mem.mapped values (these values are in megabytes). If you are running with journaling disabled, the difference should be relatively small compared to total RAM on the machine. If you are running with journaling enabled, compare mem.virtual to 2*mem.mapped.   Also watch the delta over time; if it is increasing consistently, that could indicate a leak.

The mem.mapped value reflects the size of all databases currently open. When replication is on, this includes the size of the local database which includes the oplog. When journaling is enabled each file is mapped twice, once as a writable memmapped view and once as a protected view. The total amount of RAM used is roughly the same; the larger virtual memory size should not be a cause for concern.

One large component of the difference between mem.virtual and mem.mapped (or 2*mem.mapped when journaling is enabled) can be stack memory. In particular, each connection that you have open has a stack frame. The size of each stack frame is determined by the stack size; in Linux this typically defaults to 8MB, which means that each connection will use 8MB on the server. If you are using many connections and are concerned with memory usage you should reduce the stack size to 1MB (this is automatic in the upcoming v2.0 release).

On Linux, extra_info will contain information about the total heap size in a heap bytes field.

You can also see the virtual size and mapped values in the mongostat utility's output.

While increasing virtual size can indicate a memory leak, increasing resident size (ie, what is reflected as RES in the output of top) indicates that the operating system is using a larger portion of available memory to hold mongodb data, which often occurs under normal operations as a system warms up.

Note: OS X includes the operating system image in virtual size (~2GB on a small program). Thus interpretation of the values on OS X is a bit harder.

Working Set Size

In MongoDB it is fine if databases (and thus virtual size) are much larger than ram (terabytes for example); however, you will want your *working set* to stay in memory to achieve good performance. Otherwise lots of random disk IO's will occur, and unless you are using SSD, this can be quite slow. One area to watch specifically in managing the size of your working set is index access patterns. If you are inserting into indexes at random locations (for example, with id's which are effectively randomly generated by hashes), you will continually be updating the whole index. If instead you are able to create your id's in approximately ascending order (for example, day concatenated with a random id), all the updates will occur at the right side of the b-tree and the working set size for index pages will be much smaller.

Eatmem utility

Measuring working set size can be difficult; even if it is much smaller than total RAM, if the db has been up for a while and the db is much larger than RAM in total, all memory will be indicated as in use for the cache. Thus we need some other way to estimate our working set size.

One technique is to use a utility which reserves a certain amount of system memory for itself. So one could run this with a certain amount specified and see if the server continues to perform well. If not, the working set is largely than (total_ram - eaten_ram). Note this test will eject some data from the file system cache which may take time to page back in after the `eatmem` utility is terminated.

Running eatmem continuously with a small percentage of total RAM (say, 20%) is a good technique to get an "early warning" of memory being too low. If disk i/o activity increases significantly, terminate eatmem to migigate the problem for the moment until further steps can be taking.

Asymmetry

In replica sets if one server is underpowered this could one again help as an early warning mechanism for server capacity. Of course the server must be receiving representative traffic to get an indication here.

Unix Utilities

`mongod` uses memory-mapped files; thus the memory stats in top require interpretation in a special way. On a large database, virtual bytes/VSIZE will tend to be the size of the entire database, and if the server doesn't have other processes running, resident bytes/RSIZE will be the total memory of the machine (as this counts file system cache contents).

`vmstat` can be useful – try running `vmstat 2`. on OS X, just `vm_stat`.

Historical Memory Leak Bugs

(21 issues)			
Key	**Status**	**FixVersion**	**Summary**
SERVER-2497	Closed	1.7.6	memory leak in ModSet::prepare()
SERVER-2522	Closed	1.7.6	State::reduceInMemory leaks InMemory on exception
SERVER-3522	Closed	2.1.0	Memory Leak in dbtests/perftests.cpp
SERVER-4008	Closed	2.1.0	A memory leak per line of typed text in the shell
SERVER-2520	Closed	1.9.1	add comments in code about intentional memory leak in ReplSetImpl::initFroı
SERVER-3911	Resolved		memory leak with journaling in windows
SERVER-2122	Closed		memory leak of shard + replication
SERVER-2511	Closed	1.9.0	memory leak of _MultiCommandJob
SERVER-1827	Closed		Memory leak when there's multiple query plans with empty result
SERVER-1897	Closed		admin page plugins and handlers leak memory
SERVER-5589	Open		Possible memory leak in Linux 64-bit server
SERVER-768	Closed	1.3.4	Memory leak and high memory usage from snapshots thread
SERVER-774	Closed		MessagingPorts are leaking
SERVER-4089	Closed		Memory leak
SERVER-2558	Closed	1.8.0-rc0	memory and cursor leak in FindingStartCursor
SERVER-4354	Closed		Waiting mongo process leaks memory.
SERVER-2498	Open	Planning Bucket B	small memory leak when closing a database
SERVER-2521	Open	Planning Bucket A	1300 byte memory leak in PiggyBackData on socket exception
SERVER-3157	Closed		Replicaset becomes inaccessable and instable after mapreduce job
SERVER-5356	Open	debugging with submitter	mongos OOM
SERVER-4127	Closed	debugging with submitter	appears to be a memoryleak in mongodump

See Also

- The Linux Out of Memory OOM Killer

collStats Command

Statistics on a collection. The command name is "collStats" and the shell helper is *database.collection*.stats().

```
> db.commandHelp("collStats")
help for: collStats { collStats:"blog.posts" , scale : 1 } scale divides sizes e.g. for KB use 1024
```

In the shell:

```
> // to see params etc.:
> db.foo.stats
function (scale) {
    return this._db.runCommand({collstats:this._shortName, scale:scale});
}

// to run:
> db.foo.stats()
{
        "ns" : "test.foo",
        "count" : 9,           // number of documents
"size" : 432,          // collection size
"avgObjSize" : 48,     // average object size in bytes
"storageSize" : 3840,  // (pre)allocated space for the collection
"numExtents" : 1,      // extents are contiguously allocated
// chunks of datafile space
"nindexes" : 2,
        "lastExtentSize" : 3840,
        "paddingFactor" : 1,    // padding can make updates faster
// if documents grow
"flags" : 1,
        "totalIndexSize" : 16384,
        "indexSizes" : {
                "_id_" : 8192,
                "x_1" : 8192
        },
        "ok" : 1
}
```

- Slave ok : true
- Lock type : read
- Slow to run : no

Database Profiler

Mongo includes a profiling tool to analyze the performance of database operations.

See also the currentOp command.

Enabling Profiling

Through the profile command

You can enable and disable profiling from the mongo shell, or through a driver, via the profile command.

```
> db.commandHelp("profile") // see how to run from drivers
```

When using the profile command, profiling is enabled or disabled per database. A system.profile collection will be created for the database.

To enable profiling, from the `mongo` shell invoke:

```
> db.setProfilingLevel(2);
{"was" : 0 , "slowms" : 100, "ok" : 1} // "was" is the old setting
> db.getProfilingLevel()
2
```

Profiling levels are:

- 0 - off
- 1 - write slow operations (by default, >100ms is considered slow) to the system.profile collection
- 2 - write all operations to the system.profile collection

In addition to the default levels you can also specify a `slowms` option:

```
> db.setProfilingLevel(1,20) // log slow operations, slow threshold=20ms
> db.getProfilingStatus() // new shell helper method as of v1.7+
{ "was" : 1, "slowms" : 20 }
```

Note: the profiling level controls which operations get written to the system.profile collection. However, even if the profiler is off, queries slower than the "slowms" level will get written to the logs.

Through the command-line/config-file

You can also enable profiling on the command line; for example:

```
$ mongod --profile=1 --slowms=15
```

Using with Sharding

Enabling and aggregating profile data globally is not yet available. For now, connect directly to a `mongod` of interest with the shell, and follow the instructions on this page. You can then repeat the same procedure on other shards if necessary.

Viewing the Data

Profiling data is recorded in the database's `system.profile` collection. Query that collection to see the results.

> Because profile data is written to `system.profile`, you will see write activity on the database, even if only reading, because of the writes to `system.profile`.

```
> db.system.profile.find()
{"ts" : "Thu Jan 29 2009 15:19:32 GMT-0500 (EST)" , "info" : "query test.$cmd ntoreturn:1 reslen:66
nscanned:0  <br>query: { profile: 2 }  nreturned:1 bytes:50" , "millis" : 0}
...
```

Filtering example

As an example, to see output without $cmd (command) operations, invoke:

```
db.system.profile.find( function() { return this.info.indexOf('$cmd')<0; } )
```

Likewise we could query for indexOf(...)>=0 to see only those lines.

View stats for only one collection example

To view operations for a particular collection:

```
> db.system.profile.find( { info: /test.foo/ } )
{"ts" : "Thu Jan 29 2009 15:19:40 GMT-0500 (EST)" , "info" : "insert test.foo" , "millis" : 0}
{"ts" : "Thu Jan 29 2009 15:19:42 GMT-0500 (EST)" , "info" : "insert test.foo" , "millis" : 0}
{"ts" : "Thu Jan 29 2009 15:19:45 GMT-0500 (EST)" , "info" : "query test.foo ntoreturn:0 reslen:102
nscanned:2  <br>query: {}  nreturned:2 bytes:86" , "millis" : 0}
{"ts" : "Thu Jan 29 2009 15:21:17 GMT-0500 (EST)" , "info" : "query test.foo ntoreturn:0 reslen:36
nscanned:2  <br>query: { $not: { x: 2 } }  nreturned:0 bytes:20" , "millis" : 0}
{"ts" : "Thu Jan 29 2009 15:21:27 GMT-0500 (EST)" , "info" : "query test.foo ntoreturn:0 exception
bytes:53" , "millis" : 88}
```

View slow operations only

To view operations slower than a certain number of milliseconds:

```
> db.system.profile.find( { millis : { $gt : 5 } } )
{"ts" : "Thu Jan 29 2009 15:21:27 GMT-0500 (EST)" , "info" : "query test.foo ntoreturn:0 exception
bytes:53" , "millis" : 88}
```

To see newest information first

```
> db.system.profile.find().sort({$natural:-1})
```

To view information from a certain time range

```
> db.system.profile.find(
...{ts:{$gt:new ISODate("2011-07-12T03:00:00Z"),
...     $lt:new ISODate("2011-07-12T03:40:00Z")}
...})
```

In the next example we look at the time range, suppress the `user` field from the output to make it easier to read, and sort the results by how long each operation took to run.

```
> db.system.profile.find(
...{ts:{$gt:new ISODate("2011-07-12T03:00:00Z"),
...     $lt:new ISODate("2011-07-12T03:40:00Z")}
...}
...,{user:0}).sort({millis:-1})
```

The show `profile` shell helper

The mongo shell includes a helper to see the most recent 5 profiled events that took at least 1ms to execute. Type

```
show profile
```

at the command prompt to use this feature.

Understanding the Output

The output reports the following values:

- `ts` Timestamp of the profiled operation.
- `millis` Time, in milliseconds, to perform the operation. This time does not include time to acquire the lock or network time, just the time for the server to process.
- `info` Details on the operation.
    - `query` A database query operation. The query info field includes several additional terms:
        - `ntoreturn` Number of objects the client requested for return from a query. For example, <code>findOne()</code> sets ntoreturn to 1. <code>limit()</code> sets the appropriate limit. Zero indicates no limit.
        - `query` Details of the query spec.
        - `nscanned` Number of objects scanned in executing the operation.
        - `reslen` Query result length in bytes.
        - `nreturned` Number of objects returned from query.
    - `update` A database update operation. <code>save()</code> calls generate either an update or insert operation.
        - `fastmod` Indicates a fast modify operation. See Updates. These operations are normally quite fast.
        - `fastmodinsert` - indicates a fast modify operation that performed an upsert.
        - `upsert` Indicates on upsert performed.
        - `moved` Indicates the update moved the object on disk (not updated in place). This is slower than an in place update, and normally occurs when an object grows.
        - `key updates` How many index keys changed during the update. Key updates are a little bit expensive since the db needs to remove the old key and insert a new key into the b-tree index.
    - `insert` A database insert.
    - `getmore` For large queries, the database initially returns partial information. `getmore` indicates a call to retrieve further information.

Optimizing Query Performance

- If `nscanned` is much higher than `nreturned`, the database is scanning many objects to find the target objects. Consider creating an index to improve this.

- `reslen` A large number of bytes returned (hundreds of kilobytes or more) causes slow performance. Consider passing <code>find()</code> a second parameter of the member names you require.

`Note:` There is a cost for each index you create. The index causes disk writes on each insert and some updates to the collection. If a rare query, it may be better to let the query be "slow" and not create an index. When a query is common relative to the number of saves to the collection, you will want to create the index.

Optimizing Update Performance

- Examine the `nscanned` info field. If it is a very large value, the database is scanning a large number of objects to find the object to update. Consider creating an index if updates are a high-frequency operation.
- Use fast modify operations when possible (and usually with these, an index). See Updates.

Profiler Overhead

When enabled, profiling affects performance, although not severely.

Profile data is stored in the database's `system.profile` collection, which is a Capped Collection. By default it is set to a very small size and thus only includes recent operations.

Profiling and Replication

In v1.9+, you can use profiling on secondaries in addition to the current primary. In older versions of MongoDB, use profiling on the primary only.

Changing the system.profile Collection Size

Profiling information is written to the system.profile capped collection. There is a separate profile collection per database. By default the collection is very small and like all capped collections works in a rotating RRD-like style. To make it bigger you can create it explicitly. You will need to drop it first; you may need to disable profiling before the drop/recreate. Example in the shell:

```
> db.system.profile.drop()
> db.createCollection("system.profile", {capped:true, size:4000000})
> db.system.profile.stats()
```

Alternatives to Profiling

The profiler can generate write locks as it writes to the profile collection. Thus other tools to consider for optimizing queries are:

1. Running db.currentOp(), perhaps many times in a row to get a good sample;
2. Using the explain() helper in the shell

See Also

- Optimization
- explain()
- Viewing and Terminating Current Operation

mongoperf

mongoperf is a utility for checking disk i/o performance of a server independent of MongoDB. It performs simple timed random disk i/o's.

The utility is new and will likely be more sophisticated in the future.

```
# get help:
mongoperf -h

# example invocation:
echo "{nThreads:16,fileSizeMB:1000,r:true}" | ./mongoperf | tee out
```

Building

Run `scons mongoperf` to build.

mmf:false mode (default mode)

In the default mode of operation, random 4KB direct disk i/o's are performed (i.e., O_DIRECT is used on Linux). Thus this is a physical disk i/o test.

Example run

mmf:true mode

If `mmf:true` is specified as an option, tests are performed using memory-mapped files. These files are openedin a "normal" fashion and thus caching is allowed. This sometimes can be used to test file system cache behavior with memory mapped files.

syncDelay option

The `syncDelay:secs` option instructs mongoperf to perform an asynchronous fsync of the test mmap file at the specified interval. mongod does something similar every 60 seconds, thus this can be useful to test basic system behavior in a simpler setting. This option is applicable only when using `mmf:true` mode.

fileSizeMB option

This specifies the size of the test data file; you will need this much free disk space. The file will be placed in the current directory.

Specify a very large test file size to create a realistic simulation. A very small file (1MB) could be cached by your disk controller compltely. A file that is say, 100MB in size would involve a small number of disk cylinders – track-to-track seeks are much faster than a drive's average seek time.

The file size is particularly important when `mmf:true` is specified, as the file system cache is then involved. A file size much larger than RAM will result in much different performance results than a file size smaller than RAM.

See Also

As the utility is quite simple you might wish to also take a look at the source code.

Munin configuration examples

Overview

Munin can use be used for monitoring aspects of a running system. The following is a mini tutorial to help you set up and use the MongoDB plugin with munin.

Setup

Munin is made up of two components

- agent and plugins that are installed on the system you want to monitor
- server which polls the agent(s) and creates the basic web pages and graphs to visualize the data

Install

You can download from SourceForge , but prebuilt packages are also available. For example on Ubuntu you can do the following:

Agent install

To install the agent, repeat the following steps on each node you want to monitor.

```
shell> sudo apt-get install munin-node
```

Server install

The server needs to be installed once. It relies on apache2, so you will need to ensure that it is installed as well.

```
shell> apt-get install apache2
shell> apt-get install munin
```

Configuration

Both the agent(s) and server need to be configured with the IP address and port to contact each other. In the following examples we will use these nodes:

- db1 : 10.202.210.175
- db2 : 10.203.22.38
- munin-server : 10.194.102.70

Agent configuration

On each node, add an entry as follows into
for db1:

```
/etc/munin/munin-node.conf
host_name db1-ec2-174-129-52-161.compute-1.amazonaws.com
allow ^10\.194\.102\.70$
```

for db2:

```
/etc/munin/munin-node.conf
host_name db2-ec2-174-129-52-161.compute-1.amazonaws.com
allow ^10\.194\.102\.70$
```

* host_name : can be whatever you like, this name will be used by the server

- allow : this is the IP address of the server, enabling the server to poll the agent

Server configuration

Add an entry for each node that is being monitored as follows in

```
[db1-ec2-174-129-52-161.compute-1.amazonaws.com]
address 10.202.210.175
use_node_name no

[db2-ec2-184-72-191-169.compute-1.amazonaws.com]
address 10.203.22.38
use_node_name no
```

* the name in between the [] needs to match the name set in the agents munin-node.conf

- address : IP address of the node where the agent is running
- use_node_name : dtermine if the IP or the name between [] is used to contact the agent

MongoDB munin plugin

A plugin is available that provide metrics for

- B-Tree stats
- Current connections
- Memory usage
- Database operations (inserts, updates, queries etc.)

The plugin can be installed as follows on each node where MongoDB is running

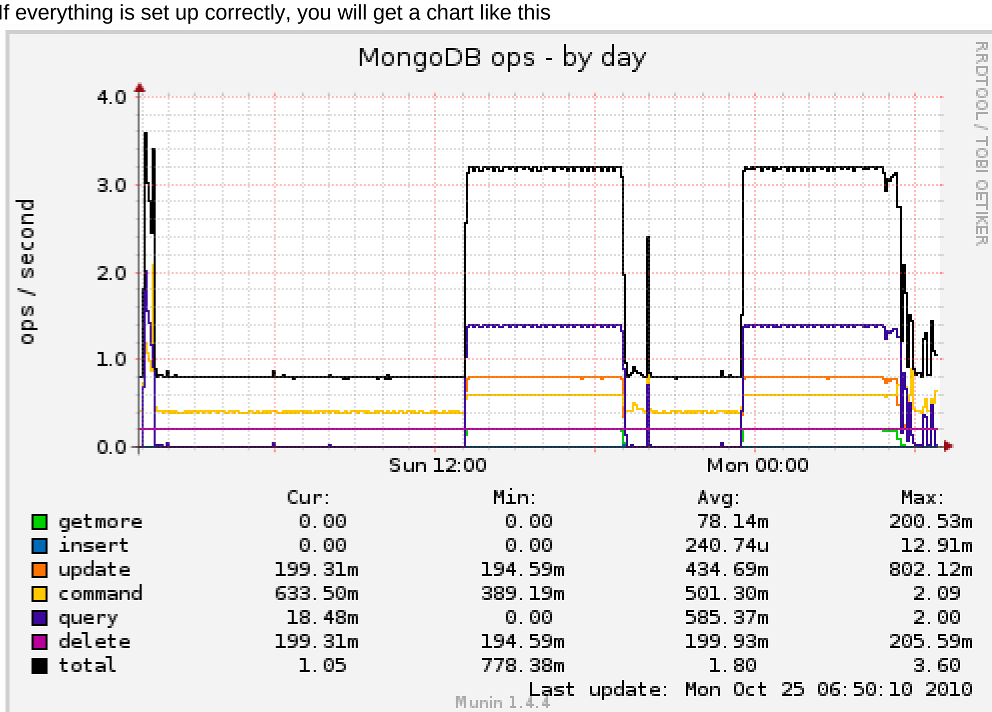
```
shell> wget http://github.com/erh/mongo-munin/tarball/master
shell> tar xvf erh-mongo-munin-*tar.gz
shell> cp erh-mongo-munin-*/mongo_* /etc/munin/plugins/
```

Check your setup

After installing the plugin and making the configuration changes, force the server to update the information to check your setup is correct using the following

```
shell> sudo -u munin /usr/share/munin/munin-update
```

If everything is set up correctly, you will get a chart like this



Advanced charting

If you are running a large MongoDB cluster, you may want to aggregate the values (e.g. inserts per second) across all the nodes in the cluster. Munin provides a simple way to aggregate.

```
/etc/munin/munin.conf
[compute-1.amazonaws.com;CLUSTER]
update no
```

* Defines a new segment called CLUSTER

- update no : munin can generate the chart based on existing data, this tell munin not to poll the agents for the data

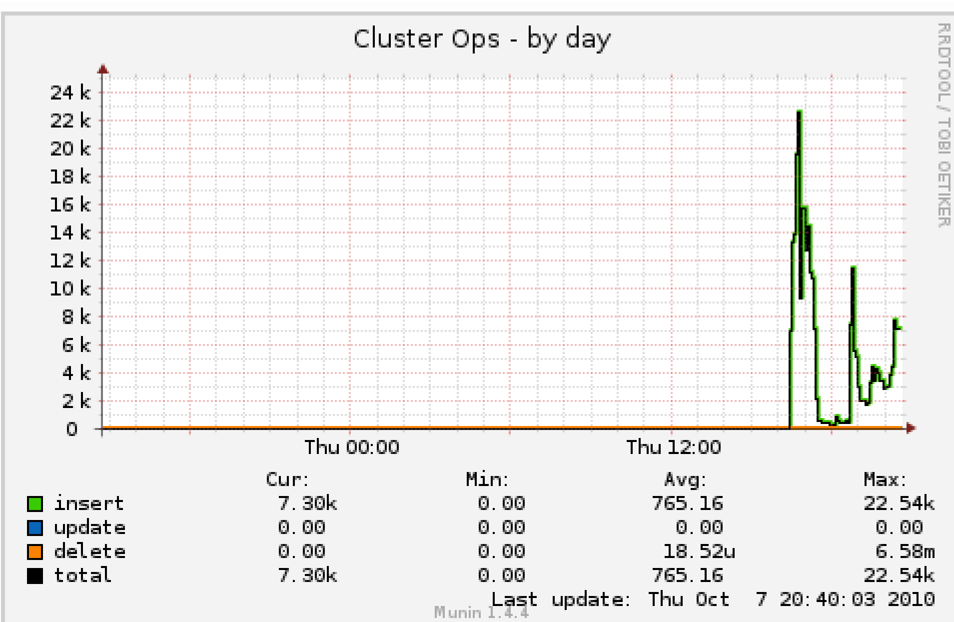
Now lets define a chart to aggregate the inserts, updates and deletefor the cluster

```
cluster_ops.graph_title Cluster Ops
cluster_ops.graph_category mongodb
cluster_ops.graph_total total
cluster_ops.total.graph no
cluster_ops.graph_order insert update delete
cluster_ops.insert.label insert
cluster_ops.insert.sum \
  db1-ec2-174-129-52-161.compute-1.amazonaws.com:mongo_ops.insert \
  db2-ec2-184-72-191-169.compute-1.amazonaws.com:mongo_ops.insert
cluster_ops.update.label update
cluster_ops.update.sum \
  db1-ec2-174-129-52-161.compute-1.amazonaws.com:mongo_ops.update \
  db2-ec2-184-72-191-169.compute-1.amazonaws.com:mongo_ops.update
cluster_ops.delete.label delete
cluster_ops.delete.sum \
  db1-ec2-174-129-52-161.compute-1.amazonaws.com:mongo_ops.delete \
  db2-ec2-184-72-191-169.compute-1.amazonaws.com:mongo_ops.delete
```

* cluster_ops : name of this chart

- cluster_ops.graph_category mongodb : puts this chart into the "mongodb" category. Allows you to collect similar charts on a single page
- cluster_ops.graph_order insert update delete : indicates the order of the line son the key for the chart
- cluster_ops.insert : represents a single line on the chart, in this case the "insert"
- cluster_ops.insert.sum : indicates the values are summed
  - db1-ec2-174-129-52-161.compute-1.amazonaws.com : indicates the node to aggregate
  - mongo_ops.insert : indicates the chart (mongo_ops) and the counter (insert) to aggregate

And this is what it looks like



serverStatus Command

The serverStatus command provides very useful diagnostic information for a mongod instance.

From the shell:

```
> db.commandHelp("serverStatus")
help for: serverStatus returns lots of administrative server statistics

> db.serverStatus()
```

Field	Example Value	Explanation
host	my.node.com	The hostname of this server
version	1.8.0-rc1-pre-	The version number of this server
process	mongod	What is the process? (mongod, mongos)
uptime	14143	Uptime in seconds
uptimeEstimate	12710	Uptime based on MongoDB's internal coarse grained timer
localTime	ISODate("2011-03-01T05:30:16.682Z")	The local time at this server (time is in UTC).
globalLock.totalTime	14143457121	The number of microseconds since the global lock was created
globalLock.lockTime	17166	The number of microseconds that the global lock has been held since it was created
globalLock.ratio	0.0000012137060870720337	The ratio between lockTime & totalTime
globalLock.currentQueue.total	12	The current number of operations queued waiting for the global lock
globalLock.currentQueue.readers	10	The current number of operations queued waiting on a read lock
globalLock.currentQueue.writers	2	The current number of operations queued waiting for a write lock
globalLock.activeClients.total	3	Total number of active clients connected to this server
globalLock.activeClients.readers	2	The total number of active clients currently performing read operations
globalLock.activeClients.writers	1	The total number of active clients currently performing write operations
mem.bits	64	Is this a 32 or 64 bit architecture?
mem.resident	20	number of megabytes resident. It is typical over time, on a dedicated database server, for this number to approach the amount of physical ram on the box.
mem.virtual	2502	virtual megabytes for the mongod process. Generally virtual should be a little larger than mapped, but if virtual is many gigabytes larger, that could indicate a memory leak - with journalling, virtual is twice mapped
mem.supported	true	Whether or not this machine supports extended memory info. If this is false, other values in 'mem' may not be present
mem.mapped	80	Megabytes of data mapped by the database. As MongoDB memory maps all the data files, this number is likely similar to your total database(s) size.
connections.current	23	The number of currently active connections to this server
connections.available	50	The number of available connections remaining
extra_info.heap_usage_bytes	234342	The number of bytes of heap used by this process. Only available on linux

extra_info.page_faults	2523	The number of page faults on this process which required loading from disk. Only available on Linux
indexCounters.btree.accesses	2342	The number of times the btree indexes have been accessed
indexCounters.btree.hits	2340	The number of times a btree page was in memory
indexCounters.btree.misses	2	The number of times a btree page was not in memory
indexCounters.btree.resets	0	The number of times the index counters have been reset to 0
indexCounters.btree.missRatio	0	The ratio of misses to hits on the btree
backgroundFlushing.flushes	214	The number of times the database has flushed writes to disk
backgroundFlushing.total_ms	1357	The total number of ms that the database has spent flushing data to disk
backgroundFlushing.average_ms	6.341121495327103	The average number of ms it takes to perform a single flush
backgroundFlushing.last_ms	7	The number of ms that the last flush took to complete
backgroundFlushing.last_finished	ISODate("2011-03-01T05:29:44.124Z")	The timestamp from when the last flush was completed
cursors.totalOpen	0	The total number of cursors that the server is maintaining for clients
cursors.clientCursors_size	[deprecated] Same as cursors.totalOpen	
cursors.timedOut	0	The number of cursors that have timed out since this server was started
network.bytesIn	1430	The total number of bytes sent to this database
network.bytesOut	2140	The total number of bytes sent from this database
network.numRequests	20	The total number of requests that have been sent to this database
repl.setName	mySet	The name of the replica set that this server is a part of
repl.ismaster	true	Whether or not this node is the master of the replica set
repl.secondary	false	Whether or not this node is a secondary of the replica set
repl.hosts	["my.node.com:27017", "other.node.com:27017", "third.node.com:27017"]	The set of hosts in this replica sets
opcounters.insert	0	The total number of inserts performed since this process started
opcounters.query	9	The total number of queries performed since this process started
opcounters.update	0	The total number of updates performed since this process started
opcounters.delete	0	The total number of deletes performed since this process started
opcounters.getmore	0	The total number of times getMore has been called on any cursor since this process started
opcounters.command	13	The total number of other commands performed since this process started
asserts.regular	0	The number of regular asserts raised since this process started
asserts.warning	0	The number of warnings raised since this process started
asserts.msg	0	The number of message asserts. These are internal server errors that have a well defined text string. Stack traces are logged for these
asserts.user	0	The number of user asserts. These are errors that can be generated by a user such as out of disk space or duplicate key
asserts.rollovers	0	The number of times the assert counters have rolled over since this process started

writeBacksQueued	false	Whether or not there are any operations from a mongos that have to be retried
dur.commits	0	The number of commits to the journal that have occured in the last interval
dur.journaledMB	0	MBs of data written to the journal in the last interval
dur.writeToDataFilesMB	0	MBs of data written from journal to data files in the last interval
dur.commitsInWriteLock	0	The number of commits in the last interval which were in a write lock. Commits in a write lock are undesirable
dur.earlyCommits	0	Number of times a commit was requested before the scheduled time
dur.timeMs.dt	3011	The time length of the interval over which the dur statswere collected
dur.timeMs.prepLogBuffer	0	The amount of time spent preparing to write to the journal
dur.timeMs.writeToJournal	0	The amount of time spent actually writing to the journal
dur.timeMs.writeToDataFiles	0	The amount of time spent writing to datafiles after journaing
dur.timeMs.remapPrivateView	0	The amount of time spent remapping copy-on-write memory mapped views
ok	1	Whether or not serverStatus returned correctly

- Slave ok : true
- Lock type : none

Http Interface

REST Interfaces

Sleepy Mongoose (Python)

Sleepy Mongoose is a full featured REST interface for MongoDB which is available as a separate project.

MongoDB Rest (Node.js)

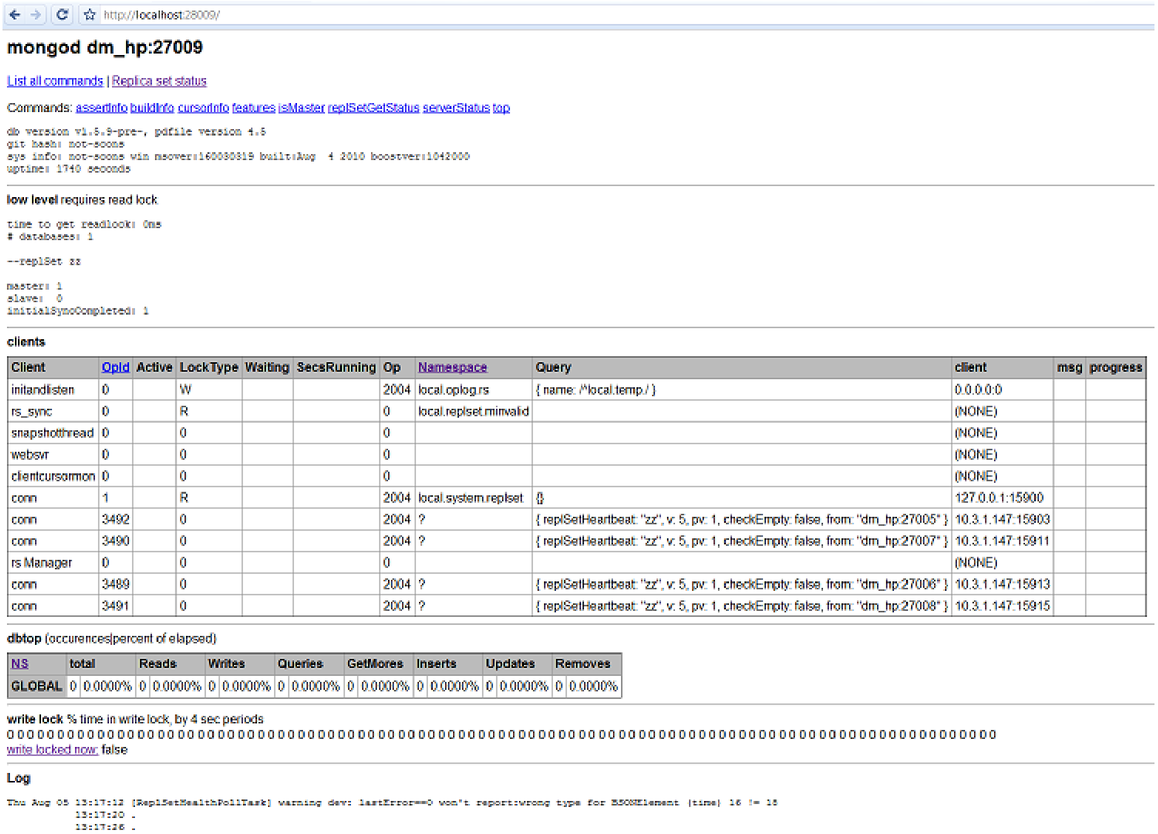
MongoDB Rest is an **alpha** REST interface to MongoDB, which uses the MongoDB Node Native driver.

Mongodb Java REST server

Mongodb Java REST server based on Jetty.

HTTP Console

MongoDB provides a simple http interface listing information of interest to administrators. This interface may be accessed at the port with numeric value 1000 more than the configured mongod port; the default port for the http interface is 28017. To access the http interface an administrator may, for example, point a browser to http://localhost:28017 if mongod is running with the default port on the local machine.



http://localhost:28009/

mongod dm_hp:27009

List all commands | Replica set status

Commands: assertInfo buildInfo cursorInfo features isMaster replSetGetStatus serverStatus top

```
db version v1.5.9-pre-, pdfile version 4.5
git hash: not-scons
sys info: not-scons win msover:160030319 built:Aug  4 2010 boostver:1042000
uptime: 1740 seconds
```

low level requires read lock

```
time to get readlock: 0ms
# databases: 1

--replSet zz

master: 1
slave:  0
initialSyncCompleted: 1
```

clients

Client	OpId	Active	LockType	Waiting	SecsRunning	Op	Namespace	Query	client	msg	progress
initandlisten	0		W			2004	local.oplog.rs	{ name: /^local.temp./ }	0.0.0.0:0		
rs_sync	0		R			0	local.replset.minvalid		(NONE)		
snapshotthread	0		0			0			(NONE)		
websvr	0		0			0			(NONE)		
clientcursormon	0		0			0			(NONE)		
conn	1		R			2004	local.system.replset	{}	127.0.0.1:15900		
conn	3492		0			2004	?	{ replSetHeartbeat: "zz", v: 5, pv: 1, checkEmpty: false, from: "dm_hp:27005" }	10.3.1.147:15903		
conn	3490		0			2004	?	{ replSetHeartbeat: "zz", v: 5, pv: 1, checkEmpty: false, from: "dm_hp:27007" }	10.3.1.147:15911		
rs Manager	0		0			0			(NONE)		
conn	3489		0			2004	?	{ replSetHeartbeat: "zz", v: 5, pv: 1, checkEmpty: false, from: "dm_hp:27006" }	10.3.1.147:15913		
conn	3491		0			2004	?	{ replSetHeartbeat: "zz", v: 5, pv: 1, checkEmpty: false, from: "dm_hp:27008" }	10.3.1.147:15915		

dbtop (occurences|percent of elapsed)

NS	total		Reads		Writes		Queries		GetMores		Inserts		Updates		Removes	
GLOBAL	0	0.0000%	0	0.0000%	0	0.0000%	0	0.0000%	0	0.0000%	0	0.0000%	0	0.0000%	0	0.0000%

write lock % time in write lock, by 4 sec periods
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
write locked now: false

Log

```
Thu Aug 05 13:17:12 [ReplSetHealthPollTask] warning dev: lastError==0 won't report:wrong type for BSONElement (time) 16 != 18
           13:17:20 .
           13:17:26 .
```

Here is a description of the informational elements of the http interface:

element	description
db version	database version information
git hash	database version developer tag
sys info	mongod compilation environment
dblocked	indicates whether the primary mongod mutex is held
uptime	time since this mongod instance was started
assertions	any software assertions that have been raised by this mongod instance
replInfo	information about replication configuration
currentOp	most recent client request
# databases	number of databases that have been accessed by this mongod instance
curclient	last database accessed by this mongod instance
Cursors	describes outstanding client cursors
master	whether this mongod instance has been designated a master
slave	whether this mongod instance has been designated a slave
initialSyncCompleted	whether this slave or repl pair node has completed an initial clone of the mongod instance it is replicating
DBTOP	Displays the total time the mongod instance has devoted to each listed collection, as well as the percentage of available time devoted to each listed collection recently and the number of reads, writes, and total calls made recently
dt	Timing information about the primary mongod mutex

HTTP Console Security

If security is configured for a mongod instance, authentication is required for a client to access the http interface from another machine.

Simple REST Interface

The mongod process includes a simple read-only (no support for insert/update/remove) REST interface for convenience. For full REST capabilities we recommend using an external tool such as Sleepy.Mongoose.

v1.4+: This interface is disabled by default. Use `--rest` on the command line to enable.

To get the contents of a collection (note the trailing slash):

```
http://127.0.0.1:28017/databaseName/collectionName/
```

To add a limit:

```
http://127.0.0.1:28017/databaseName/collectionName/?limit=-10
```

To skip:

```
http://127.0.0.1:28017/databaseName/collectionName/?skip=5
```

To query for {a : 1}:

```
http://127.0.0.1:28017/databaseName/collectionName/?filter_a=1
```

Separate conditions with an &:

```
http://127.0.0.1:28017/databaseName/collectionName/?filter_a=1&limit=-10
```

Same as `db.$cmd.findOne({listDatabase:1})` on the "admin" database in the shell:

```
http://localhost:28017/admin/$cmd/?filter_listDatabases=1&limit=1
```

To count documents in a collection:

```
http://host:port/db/$cmd/?filter_count=collection&limit=1
```

JSON in the simple REST interface

The simple ReST interface uses strict JSON (as opposed to the shell, which uses Dates, regular expressions, etc.). To display non-JSON types, the web interface wraps them in objects and uses the key for the type. For example:

```
# ObjectIds just become strings
"_id" : "4a8acf6e7fbadc242de5b4f3"

# dates
"date" : { "$date" : 1250609897802 }

# regular expressions
"match" : { "$regex" : "foo", "$options" : "ig" }
```

The code type has not been implemented yet and causes the DB to crash if you try to display it in the browser.

See Mongo Extended JSON for details.

See Also

- Replica Set Admin UI
- Diagnostic Tools

mongostat

Use the mongostat utility to quickly view statistics on a running mongod instance.

```
connected to: 127.0.0.1
insert  query update delete getmore command flushes mapped  vsize    res locked % idx miss %     qr|qw   ar|aw  netIn netOut  conn       time
     0      0      0      0       0       1       0     0m  2.36g     4m        0          0       0|0     0|0    62b     1k     1   17:25:35
     0      0      0      0       0       1       0     0m  2.36g     4m        0          0       0|0     0|0    62b     1k     1   17:25:36
     0      0      0      0       0       1       0     0m  2.36g     4m        0          0       0|0     0|0    62b     1k     1   17:25:37
     0      0      0      0       0       1       0     0m  2.36g     4m        0          0       0|0     0|0    62b     1k     1   17:25:38
```

Run *mongostat --help* for help.

```
Fields:
   insert       - # of inserts per second (* means replicated op)
   query        - # of queries per second
   update       - # of updates per second
   delete       - # of deletes per second
   getmore      - # of get mores (cursor batch) per second
   command      - # of commands per second (on a slave, it's local|replicated)
   flushes      - # of fsync flushes per second
   mapped       - amount of data mmaped (total data size) megabytes
   vsize        - virtual size of process in megabytes
   res          - resident size of process in megabytes
   faults       - # of pages faults/sec (linux only)
   locked       - percent of time in global write lock
   idx miss     - percent of btree page misses (sampled)
   qr | qw      - queue lengths for clients waiting (read|write)
   ar | aw      - active clients (read|write)
   netIn        - network traffic in - bits
   netOut       - network traffic out - bits
   conn         - number of open connections
   set          - replica set name
   repl         - replication type
                    M    - master
                    SEC  - secondary
                    REC  - recovering
                    UNK  - unknown
                    SLV  - slave
                    RTR  - router
```

multiple servers:

```
mongostat --host a,b,c
```

find all connected servers (v1.8+):

```
mongostat --discover (--host optional)
```

Note: When reporting statistics on a secondary or slave instance, replicated operations are marked with a preceding asterisk.

mongosniff

Unix releases of MongoDB include a utility called mongosniff. This utility is to MongoDB what tcpdump is to TCP/IP; that is, fairly low level and for complex situations. The tool is quite useful for authors of driver tools.

```
$ ./mongosniff --help
Usage: mongosniff [--help] [--forward host:port] [--source (NET <interface> | FILE <filename>)]
[<port0> <port1> ...]
--forward       Forward all parsed request messages to mongod instance at
                specified host:port
--source        Source of traffic to sniff, either a network interface or a
                file containing perviously captured packets, in pcap format.
                If no source is specified, mongosniff will attempt to sniff
                from one of the machine's network interfaces.
<port0>...      These parameters are used to filter sniffing.  By default,
                only port 27017 is sniffed.
--help          Print this help message.
```

Building

mongosniff is including in the binaries for Unix distributions. As mongosniff depends on libpcap, the MongoDB SConstruct only builds mongosniff if libpcap is installed.

```
$ # Red Hat
$ sudo yum install libpcap-devel
$
$ # Ubuntu/Debian
$ sudo apt-get install libpcap-dev
$
$ scons mongosniff
```

Example

To monitor localhost:27017, run `ifconfig` to find loopback's name (usually something like `lo` or `lo0`). Then run:

```
mongosniff --source NET lo
```

If you get the error message "error opening device: socket: Operation not permitted" or "error finding device: no suitable device found", try running it as root.

Other Tools

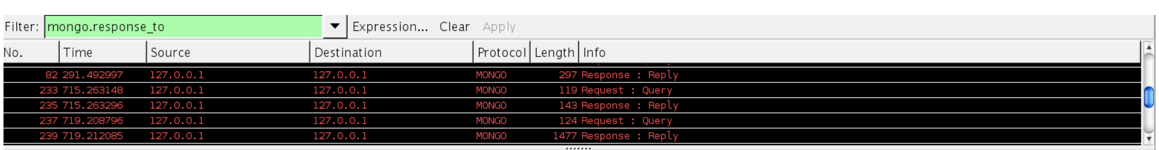
If you want to use a GUI with more detailed introspection, there is Wireshark support for MongoDB.

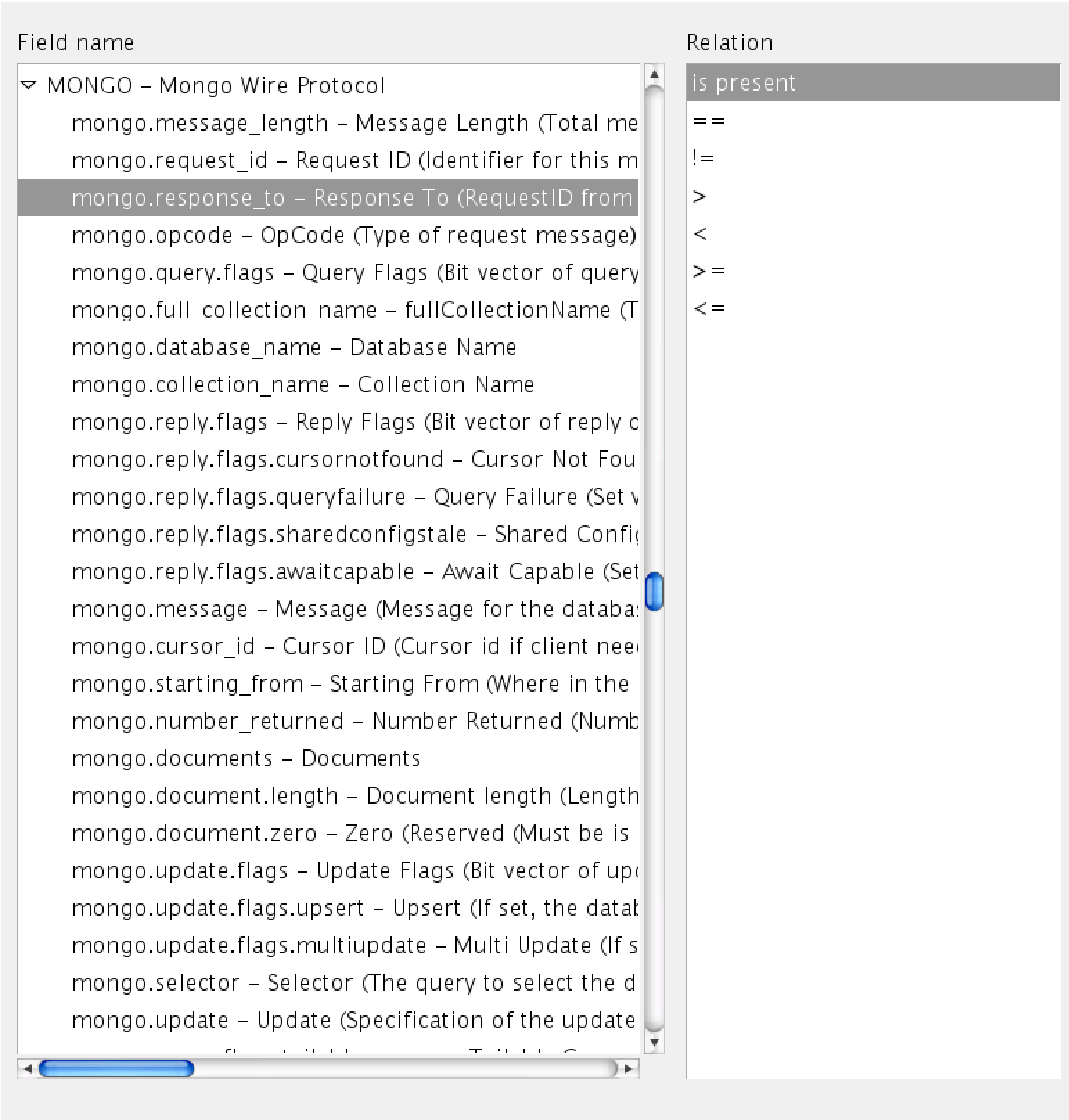
Wireshark Support for MongoDB Protocol

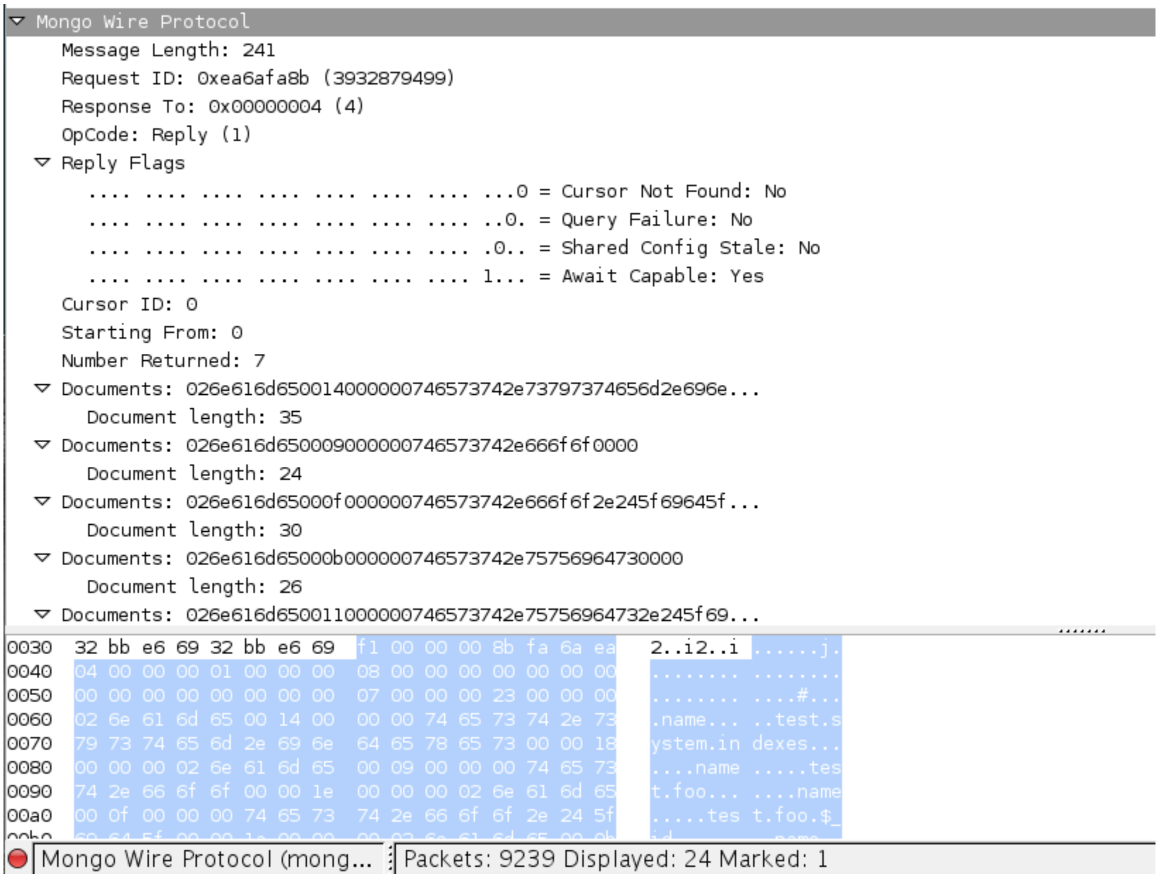
Wireshark, an advanced interactive network traffic sniffer, has full support for the MongoDB Wire protocol.

You can visually inspect MongoDB traffic, do complex filters on specific values of MongoDB wire messages and dig into individual documents both sent and received.

Note: wireshark looks for port 27017 and infers MongoDB protocol from this. If you are running on a different port number, go to Preferences...Protocols...Mongo and set your port number and it should then interpret the data.




Field name
MONGO – Mongo Wire Protocol
mongo.message_length – Message Length (Total me
mongo.request_id – Request ID (Identifier for this m
mongo.response_to – Response To (RequestID from
mongo.opcode – OpCode (Type of request message)
mongo.query.flags – Query Flags (Bit vector of query
mongo.full_collection_name – fullCollectionName (T
mongo.database_name – Database Name
mongo.collection_name – Collection Name
mongo.reply.flags – Reply Flags (Bit vector of reply c
mongo.reply.flags.cursornotfound – Cursor Not Fou
mongo.reply.flags.queryfailure – Query Failure (Set w
mongo.reply.flags.sharedconfigstale – Shared Confi
mongo.reply.flags.awaitcapable – Await Capable (Set
mongo.message – Message (Message for the databa
mongo.cursor_id – Cursor ID (Cursor id if client nee
mongo.starting_from – Starting From (Where in the
mongo.number_returned – Number Returned (Numb
mongo.documents – Documents
mongo.document.length – Document length (Length
mongo.document.zero – Zero (Reserved (Must be is
mongo.update.flags – Update Flags (Bit vector of up
mongo.update.flags.upsert – Upsert (If set, the datab
mongo.update.flags.multiupdate – Multi Update (If s
mongo.selector – Selector (The query to select the d
mongo.update – Update (Specification of the update
Relation
is present
==
!=
>
<
>=
<=



Backups

- Sharded Backup
- Snapshot (Journaling must be Enabled)
- Mongodump
  - Replication
- Shutdown and Backup
- Write Lock, Fsync, and Backup
- Slave Backup
- Community Stuff
- Presentations

Several strategies exist for backing up MongoDB databases. A word of warning: if you do not have journaling enabled, then it's not safe to simply copy the mongod data files (by default in /data/db/) while the database is running and writes are occurring; such a backup may turn out to be corrupt unless you follow the specific directions below.

> With all these backup methods we suggest that you be careful how this could affect your live system when choosing when and where you backups are run. Try not to mix too many distinct work loads on the same resource, like doing backup and analytics or reporting for example. This could overwhelm those resource and negatively effect your users.

Sharded Backup

The instructions below are for backing up a single server or a single shard. To back up an entire cluster, see the Backing Up Sharded Cluster documentation.

Snapshot (Journaling must be Enabled)

If the storage infrastructure (SAN, lvm, etc.) supports it, is it safe to snapshot the entire dbpath directory of a mongod that is running if journaling is enabled (journaling defaults to on in v2.0+ for 64 bit MongoDB). Take an lvm/ebs snapshot of the entire dbpath directory of a mongod running with journaling. All files and directories (start from the dbpath directory) must be included (especially the journal/ subdirectory). As long as all files are snapshotted at the same point in time, you don't need to fsync-lock the database.

Amazon EBS qualified if you are not raiding the volumes yourself. When raided the snapshot would be separate for each embedded volume.

Thus in this case one should still use lock+fsync.

Mongodump

Mongodump can be used to do a live backup of your data, or can work against an inactive set of database files. The mongodump utility may be used to dump an entire cluster/server/database/collection (or part of a collection with a query), even when the database is running and active.

Replication

If you are backing up a replica (from a replica set, or master/slave) you can use the

```
--oplog
```

to do a point in time backup; that point in time will be at the end of the backup. When you restore you will need to use the corresponding

```
--oplogReplay
```

to use this extra backup information.

Shutdown and Backup

A simple approach is just to stop the database, back up the data files, and resume. This is safe but of course requires downtime. This can be done on a secondary without requiring downtime, but you must ensure your oplog is large enough to cover the time the secondary is unavailable so that it can catch up again when you restart it.

Write Lock, Fsync, and Backup

MongoDB supports an fsync and lock command with which we can lock the database to prevent writing, flush writes, and then backup the datafiles.

While in this locked mode, all writes will block (including replication, for secondaries). If this is a problem consider one of the other methods.

> A write attempt will request a lock and may block new readers. This will be fixed in a future release. Thus currently, fsync and lock works best with storage systems that do quick snapshots.

For example, you could use LVM2 to create a snapshot after the fsync+lock, and then use that snapshot to do an offsite backup in the background. This means that the server will only be locked while the snapshot is taken. Don't forget to unlock after the backup/snapshot is taken.

Slave Backup

Another good technique for backups is replication to a slave/secondary server. The replica is continuously kept up to date through Replication and thus always has a nearly-up-to-date copy of primary/master.

We then have several options for backing up from a replica:

1. Fsync, write lock, and backup the slave.
2. Shut it down, backup, and restart.
3. Dump from the slave.

For methods 1 and 2, after the backup the slave will resume replication, applying any changes made which during the backup period.

Using a replica is advantageous because we then always have backup database machine ready in case primary/master fails (failover). But a replica also gives us the chance to back up the full data set without affecting the performance of the primary/master server.

Community Stuff

- http://github.com/micahwedemeyer/automongobackup

Presentations

- Backing up your MongoDB Cluster - MongoSF (May 2011)

EC2 Backup & Restore

Overview

This article describes how to backup, verify & restore a MongoDB running on EC2 using EBS Snapshots.

Backup

How you backup MongoDB will depend on whether you are using the --journal option in 1.8 (or above) or not.

Backup with --journal

The journal file allows for roll forward recovery. The journal files are located in the dbpath directory so will be snapshotted at the same time as the database files.

If the dbpath is mapped to a single EBS volume then proceed to the Backup the Database Files section.

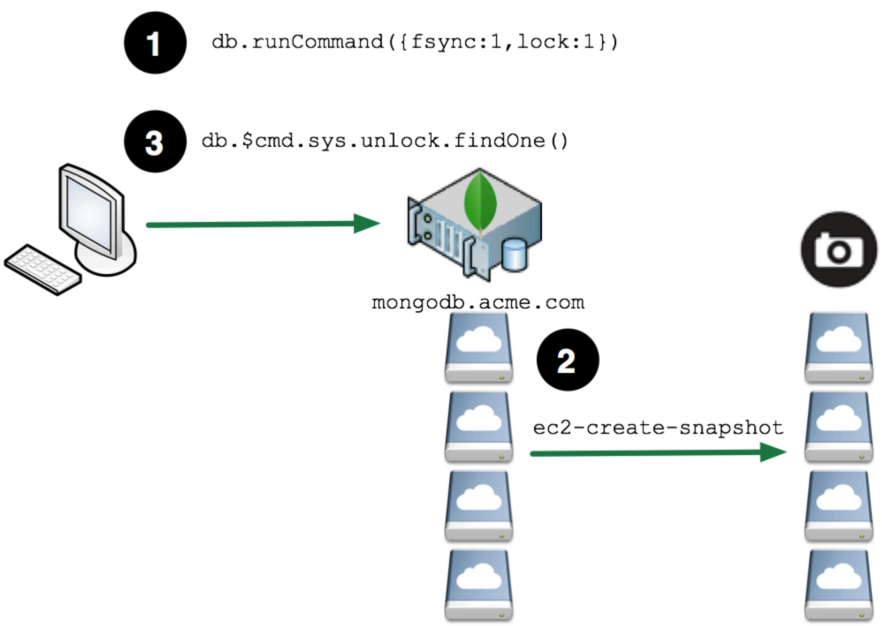
If you dbpath is mapped to multiple EBS volumes, in order to guarantee the stability of the file-system then you will need to Flush and Lock the Database section.

> Note that snapshotting with the journal is only possible if the journal resides on the same volume as the data files, so that one snapshot operation captures the journal state and data file state atomically.

Backup without --journal

In order to correctly backup a MongoDB, you need to ensure that writes are suspended to the file-system before you backup the file-system. If writes are not suspended then the backup may contain partially written or data which may fail to restore correctly.

Backing up MongoDB is simple to achieve using the fsync + lock command. If the file-system is being used only by the database, then you can then use the snapshot facility of EBS volumes to create a backup. If you are using the volume for any other application then you will need to ensure that the file-system is frozen as well (e.g. on XFS file-system use xfs_freeze) before you initiate the EBS snapshot. The overall process looks like:



Flush and Lock the database

Writes have to be suspended to the file-system in order to make a stable copy of the database files. This can be achieved through the MongoDB shell using the fsync + lock command.

```
mongo shell> use admin
mongo shell> db.runCommand({fsync:1,lock:1});
{
 "info" : "now locked against writes, use db.$cmd.sys.unlock.findOne() to unlock",
 "ok" : 1
}
```

During the time the database is locked, any write requests that this database receives will be rejected. Any application code will need to deal with these errors appropriately.

Backup the database files

There are several ways to create a EBS Snapshot, for example with Elastic Fox or the AWS Command line. The following examples use the AWS command line tool.

Find the EBS volumes associated with the MongoDB

If the mapping of EBS Block devices to the MongoDB data volumes is already known, then this step can be skipped. The example below shows how to determine the mapping for an LVM volume, please confirm with your System Administrator how the original system was setup if you are unclear.

Find the EBS block devices associated with the running instance

```
shell> ec2-describe-instances
RESERVATION r-eb09aa81 289727918005 tokyo,default
INSTANCE i-78803e15 ami-4b4ba522 ec2-50-16-30-250.compute-1.amazonaws.com
ip-10-204-215-62.ec2.internal running scaleout 0  m1.large 2010-11-04T02:15:34+0000 us-east-1a
aki-0b4aa462   monitoring-disabled 50.16.30.250 10.204.215.62   ebs paravirtual
BLOCKDEVICE /dev/sda1 vol-6ce9f105 2010-11-04T02:15:43.000Z
BLOCKDEVICE /dev/sdf vol-96e8f0ff 2010-11-04T02:15:43.000Z
BLOCKDEVICE /dev/sdh vol-90e8f0f9 2010-11-04T02:15:43.000Z
BLOCKDEVICE /dev/sdg vol-68e9f101 2010-11-04T02:15:43.000Z
BLOCKDEVICE /dev/sdi vol-94e8f0fd 2010-11-04T02:15:43.000Z
```

As can be seen in this example, there are a number of block devices associated with this instance. We have to determine which volumes make up the file-system we need to snapshot.

Determining how the dbpath is mapped to the file-system

Log onto the running MongoDB instance in EC2. To determine where the database file are located, either look at the startup parameters for the mongod process or if mongod is running, then you can examine the running process.

```
root> ps -ef | grep mongo
ubuntu   10542     1  0 02:17 ?        00:00:00 /var/opt/mongodb/current/bin/mongod --port 27000
--shardsvr --dbpath /var/lib/mongodb/tokyo0 --fork --logpath /var/opt/mongodb/log/server.log
--logappend --rest
```

dbpath is set to /var/lib/mongodb/tokyo0 in this example.

Mapping the dbpath to the physical devices

Using the df command, determine what the --dbpath directory is mapped to

```
root> df /var/lib/mongodb/tokyo0
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/data_vg-data_vol
                     104802308      4320 104797988   1% /var/lib/mongodb
```

Next determine the logical volume associated with this device, in the example above /dev/mapper/data_vg-data_vol

```
root> lvdisplay /dev/mapper/data_vg-data_vol
  --- Logical volume ---
  LV Name                /dev/data_vg/data_vol
  VG Name                data_vg
  LV UUID                fixOyX-6Aiw-PnBA-i2bp-ovUc-u9uu-TGvjxl
  LV Write Access        read/write
  LV Status              available
  # open                 1
  LV Size                100.00 GiB
...
```

This output indicates the volume group associated with this logical volume, in this example data_vg. Next determine how this maps to the physical volume.

```
root> pvscan
  PV /dev/md0   VG data_vg   lvm2 [100.00 GiB / 0    free]
  Total: 1 [100.00 GiB] / in use: 1 [100.00 GiB] / in no VG: 0 [0   ]
```

From the physical volume, determine the associated physical devices, in this example /dev/md0.

```
root> mdadm --detail /dev/md0
/dev/md0:
        Version : 00.90
  Creation Time : Thu Nov  4 02:17:11 2010
     Raid Level : raid10
     Array Size : 104857472 (100.00 GiB 107.37 GB)
  Used Dev Size : 52428736 (50.00 GiB 53.69 GB)
   Raid Devices : 4
...
           UUID : 07552c4d:6c11c875:e5a1de64:a9c2f2fc (local to host ip-10-204-215-62)
         Events : 0.19

    Number   Major   Minor   RaidDevice State
       0       8       80        0      active sync   /dev/sdf
       1       8       96        1      active sync   /dev/sdg
       2       8      112        2      active sync   /dev/sdh
       3       8      128        3      active sync   /dev/sdi
```

We can see that block devices /dev/sdf through /dev/sdi make up this physical devices. Each of these volumes will need to be snapped in order to complete the backup of the file-system.

Create the EBS Snapshot

Create the snapshot for each devices. Using the ec2-create-snapshot command, use the Volume Id for the device listed by the ec2-describe-instances command.

```
shell> ec2-create-snapshot -d backup-20101103 vol-96e8f0ff
SNAPSHOT snap-417af82b vol-96e8f0ff pending 2010-11-04T05:57:29+0000  289727918005 50 backup-20101103
shell> ec2-create-snapshot -d backup-20101103 vol-90e8f0f9
SNAPSHOT snap-5b7af831 vol-90e8f0f9 pending 2010-11-04T05:57:35+0000  289727918005 50 backup-20101103
shell> ec2-create-snapshot -d backup-20101103 vol-68e9f101
SNAPSHOT snap-577af83d vol-68e9f101 pending 2010-11-04T05:57:42+0000  289727918005 50 backup-20101103
shell> ec2-create-snapshot -d backup-20101103 vol-94e8f0fd
SNAPSHOT snap-2d7af847 vol-94e8f0fd pending 2010-11-04T05:57:49+0000  289727918005 50 backup-20101103
```

Unlock the database

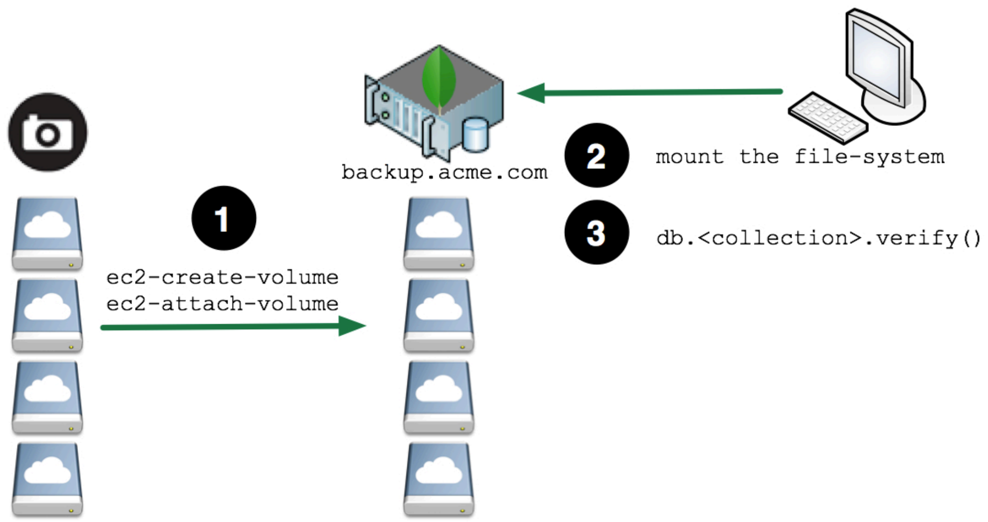
After the snapshots have been created, the database can be unlocked. After this command has been executed the database will be available to process write requests.

```
mongo shell> db.$cmd.sys.unlock.findOne();
{ "ok" : 1, "info" : "unlock requested" }
```

Verifying a backup

In order to verify the backup, the following steps need to be completes

- Check the status of the snapshot to ensure that they are "completed"
- Create new volumes based on the snapshots and mount the new volumes
- Run mongod and verify the collections



Typically, the verification will be performed on another machine so that you do not burden your production systems with the additional CPU and I/O load of the verification processing.

Describe the snapshots

Using the ec2-describe-snapshots command, find the snapshots that make up the backup. Using a filter on the *description* field, snapshots associated with the given backup are easily found. The search text used should match the text used in the *-d* flag passed to ec2-create-snapshot command when the backup was made.

```
backup shell> ec2-describe-snapshots --filter "description=backup-20101103"
SNAPSHOT snap-2d7af847 vol-94e8f0fd completed 2010-11-04T05:57:49+0000 100% 289727918005 50
backup-20101103
SNAPSHOT snap-417af82b vol-96e8f0ff completed 2010-11-04T05:57:29+0000 100% 289727918005 50
backup-20101103
SNAPSHOT snap-577af83d vol-68e9f101 completed 2010-11-04T05:57:42+0000 100% 289727918005 50
backup-20101103
SNAPSHOT snap-5b7af831 vol-90e8f0f9 completed 2010-11-04T05:57:35+0000 100% 289727918005 50
backup-20101103
```

Create new volumes based on the snapshots

Using the ec2-create-volume command, create a new volumes based on each of the snapshots that make up the backup.

```
backup shell> ec2-create-volume --availability-zone us-east-1a --snapshot snap-2d7af847
VOLUME vol-06aab26f 50 snap-2d7af847 us-east-1a creating 2010-11-04T06:44:27+0000
backup shell> ec2-create-volume --availability-zone us-east-1a --snapshot snap-417af82b
VOLUME vol-1caab275 50 snap-417af82b us-east-1a creating 2010-11-04T06:44:38+0000
backup shell> ec2-create-volume --availability-zone us-east-1a --snapshot snap-577af83d
VOLUME vol-12aab27b 50 snap-577af83d us-east-1a creating 2010-11-04T06:44:52+0000
backup shell> ec2-create-volume --availability-zone us-east-1a --snapshot snap-5b7af831
VOLUME vol-caaab2a3 50 snap-5b7af831 us-east-1a creating 2010-11-04T06:45:18+0000
```

Attach the new volumes to the instance

Using the ec2-attach-volume command, attach each volume to the instance where the backup will be verified.

```
backup shell> ec2-attach-volume --instance i-cad26ba7 --device /dev/sdp vol-06aab26f
ATTACHMENT vol-06aab26f i-cad26ba7 /dev/sdp attaching 2010-11-04T06:49:32+0000
backup shell> ec2-attach-volume --instance i-cad26ba7 --device /dev/sdq vol-1caab275
ATTACHMENT vol-1caab275 i-cad26ba7 /dev/sdq attaching 2010-11-04T06:49:58+0000
backup shell> ec2-attach-volume --instance i-cad26ba7 --device /dev/sdr vol-12aab27b
ATTACHMENT vol-12aab27b i-cad26ba7 /dev/sdr attaching 2010-11-04T06:50:13+0000
backup shell> ec2-attach-volume --instance i-cad26ba7 --device /dev/sds vol-caaab2a3
ATTACHMENT vol-caaab2a3 i-cad26ba7 /dev/sds attaching 2010-11-04T06:50:25+0000
```

Mount the volumes groups etc.

Make the file-system visible on the host O/S. This will vary by the Logical Volume Manager, file-system etc. that you are using. The example below shows how to perform this for LVM, please confirm with your System Administrator on how the original system system was setup if you are unclear.

Assemble the device from the physical devices. The UUID for the device will be the same as the original UUID that the backup was made from, and can be obtained using the *mdadm* command.

```
backup shell> mdadm --assemble --auto-update-homehost -u 07552c4d:6c11c875:e5a1de64:a9c2f2fc
--no-degraded /dev/md0
mdadm: /dev/md0 has been started with 4 drives.
```

You can confirm that the physical volumes and volume groups appear correctly to the O/S by executing the following:

```
backup shell> pvscan
  PV /dev/md0   VG data_vg   lvm2 [100.00 GiB / 0    free]
  Total: 1 [100.00 GiB] / in use: 1 [100.00 GiB] / in no VG: 0 [0   ]

backup shell> vgscan
  Reading all physical volumes.  This may take a while...
  Found volume group "data_vg" using metadata type lvm2
```

Create the mount point and mount the file-system:

```
backup shell> mkdir -p /var/lib/mongodb

backup shell> cat >> /etc/fstab << EOF
/dev/mapper/data_vg-data_vol /var/lib/mongodb xfs noatime,noexec,nodiratime 0 0
EOF

backup shell> mount /var/lib/mongodb
```

Startup the database

After the file-system has been mounted, MongoDB can be started. Ensure that the owner of the files is set to the correct user & group. Since the backup was made with the database running, the lock file will need to be removed in order to start the database.

```
backup shell> chown -R mongodb /var/lib/mongodb/toyko0
backup shell> rm /var/lib/mongodb/tokyo0/mongod.lock
backup shell> mongod --dbpath /var/lib/mongodb/tokyo0
```

Verify the collections

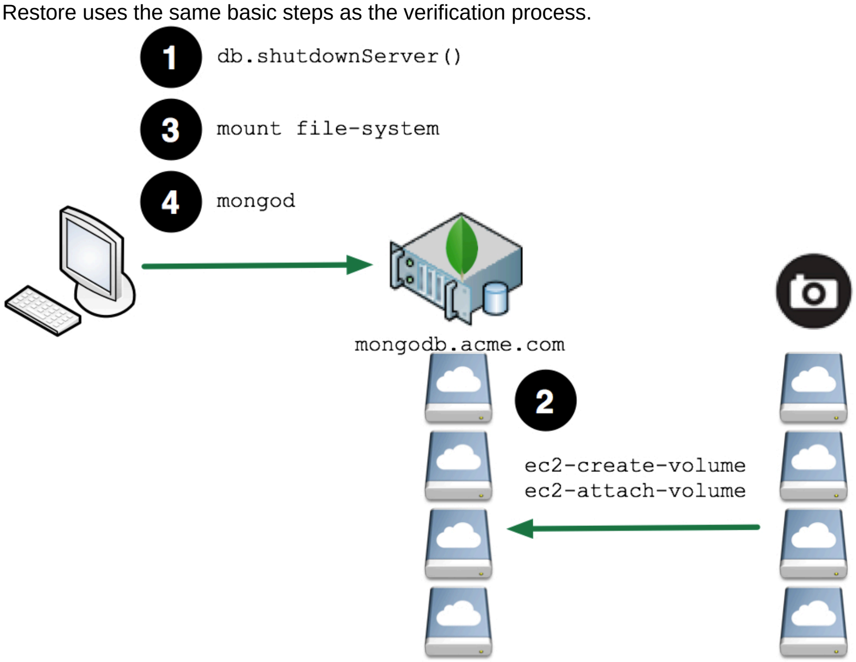
Each collection can be verified in order to ensure that it valid and does not contain any invalid BSON objects.

```
mongo shell> db.blogs.validate({full:true})
```

Validate Command

Restore

Restore uses the same basic steps as the verification process.



After the file-system is mounted you can decide to

- Copy the database files from the backup into the current database directory
- Startup mongod from the new mount point, specifying the new mount point in the --dbpath argument

After the database is started, it will be ready to transact. It will be at the specific point in time from the backup, so if it is part of a master/slave or replica set relationship, then the instance will need to synchronize itself to get itself back up to date.

How to do Snapshotted Queries in the Mongo Database

> ⚠ This document refers to query snapshots. For backup snapshots of the database's datafiles, see the fsync lock page.

MongoDB does not support full point-in-time snapshotting. However, some functionality is available which is detailed below.

Cursors

A MongoDB query returns data as well as a cursor ID for additional lookups, should more data exist. Drivers lazily perform a "getMore" operation as needed on the cursor to get more data. Cursors may have latent getMore accesses that occurs after an intervening write operation on the database collection (i.e., an insert, update, or delete).

Conceptually, a cursor has a current position. If you delete the item at the current position, the cursor automatically skips its current position forward to the next item.

MongoDB cursors do not provide a snapshot: if other write operations occur during the life of your cursor, it is unspecified if your application will see the results of those operations. In fact, it is even possible (although unlikely) to see the same object returned twice if the object were updated and grew in size (and thus moved in the datafile). To assure no update duplications, use snapshot() mode (see below).

Snapshot Mode

snapshot() mode assures that objects which update during the lifetime of a query are returned once and only once. This is most important when doing a find-and-update loop that changes the size of documents that are returned ($inc does not change size).

```
> // mongo shell example
> var cursor = db.myCollection.find({country:'uk'}).snapshot();
```

Even with snapshot mode, items inserted or deleted during the query may or may not be returned; that is, this mode is not a true point-in-time snapshot.

Because snapshot mode traverses the _id index, it may not be used with sorting or explicit hints. It also cannot use any other index for the query.

You can get the same effect as snapshot by using any unique index on a field(s) that will not be modified (probably best to use explicit hint() too). If you want to use a non-unique index (such as creation time), you can make it unique by appending _id to the index at creation time.

Import Export Tools

If you just want to copy a database from one mongod server to another, use the copydb Command instead of these tools.

These tools work with the raw data (the BSON documents in the collections, both user and system); they do not save, or load certain metadata such as (capped) collection properties. You will need to (re)create those yourself in a separate step, before loading that data. Vote for SERVER-808 to change this. (Consider using the copydb command which does preserve these properties.)

Data Import and Export

mongoimport

This utility takes a single file that contains 1 JSON/CSV/TSV string per line and inserts it. You have to specify a database and a collection.

```
options:
  --help                    produce help message
  -v [ --verbose ]          be more verbose (include multiple times for more
                            verbosity e.g. -vvvvv)
  -h [ --host ] arg         mongo host to connect to ("left,right" for pairs)
  --port arg                server port. (Can also use --host hostname:port)
  --ipv6                    enable IPv6 support (disabled by default)
  -d [ --db ] arg           database to use
  -c [ --collection ] arg collection to use (some commands)
  -u [ --username ] arg     username
  -p [ --password ] arg     password
  --dbpath arg              directly access mongod data files in the given path,
                            instead of connecting to a mongod instance - needs to
                            lock the data directory, so cannot be used if a
                            mongod is currently accessing the same path
  --directoryperdb          if dbpath specified, each db is in a separate
                            directory
  -f [ --fields ] arg       comma seperated list of field names e.g. -f name,age
  --fieldFile arg           file with fields names - 1 per line
  --ignoreBlanks            if given, empty fields in csv and tsv will be ignored
  --type arg                type of file to import.  default: json (json,csv,tsv)
  --file arg                file to import from; if not specified stdin is used
  --drop                    drop collection first
  --headerline              CSV,TSV only - use first line as headers
  --upsert                  insert or update objects that already exist
  --upsertFields arg        comma-separated fields for the query part of the
                            upsert. You should make sure this is indexed.
  --stopOnError             stop importing at the first error rather
                            than continuing
  --jsonArray               load a json array, not one item per line.
                            Currently limited to 4MB.
```

Note that the following options are only available in 1.5.3+: `upsert`, `upsertFields`, `stopOnError`, `jsonArray`

Example: Import file format

The import file should contain one document per line (with a few exceptions: if using --jsonArray, if importing a CSV then one document may span multiple lines if it contains multi-line string, if importing a CSV with --headerline then the first line doesn't correspond to a document but instead specifies which fields are being imported).

When using the standard JSON import format, each line in input file must be one JSON document which will be inserted directly into the database.

For example, if you imported a file that looks like this:

```
{ "_id" : { "$oid" : "4e5bb37258200ed9aabc5d65" }, "name" : "Bob", "age" : 28, "address" : "123 fake
street" }
```

by running

```
mongoimport -d test -c foo importfile.json
```

you'd get this imported:

```
> db.foo.find()
{ "_id" : ObjectId("4e5bb37258200ed9aabc5d65"), "name" : "Bob", "age" : 28, "address" : "123 fake
street" }
```

Example: Importing with `upsert`

The following command will import data from `temp.csv` into database `foo`, collection `bar` on localhost. Additionally it will perform an `upsert` of the data. By default the `upsert` will use the field marked as `_id` as the key for updates.

```
mongoimport --host localhost --db foo --collection bar --type csv --file temp.csv --headerline
--upsert
```

If the file does not have an `_id` field, you can update on alternate fields by using `upsertFields`. Note that when using this with **sharding**, the `upsertField` **must** be the shardkey.

> ⚠ Even though using --upsert may result in an update, every document in the input file must be formatted in a way that is compatible to insert. Therefore, no update modifiers are allowed.

Example: Importing Interesting Types

MongoDB supports more types that JSON does, so it has a special format for representing some of these types as valid JSON. For example, JSON has no date type. Thus, to import data containing dates, you structure your JSON like:

```
{"somefield" : 123456, "created_at" : {"$date" : 1285679232000}}
```

Then `mongoimport` will turn the `created_at` value into a Date.

Note: the $-prefixed types must be enclosed in double quotes to be parsed correctly.

mongoexport

mongoexport takes a collection and exports to either JSON or CSV. You can specify a filter for the query, or a list of fields to output.

See the mongoexport page for more information.

mongodump and mongorestore

The are many ways to do backups and restores. (Here are some other backup strategies)

mongodump

This takes a database and outputs it in a binary representation. This is used for doing (hot) backups of a database.

> ⚠ If you're using sharding and trying to migrate data this way, this will dump shard configuration metadata information (from the config db) and overwrite configurations upon restore. This is true because without any options `mongodump` will dump all dbs/collections, including the config db where this information is kept.

```
options:
  --help                    produce help message
  -v [ --verbose ]          be more verbose (include multiple times for more
                            verbosity e.g. -vvvvv)
  -h [ --host ] arg         mongo host to connect to ("left,right" for pairs)
  -d [ --db ] arg           database to use
  -c [ --collection ] arg   collection to use (some commands)
  -u [ --username ] arg     username
  -p [ --password ] arg     password
  --dbpath arg              directly access mongod data files in the given path,
                            instead of connecting to a mongod instance - needs
                            to lock the data directory, so cannot be used if a
                            mongod is currently accessing the same path
  --directoryperdb          if dbpath specified, each db is in a separate
                            directory
  -o [ --out ] arg (=dump) output directory
  -q [ --query ] arg        json query
  --oplog                   point in time backup (requires an oplog)
  --repair                  repairs documents as it dumps from a corrupt db (requires --dbpath and
-d/--db)
  --forceTableScan          force a table scan (do not use $snapshot)
```

Example: Dumping Everything

To dump all of the collections in all of the databases, run `mongodump` with just the `--host`:

```
$ ./mongodump --host prod.example.com
connected to: prod.example.com
all dbs
DATABASE: log    to   dump/log
        log.errors to dump/log/errors.bson
                713 objects
        log.analytics to dump/log/analytics.bson
                234810 objects
DATABASE: blog    to    dump/blog
        blog.posts to dump/log/blog.posts.bson
                59 objects
DATABASE: admin    to    dump/admin
```

You'll then have a folder called "dump" in your current directory.

If you're running `mongod` locally on the default port, you can just do:

```
$ ./mongodump
```

Example: Dumping a Single Collection

If we just want to dump a single collection, we can specify it and get a single .bson file.

```
$ ./mongodump --db blog --collection posts
connected to: 127.0.0.1
DATABASE: blog        to     dump/blog
        blog.posts to dump/blog/posts.bson
                 59 objects
```

> Currently indexes for a single collection will not be backed up. Please follow SERVER-808

Example: Dumping a Single Collection to Stdout

In version 1.7.0+, you can use stdout instead of a file by specifying `--out stdout`:

```
$ ./mongodump --db blog --collection posts --out - > blogposts.bson
```

mongodump creates a file for each database collection, so we can only dump one collection at a time to stdout.

Example: Dumping a Single Collection with a query

Using the -q argument, you can specify a JSON query to be passed. The example below dumps out documents where the "created_at" is between 2010-12-01 and 2010-12-31.

```
$ ./mongodump --db blog --collection posts
    -q '{"created_at" : { "$gte" : {"$date" : 1293868800000},
                          "$lt"  : {"$date" : 1296460800000}
                        }
        }'
```

Example: Using --oplog to get a point-in-time backup

If data is changed over the course of a backup then the resulting dump may wind up in an inconsistent state that doesn't correspond to how the data looked in the DB at any one moment. This can be avoided by using --oplog in mongodump and --oplogReplay in mongorestore. If you use --oplog then when the backup is started, mongodump will note the time of the most recent entry in the oplog. When the dump is finished, mongodump will then find all the oplog entries since the dump started and will dump those as well. When you run mongorestore with

--oplogReplay, after it finishes restoring the main dump, it will replay the entries in the oplog dump so that the data restored ends up in a consistent state corresponding to the moment that the original dump finished.

```
$ ./mongodump --host localhost --oplog
connected to: localhost
all dbs
DATABASE: admin  to     dump/admin
        admin.system.users to dump/admin/system.users.bson
                 1 objects
        admin.system.indexes to dump/admin/system.indexes.bson
                 1 objects
DATABASE: test   to     dump/test
        test.foo to dump/test/foo.bson
                 297110 objects
        test.system.indexes to dump/test/system.indexes.bson
                 1 objects
        local.oplog.rs to dump/oplog.bson
                 11304 objects
```

It is not valid to use --oplog when dumping from a `mongos`. You can use it to dump an individual shard though – see the Backing Up Sharded Cluster page. Likewise, --oplog can only be used on the *master* in a Master Slave configuration because the *slave* does not store an oplog.

Performance Tips

The default dump mode is to do a "snapshot" query. This results in the dump query walking through the `_id` index and returning results in that order. **If you use a custom `_id` value, not the default `ObjectId` type, then this could cause much more disk activity to do a dump; it could (dramatically) slow down things.**

In 1.9.1+ you can force a walk of the data without using an index:

```
$ ./mongodump --forceTableScan ...
```

In earlier versions you can cause this to behavior with a special query (one that cannot use the index):

```
$ ./mongodump -q "{xxxx : { $ne : 0 } }" ...
```

Note: In some shells (like bash) you must escape the "$" in this command like so "\$".

If mongodump seems to skip documents...

There is a maximum key size in the indexes, currently approximately 800 bytes. **This limit also applies to the default index on `_id`.** Any document with an _id key larger than 800 bytes will not be indexed on `_id`. By default mongodump walks the `_id` index and will skip documents with keys too large to index.

```
> use bigid
> db.foo.count()
3

$ mongodump -d bigid -c foo
connected to: 127.0.0.1
DATABASE: bigid  to  dump/bigid
 bigid.foo to dump/bigid/foo.bson
   0 objects
```

You can work around this issue with either of the options listed in the previous section:

```
$ mongodump -d bigid -c foo -q "{xxxx : { \$ne : 0 } }"
connected to: 127.0.0.1
DATABASE: bigid  to  dump/bigid
 bigid.foo to dump/bigid/foo.bson
   3 objects
```

mongorestore

`mongorestore` takes the output from mongodump and restores it. Indexes will be created on a restore. **`mongorestore` just does inserts with the data to restore**; if existing data (like with the same _id) is there it will not be replaced. This can be done with an existing database, or mongorestore will create a new one if the database does not exist. Mongorestore is mostly non-blocking (it just calls a series of normal inserts), though if the dump included indexes it may cause the DB to block as the indexes are rebuilt.

> If you do not wish to create indexes you can remove the system.indexes.bson file from your database(s) dump directory before restoring. (The default _id indexes will always be created.)

```
usage: ./mongorestore [options] [directory or filename to restore from]
options:
  --help                  produce help message
  -v [ --verbose ]        be more verbose (include multiple times for more
                          verbosity e.g. -vvvvv)
  --version               print the program's version and exit
  -h [ --host ] arg       mongo host to connect to ( <set name>/s1,s2 for sets)
  --port arg              server port. Can also use --host hostname:port
  --ipv6                  enable IPv6 support (disabled by default)
  -u [ --username ] arg   username
  -p [ --password ] arg   password
  --dbpath arg            directly access mongod database files in the given
                          path, instead of connecting to a mongod  server -
                          needs to lock the data directory, so cannot be used
                          if a mongod is currently accessing the same path
  --directoryperdb        if dbpath specified, each db is in a separate
                          directory
  --journal               enable journaling
  -d [ --db ] arg         database to use
  -c [ --collection ] arg collection to use (some commands)
  --objcheck              validate object before inserting
  --filter arg            filter to apply before inserting
  --drop                  drop each collection before import
  --oplogReplay           replay oplog for point-in-time restore
  --keepIndexVersion      don't upgrade indexes to newest version
```

> The above `--help` output is from v2.0.3. Check your version and output for the relevant options for your version of `mongorestore`.

bsondump

> v1.6+

This takes a bson file (typically from mongodump) and converts it to json/debug output. Passing `type=debug` outputs an indented format that shows the type and size for each object.

```
usage: bsondump [options] <bson filename>
options:
  --help                produce help message
  -v [ --verbose ]      be more verbose (include multiple times for more
                        verbosity e.g. -vvvvv)
  --version             print the program's version and exit
  --objcheck            validate object before inserting
  --filter arg          filter to apply before inserting
  --type arg (=json)    type of output: json,debug
```

The debug format displays extra debug information for each field, including the type and size, in an indented form. The debug option also tries to validate strings are valid utf-8.

See Also

- Components

- mongofiles (GridFS tools)
- Backups

mongoexport

The mongoexport utility is a command line tool that takes a collection and exports to either JSON or CSV. You can specify a filter for the query, or a list of fields to output.

> Neither JSON nor TSV/CSV can represent all data types. Please be careful not to lose or change data (types) when using this. For full data fidelity, or backups, please use `mongodump`.

If you want to output CSV, you have to specify the fields in the order you want them output.

Command Line

```
options:
  --help                  produce help message
  -v [ --verbose ]        be more verbose (include multiple times for more
                          verbosity e.g. -vvvvv)
  -h [ --host ] arg       mongo host to connect to ("left,right" for pairs)
  -d [ --db ] arg         database to use
  -c [ --collection ] arg where 'arg' is the collection to use
  -u [ --username ] arg   username
  -p [ --password ] arg   password
  --dbpath arg            directly access mongod data files in the given path,
                          instead of connecting to a mongod instance - needs to
                          lock the data directory, so cannot be used if a
                          mongod is currently accessing the same path
  --directoryperdb        if dbpath specified, each db is in a separate
                          directory
  -q [ --query ] arg      query filter, as a JSON string
  -f [ --fields ] arg     comma seperated list of field names e.g. -f name,age
  --csv                   export to csv instead of json, requires -f
  -o [ --out ] arg        output file; if not specified, stdout is used
```

Dates

To pass a date in a query on the command line, use the syntax:

```
Date(<millis_since_epoch>)
```

You can get the numeric value for a date in the mongo shell:

```
$ mongo --nodb
> Date("2011-01-12T12:00:00Z").valueOf()
1294833600000
```

and then exit the shell and use that value in your mongoexport -q parameter:

```
mongoexport -q "{when:new Date(1294833600000)}" ...
```

Durability and Repair

MongoDB (specifically, the mongod process) is normally ran with *journaling* enabled. This makes it crash-safe.

Journaling is on by default in v2.0+ for 64 bit builds. (Use journaling, leave it enabled.)

Journaling Enabled

If you are running with Journaling you should not do a repair to recover to a consistent state. When you start with journaling files they will automatically be replayed to a consistent state.

> The `--dur` option was used before 1.8; now the option is `--journal`, and is on by default in version 1.9.2+ on 64-bit platforms

When using journaling, you may see the message:

```
**************
old lock file: mongod.lock.  probably means unclean shutdown,
but there are no journal files to recover.
this is likely human error or filesystem corruption.
found 23 dbs.
see: http://dochub.mongodb.org/core/repair for more information
*************
```

You may want to check:

- If someone moved the journal files
- The integrity of your disk.

Replication without Journaling

If you have a replica set then it is favorable to re-sync the failed node from scratch or a backup than to do a repair.

No Replication nor Journaling

Recent Backup

If you have a recent backup then it makes sense to use that instead of repair if you are concerned with application data consistency.

Repair Command

When not using journaling (`--nojournal`), after a machine crash or `kill -9` termination, run the *repairDatabase* command. This command will check all data for corruption, remove any corruption found, and compact data files a bit. Repair is analogous to running *fsck* for a file system.

When journaling is enabled, it should not be necessary to run repair. However one could still use the repair command to compact a database.

From the command line:

```
mongod --repair
```

From the shell (you have to do for all dbs including local if you go this route):

```
> db.repairDatabase();
```

During a repair operation, *mongod* must store temporary files to disk. By default, *mongod* creates temporary directories under the dbpath for this purpose. Alternatively, the `--repairpath` command line option can be used to specify a base directory for temporary repair files.

Note that repair is a slow operation which inspects the entire database.

After running with `--repair`, *mongod* will start up normally.

> When running the repairDatabase command on a non-primary server (replica set secondary), you will get an error stating that the server is not master. In order to run the repair, restart the server without the `--replSet` option so that the server is in single server mode, and run the repair. When you restart, make sure to do it on a different port, so as not to confused the other members. Then restart one more time with the `--replSet` option on. This may put the replica server back in a consistent state, but it is highly recommended to check the data validity by comparing a dump of the master/primary and repaired-replica. If there is a suspicion of data being corrupted, it is safer to resync the replica from scratch.

Because `mongod` rewrites all of the database files during the repair routine, if you do not run `--repair` under the same user account as `mongod` usually runs, you will need to run `chown` on your database files to correct the permissions before starting `mongod` again.

`mongod.lock`

Do not remove the `mongod.lock` file. If `mongod` is unable to start, use one of the methods above to correct the situation.

Removing the lock file will allow the database to start when its data may be corrupt. In general, you should never force the database to start with possibly corrupt data. In an emergency situation, you may want to remove the lock file to pull whatever data you can off the server. If you have ever manually removed the lock file and started the server back up, you should not consider that server "healthy."

Checking Data Integrity

You can use the *validate* command on to check if the contents of a collection are valid.

For example, here we validate the `users` collection:

```
> db.users.validate();
{
 "ns" : "test.users",
 "result" : " validate
  details: 0x1243dbbdc ofs:740bdc
  firstExtent:0:178b00 ns:test.users
  lastExtent:0:178b00 ns:test.users
  # extents:1
  datasize?:44 nrecords?:1 lastExtentSize:8192
  padding:1
  first extent:
    loc:0:178b00 xnext:null xprev:null
    nsdiag:test.users
    size:8192 firstRecord:0:178bb0 lastRecord:0:178bb0
  1 objects found, nobj:1
  60 bytes data w/headers
  44 bytes data wout/headers
  deletedList: 0000000010000000000
  deleted: n: 1 size: 7956
  nIndexes:2
    test.users.$_id_ keys:1
    test.users.$username_1 keys:1 ",
 "ok" : 1,
 "valid" : true,
 "lastExtentSize" : 8192
}
```

This is a slow command, as it has to check every document in a collection.

If journaling is disabled

If the databases exited uncleanly and you attempt to restart the database, *mongod* will prin

```
**************
old lock file: /data/db/mongod.lock.  probably means unclean shutdown
recommend removing file and running --repair
see: http://dochub.mongodb.org/core/repair for more information
*************
```

Then it will exit.

See Also

- What About Durability? (MongoDB Blog)
- *fsync* Command
- MongoDB (Single-Server) Data Durability Guide

Security and Authentication

MongoDB includes basic authentication functionality. By default, authentication is off. When off, it is then required that one runs the database in a trusted (network restricted) environment.

> Authentication is not available with sharding before v2.0. Older sharded environments must run with the database in a trusted environment.

Please also see the replica set authentication documentation.

- Running Without Security (Trusted Environment)
    - Firewall Rules
- Mongo Security
- Configuring Authentication and Security
    - Viewing Users
    - Changing Passwords
    - Deleting Users
- Replica Set and Sharding Authentication
    - Sharded Security
        - Sharded Database Users
    - Enabling/disabling authentication on an existing cluster
    - About the Key File
        - Key file permissions
    - Ports
    - IP Address Binding
- Report a Security Issue
- FAQ

Running Without Security (Trusted Environment)

Trusted environment is the default option and is recommended. It is often valid to run the database in a trusted environment with no in-database security and authentication (much like how one would use, say, memcached). Of course, in such a configuration, one must be sure only trusted machines can access database TCP ports.

Firewall Rules

Given the default ports here are the basic firewall rules needed.

Process	Direction	Ports	Notes
mongod	Incoming	27017/*	client connections
mongod	Outgoing	*/27017	replication from other members, shard migrations
mongos	Incoming	27017/*	client connections
mongos	Outgoing	*/[27019, 27018]	connections to shards and config servers

- Replica Sets
    - All clients need to connect to all non-hidden nodes and all nodes need to communicate with each other.

- Sharding
  - Mongos must be able to connect to all config servers and shards (every replica set member)
  - All shards (all replicas if a set) must connect to each other, and the config servers

Mongo Security

The current version of Mongo supports only basic security. You authenticate a username and password in the context of a particular database. Once authenticated, a normal user has full read and write access to the database. You can also create read-only users that only have read access.

The `admin` database is special. Several administrative commands can only run on the `admin` database (so can only be run by an admin user). Also, authentication on `admin` gives a user read and write access to all databases on the server.

Logging in using an admin account

Although admin user accounts can access any database, you must log into the admin database. For example, if someAdminUser has an admin account, this login will fail:

```
> use test
> db.auth("someAdminUser", password)
```

This one will succeed:

```
> use admin
> db.auth("someAdminUser", password)
```

To enable security, run the database (`mongod` process) with the `--auth` option (or `--keyFile` for replica sets and sharding). You **must** either have added a user to the `admin` db before starting the server with authentication, or add the first user from the localhost interface (you cannot add the first user from a connection that is not local with respect to mongod).

Configuring Authentication and Security

We should first create an administrator user for the entire db server process. This user is stored under the special `admin` database.

If there are no admin users, one may access the database from the localhost interface without authenticating. Thus, from the server running the database (and thus on localhost), run the database shell and configure an administrative user:

```
$ ./mongo
> use admin
> db.addUser("theadmin", "anadminpassword")
```

We now have a user created for database `admin`. Note that if we have not previously authenticated, we now must if we wish to perform further operations, as there is now an admin user.

```
> db.auth("theadmin", "anadminpassword")
```

Now, let's configure a "regular" user for another database.

```
> use projectx
> db.addUser("joe", "passwordForJoe")
```

Finally, let's add a readonly user. (only supported in 1.3.2+)

```
> use projectx
> db.addUser("guest", "passwordForGuest", true)
```

Viewing Users

User information is stored in each database's `system.users` collection. For example, on a database *projectx*, `projectx.system.users` will contain user information.

We can view existing users for the current database with the query:

```
> db.system.users.find()
```

Changing Passwords

The shell `addUser` command may also be used to update a password: if the user already exists, the password simply updates.

Some Mongo drivers provide a helper function equivalent to the db shell's `addUser` method.

Deleting Users

To delete a user:

```
db.removeUser( username )
```

or

```
db.system.users.remove( { user: username } )
```

Replica Set and Sharding Authentication

The keyFile is required!
You are required to use the keyFile option for sharding or replica set authentication. Just using the auth option will not work.

Replica sets and sharding can use authentication in v1.9.1+ (replica sets without sharding in v1.8). From the client's perspective, it is identical to using single-server authentication: connect to your *mongos*, create users, authenticate when you make connections.

The only difference is that the servers use a *key file* to authenticate internal communication. A key file is basically a plaintext file which is hashed and used as an internal password.

To set up replica sets and/or sharding with authentication:

1. Create a key file that can be copied to each server in the set. A key file is composed of characters in the Base64 set, plus whitespace and newlines (see About the Key File for details).
2. Modify this file's permissions to be only readable by the current user.
3. Start each server in the cluster (including all replica set members, all config servers, and all *mongos* processes) with the *--keyFile /path/to/file* option.
4. Each client connection to the database must be authenticated before it can be used, as with single-server authentication.

You do not need to use the *--auth* option, too (although there's no harm in doing so), *--keyFile* implies *--auth*. *--auth* does **not** imply *--keyFile*.

Sharded Security

When you enable authentication on the sharded cluster you will be securing connections via the `mongos` instances and not on individual resources directly like with a single server or a replica set. This is partly because the security information is stored in the sharding config database and not directly on the shards for the admin users. It is also important in a sharded cluster to **always** have clients connect through a `mongos` instance and not directly to any resource like the shard servers or config server.

If you do add individual users on sharded resources, the individual shard members, then those users are separate from the sharding "admin" users. These two users sources are different between the sharded cluster and the shards for admin users; care should be taken not to confuse the two or to think that they are same set of users.

Sharded Database Users

Users are stored on the primary shard so authentication can be done there. This may lead to a bit of confusion when you have sharded collections because it may not be obvious that the there is no "system.users" collection on the non-primary shards.

Enabling/disabling authentication on an existing cluster

To enable authentication on an existing cluster, shut down all members and restart them with the *--keyFile* option. Remember that you must add users before you can access the data remotely.

You can disable authentication by restarting all members without the *--keyFile* option.

About the Key File

A key file must contain at least 6 Base64 characters and be no larger than 1KB (whitespace included). Whitespace characters are stripped (mostly for cross-platform convenience), so the following keys are identical to the database:

```
$ echo -e "my secret key" > key1
$ echo -e "my secret key\n" > key2
$ echo -e "my    secret    key" > key3
$ echo -e "my\r\nsecret\r\nkey\r\n" > key4
```

If you run *mongod* with *-v*, the key will be printed in the log.

Key file permissions

On *NIX, group and everyone must have 0 permissions (up to 700 is allowed). If the permissions are too open on the key file, MongoDB will exit with an error to that effect.

At the moment, permissions are not checked by *mongod* on Windows.

Ports

Default TCP port numbers for MongoDB processes are as follows:

- Standalone *mongod* : 27017
- *mongos* : 27017
- shard server (`mongod --shardsvr`) : 27018
- config server (`mongod --configsvr`) : 27019
- web stats page for *mongod* : add 1000 to port number (28017, by default). Most stats pages in the HTTP UI are unavailable unless the --rest option is specified. To disable the "home" stats page use --nohttpinterface. (This port should be secured, if used, however, the information on the stats home page is read only regardless.)

You can change the port number using the 'port' option when starting mongod.

> Do not rely on a non-standard port as a way to secure a Mongo server. Vulnerability scanners will scan across many port numbers looking for a response.

IP Address Binding

By default, a mongod server will listen on all available IP addresses on a machine. You can restrict this to a single IP address with the 'bind_ip' configuration option for mongod.

Typically, this would be set to 127.0.0.1, the loopback interface, to require that mongod only listen to requests from the same machine (localhost).

To enable listening on all interfaces, remove the bind_ip option from your server configuration file.

> To accept requests on external interfaces you may also have to modify your computer's firewall configuration to allow access to the ports used by mongo (see above).

Report a Security Issue

If you have discovered any potential security issues in MongoDB, please email security@10gen.com with any and all relevant information.

FAQ

- Are passwords sent over the wire encrypted?
    - Yes. (Actually a nonce-based digest is passed.)

- Are database operations, after authenticating, passed over the wire encrypted?
    - No.

Admin UIs

MongoDB does not include a GUI-style administrative interface. Instead most administration is done from command line tools such as the mongo shell. However some UI's are available as separate community projects and are listed below. Some are focused on administration, while some focus on data viewing.

See also:

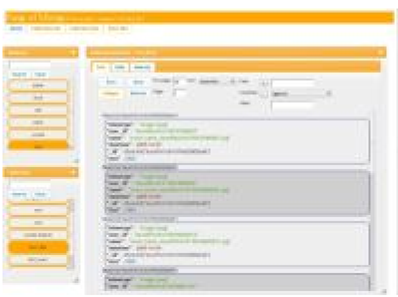
- The Mongo Monitoring Service (MMS) from 10gen.
- Tim Gourley's blog has a good summary of the tools.
- The built-in replica set admin UI page.

Tools

Fang of Mongo

- http://blueone.pl:8001/fangofmongo/
- http://github.com/Fiedzia/Fang-of-Mongo

A web-based user interface for MongoDB build with django and jquery.



It will allow you to explore content of mongodb with simple but (hopefully) pleasant user interface.

Features:

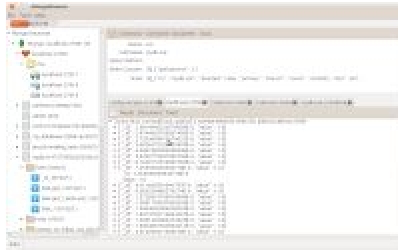
- field name autocompletion in query builder
- data loading indicator
- human friendly collection stats
- disabling collection windows when there is no collection selected
- twitter stream plugin
- many more minor usability fixes
- works well on recent chrome and firefox

To track progress on twitter: @fangofmongo

JMongoBrowser

- home page: http://edgytech.com/jmongobrowser/
- github: http://github.com/agirbal/JMongoBrowser
- download: https://github.com/agirbal/JMongoBrowser/downloads

JMongoBrowser is a GUI app that can browse and administer a MongoDB cluster. It is available for Linux, Windows and Mac OSX.



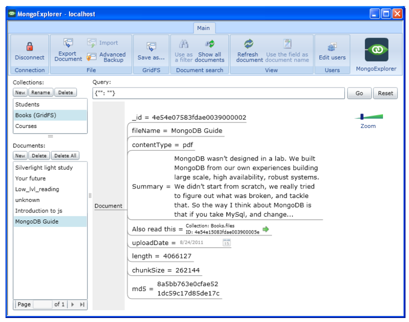
MongoExplorer

- http://mongoexplorer.com/

MongoExplorer is a MongoDB management tool, written in Silverlight (.net – works in windows/osx/?linux?).

Features:

- Easy to use
- Shows all the collections and documents of the database
- Uses a convenient tree view for documents
- Drag'n'drop is fully supported
- Document in-place editing



MongoHub

- http://mongohub.todayclose.com/

MongoHub is a native OS X GUI.

MongoVision

- http://code.google.com/p/mongo-vision/

MongoVision is a MongoDB management tool, written for Prudence.

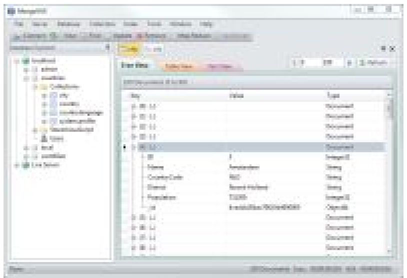
Features:

- Extended JSON support
- Tabular view
- Click to sort
- Filter boxes to alter query
- Auto-refresh

MongoVUE

- http://blog.mongovue.com

MongoVUE is a .NET GUI for MongoDB.



mViewer

- http://imaginea.com/mviewer

mViewer is a web-based MongoDB administration tool.

Opricot

- http://www.icmfinland.fi/oss/opricot/

Opricot is a hybrid GUI/CLI/Scripting web frontend implemented in PHP to manage your MongoDB servers and databases. Use as a point-and-click adventure for basic tasks, utilize scripting for automated processing or repetitive things.

Opricot combines the following components to create a fully featured administration tool:

- An interactive console that allows you to either work with the database through the UI, or by using custom Javascript.
- A set of simple commands that wrap the Javascript driver, and provide an easy way to complete the most common tasks.
- Javascript driver for Mongo that works on the browser and talks with the AJAX interface.
- Simple server-side AJAX interface for communicating with the MongoDB server (currently available for PHP).

PHPMoAdmin

- http://www.phpmoadmin.com/

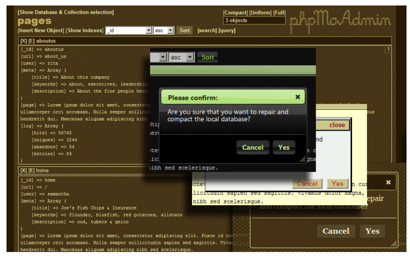
PHPMoAdmin is a MongoDB administration tool for PHP built on a stripped-down version of the Vork high-performance framework.

- Nothing to configure - place the moadmin.php file anywhere on your web site and it just works!
- Fast AJAX-based XHTML 1.1 interface operates consistently in every browser!
- Self-contained in a single 95kb file!
- Works on any version of PHP5 with the MongoDB NoSQL database installed & running.
- Super flexible - search for exact-text, text with * wildcards, regex or JSON (with Mongo-operators enabled)
- Option to enable password-protection for one or more users; to activate protection, just add the username-password(s) to the array at the top of the file.

- E_STRICT PHP code is formatted to the Zend Framework coding standards + fully-documented in the phpDocumentor DocBlock standard.
- Textareas can be resized by dragging/stretching the lower-right corner.
- Free & open-source! Release under the GPLv3 FOSS license!
- Option to query MongoDB using JSON or PHP-array syntax
- Multiple design themes to choose from
- Instructional error messages - phpMoAdmin can be used as a PHP-Mongo connection debugging tool

PHPMoAdmin can help you discover the source of connection issues between PHP and Mongo. Download phpMoAdmin, place the moadmin.php file in your web site document directory and navigate to it in a browser. One of two things will happen:

- You will see an error message explaining why PHP and Mongo cannot connect and what you need to do to fix it
- You will see a bunch of Mongo-related options, including a selection of databases (by default, the "admin" and "local" databases always exist) - if this is the case your installation was successful and your problem is within the PHP code that you are using to access MongoDB, troubleshoot that from the Mongo docs on php.net



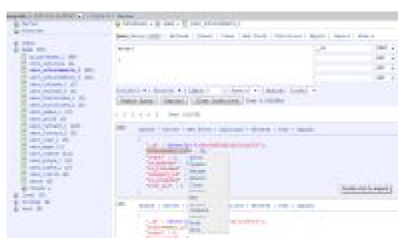
RockMongo

- http://code.google.com/p/rock-php/wiki/rock_mongo

RockMongo is a MongoDB management tool, written in PHP 5.

Main features:

- easy to install, and open source
- multiple hosts, and multiple administrators for one host
- password protection
- query dbs
- advanced collection query tool
- read, insert, update, duplicate and remove single row
- query, create and drop indexes
- clear collection
- remove and change (only work in higher php_mongo version) criteria matched rows
- view collection statistics



Meclipse

Eclipse plugin for mongodb: http://update.exoanalytic.com/org.mongodb.meclipse/

Humongous

A standalone MongoDB browser built in Ruby: http://github.bagwanpankaj.com/humongous/

MongoDB ODA plugin for BIRT

The MongoDB ODA plugin for BIRT is an Eclipse based plugin which enables you to connect to a Mongo database and pull out data to display in your BIRT report. The interface is simple and an extensive user guide is also provided with the release.

http://code.google.com/a/eclipselabs.org/p/mongodb-oda-birt-plugin/

Commercial

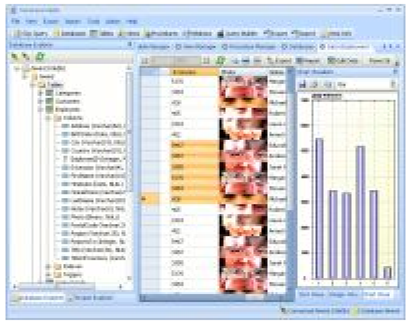
Database Master

Database Master from Nucleon Software.

Seems to be written in .net for windows (windows installer).

Features:

- Tree view for dbs and collections
- Create/Drop indexes
- Server/DB stats
- Support RDMBS (MySQL, postgres, ...)



Data Viewers

mongs

- http://www.whit537.org/mongs/

Starting and Stopping Mongo

MongoDB is run as a standard program from the command line. Please see Command Line Parameters for more information on those options.

The following examples assume that you are in the directory where the `mongod` executable resides. `mongod` is the primary database process that runs on an individual server. `mongos` is the sharding process. `mongo` is the administrative shell. This page discusses `mongod`.

Starting mongod

Default Data Directory, Default Port

To start Mongo in default mode, where data will be stored in the `/data/db` directory (or `c:\data\db` on Windows), and listening on port 27017, just type

```
$ ./mongod
```

Alternate Data Directory, Default Port

To specify a directory for Mongo to store files, use the `--dbpath` option:

```
$ ./mongod --dbpath /var/lib/mongodb/
```

Note that you must create the directory and set its permissions appropriately ahead of time -- Mongo will not create the directory if it doesn't exist.

Alternate Port

You can specify a different port for Mongo to listen on for connections from clients using the `--port` option

```
$ ./mongod --port 12345
```

This is useful if you want to run more than one instance of Mongo on a machine (e.g., for running a master-slave pair).

Running as a Daemon

Note: these options are only available in MongoDB version 1.1 and later.

This will fork the Mongo server and redirect its output to a logfile. As with `--dbpath`, you must create the log path yourself, Mongo will not create parent directories for you.

```
$ ./mongod --fork --logpath /var/log/mongodb.log --logappend
```

Stopping mongod

Control-C

If you have Mongo running in the foreground in a terminal, you can simply "Ctrl-C" the process. This will cause Mongo to do a clean exit, flushing and closing it's data files. Note that it will wait until all ongoing operations are complete.

Sending shutdownServer() message from the mongo shell

The shell can request that the server terminate.

```
$ ./mongo
> use admin
> db.shutdownServer()
```

This command only works from localhost or if one is authenticated.

From a driver (where the helper function may not exist), one can run the command

```
{ "shutdown" : 1 }
```

If this server is the primary in a replica set, it will go through the following process (version 1.9.1+):

1. Check how up-to-date the secondaries are.
2. If no secondary within 10 seconds of the primary, return that we won't shut down (optionally pass the `timeoutSecs` option to wait for a secondary to catch up.
3. If there is a secondary within 10 seconds of the primary, the primary will step down and wait for the secondary to catch up.
4. After 60 seconds or once the secondary has caught up, the primary will shut down.

If there is no up-to-date secondary and you want the primary to shut down anyway, you can use `force : true`:

```

> db.adminCommand({shutdown : 1, force : true})
> // or
> db.shutdownServer({force : true})
```

You can also specify `timeoutSecs : N`, which will keep checking the secondaries for N seconds if none are immediately up-to-date. If any of the secondaries catch up within N seconds, the primary will shut down. If no secondaries catch up, it will not shut down.

```
> db.adminCommand({shutdown : 1, timeoutSecs : 5})
> // or
> db.shutdownServer({timeoutSecs : 5})
> // sample output:
{
        "closest" : NumberLong(1307651781),
        "difference" : NumberLong(1307651808),
        "errmsg" : "no secondaries within 10 seconds of my optime",
        "ok" : 0
}
```

Sending a Unix INT or TERM signal

You can cleanly stop `mongod` using a SIGINT or SIGTERM signal on Unix-like systems. Either ^C, "`kill -2 PID`," or `kill -15 PID` will work.

> Sending a KILL signal `kill -9` will probably cause damage if `mongod` is not running with the `--journal` option. (In such a scenario, run repairDatabase command.)

After a hard crash, when not using `--journal`, MongoDB will say it was not shutdown cleanly, and ask you to do a repair of the database.

Memory Usage

Mongo uses memory mapped files to access data, which results in large numbers being displayed in tools like top for the mongod process. This is not a concern, and is normal when using memory-mapped files. Basically, the size of mapped data is shown in the virtual size parameter, and resident bytes shows how much data is being cached in RAM.

You can get a feel for the "inherent" memory footprint of Mongo by starting it fresh, with no connections, with an empty /data/db directory and looking at the resident bytes.

getCmdLineOpts command

The `getCmdLineOpts` command return both raw and formatted versions of the command line options used to start `mongod`.

```
use admin
db.runCommand({getCmdLineOpts: 1})
{
 "argv" : [
  "mongod",
  "--replSet",
  "replica-set-foo",
  "--logpath",
  "/data/mongod.log",
  "--oplogSize",
  "512",
  "--nojournal",
  "--dbpath",
  "/data/rs-30000",
  "--port",
  "30000",
  "--fork"
 ],
 "parsed" : {
  "dbpath" : "/data/rs-30000",
  "fork" : true,
  "logpath" : "/data/mongod.log",
  "nojournal" : true,
  "oplogSize" : 512,
  "port" : 30000,
  "replSet" : "replica-set-foo"
 },
 "ok" : 1
}
```

The "parsed" section is new in version 2.x

Logging

- Command Line Options
- Rotating the log files
    - From the mongo shell
    - From the unix shell
- Accessing Logs via Shell
    - Get a list of available loggers
    - Get a log

MongoDB outputs some important information to stdout while its running. There are a number of things you can do to control this

Command Line Options

- `--quiet` - less verbose output (more details)
- `-v` - more verbose output. use more v's (such as `-vvvvvv`) for higher levels of verbosity. To change the logging verbosity on a running instance, you can use the setParameter Command
- `--logpath <file>` - output to file instead of stdout
    - If you use `logpath`, you can rotate the logs by either running the `logRotate` command (1.3.4+) or sending SIGUSR1
    - You should always use `--logappend` with `--logpath` to append to the existing log file, instead of overwriting it

Rotating the log files

Log file rotation renames the current log file to the same name with a timestamp file extension appended, then continues logging to a new file with the original name. The timestamp is the time that the logRotate command was executed, expressed in UTC (GMT) and formatted as ISO but with dashes instead of colons for the time portion.

For example:

```
$ ./mongod -v --logpath /var/log/mongodb/server1.log --logappend
```

will start mongod with verbose logging to `/var/log/mongodb/server1.log`, appending to any existing log file.

In another terminal, list the matching files:

```
$ ls /var/log/mongodb/server1.log*
server1.log
```

Rotate the log file using one of the methods described below, then list the files again:

```
$ ls /var/log/mongodb/server1.log*
server1.log  server1.log.2011-11-24T23-30-00
```

This indicates a log rotation performed at exactly 11:30 pm on November 24th, 2011 UTC, which will be the local time offset by the local time zone. The original log file is the one with the timestamp, and new log lines are now being written to the `server1.log` file.

If another logRotate command is given one hour later, an additional file will appear:

```
$ ls /var/log/mongodb/server1.log*
server1.log  server1.log.2011-11-24T23-30-00  server1.log.2011-11-25T00-30-00
```

The `server1.log.2011-11-24T23-30-00` file is unchanged from before, while `server1.log.2011-11-25T00-30-00` is the previous `server1.log` file renamed and `server1.log` is a new empty file that will receive new log output.

From the mongo shell

```
> use admin
> db.runCommand("logRotate");
```

Windows
The logRotate command is available on Windows in version 2.0.3 and higher

From the unix shell

Rotate logs for a single process

```
shell> kill -SIGUSR1 <mongod process id>
```

Rotate logs for all mongo processes on a machine

```
shell> killall -SIGUSR1 mongod
```

Windows
Windows does not have an equivalent to the unix `kill -SIGUSR1` feature, but the mongo shell can be used from the Windows command line using a JavaScript command file to issue a logRotate command.

```
C:\> type logRotate.js
db.getMongo().getDB("admin").runCommand("logRotate")
C:\> mongo logRotate.js
C:\> rem Log files rotated, still at Windows command prompt
```

Accessing Logs via Shell

New in 1.9.x
See the getLog Command for more details

Get a list of available loggers

```
> show logs
> db.runCommand( { getLog : "*" } )
```

Get a log

```
> show log global
> db.runCommand( { getLog : "global" } )
```

Command Line Parameters

MongoDB can be configured via command line parameters in addition to File Based Configuration. You can see the currently supported set of command line options by running the database with `-h [ --help ]` as a single parameter:

```
$ ./mongod --help
```

Information on usage of these parameters can be found in Starting and Stopping Mongo.

The following list of options is not complete; for the complete list see the usage information as described above.

The command line parameters passed to `mongod` can be viewed while the instance is running via the getCmdLineOpts command.

Basic Options

<table>
<tr><td>-h | --help</td><td>Shows all options</td></tr>
<tr><td>-f | --config <file></td><td>Specify a configuration file to use</td></tr>
<tr><td>--port <portno></td><td>Specifies the port number on which Mongo will listen for client connections. Default is 27017</td></tr>
<tr><td>--dbpath <path></td><td>Specifies the directory for datafiles. Default is /data/db or c:\data\db</td></tr>
<tr><td>--fork</td><td>Fork the server process</td></tr>
<tr><td>--bind_ip <ip></td><td>Specifies a single IP that the database server will listen for</td></tr>
<tr><td>--directoryperdb</td><td>Specify use of an alternative directory structure, in which files for each database are kept in a unique directory. (more details) (v1.4+)</td></tr>
<tr><td>--journalCommitInterval</td><td>How often to group/batch commit (ms). Default 100ms</td></tr>
<tr><td>--nojournal</td><td>Disable journaling. In v2.0+, journaling is on by default for 64-bit</td></tr>
<tr><td>--quiet</td><td>Reduces amount of log output (more details)</td></tr>
<tr><td>--nohttpinterface</td><td>Disable the HTTP interface (localhost:28017)</td></tr>
<tr><td>--rest</td><td>Allow extended operations at the Http Interface</td></tr>
<tr><td>--logpath <file></td><td>File to write logs to (instead of stdout). You can rotate the logs by sending SIGUSR1 to the server.</td></tr>
<tr><td>--logappend</td><td>Append to existing log file, instead of overwritting</td></tr>
<tr><td>--syslog</td><td>Send all output to syslog. Added in 2.1.0. Not compatible with --logpath</td></tr>
<tr><td>--repairpath <path></td><td>Root path for temporary files created during database repair. Default is dbpath value.</td></tr>
<tr><td>--cpu</td><td>Enables periodic logging of CPU utilization and I/O wait</td></tr>
<tr><td>--noauth</td><td>Turns off security. This is currently the default</td></tr>
<tr><td>--auth</td><td>Turn on security</td></tr>
<tr><td>-v[v[v[v[v]]]] | --verbose</td><td>Verbose logging output (-vvvvv is most verbose, -v == --verbose)</td></tr>
</table>

--objcheck	Inspect all client data for validity on receipt (useful for developing drivers)
--syncdelay arg (=60)	seconds between disk syncs. Do not use zero. (more details)
--diaglog <n>	Set oplogging level where n is 0=off (default) 1=W 2=R 3=both 7=W+some reads
--nocursors	Diagnostic/debugging option – limits results to single batch request. Do NOT use in production. This option will be removed in 2.1.0.
--nohints	Ignore query hints
--noscripting	Turns off server-side scripting. This will result in greatly limited functionality
--notablescan	Turns off table scans. Any query that would do a table scan fails
--noprealloc	Disable data file preallocation
--smallfiles	Use a smaller initial file size (16MB) and maximum size (512MB)
--nssize <MB>	Specifies .ns file size for new databases
--slowms <value>	Specifies the threshold (in milliseconds) above which long-running queries will appear in the log, and in the system.profiler collection if profiling is enabled.
--sysinfo	Print system info as detected by Mongo and exit
--nounixsocket	disable listening on unix sockets (will not create socket files at /tmp/mongodb-<port>.sock)
--upgrade	Upgrade database files to new format if necessary (required when upgrading from <= v1.0 to v1.1+)

Replica Set Options

--replSet <setname>[/<seedlist>]	Use replica sets with the specified logical set name. Typically the optional seed host list need not be specified.
--oplogSize <MB>	Custom size for replication operation log

Master/Slave Replication Options

Please use replica sets instead of master/slave.

--master	Designate this server as a master in a master-slave setup
--slave	Designate this server as a slave in a master-slave setup
--source <server:port>	Specify the source (master) for a slave instance
--only <db>	Slave only: specify a single database to replicate
--arbiter <server:port>	Address of arbiter server
--autoresync	Automatically resync if slave data is stale
--oplogSize <MB>	Custom size for replication operation log
--fastsync	If the node has a completely up-to-date copy of the data, use this option to let it know it can skip the resync. Be careful – the server will assume it is caught up completely and if not so the data will be out of sync permanently.

--directoryperdb

By default data files for MongoDB are all created in a single directory. This directory is defined by the `--dbpath` variable and defaults to `/data/db`.

The "directory per DB" option (`--directoryperdb`) allows for a separate directory for the files of each database.

Uses

The most common use for this option is to enable data to be stored on different physical disks. This is generally accomplished by creating a symbolic link to the appropriate underlying hardware.

Example

The following sample demonstrates the difference in directory structures. The `db/` directory is run without "directory per DB", the `db2/` directory is run *with* "directory per DB".

```
db:
total 417M
-rw------- 1 mongo mongo  64M 2011-04-05 12:50 foo.0
-rw------- 1 mongo mongo 128M 2011-04-05 12:50 foo.1
-rw------- 1 mongo mongo  16M 2011-04-05 12:50 foo.ns
-rw------- 1 mongo mongo  64M 2011-04-05 12:48 test.0
-rw------- 1 mongo mongo 128M 2011-04-05 12:48 test.1
-rw------- 1 mongo mongo  16M 2011-04-05 12:48 test.ns
-rwxr-xr-x 1 mongo mongo    5 2011-04-05 12:47 mongod.lock

db2:
total 16K
drwxr-xr-x 2 mongo mongo 4.0K 2011-04-05 12:50 foo
-rwxr-xr-x 1 mongo mongo    5 2011-04-05 12:47 mongod.lock
drwxr-xr-x 2 mongo mongo 4.0K 2011-04-05 12:48 test

db2/foo:
total 209M
-rw------- 1 mongo mongo  64M 2011-04-05 12:50 foo.0
-rw------- 1 mongo mongo 128M 2011-04-05 12:50 foo.1
-rw------- 1 mongo mongo  16M 2011-04-05 12:50 foo.ns

db2/test:
total 209M
-rw------- 1 mongo mongo  64M 2011-04-05 12:48 test.0
-rw------- 1 mongo mongo 128M 2011-04-05 12:48 test.1
-rw------- 1 mongo mongo  16M 2011-04-05 12:48 test.ns
```

--quiet

When the `mongod` process is run with `--quiet`, certain messages are suppressed from the logs.

The following log messages are suppressed:

1. Connection accepted & connection closed
2. Commands: like... `drop`, `dropIndex`, `diaglogging`, `validate`, `clean`, ...
3. Replication syncing activity: sleeping, "repl: from... host"

> we are not suppressing the actual commands, just suppressing the logging of the execution of those commands

The setting of `quiet` does not influence the setting of `verbose`. A server can be both quiet and verbose.

Example log entries suppressed

```
#1: Connection accepted & closed
[initandlisten] connection accepted from 127.0.0.1:52798 #3
[conn3] end connection 127.0.0.1:52798

#2: Commands
[conn10] CMD: drop test.foo
[conn10] CMD: dropIndexes test.foo

#3: Replication syncing activity
[replslave] repl: sleep 1sec before next pass
[replslave] repl: from host:127.0.0.1:6900
```

--syncdelay

This command line option specifies the number of seconds between data file flushes/syncs.

The MongoDB journal file flushes almost immediately, but data files are flushed lazily to optimize performance.

The default is 60 seconds.

A 0 setting means never, but is not recommended and should **never** be used with journaling enabled.

On Linux, longer settings will likely be ineffective unless `/proc/sys/vm/dirty_expire_centisecs` is adjusted also. Generally, set syncdelay to 4x the desired dirty_expire_centiseconds value.

If backgroundFlushing.average_ms is relatively large (>= 10,000 milliseconds), consider increasing this setting as well as `dirty_expire_centiseconds`.

Under extremely high write volume situations, a higher syncdelay value may result in more journal files as they cannot be rotated (deleted) as quickly.

File Based Configuration

In addition to accepting Command Line Parameters, MongoDB can also be configured using a configuration file. A configuration file can be specified using the `-f` or `--config` command line options. On some packaged installs of MongoDB (for example Ubuntu & Debian), the default file can be found in /etc/mongodb.conf, which is automatically used when starting and stopping MongoDB from the service.

The following example configuration file demonstrates the syntax to use:

```
# This is an example config file for MongoDB.
dbpath = /var/lib/mongodb
bind_ip = 127.0.0.1
noauth = true # use 'true' for options that don't take an argument
verbose = true # to disable, comment out.
```

Parameters

Before 2.0
Unfortunately, flag parameters like "quiet" will register as true no matter what value you put after them. So don't write this: "quiet=false", just comment it out, or remove the line.

In 2.0, you can use `option=false` in the config file and it will be handled correctly (the option will be ignored).

Suppose you were running 1.8 and using a config file with the contents:

```
auth = false
```

This would actually **enable** authentication in 1.8. If you upgrade to 2.0, this would suddenly be parsed correctly, turning off authentication (which you might not want, if you've been running with it on forever). Thus, if you start up with 2.0+ and you have any `option=false` options in your config file, you'll get a warning that looks like:

```
warning: remove or comment out this line by starting it with '#', skipping now : auth=false
```

This is to let you know that this option is being skipped. If you want this option skipped, feel free to ignore this warning.

Basic database configuration

Parameter	Meaning	Example
dbpath	Location of the database files	`dbpath = /var/lib/mongodb`
port	Port the mongod will listen on	`port = 27017`
bind_ip	Specific IP address that mongod will listen on	`bind_ip = 127.0.0.1`
logpath	Full filename path to where log messages will be written	`logpath = /var/log/mongodb/mongodb.log`
logappend	Whether the log file will be appended to or over-written (default) at start-up	`logappend = true`

Logging

Parameter	Meaning	Example
cpu	Enable periodic logging of CPU utilization and I/O wait	`cpu = true`
verbose	Verbose logging output (same as `v = true`)	`verbose=true`
v[vvvv]	Level of verbose logging output	`vvvvv = true`

Security

Parameter	Meaning	Example
noauth	Turn authorization off. Off is currently the default	`noauth = true`
auth	Turn authorization on. Off is currently the default	`auth = true`
keyFile	Private key for cluster authentication.	`keyFile = /var/lib/mongodb/key`

Administration & Monitoring

Parameter	Meaning	Example
nohttpinterface	Disable the HTTP interface. The default port is 1000 more than the dbport	`nohttpinterface = true`
rest	Turn on simple rest API	`rest = true`
noscripting	Turns off server-side scripting. This will result in greatly limited functionality	`noscripting = true`
notablescan	Turns off table scans. Any query that would do a table scan fails.	`notablescan = true`
noprealloc	Disable data file preallocation.	`noprealloc = true`
nssize	Specify .ns file size for new databases in MB	`nssize = 16`

Replication

All

Parameter	Meaning	Example
oplogSize	Custom size for replication operation log in MB.	`oplogSize = 100`

Replica Pairs (Deprecated – do not use)

Parameter	Meaning	Example

opIdMem	Size limit for in-memory storage of op ids in bytes	`opIdMem = 1000`
arbiter		

Master-slave Replication

Parameter	Meaning	Example
master	In replicated mongo databases, specify here whether this is a slave or master	`master = true`
slave	In replicated mongo databases, specify here whether this is a slave or master	`slave = true`
source	Specify the	`source = master.example.com`
only	Slave only: specify a single database to replicate	`only = master.example.com`
autoresync	Automatically resync if slave data is stale	autoresync
fastsync	Indicate that this instance is starting from a dbpath snapshot of the repl peer	

Replica Sets

Parameter	Meaning	Example
replSet	Use replica sets with the specified logical set name. Typically the optional seed host list need not be specified.	`replSet = <setname>[/<seedlist>]`

Sharding

Parameter	Meaning	Example
shardsvr	Indicates that this mongod will participate in sharding * This is optional and just changes the default port	`shardsvr = true`

Journaling

Parameter	Meaning	Example
journal	Whether writes will be journaled or not. This feature enables fast recovery from crashes	`journal = true`
nojournal	Turns off journaling if it is enabled by default (1.9.2+)	`nojournal = true`

Notes

- Lines starting with octothorpes (#) are comments
- Options are case sensitive
- The syntax is assignment of a value to an option name
- All command line options are accepted, for example "vvvv=true"

GridFS Tools

File Tools

`mongofiles` is a tool for manipulating GridFS from the command line.

Example:

```
$ ./mongofiles list
connected to: 127.0.0.1

$ ./mongofiles put libmongoclient.a
connected to: 127.0.0.1
done!

$ ./mongofiles list
connected to: 127.0.0.1
libmongoclient.a 12000964

$ cd /tmp/

$ ~/work/mon/mongofiles get libmongoclient.a

$ ~/work/mongo/mongofiles get libmongoclient.a
connected to: 127.0.0.1
done write to: libmongoclient.a

$ md5 libmongoclient.a
MD5 (libmongoclient.a) = 23a52d361cfa7bad98099c5bad50dc41

$ md5 ~/work/mongo/libmongoclient.a
MD5 (/Users/erh/work/mongo/libmongoclient.a) = 23a52d361cfa7bad98099c5bad50dc41
```

DBA Operations from the Shell

This page lists common DBA-class operations that one might perform from the MongoDB shell.

Note one may also create .js scripts to run in the shell for administrative purposes.

```
help                         show help
show dbs                     show database names
show collections             show collections in current database
show users                   show users in current database
show profile                 show most recent system.profile entries with time >= 1ms
use <db name>                set curent database to <db name>

db.addUser (username, password)
db.removeUser(username)

db.cloneDatabase(fromhost)
db.copyDatabase(fromdb, todb, fromhost)
db.createCollection(name, { size : ..., capped : ..., max : ... } )

db.getName()
db.dropDatabase()

// runs the collstats] command on each collection in the database
db.printCollectionStats()

db.currentOp() displays the current operation in the db
db.killOp() kills the current operation in the db

db.getProfilingLevel()
db.setProfilingLevel(level) 0=off 1=slow 2=all

db.getReplicationInfo()
db.printReplicationInfo()
db.printSlaveReplicationInfo()
db.repairDatabase()

db.version() current version of the server

db.shutdownServer()
```

Commands for manipulating and inspecting a collection:

```
db.foo.drop() drop the collection
db.foo.dropIndex(name)
db.foo.dropIndexes()
db.foo.getIndexes()
db.foo.ensureIndex(keypattern,options) - options object has these possible
                                         fields: name, unique, dropDups

db.foo.find( [query] , [fields])       - first parameter is an optional
                                         query filter. second parameter
                                         is optional
                                         set of fields to return.
                                    e.g. db.foo.find(
                                                 { x : 77 } ,
                                                 { name : 1 , x : 1 } )
db.foo.find(...).count()
db.foo.find(...).limit(n)
db.foo.find(...).skip(n)
db.foo.find(...).sort(...)
db.foo.findOne([query])

db.foo.getDB() get DB object associated with collection

db.foo.count()
db.foo.group( { key : ..., initial: ..., reduce : ...[, cond: ...] } )

db.foo.renameCollection( newName ) renames the collection

db.foo.stats()
db.foo.dataSize()
db.foo.storageSize() - includes free space allocated to this collection
db.foo.totalIndexSize() - size in bytes of all the indexes
db.foo.totalSize() - storage allocated for all data and indexes
db.foo.validate() (slow)

db.foo.insert(obj)
db.foo.update(query, object[, upsert_bool])
db.foo.save(obj)
db.foo.remove(query)                   - remove objects matching query
                                         remove({}) will remove all
```

See Also

- collStats (stats) command
- Commands

Architecture and Components

This is a list of the components (processes) in the MongoDB server package.

- mongod - The core database process
- mongos - Sharding controller
- mongo - The database shell (uses interactive javascript)

- Import Export Tools
  - mongoimport
  - mongoexport
  - mongodump
  - mongorestore
  - bsondump

- mongofiles - the GridFS utility

- mongostat

MongoDB has two primary components to the database server. The first is the `mongod` process which is the core database server. In many cases, mongod may be used as a self-contained system similar to how one would use mysqld on a server. Separate mongod instances on different machines (and data centers) can replicate from one to another.

Another MongoDB process, `mongos`, facilitates auto-sharding. mongos can be thought of as a "database router" to make a cluster of mongod processes appear as a single database. See the sharding documentation for more information.

See Also

- Mongo Concepts and Terminology

Windows

Windows Quick Links and Reference Center

Running MongoDB on Windows

See the Quickstart page for info on how to install and run the database for the first time.

Running as a Service

See the Windows Service page.

The MongoDB Server

Get pre-built binaries on the Downloads page. Binaries are available for both 32 bit and 64 bit Windows. MongoDB uses memory-mapped files for data storage, so for servers managing more than 2GB of data you will definitely need the 64 bit version (and a 64 bit version of Windows).

Writing Apps

You can write apps in almost any programming language – see the Drivers page. In particular C#, .NET, PHP, C and C++ work just fine.

- CSharp Language Center
- CSharp Community Projects

Building

We recommend using the pre-built binaries, but Mongo builds fine with Visual Studio 2008 and 2010. See the Building for Windows page.

Versions of Windows

We have successfully ran MongoDB (mongod etc.) on:

- Windows Server 2008 R2 64 bit
- Windows 7 (32 bit and 64 bit)
- Windows XP
- Vista

Azure

Instructions for running MongoDB on Azure (alpha)

AppHarbor

Instructions for running MongoDB on AppHarbor
Sample ASP.NET MVC app that uses MongoDB from AppHarbor

MongoDB on Azure

> ⚠ The MongoDB Replica Set Azure wrapper is currently a preview release. Please provide feedback, mongodb-dev, mongodb-user and IRC #mongodb are good places!

- Getting the package
- Components
- Initial Setup
- Building
- Deploying and Running
    - Running locally on compute/storage emulator

The MongoDB Wrapper for Azure allows you to deploy and run a MongoDB replica set on Windows Azure. Replica set members are run as Azure worker role instances. MongoDB data files are stored in an Azure Blob mounted as a cloud drive. One can use any MongoDB driver to connect to the MongoDB server instance. The MongoDB C# driver v1.3.1 is included as part of the package.

Getting the package

The MongoDB Azure Wrapper is delivered as a Visual Studio 2010 solution with associated source files. The simplest way to get the package is by downloading it from GitHub. It is recommended using the latest tagged version.

Alternatively, you can clone the repository run the following commands from a git bash shell:

```
$ cd <parentdirectory>
$ git config --global core.autocrlf true
$ git clone git@github.com:mongodb/mongo-azure.git
$ cd mongo-azure
$ git config core.autocrlf true
```

You must set the global setting for core.autocrlf to true before cloning the repository. After you clone the repository, we recommend you set the local setting for core.autocrlf to true (as shown above) so that future changes to the global setting for core.autocrlf do not affect this repository. If you then want to change your global setting for core.autocrlf to false run:

```
$ git config --global core.autocrlf false
```

Components

Once you have unzipped the package or cloned the repository, you will see the following directories:

- SampleApplications - This directory contains sample applications that run against MongoDB on Azure. More information can be found in the README

- ReplicaSets - This directory contains the bare MongoDB Replica Set worker role code. Use this if you are familiar with Azure and are ready to build your own application on top of MongoDB running on Azure

Initial Setup

Run the appropriate `*setup.cmd` to setup up the solution for building. The script only needs to be run once. The script does the following:

- Creates ServiceConfiguration.Cloud.cscfg as a copy of configfiles/ServiceConfiguration.Cloud.cscfg.ref
- Downloads the MongoDB binaries (currently 2.1.0-pre) to the appropriate solution location

Building

The prerequisites can be found in the Github readme

Once these are installed, you can open MongoAzure.sln from Visual Studio and build the solution.

Deploying and Running

Running locally on compute/storage emulator

The following instructions are for running the sample application.

To start, you can test out your setup locally on your development machine. The default configuration has 3 replica set members running on ports 27017, 27018 and 27019 with a replica set name of 'rs'.

In Visual Studio, run the solution using F5 or Debug->Start Debugging. This will start up the replica set and the MvcMovie sample application.

You can verify this by using the MvcMovie application in the browser or by running mongo.exe against the running instances.

Deploying to Azure

Once you have the application running locally, you can deploy the sample app solution to Windows Azure. **Note** You cannot execute locally against storage in Azure due to the use of Windows Azure Drive.

- Detailed configuration options are outlined here
- Step-by-step deployment instructions are here

Additional Information

The azure package wraps and runs mongod.exe with the following mongod command line options:

```
--dbpath --port --logpath --journal --nohttpinterface --logappend --replSet
```

MongoDB 2.1.0-pre is currently used in this package.

MongoDB creates the following containers and blobs on Azure storage:

- Mongo Data Blob Container Name - mongoddatadrive(replica set name)
- Mongo Data Blob Name - mongoddblob(instance id).vhd

FAQ/Troubleshooting

- Can I run mongo.exe to connect?
  - Yes if you set up remote desktop. Then you can connect to the any of the worker role instances and run e:\approot\MongoDBBinaries\bin\mongo.exe.
- Role instances do not start on deploy to Azure
  - Check if the storage URLs have been specified correctly.

Known issues/Where do I file bugs?

https://jira.mongodb.org/browse/AZURE

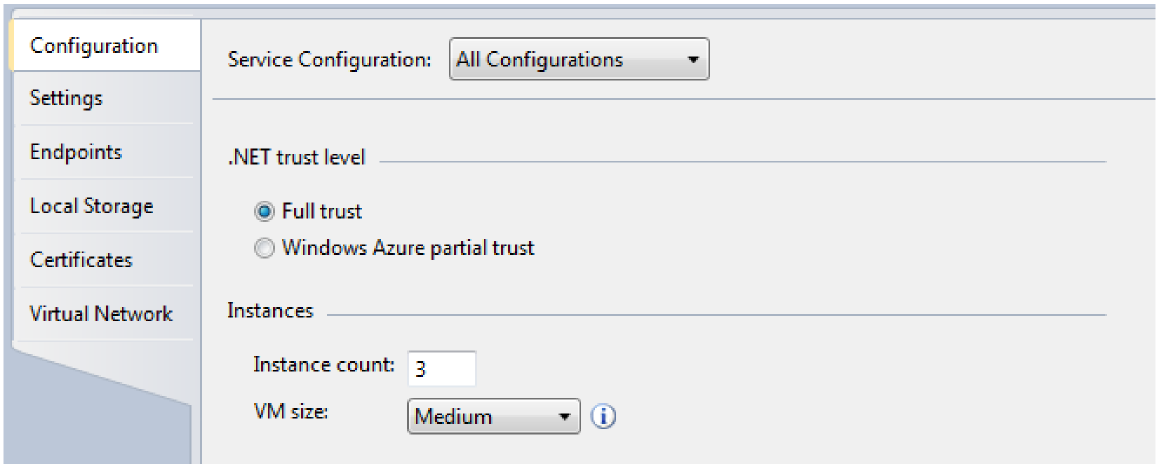
Azure Configuration

- ReplicaSetRole configuration

The following are the configuration operations available as part of the MongoDB Replica Sets Azure package.

ReplicaSetRole configuration

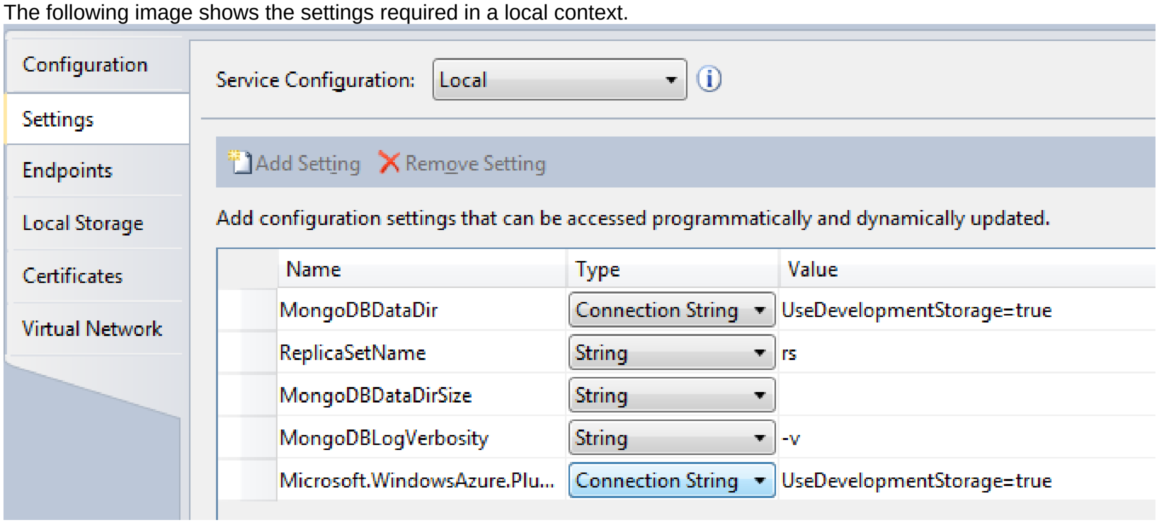
- Configuration
  - **Instance count** - Set to the number of replica set members you require. Default is 3.
    - Setting this to 1 would run a replica set with 1 instance (equivalent to stand alone)
  - **VM Size** - Choose size of Medium or higher. **Note** The I/O characteristics of small or extra small instance make these configurations unsuitable for MongoDB.



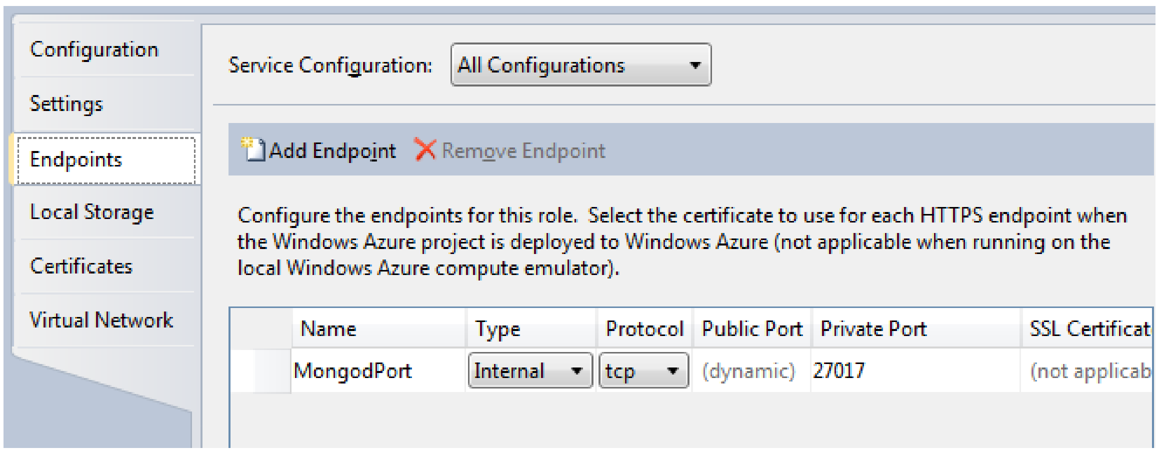
- Settings
  - **MongoDBDataDir** - Storage for mongo data files --dbpath. Configure for development or deployment. Data files are in a subdirectory called `data`.
  - **MongoDBDataDirSize** - Size of blob (in MegaBytes) allocated for mongodb data directory. Default is 1GB for the emulator and 100GB for deployed.
  - **ReplicaSetName** - Name of the mongodb replica set. Default is rs.

- **MongoDBLogVerbosity** - Verbosity to start mongod with. Default is -v
- **Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString** - Storage for diagnostics information.

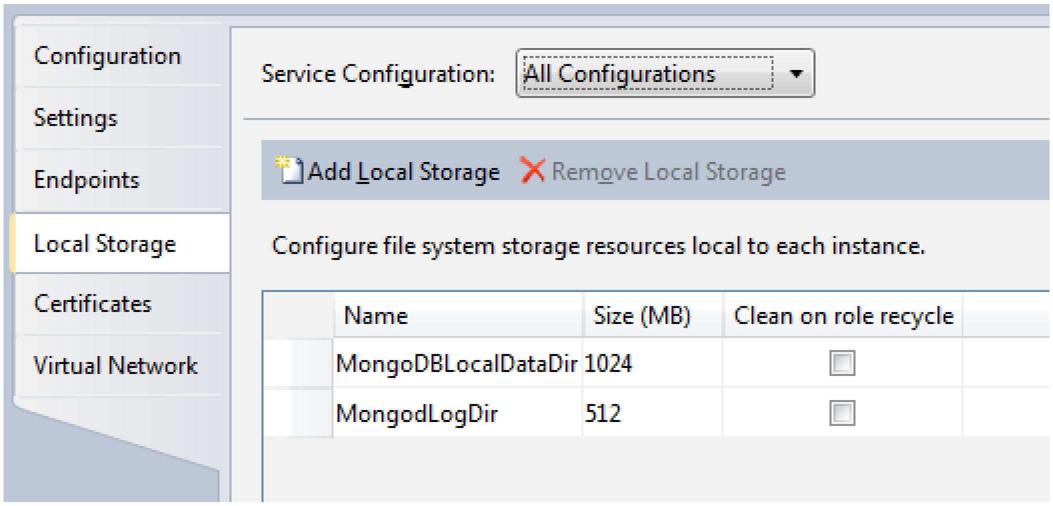
The following image shows the settings required in a local context.



- Endpoints
  - **MongodPort** - Port on which mongod listens. Default is 27017. If running in the Azure emulator this is port for the first instance that is started.



- Local Storage
  - **MongoDBLocalDataDir** - This specifies the size of the local storage used as a cache for the Windows Azure Drive used as the mongod data directory. Larger sizes provide better read performance.
  - **MongodLogDir** - Size of storage allocated for mongod.log. Make this large enough depending on verbosity chosen and instance size.
    Use the default settings for local storage as specified below when running on the Azure emulator. When deploying to Azure, change local storage size based on instance size chosen. MongoDBLocalData



Azure Deployment

In a development environment

The solution can be built and run in a development environment using the Azure emulators as is using Visual Studio 2010 (if adequate disk space is available). Since this solution uses Cloud Drive you cannot run from a development environment against Azure cloud storage. Instead, when you run in your development environment it uses development storage:

- The mongod log file is at `C:\Users\<user>\AppData\Local\dftmp\Resources\<deploymentid>\directory\MongodLogDir` **Note** - On a development environment the port mongod listens on would be configured port (27017 by default) + instance id.
- The mongod data files are at `C:\Users\<user>\AppData\Local\dftmp\wadd\devstoreaccount1\mongoddatadrive(replica set name)\mongoddblob(instance id).vhd\data`.
  Note - Additionally when the app is running you should be able to access the drives on mounting as you would access any mounted drive.

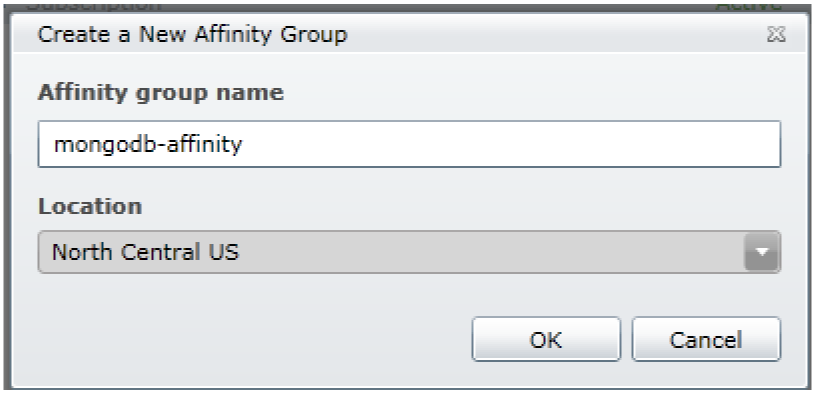
In the Azure environment

Login to the Azure management portal using your azure credentials

Azure setup (first time only per deployment)

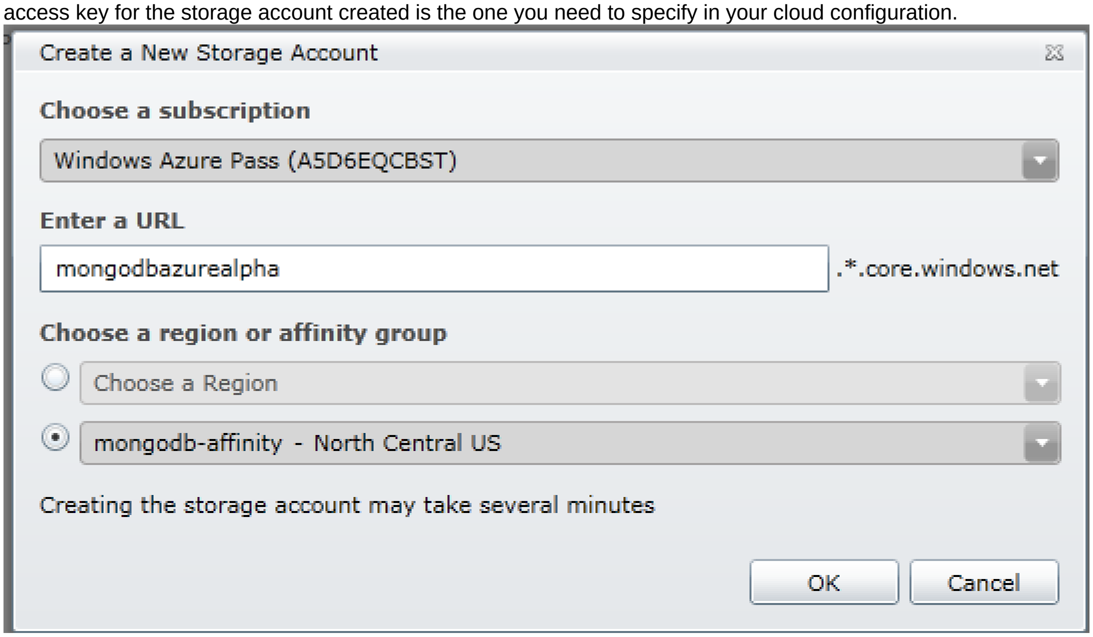
Affinity Group

Create an affinity group for your deployment. Choose your required location for the group.



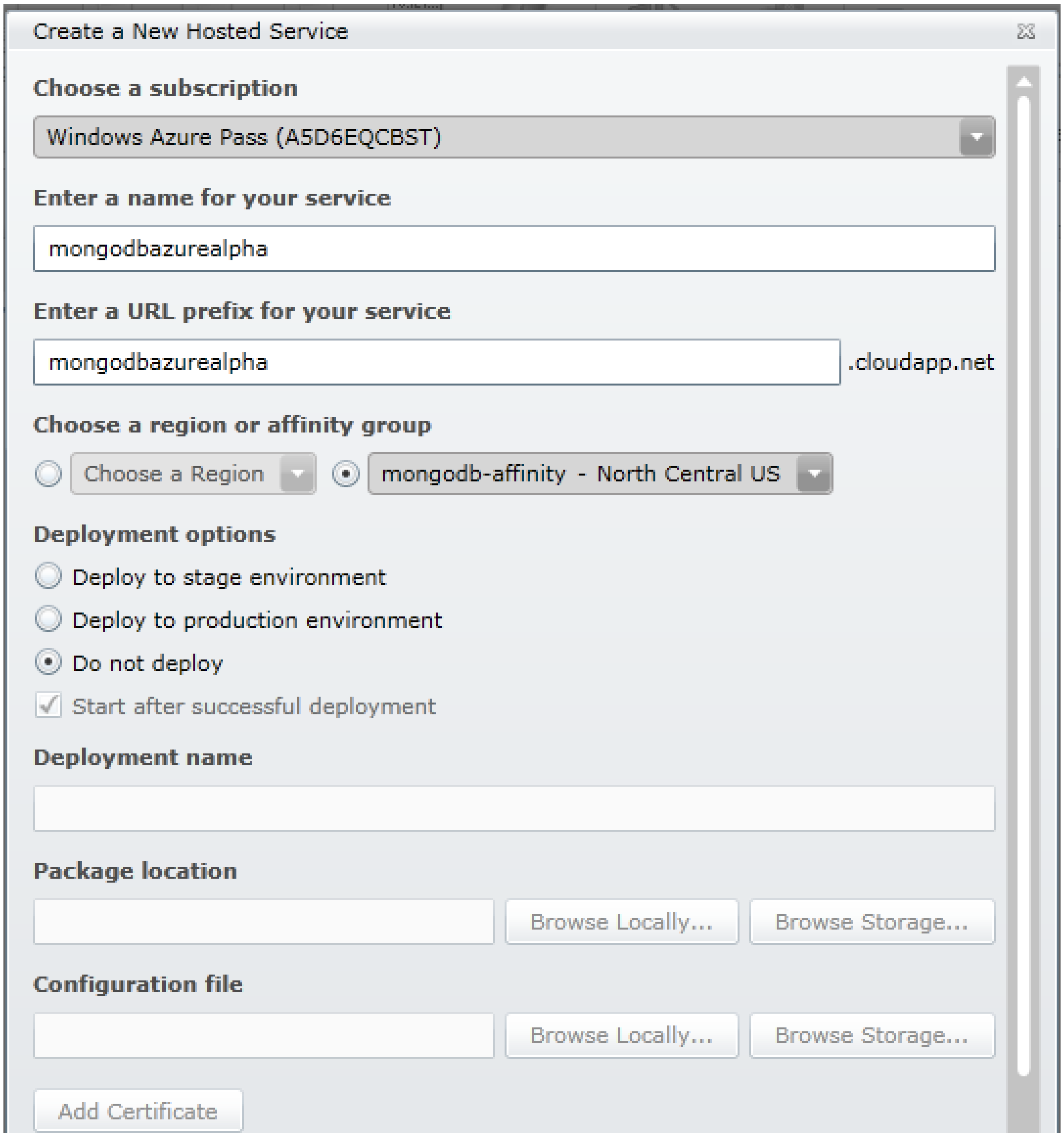
Storage account

Create the required number of storage account(s) to store the blobs. For region/affinity choose the affinity group you created earlier. **Note** The access key for the storage account created is the one you need to specify in your cloud configuration.



Service

Create a new hosted service to host your Mongo package. For region/affinity group use the same affinity group as your storage account. **Note** for cloud drives the compute and storage instances should be in the same azure domain. Choose do not deploy



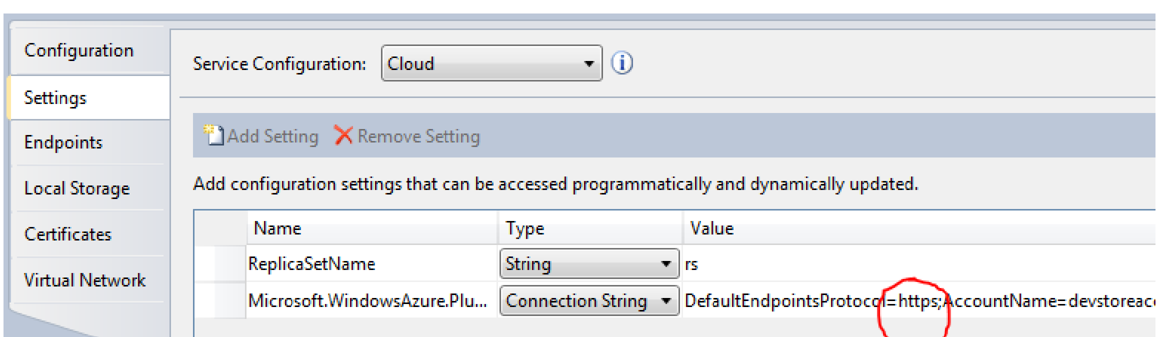
Deployment configuration for Mvc Role

If deploying the sample app you can use the default settings as is. You would only need to set the storage settings for diagnostics

Settings

In the Settings tab

- **ReplicaSetName** - This should be the same as the replica set name specified in the ReplicaSetRole
- **Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString** - Specify your azure storage credentials. Ensure that the connection mode is https.



Deployment configuration for Replica Set Role

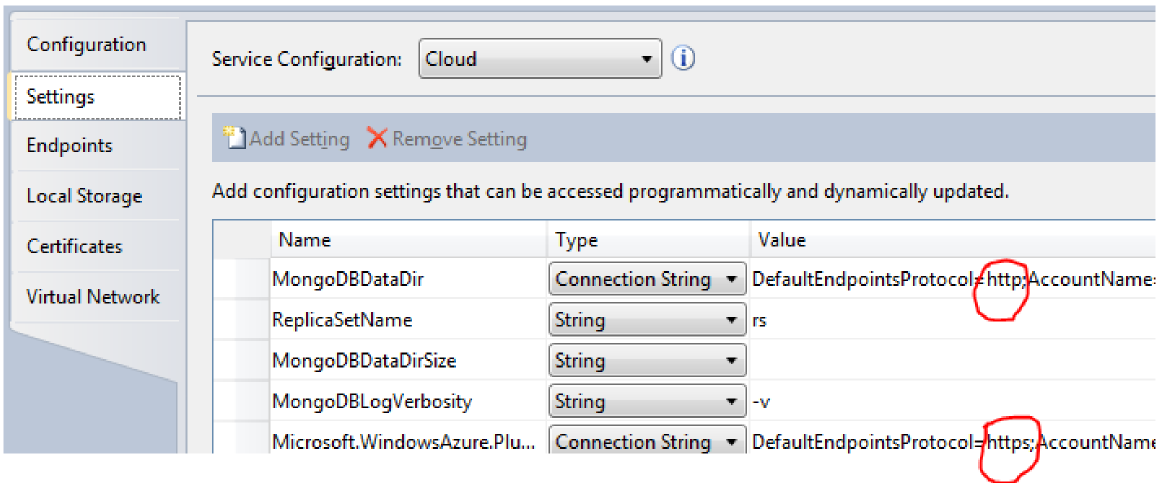
Configuration

- Ensure VM size for ReplicaSetRole is at least Medium. Larger instance sizes provide more RAM and also greater bandwidth. More information on instance sizes can be found here
- Set the Instance Count equivalent to the required number of replica set members. Default is 3.

Settings

Change connection setting from UseDevelopment storage to actual storage account credentials. It is recommended to use different storage accounts for data and diagnostics. This would allow you to give access to external monitors for diagnostics information without giving access to your data.

- **MongoDBDataDir** - Ensure that connection mode is **http**
- **ReplicaSetName** - This is the name of the replica set in the replica set configuration. This is also the suffix to the blob container created in your storage account. **Note** - This needs to be the same as the replica set name in the client application.
- **MongoDBDataDirSize** - Maximum size of your cloud drive where mongod data files are stored. Currently the maximum size can be 1TB.
- **MongoDBLogVerbosity** - Verbosity for mongod logging. Default is -v
- **Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString** - Ensure that the connection mode is **https**

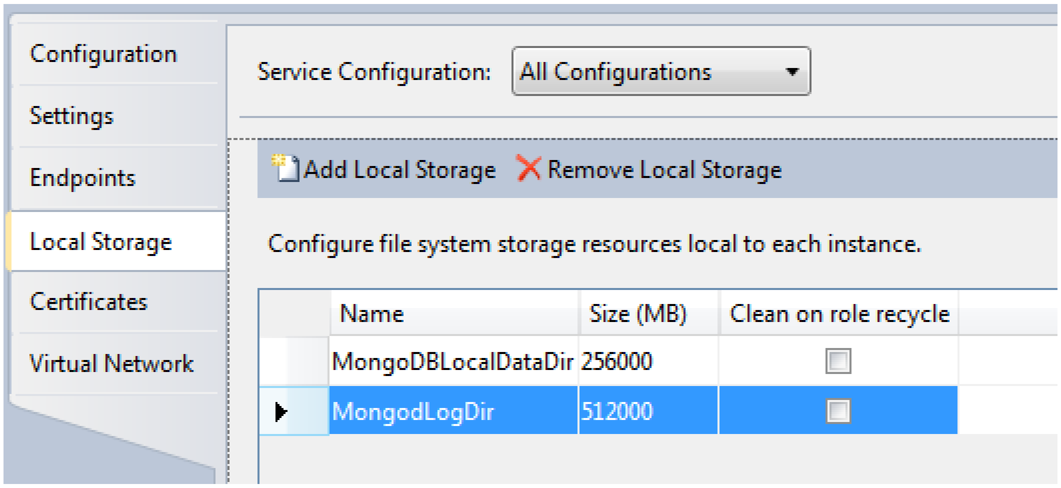


Note - If deploying multiple Azure instances make sure you use different storage accounts for each of the deployments or different replica set names if using the same storage account.

Local Storage

Configure the amount of local storage required depending on the VM size chosen. Ensure Clean on recycle role is unchecked. The following are recommendations of Local Storage **Note** All sizes are in MB.

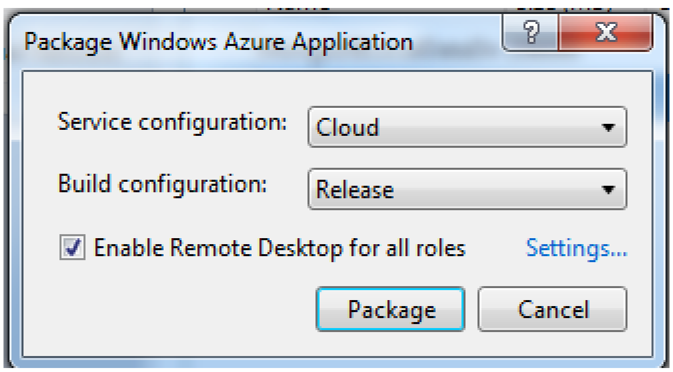
VM Size	MongoDBLocalDataDir	MongodLogDir
Medium	256000 (250 GB)	51200 (50GB)
Large	768000 (750 GB)	51200 (50GB)
Extra Large	1024000 (1000GB)	51200 (50GB)



Package and Publish

Create the Azure package by right clicking on the cloud project and choosing Package. Choose the Service Configuration as Cloud and Build Configuration as Release.

- Enable remote desktop on the role intances if required http://msdn.microsoft.com/en-us/library/gg443832.aspx



- Deploy the created package from the Azure management portal. More details on deploying can be found at http://msdn.microsoft.com/en-us/magazine/ee336122.aspx

When deploying to the Azure cloud make sure to check deployment warnings/errors to see for any breaking issues. Some common errors are

- Remote desktop is enabled but the remote desktop certificate has not been uploaded to the portal
- https is not chosen for diagnostics connection string
- If deploying the sample MvcMovie application, you can safely ignore the warning that indicates you are only running 1 instance of it.

Troubleshooting

mongod process "disappeared"

Scenario here is the log ending suddenly with no error or shutdown messages logged.

On Unix, check /var/log/messages:

```
$ sudo grep mongod /var/log/messages
$ sudo grep score /var/log/messages
```

Socket errors in sharded clusters and replica sets

If you're experiencing unexplainable socket errors between nodes in a sharded cluster or replica set, you may want to check your TCP keepalive time. The default Linux keepalive time is 7200 seconds (2 hours); we recommed that you change this to 300 seconds (five minutes).

To check your current setting:

```
cat /proc/sys/net/ipv4/tcp_keepalive_time
```

You'll usually see this:

```
7200
```

To change this setting:

```
echo 300 > /proc/sys/net/ipv4/tcp_keepalive_time
```

Note that you must alter the `tcp_keepalive_time` value on all machines hosting MongoDB processes. This include machines hosting `mongos` routers, config servers, and `mongod` servers. You should not need to restart the mongo processes themselves, the setting will take effect immediately.

See Also

- Diagnostic Tools

Excessive Disk Space

- Understanding Disk Usage
    - local.* files and replication
    - Datafile Preallocation
    - Recovering Deleted Space
    - Running out of disk space
    - Checking Size of a Collection
    - Helpful scripts

Understanding Disk Usage

You may notice that for a given set of data the MongoDB datafiles in `/data/db` are larger than the data set inserted into the database. There are several reasons for this.

local. files and replication*

The replication oplog is preallocated as a capped collection in the local database.

The default allocation is approximately 5% of disk space (*64 bit installations*).

If you would like a smaller oplog size use the `--oplogSize` command line parameter.

Datafile Preallocation

Each datafile is preallocated to a particular size. (This is done to prevent file system fragmentation, among other reasons.) The first filename for a database is <dbname>.0, then <dbname>.1, etc. <dbname>.0 will be 64MB, <dbname>.1 128MB, et cetera, up to 2GB. Once the files reach 2GB in size, each successive file is also 2GB.

Thus, if the last datafile present is, say, 1GB, that file might be 90% empty if it was recently created.

Additionally, on Unix, mongod will preallocate an additional datafile in the background and do background initialization of this file. These files are prefilled with zero bytes. This initialization can take up to a minute (less on a fast disk subsystem) for larger datafiles. Pre-filling in the background prevents significant delays when a new database file is next allocated.

On Windows, additional datafiles are not preallocated. NTFS can allocate large files filled with zeroes relatively quickly, rendering preallocation unnecessary.

As soon as a datafile starts to be used, the next one will be preallocated.

You can disable preallocation with the `--noprealloc` command line parameter. This flag is nice for tests with small datasets where you drop the database after each test. It should not be used on production servers.

For large databases (hundreds of GB or more), this is of no significant consequence as the unallocated space is relatively small.

On Linux systems you can use `hdparam` to allocate files to get an idea of how costly allocation might be:

```
time hdparm --fallocate $((1024*1024)) testfile
```

Recovering Deleted Space

MongoDB maintains lists of deleted blocks within the datafiles when objects or collections are deleted. This space is reused by MongoDB but never freed to the operating system.

To shrink the amount of physical space used by the datafiles themselves, by reclaiming deleted blocks, you must rebuild the database by using (`db.repairDatabase()`).

> repairDatabase copies all the database records to new file(s). You will need enough free disk space to hold both the old and new database files while the repair is running. Be aware that repairDatabase will block and will take a long time to complete.

Rather than compacting an entire database, you can compact just a single collection by using `db.runCommand({compact:'collectionname'})`. This does not shrink any datafiles, however; it only defragments deleted space so that larger objects might reuse it. The compact command will never delete or shrink database files, and in general requires extra space to do its work. Thus, it is not a good option when you are running critically low on disk space.

> When testing and investigating the size of datafiles, if your data is just test data, use db.dropDatabase() to clear all datafiles and start fresh.

Running out of disk space

If your server runs out of disk space you will see something like this in the log:

```
Thu Aug 11 13:06:09 [FileAllocator] allocating new datafile dbms/test.13, filling with zeroes...
Thu Aug 11 13:06:09 [FileAllocator] error failed to allocate new file: dbms/test.13 size: 2146435072
errno:28 No space left on device
Thu Aug 11 13:06:09 [FileAllocator]     will try again in 10 seconds
Thu Aug 11 13:06:19 [FileAllocator] allocating new datafile dbms/test.13, filling with zeroes...
Thu Aug 11 13:06:19 [FileAllocator] error failed to allocate new file: dbms/test.13 size: 2146435072
errno:28 No space left on device
Thu Aug 11 13:06:19 [FileAllocator]     will try again in 10 seconds
```

The server remains in this state forever blocking all writes including deletes. However, reads still work. To delete some data and compact (see above), you must restart the server first.

Checking Size of a Collection

Use the validate command to check the size of a collection -- that is from the shell run:

```
> db.<collectionname>.validate();

> // these are faster:
> db.<collectionname>.dataSize(); // just data size for collection
> db.<collectionname>.storageSize(); // allocation size including unused space
> db.<collectionname>.totalSize(); // data + index
> db.<collectionname>.totalIndexSize(); // index data size
```

This command returns info on the collection data but note there is also data allocated for associated indexes. These can be checked with validate too, if one looks up the index's namespace name in the system.namespaces collection. For example:

```
> db.system.namespaces.find()
{"name" : "test.foo"}
{"name" : "test.system.indexes"}
{"name" : "test.foo.$_id_"}
> > db.foo.$_id_.validate()
{"ns" : "test.foo.$_id_" , "result" : "
validate
  details: 0xb3590b68 ofs:83fb68
  firstExtent:0:8100 ns:test.foo.$_id_
  lastExtent:0:8100 ns:test.foo.$_id_
  # extents:1
  datasize?:8192 nrecords?:1 lastExtentSize:131072
  padding:1
  first extent:
    loc:0:8100 xnext:null xprev:null
    ns:test.foo.$_id_
    size:131072 firstRecord:0:81b0 lastRecord:0:81b0
  1 objects found, nobj:1
  8208 bytes data w/headers
  8192 bytes data wout/headers
  deletedList: 0000000000001000000
  deleted: n: 1 size: 122688
  nIndexes:0
" , "ok" : 1 , "valid" : true , "lastExtentSize" : 131072}
```

Helpful scripts

These one-line scripts will print the stats for each db/collection:

```
db._adminCommand("listDatabases").databases.forEach(function (d) {mdb = db.getSiblingDB(d.name);
printjson(mdb.stats())})

db._adminCommand("listDatabases").databases.forEach(function (d) {mdb = db.getSiblingDB(d.name);
mdb.getCollectionNames().forEach(function(c) {s = mdb[c].stats(); printjson(s)})})
```

The Linux Out of Memory OOM Killer

The Linux out of memory killer kills processes using too much memory. On a kill event you will see a line such as the following in the system log file:

```
Feb 13 04:33:23 hostm1 kernel: [279318.262555] mongod invoked oom-killer: gfp_mask=0x1201d2, order=0,
oomkilladj=0
```

On such an event, check in future runs that memory is not leaking. This can be checked by verifying that virtualbytes - mappedbytes for mongod is bounded. Check this in serverStatus and/or mongostat.

```
> db.serverStatus()
```

ulimit

`ulimit -m` with `mongod` may cause kills that are false positives. This setting is not recommended. MongoDB uses memory mapped files. The entire data is mapped. Over time, if there is no memory pressure, the `mongod` resident bytes may approach total memory, as the resident bytes includes file system cache bytes for the file pages open and touched by `mongod`.

swap

Please see the production notes about swap

Too Many Open Files

If you receive the error "too many open files" or "too many open connections" in the mongod log, there are a couple of possible reasons for this.

First, to check what file descriptors are in use, run lsof (some variations shown below):

```
lsof | grep mongod
lsof | grep mongod | grep TCP
lsof | grep mongod | grep data | wc
```

If most lines include "TCP", there are many open connections from client sockets. If most lines include the name of your data directory, the open files are mostly datafiles.

ulimit

If the numbers from lsof look reasonable, check your ulimit settings. The default for file handles (often 1024) might be too low for production usage. Run ulimit -a (or limit -a depending on shell) to check.

Use ulimit -n X to change the max number of file handles to X. If your OS is configured to not allow modifications to that setting you might need to reconfigure first. On ubuntu you'll need to edit /etc/security/limits.conf and add a line something like the following (where user is the username and X is the desired limit):

```
user hard nofile X
```

Upstart uses a different mechanism for setting file descriptor limits - add something like this to your job file:

```
limit nofile X
```

High TCP Connection Count

If lsof shows a large number of open TCP sockets, it could be that one or more clients is opening too many connections to the database. Check that your client apps are using connection pooling.

Mongod (hard) connection limit

Currently, there is a limit of 20,000 connections per process; this will be changed when better connection management code (maybe async io) will be implemented.

Data files count with very large databases

If your database is many terabytes, there will be a fairly large number of datafiles – be sure your ulimit setting is appropriate. You will need approximately 500 file handles per terabyte. (Also note if you run a repairdatabase operation, you will need double as two copies will be open at the same time.)

Estimating ulimit setting

You can always set the ulimit to some very large number like 64,000 or you can estimate the number. In general you will need to take your database sizes in GB and divide by 2GB plus the number of database, the number of possible connections and some overhead to get the number of files needed.

```
//TotalSize = `du -gs <db-path> | cut  -f 1`
# = (TotalSizeGB/2) + (NumberOfDBs) + (max(maxConns, 20,000)) + 5%-overhead
```

Production Deployments

If you're using MongoDB in production, we'd love to list you here! Please complete this web form and we will add you.

Archiving

Company	Use Case
craigslist	Craigslist uses MongoDB to archive billions of records. • Lessons Learned from Migrating 2+ Billion Documents at Craigslist - MongoSF (May 2011) • MongoDB Live at Craigslist - MongoDB blog (May 2011) • MongoDB and Craigslist's Data Storage Evolution - MongoSV (December 2010)

Content Management

Company	Use Case

	SAP uses MongoDB a core component of SAP's platform-as-a-servic (PaaS) offering. MongoDB was selec for the enterprise co management (ECM) section of the platfor as its flexibility and scalability will enable SAP to scale its con management service its PaaS offering to r customers' requirem while managing data from different applications. More information available the 10gen blog. • Introduction SAP's Java Platform as Service - Di Guendisch's Presentation MongoUK (September 2011)
	MongoDB is the repository that powe MTV Networks' next-generation CMS which will eventually used to manage and distribute content for of MTV Networks' m websites. Current deployments include SpikeTV.com and Comedy Central Indecision, and MTV Networks will be rolli out MongoDB on ma other major sites wit the next year, most l including gametrailers.com, thedailyshow.com, comedycentral.com, nick.com, and nume international properti Read more on the MongoDB blog. • How MTV Leverages MongDB for CMS - Jeff Yemin's Presentation MongoBosto 2011 (Octob 2011)
	MongoDB is used fo back-end storage on SourceForge front pages, project pages and download pages all projects.

- Realtime Analytics us MongoDB, Python, Gev and ZeroMQ Rick Copela Presentation MongoSV (December 2011)
- Achieving 2 Years of Mo Stability and Performanc SF.net - Ma Ramm's Presentation MongoDalla (November 2011)
- Rapid and Scalable Developmen with Mongo PyMongo an Ming - Mark Ramm's Presentation MongoDalla (November 2011)
- Allura - An Open-Sourc MongoDB-b Document Oriented SourceForg Rick Copela Presentation MongoSF (M 2011)
- How SourceForg Uses Mongo - Rick Copeland's Presentation MongoAtlan (February 2
- Scaling SourceForg with Mongo OSCON Presentation (July 2010)
- MongoDB a SourceForg QCon Lond Presentation (March 201
- How Python TurboGears and Mongo are Transformin SourceForg - PyCon (February 2
- SourceForg releases Mi SourceForg blog (Decem 2009)

- TurboGears
Sourceforge
Compound
Thinking (Ju
2009)

Wordnik stores its er
text corpus in Mongo
- 3.5T of data in 20 b
records. The speed
query the corpus wa
to 1/4 the time it took
prior to migrating to
MongoDB. More abo
MongoDB at Wordni

	- From the Cl and Back - Presentation MongoSV (December 2011) - What drove Wordnik non-relation (August 201 - Building a Directed Gra with Mongo Tony Tam's Presentation MongoSF (M 2011) - Managing a MongoDB Deployment Presentation Large Scale Production Engineering Meetup (February 2 - Wordnik: 10 million API requests a d on MongoD High Scalab Blog (Febru 2011) - Keeping the Lights on wi MongoDB - Tony Tam's Presentation MongoSV 2 (December 2010) - 12 Months w MongoDB - Wordnik Blo (October 20 - B is for Billio Wordnik Blo (July 2010) - MongoDB: Migration fro Mysql at Wordnik - Scalable We Architecture (May 2010) - Tony Tam's Presentation MongoSF (A 2010) - What has technology d for words la - Wordnik bl (February 2

	Harmony is a power web-based platform creating and managi websites. It helps developers with cont editors work togethe with unprecedented flexibility and simplic From stylesheets, images and template pages, blogs, and comments, every pie of Harmony data is stored in MongoDB. Switching to MongoI from MySQL drastica simplified Harmony's data model and increased the speed which we can delive features. • Real World Modeling wi MongoDB a Harmony - S Smith's Presentation MongoBosto (September 2010) • Steve Smith presentation about Harm at MongoSF (April 2010)
	Boundless Learning MongoDB as a rapid-prototyping technology to develo robust content management system

	eHow, a web proper Demand Media, rece 100MM+ unique use on a large content database. MongoDB perfectly suited for eHow's content management, as the document model exa matches the delivery model of documents pages. The Demand Media team rewrote legacy inherited .NET/SQL system to PHP/Python/MongoI a cachless, tierless design, all on one machine with replica
	MeteKamil uses MongoDB for browsi hundreds of thousan of videos on the websites: YouUnbox.com YouGamePlay.com YouOverClock.com DroidFather.com YouTouchDown.com

Ecommerce

Company	Use Case

At CustomInk, we be
custom t-shirts have
power to turn your g
into a team, elevate
gathering to an even
make your special m
more... well, momen
That's why we make
to create awesome s
CustomInk uses Mo
for a number of diffe
applications includin

- Supply Cha
Managemen
MongoDB s
information
t-shirt suppl
local screen
printers.
- Content
Managemen
MongoDB is
to store the
catalog, Cus
sells hundre
products, no
t-shrits.
- Logging - M
is used for l
interactions
website and
central store
application l
Learn more
how Custom
using Mong
from their
presentation
at MongoSV

	OpenSky is a free or platform that helps p discover amazing pr share them with thei friends, family and fo and earn money. Op uses MongoDB, Syn Doctrine 2, PHP 5.3, PHPUnit 3.5, jQuery node.js, Git (with gitf and a touch of Java Python. OpenSky us MongoDB for just ab everything (not just analytics). Along the they've developed MongoODM (PHP) a MongoDB drivers for and CAS. • Blending Mo with RDBMS e-commerce Steve Franc presentation MongoNYC 2011) • Augmenting RDBMS with NoSQL for e-commerce Steve Franc presentation East (March • MongoDB & Ecommerce Perfect Combination Video from S Francia's presentation New York MongoDB U Group (Octo 2010)
	Gilt Groupe is an inv only luxury shopping Gilt uses MongoDB time ecommerce ana • Gilt CTO Mi Bryzek's presentation MongoSF in 2010. • Hummingbir real-time we visualization developed b and powere MongoDB

	edelight is a social s portal for product recommendations. • MongoDB a edelight: A Pain and Su in 10 Slides Fabian Schl Presentatio MongoBerli (October 20 • MongoDB: Edelight sta MySQL auf MongoDB s Exciting Ecommerce (September
	Totsy offers moms on-the-go and moms access to brand-spe sales, up to 70% off Totsy was re-built up and MongoDB to co performance and sca limitations incurred v the prior relational-d platform. The transiti MongoDB has resolv of Totsy's performan scaling issues. • MongoDB Ecommerce Study: Totsy Pirtle's pres at Mongo B (September
	PlumWillow is a Soc Shopping Network w girls who like fashion build outfits by drag-and-drop, selec from thousands of to items. PlumWillow w by a "dream team" o core-developers/con to PHP, jQuery and MongoDB who utiliz Agile efficiency of M and the Vork Entepr Framework to bring PlumWillow from concept-to-launch in few months.

	Shopwiki uses Mong data store for its sho search engine, wher commit all the data generated, such as c analytics. Mongo's performance is such ShopWiki uses it in c where MySQL would not be practical. Sho also using it as a sto engine for all R&D a data-mining efforts v MongoDB's docume oriented architecture maximum flexibility. • Avery's Talk MongoNYC ShopWiki D (June 2010)
	Yabblr uses MongoD everything, including inventory manageme orders. Yabblr is an E-Commerce platfor flips ‘deals’ on its he Instead of pushing ra offers at you, Yabblr empowers you to gro together with friends deals on products yo select. Yabblr uses MongoDB exclusivel data store for the en platform, including: - data - Real time met are used for analyzin behavior and system - Inventory manager Customer orders - E queues and messag between different sy

Finance

Company	Use Case
	SecondMarket is the online marketplace for alternative assets such as private company stock, structured products & bankruptcy claims. SecondMarket uses mongoDB for storing its diverse asset classes in a single collection utilizing the power of the schema free nature of mongoDB. We also store organization news and activity in mongoDB. • Mongo & Mongeez on the SecondMarket Engineering Blog - Feb 2012 • Mongeez the open source, change management tool developed for MongoDB.] • Second Market, MongoDB and Mongeez for Change Management - Michael Lysaght's Presentation at MongoNYC (June 2011) • Mongeez at SecondMarket -Michael Lysaght's presentation at the MongoDB User Group (January 2012)
	Athena Capital Research is a quantitative investment manager, specializing in automated trading strategies. • How a Hedge Fund Uses MongoDB - Roman Shtylman's Presentation at MongoNYC (June 2011) • Low Latency Event Logging with BSON - Andrew Morrow's Presentation at MongoSV (December 2010)

	Equilar uncovers the direct pathways to the most influential high net worth individuals, and delivers immediate and actionable insight into the equity wealth events that drive business development opportunities. Equilar uses Mongo DB to map and analyze the connections between over 300,000 executives and board members in companies worldwide.
	Auriga USA is a financial services company that operates in the residential mortgage space. Moving to MongoDB solved a host of problems, and assisted Auriga USA in upgrading the functionality of their loan inventory management system to handle many new features and different types of assets, including student loans, credit cards, and asset-back securities.

Gaming

Company	Use Case
	Disney built a common set of tools and APIs for all games within the Interactive Media Group, using MongoDB as a common object repository to persist state information. • A Year with MongoDB: Running Operations to Keep the Game Magic Alive - Curt Steven's Presentation at MongoSV (December 2011) • Disney Central Services Storage: Leveraging Knowledge and skillsets - MongoSF (May 2011)
	IGN Entertainment, a unit of News Corporation, is a leading Internet media and services provider focused on the videogame and entertainment enthusiast markets. IGN's properties reached more than 37.3 million unique users worldwide February 2010, according to Internet audience measurement firm comScore Media Metrix. MongoDB powers IGN's real-time traffic analytics and RESTful Content APIs. • Using MongoDB for IGN's Social Platform - Presentation to San Francisco MongoDB User Group (February 2011) • Confessions of a recovering relational addict - Presentation by Chandra Patni at MongoSV (December 2010)
	WordSquared (formerly Scrabbly) is a massively multiplayer online ~~scrabble~~ crossword. Uses MongoDB geospatial indexing. • Mapping Flatland: Using MongoDB for an MMO Crossword Game - Presentation at Mongo Seattle 2011 • Building a Scrabble MMO in 48 Hours - Startup Monkeys Blog (September 2010)
	Shadelight is a unique fantasy roleplaying game where you play one of the legendary Guardians of Elumir. Set out on magical quests, battle mysterious creatures and explore a truly unique fantasy world.

METAMOKI	Metamoki builds social games like Mob Wars, Cityzen, and several others on top of Facebook. From Cityzen onwards, MongoDB is used for all games in production for pretty much everything.
adyaX	Freerice is developped by World Food Porgramme, the largest UN humanitary organisation. Freerice.com is a social quizz-like game where each good answer to a question make you win 10 rice grains. Since the begining more than 85 billions of rice grains were won trough freerice. Fully connected with twitter, facebook and groups, freerice tracks down each gain of each person. This generates almost 1 MongoDB row for each right answer, wich mean 8 billions of rows currently stored in Freerice MongoDB. Numerous totals are provided to the end user, which are all stored in Mongo too.

Government

Company	Use Case
the national archives	The National Archives (UK) made details of 11m records available through an application interface it published today as part of an ongoing programme to get more official records online. is consolidating numerous existing electronic archives, either by porting them directly or putting them in a service wrapper that can communicate with its unified system. The unified system uses MongoDB and is built on a Microsoft software stack. • National Archives releases public application programming interface for 11m records - ComputerWorld (Sept 2011) • From SQL Server to MongoDB - Presentation at MongoDB UK (Sept 2011)
HM Government	The British Government launched a beta of its GOV.UK platform, testing a single domain for that could be used throughout government. According to James Stewart, the Tech Lead on the beta of GOV.UK, "We started out building everything using MySQL but moved to MongoDB as we realised how much of our content fitted its document-centric approach," said Stewart. "Over time we've been more and more impressed with it and expect to increase our usage of it in the future." • Colophon for the GOV.UK beta - Feb 2012 • With GOV.UK, British government redefines the online government platform - O'Reilly Radar (Feb 2012)
Sunlight LABS	Sunlight Labs is a community of open source developers and designers dedicated to opening up our government to make it more transparent, accountable and responsible. MongoDB powers the National Data Catalog, and the Drumbone API, which is an aggregator of data about members of Congress. • MongoDB at Sunlight - Luis Martinez's Presentation Slides • Civic Hacking - Video from Luigi Montanez's presentation at MongoNYC (May 2010) • How We Use MongoDB at Sunlight blog post (May 2010)

Metadata Storage

Company	Use Case

shutterfly	Shutterfly is an Internet-based social expression and personal publishing service. MongoDB is used for various persistent data storage requirements within Shutterfly. MongoDB helps Shutterfly build an unrivaled service that enables deeper, more personal relationships between customers and those who matter most in their lives. • Performance Tuning and Scalability - Kenny Gorman's Presentation at MongoSV (December 2011) • The Shutterfly Data Layer and Schema Definitions - Luciano Resende's Presentation at MongoSV (December 2011) • MongoDB Profiling and Tuning - MongoSF (May 2011) • Q & A with Shutterfly Data Architect Kenny Gorman (Jan 2011) • Sharing Life's Joy using MongoDB: A Shutterfly Case Study - Kenny Gorman's presentation at MongoSV (December 2010) • *Implementing MongoDB at Shutterfly* from MongoSF (April 2010): Slides and Video
Lulu	Lulu's open publishing platform empowers more creators to sell more content to more readers more profitably than ever before. Lulu uses MongoDB to store the bibliographic, product, and file metadata for its creators' works. MongoDB's document-based data model makes it easy to manage the differences between print books, eBooks, photo books, and calendars and to keep up with the rapidly evolving world of self publishing. • Why we decided NoSQL was right for us, How we came to choose MongoDB - Sept 2011 Presentation

News & Media

Company	Use Case
theguardian	guardian.co.uk is a leading UK-based news website. We've spent ten years fighting relational database representations of our domain model, until the implementation of our API made us realise that if only we could store documents everything got simpler. MongoDB is a key part of the Guardian's infrastructure. • Evolving from relational to document store - MongoSF (May 2011) • Why I Chose MongoDB for guardian.co.uk - QCon London (August 2011) • MongoDB at the Guardian - MongoDB UK (Sept 2011)
examiner.com the insider source for local	Examiner.com is the fastest-growing local content network in the U.S., powered by the largest pool of knowledgeable and passionate contributors in the world. Launched in April 2008 with 60 cities, Examiner.com now serves hundreds of markets across the U.S. and Canada. Examiner.com migrated their site from Cold Fusion and SQL Server to Drupal 7 and MongoDB. Details of the deployment are outlined in an Acquia case study
BUSINESS INSIDER	Business Insider has been using MongoDB since the beginning of 2008. All of the site's data, including posts, comments, and even the images, are stored on MongoDB. For more information: • An Inside Look At The Guts Of Our Tech Infrastructure (June 2011 Article) • How Business Insider Uses MongoDB (May 2010 Presentation) • How This Web Site Uses MongoDB (November 2009 Article)
Forbes	Forbes has been around for nearly 100 years and on the web for more than a decade; recently the media space has been a changing landscape. Once of the changes that Forbes has undertaken starting in 2010 was the “opening up” our digital and print platforms to a global collection of content creators, marketers and audience members. This necessitated the changing of how our site worked, allowing content to flow easily through the site. We began to evaluate how we store and serve our content and decided to use MongoDB. We are currently using MongoDB for storing articles and companies data and are working to move more of our core assets onto it. • Supporting Distributed Global Workforce of Contributors with MongoDB David Dunlop's Presentation at MongoNYC (June 2011)

	The New York Times is using MongoDB in a form-building application for photo submissions. Mongo's dynamic schema gives producers the ability to define any combination of custom form fields. For more information: • The New York Times R&D Lab and MongoDB • Building a Better Submission Form - NYTimes Open Blog (May 25, 2010) • A Behind the Scenes Look at the New York Times Moment in Time Project - Hacks/Hackers Blog (July 28, 2010)
	The Chicago Tribune uses MongoDB in its Illinois School Report Cards application, which is generated from a nearly 9,000 column denormalized database dump produced annually by the State Board of Education. The application allows readers to search by school name, city, county, or district and to view demographic, economic, and performance data for both schools and districts.

Online Advertising

Company	Use Case
	MongoDB will be the central data store in our system, providing low-latency, high-volume reads in a distributed cluster behind our pixel servers, and near-real-time aggregate counts to our clients for forecasting. We're excited by the speed, replication features, flexible data model, and map-reduce style query access, which will let us dramatically simplify our system relative to the current architecture using a key-value store. • Tracking & Analytics with MongoDB at Signpost; MongoDB for Online Advertising - Matt Insler's Presentation for NYC Meetup Group • MongoDB Analytics for Online Advertising at Magnetic - Webinar hosted by Mark Weiss
	MediaMath is the leader in the new and rapidly growing world of digital media trading.
	Konverta.ru is the first Real Time Bidding (RTB) ad exchange on the Russian and CIS online advertising market. MongoDB is used to store all ads, ad impressions, clicks, and other data, as well as for real-time reporting and optimization.
	G5 is the largest and fastest growing provider of vertical-specific Local Marketing Solutions that help mid-market companies get found online, generate more qualified leads, convert more leads into new customers, track marketing performance - including offline, and optimize to the marketing sources with the best return on investment. G5 migrated our analytics platform from MySQL to MongoDB due to the heavy demands of storing & processing analytical data. MongoDB has proven to be fast, scalable, flexible, & maintainable. Best of all, MongoDB is supported by a fantastic community!
	Yodle uses MongoDB to persist queues of items to be synchronized with several partner APIs. Mongo is ideally suited for this update-heavy performance sensitive workload.
	The Localstars platform makes it easy for local advertisers to build and manage locally targeted advertising campaigns. The new Localstars advert server cluster uses sharded MongoDB to provide super high performance real time ad serving decision making and campaign statistics.

everyscreen media	EveryScreen is built on a scalable, high-performance, real-time mobile advertising bidding platform in Amazon EC2.
Localytics	Localytics uses MongoDB to process over 100M datapoints every day for their mobile analytics service.

Online Collaboration

Company	Use Case
Trello	Trello is a collaboration tool that organizes your projects into boards. In one glance, Trello tells you what's being worked on, who's working on what, and where something is in a process.Trello stores all historical and non-ephemeral data in MongoDB. The Trello team was attracted to MongoDB for its speed. MongoDB offers very fast writes, faster reads, and better denormalization support — it allows them to store the data of each individual card on the Trello board in a single document in the database and still have the ability to query into (and index) subfields of the document. They were also attracted to MongoDB's reliability: it is really easy to replicate, back up, and restore. Other benefits for the Trello team: using a loose document store makes it easy to run different versions of the Trello code against the same database without fooling around with DB schema migrations. This has a lot of benefits when they push a new version of Trello; there is seldom (if ever) a need to stop access to the app while we do a DB update or backfill. • What's in Trello Brett Keifer and the Trello team's presentation at the NYC MongoDB User Group, January 23, 2012. • The Trello Stack January 2012 from the Fog Creek Blog • Trello Architecture
Flowdock	Flowdock is a modern web-based team messenger, that helps your team to become more organized simply by chatting. Flowdock backend uses MongoDB to store all messages. • Why Flowdock migrated from Cassandra to MongoDB - Flowdock Blog (July 2010)
moxie Software™	Moxie Software™ is an innovative software company that provides integrated social enterprise software for employee and customer engagement through its Spaces™ by Moxie platform. Designed in the social media era and using MongoDB, Employee Spaces™ is an enterprise social software built for "The Way People Work" . It enables employees to collaborate on projects, innovate by sharing and co-creating knowledge and accelerate innovation.

	Sherl.tv is service for sharing plain text terminal screencasts from a unix terminal that uses mongodb as the main data storage for records.

Real-time stats/analytics

Company	Use Case
	Intuit is one of the world's largest providers of software and services for small businesses and individuals. Intuit uses MongoDB to track user engagement and activity in real-time across its network of websites for small businesses. • Deriving deep customer insights using MongoDB - Presentation at MongoSV (December 2010)
	The Buddy Media Platform gives brands the tools to manage their Facebook Pages and engage their Facebook fans. The second iteration of the Publisher tool on the Buddy Media Platform is powered by MongoDB. • Social Analytics on MongoDB (Video and Slides) - Presentation from February 2011 New York MongoDB User Group • The New Buddy Media Publisher: What You Need To Know - Buddy Media blog (November 2010)
	bit.ly allows users to shorten, share, and track links. bit.ly uses MongoDB to store user history. For more information: • bit.ly user history, auto-sharded presentation at MongoNYC (May 2010)
	10gen is the initiator, contributor and continual sponsor of the MongoDB project. 10gen built MongoDB Monitoring Service (MMS), a scalable software as a service monitoring tool built using MongoDB. MMS displays data in charts that track performance, resource utilization, availability, and response times. 10gen built MMS to provide operational insight into thousands of MongoDB deployments, and uses it to better diagnose and resolve customer issues.
	Chartbeat is a revolutionary real-time analytics service that enables people to understand emergent behaviour in real-time and exploit or mitigate it. Chartbeat stores all historical analytics data in MongoDB. • MongoDB & EC2: A love story? - NY MongoDB User Group (Oct 2011) • The Secret Weapons Behind Chartbeat - Kushal's coding blog (April 2010) • Kushal Dave's Presentation at MongoNYC (May 2010)

Boxed Ice	Server Density is a server monitoring tool from Boxed Ice. They have used MongoDB since June 2009 and are now processing billions of documents every month. Server Density also includes an addon for MongoDB monitoring and they have written extensively about MongoDB itself. Blog posts: • Why we migrated from mysql to mongodb • Automating partitioning, sharding and failover with MongoDB • Notes from a production MongoDB deployment • Many more Presentations: • MongoDB Monitoring and Queueing - David Mytton's Presentation at MongoSF (May 2011) • Monitoring MongoDB - David Mytton's Presentation at MongoSV (December 2010) • Humongous Data at Server Density - MongoUK Presentation (June 2010) • MongoDB in Production at Boxed Ice - Webinar (May 2010)
Zuberance The Leaders in Brand Advocacy	Zuberance started using MongoDB as a reporting engine in their enterprise product. In Q3 of 2011, Zuberance decided to build their Self serve product to scale it to thousands of brand marketers to have them identify, energize their advocates while giving them real-time analytics. • Why We Like MongoDB at Zuberance - Zuberance Blog (January 2012)
ShareThis	ShareThis makes it easy to share ideas and get to the good stuff online. ShareThis is the world's largest sharing network reaching over 400 million users across 150,000 sites and 785,000 domains across the web • MongoDB is Powering ShareThis Count System - Lenin Gali's Presentation at MongoSV (December 2010)
github SOCIAL CODING	GitHub, the social coding site, is using MongoDB for an internal reporting application.
Eventbrite Events Made Easy	Eventbrite gives you all the online tools you need to bring people together for an event and sell tickets. • Building a Social Graph with MongoDB at Eventbrite - Brian Zambrano's presentation at MongoSV (December 2010) • Tech Corner: Auto recovery with MongoDB replica sets - Eventbrite Blog (October 2010) • Why you should track page views with MongoDB - Eventbrite Blog (June 2010)
BLiNQ MEDIA Social Engagement Advertising SM	BLiNQ Media, an employee owned social media advertising and technology company, is one of 12 companies globally with access to the Facebook advertising API and the only company that is building its technology, BAM (BLiNQ Ad Manager) in Ruby on Rails. BLiNQ works with some of the world's leading media agencies and brands, helping them place and optimize advertising on Facebook through our proprietary technology. The technology team is headquartered in Midtown Atlanta at the Advanced Technology Development Center (ATDC), one of the world's top 10 incubators. The company's sales and client services headquarters is in TechSpace in New York City's Union Square area. BAM utilizes MongoDb as an operational data store to support millions of Facebook user and advertising campaign data. The data stored in MongoDB is then used for real-time reporting, analysis and optimization against key performance indicators. Since BAM went live in July 2010, it is storing over 7 million records and averaging in excess of 30,000 transactions a day. • MongoDB Delivers Results for Facebook Advertisers - Presentation at Mongo Atlanta (February 2011)
yottaa ALPHA	Yottaa offers Performance Analytics, a cloud service that monitors, ranks and analyzes the performance of millions of web sites, providing an open database to answer questions such as "why performance matters" and "how fast is my site". Yottaa is using Ruby on Rails and MongoDB to build their scalable analytics engine. • How Yottaa Uses MongoDB - Jared Rosoff's presentation at MongoBoston (September 2010) • Scalable Event Analytics with MongoDB and Ruby - Jared Rosoff's presentation at RubyConfChina (June 2010)

	BuzzFeed is a trends aggregator that uses a web crawler and human editors to find and link to popular stories around the web. BuzzFeed moved an analytics system tracking over 400 million monthly events from MySQL to MongoDB.
	uberVU is an intuitive social media management and analytics service used by companies of all sizes in 20+ countries. uberVU uses mongo as their primary data store because of its fast writes, its schemaless approach and reliability. • Intelligent Stream-Filtering using MongoDB - Mihnea Giurgea's Presentation at MongoUK (September 2011)
	CopperEgg RevealCloud – Awesome Cloud Monitoring – CopperEgg is able to achieve its super real-time updates and alerts of the RevealCloud product with the help of MongoDB. Mongo was the only DB that fit the requirements of extremely rapid storage and retrieval of customer monitoring data to provide on-screen updates every few seconds (NOT minutes). CopperEgg monitors thousands of end points world-wide in real-time. RevealCloud is available to application development organizations to give the instant visibility of system performance in any stage of the development lifecycle, from initial design through production, helping ensure SLAs…and giving your customers the response times they expect – right now. • Designing Algorithms that Scale Horizontally with MongoDB - Luke Ehresman's Presentation at MongoDallas (November 2011)
	Loggly uses MongoDB as a repository of statistics and summary information for time series log data. Loggly collects and stores metrics into MongoDB including the size and count of log events, and the results of saved searches run against our customized fulltext search engine based on SOLR. These metrics are used to drive graphs, reports, dashboards and exciting visualizations for users.
	Pattern Builders built a .NET based streaming analytics engine using MongoDB. Relational databases and MSMQ were integrated with MongoDB for high performance and a great user experience.

Social Networks

Company	Use Case

	Foursquare is a location based social network that incorporates gaming elements. foursquare uses MongoDB to store venues and user "check-ins" into venues, sharding the data over more than 25 machines on Amazon EC2. • Experiences Deploying MongoDB on AWS - Cooper Bethea's Presentation at MongoSV (December 2011) • Practical Data Storage: MongoDB at foursquare - MongoNYC (June 2011) • MongoDB @ foursquare - MongoSF (May 2011) • Scaling foursquare with MongoDB - New York MongoDB User Group (December 2010) • MongoDB Q&A - New York MongoDB User Group (December 2010) • MongoDB at foursquare presentation: Slides and Video (May 2010)
	musweet keeps track of what artists and bands publish on the social web. • Handling Humongous Data Sets from Social Net - Nader Cserny and Grischa Andreew's Presentation at MongoBerlin (October 2010)
	Guildwork is a guild host and social network for massively online multiplayer games such as World of Warcraft. Guildwork stores nearly all data in MongoDB with the exception of sessions and chat history.

	Silentale keeps track of your contacts and conversations from multiple platforms and allows you to search and access them from anywhere. Silentale is using MongoDB as the back-end for indexing and searching on millions of stored messages of different types. More details on Silentale can be found in this TechCrunch article. • One Year with MongoDB presentation from MongoUK (June 2010): Slides and Video
	Squarespace is an innovative web publishing platform that consists of a fully hosted and managed GUI environment for creating and maintaining websites. Squarespace's new social modules utilize Mongo to store large amounts of social data that is pulled in from around the Internet and displayed in native widgets that are fully integrated with the platform.
	Shelby.tv Shelby uses a few physically distinct MongoDB replica sets as the primary data stores in development and production. That data is accessed by web and iOS through our internal Ruby API.

Telecommunications

Company	Use Case
	Viber Media is using MongoDB as the core cloud infrastructure for its popular iPhone and Android applications that offer free phone calls and text messaging between users over 3G and wireless networks. Viber selected MongoDB for its ability to scale as more users join the Viber community, and to manage peak and unpredictable data loads from its 30 million plus registered mobile users. • MongoDB at Viber Media: The Platform Enabling Free Phone Calls and Text Messaging for Over 18 Million Active Users - MyNoSQL (Jan 2012) • Viber Media Selects MongoDB as Database in the Cloud for Popular Mobile Apps - 10gen blog (Nov 2011) • Where NoSQL, cloud computing and free texting converge - GigaOm (Nov 2011)

More MongoDB Users

Company	Use Case
LexisNexis	LexisNexis Risk Solı professionals and or collection agencies, MongoDB is used by and serves as the pe
CERN	CERN uses MongoD • Holy Large
turntable	The turntable.fm bac servers, and a dozer
Catch	catch.com is the eas capture & find your r • MongoDB w 2011) • MongoDB a
bump TECHNOLOGIES	Recently named #8 simply by bumping t manage and store d
CollegeHumor	CollegeHumor is a c exchange applicatio
evite	Evite uses MongoDE • Tracking an 2010)
DISQUS	Disqus is an innovat
mongoHQ fast & easy data management	MongoHQ provides service. Our hosting hosting options.
Justin.tv	Justin.tv is the easy, analytics tools for vi provide. Read more
Springer science+business media	Realtime.springer.cc book chapters in rea provide the scientific now". MongoDB is u Map reduce jobs ger book titles.
SUGARCRM. THE CLOUD IS OPEN	Sugar CRM uses Mc comments and whet Sugar.

where www.where.com	WHERE® is a local mobile coupons in th restaurant reviews, t from local merchants here, Inc. uses Mong a hyper-local ad netw
淘宝网 Taobao.com	Founded by Alibaba consumer-to-consum online retail stores th Sellers are able to p price or by auction. sold at a fixed price; the monitoring data.
PhoneTag	PhoneTag is a servi and SMS. PhoneTag
PiCloud CLOUD COMPUTING SIMPLIFIED	PiCloud enables scie power in public and computing applicatio sets in a highly distri
HASHROCKET EXPERTLY CRAFTED WEB	Hashrocket is an exp Medication Manager resolving drug-relate
mozilla labs ubiquity	The Mozilla open-so available on bitbucke
Preferred Infrastructure.	Sedue is the enterpr currently uses Mong • MongoDB a
Codaset TM	Codaset is an open out what your friends • The awesor 2010) • Handling Da
mypunchbowl	Punchbowl.com is a datamining. • Introducing presentation • *Ryan Angilly* • MongoDB fo Presentation
photostre.am	photostre.am stream • MongoDB in

fotopedia.com The first collaborative photo encyclopedia	Fotopedia uses Mon timelines, a feature t tiny html fragment in • MongoDB: (
Grooveshark	Grooveshark current
EA	MongoDB is being u gets served to ea.co
struq	Struq develops techi time.
Pitchfork	Pitchfork is using Mc
floxee	Floxee, a web toolki award-winning Twee
Sailthru You Communicate, We Deliver	Sailthru is an innova quantity. Moving to M Passing in arbitrary . And we've launched realtime and summa schema-free data st (tag-based interests, assembled mass em onsite recommendat capabilities to rapidly • MongoDB ir
teach street LEARN SOMETHING NEW	TeachStreet helps p manage their teachir teachers with insight • Slides from
Visibiz KNOW YOUR NETWORK. SELL BETTER.	Visibiz is a socially a productivity of busin and organized profile information needed t activities through sin Visibiz uses MongoD • How Mongo
Defensio outsmarting evil spam	Defensio is a comme
TweetSaver	TweetSaver is a wel MongoDB for back-e
BLOOM	Bloom Digital's AdGe storage for AdGear.

klatu networks	KLATU Networks de manage risk, reduce status, condition, an MongoDB to store te KLATU chose Mong
this or that	This or That! is a use on comparisons. Wh user-generated com drive its leaderboard
sk songkick	songkick lets you tra • Speeding u
crowdtap	Crowdtap uses Mon engine. The targetin is core to our service
Detexify²	Detexify is a cool ap out the blog post for
Sluggy Freelance	http://sluggy.com/ is • From MySQ 2010)
EDITD	EDITD is using Mon their sentiment analy
	@trackmeet helps y
eflyover	eFlyoverleverages th over two thousand g
Shapado	Shapado is a multi-t Rails and uses Mon
Sifino	Sifino enables stude summaries, and old
GameChanger BETA	GameChanger provi distribute real-time g • Tornado, M 2010) • GameChan (September
soliMAP	soliMAP is a map-ba
MyBankTracker™	MyBankTracker iPh
bill monitor™	BillMonitor uses Mo used by the live site
Tubricator beta	Tubricator allows yo Django.

	Mu.ly uses MongoDI service. MongoDB is
	Avinu is a Content M MongoDB.
	Topsy is a search er
	Cold Hard Code mal • Codepeek is • Jarvis uses
	Similaria.pl is an onli • One Year w
	ToTuTam uses Mon information about us • One Year w
	themoviedb.org is a
	OCW Search is a se uses Sphinx to index • Full Text Se
	Mixero is the new ge noise. Mixero uses M
	Biggo is an advance collection.
	Kabisa is a web dev for many of its client
	DokDok makes it ea document - right fror Morency's presentat

enbil.nu	Enbil is a swedish w querying data about
markitfor.me	markitfor.me is a boo don't have to remem text of all of your boo pages.
backpage pics	Backpage Pics is a w MongoDB is used to
Joomla Ads	Joomla Ads uses Mo
esm Eiwa System Management	Eiwa System Manag projects since Janua
morango web people	Morango is an intern several client project • Building a C 2010)
peerpong	PeerPong discovers We index users acro available informatior
ibibo i build · i bond	ibibo ("I build, I bond represented as a sin million of these docu
ZOOFS	Zoofs is a new way t Twitter searching for
oodle	Oodle is an online cl company behind the our millions of users
FunAdvice	Funadvice relaunche MongoDB User Foru
YaSabe hablemos	Ya Sabe is using Mo search engine built f more than 14 million and in English.
LOTERIA FÚTBOL	LoteriaFutbol.com is 2010. Mongo has be uses the PHP driver

	Kehalim switched ov contextual affiliate n MongoDB. MongoDI hadoop-like alternati
	Givemebeats.net is produced by some o profile, beats informa
	Cheméo, a search e explanation of the to chemeo.com/doc/te
	Planetaki is place we MySQL for the stora • Planetaki P
	[ChinaVisual.com] is ChinaVisual.com mo major production an
	RowFeeder is an ea posts in a spreadshe across multiple socia • MongoDB f (July 2010)
	Open Dining Networ web and mobile orde
	URLi.st is a small we (using the pylons fra
	LearnBoost is a free needs. LearnBoost i extensible and simpl • Mongoose -
	Kidiso is a safe onlir setup, we are using performance (ie sea
	Carbon Calculated p everything in the wo of this platform, Carl intuitive.

Vowch	Vowch is a simple pl you. It is a platform f account. • View a vowc
HolaDoctor℠	HolaDoctor.com is th online Hispanic com to store article image save handler.
РосГос ЗАТРАТЫ Мониторинг государственных контрактов	Ros Spending is the 1,400,000 federal go more than 260,000 s information, stats an Development and la
BlueSpark	BlueSpark designs a development, we ha
Ahgora	[Aghora] is a time at governmental requir information.
MAN OF THE HOUSE	Man of the House is and at home, as a fa
PeerIndex	PeerIndex is an algo firehose of social me
sahibinden.com The address for classifieds and online shopping	sahibinden.com is an over 1.5 billion page caching.
ylastic	Ylastic is using Mong capability.
BRAINREPUBLIC	BRAINREPUBLIC is like-minded people f
	Friendmaps is a tool
jounce™	The affiliate marketir search data. As of A

	Virb Looking for a pl with Virb. You provic
	Deal Machine is a st storage. It has helpe
	arrivalguides.com is launched a new site using the NoRM Driv
	The Hype Machine k and retrieval of user performance in our v
	ChatPast synchroniz computers. Search t about. Business use (Highrise, BaseCam
	Stockopedia initially 20000+ stocks, secto for building real time and investors condu
	TravelPost is a comi reviews, photos and
	SoulGoal stores or c
	Top Twitter Trends i: technologies such a:
	bongi.mobi is a plac include: fast docume on handset capabilit tracking, click-2-call,
	CoStore is an online importing, transformi such as charts, grap wherever you need i which are MapRedu

	Vuzz answers quest charts. At Vuzz, peo that share the same wants to be your tim current. Vuzz has be applicatin database.
	Bakodo is a barcode while they are shopp information about th importantly, what the of products.
	noclouds.org is a on and share informatic all systems.
	CafeClimb.com is a which lets people sh traveling information from the user.
	Keekme is free mon Keekme you will eas MongoDB as a prim
	Vitals.com consolida to help users make i of insurance compar datasource for our F MongoDB collection our dataservers perr functionality to Mong
	P2P Financial is Car DB for storing busine
	Qwerly is people sea links to social netwo
	phpMyEngine is a fr is MongoDB.
	vsChart allows you t
	yap.TV is the ultimat guide fused with a tu fans while watching MongoDB for analyti

	BusyConf makes gre letting them collect a multiple web platforr all the details preloa complementary to it' code is much simple • BusyConf p
	Sentimnt is a person networks and allows not related to you. S instances serve arou from MS SQL to Mo
	Workbreeze is fast a global project storag
	Kompasiana is the b biggest site In Indon
	Milaap works with es community developr
	Agent Storm is a cor and manage your Re listings on your web eFlyers all at the clic performs the search then looked up in My the results are serve updated either via th property is stored in changes. All this me • Now with 50
	Mashape is a frustra distribute an API of a • How We Sw Presentation
	The UK Jobsite is ar We use MongoDB fo searches to user info of the reasons of wh http://www.theukjobs
	Tastebuds music da connect with their la immediately shown s popular events servi MongoDB has dram allows users to rapid

	Skimlinks enables p equivalent affiliate li products mentioned retailer programs in and ensures no mor impressions we get
	www.vanilladesk.co as the main databas MongoDB's docume what enables Vanilla
	Summify uses Mong (metadata and HTM only the most recent cache for URL redire
	dakwak is the easie visitors want. It is a away from getting y
	Kapost's online new produce content. We analytics for the new
	ContactMe is a "ligh many small busines ContactMe.com aim social media sites lik place, making for ea
	Moontoast Impulse i from a Facebook fa Their tools power co fan-building and rev Moontoast Impulse: and more. Moontoa
	Shopperhive is a so reviews, video revie data back-end stora
	tracknose develops reward you with disc chosen for excellent georeferenced statis
	TalkAboutHealth ma like you, learn from
	Wusoup is a free on handles all user and

	Fave is a local searc businesses, and to h releasing next-gener services. Infrastructu performance/respon call campaigns
	Doodle is the world's MongoDB as an add less data via MySQL • An introduct
	FindTheBest is an o your options and dec informed decisions b products and service traffic monitoring and
	Fuseware provides r and more. This analy makers. We use Mo that we store. We ha scalability and data a
	Wherevent is a sear other cities. The enti feature.
	Skyline Innovations We deploy custom m MongoDB for data w • Time Series 2011)
	Memrise combines t of our learning and w about schema ossifi
	Dather is a content p can invite friends, fa
	Fundastic.info is Cru investors and financi dedicated pages for understand the inves to store all its investo stored in MongoDB a

	Foofind Labs is a un and indexes every fi creation by Pablo Sc MongoDB is used to in order to allow text
	Beaconpush is a clo real-time data with a methods for older br data is used for pres low system impact a
	Easy Bill, India's first to the life of the com never-before conver biggest problems fac enterprises for bill pa grown exponentially, activity logging of ou
	DeskMetrics is a rea how their users are u application, where a happened on the use We use MongoDB to have chosen Mongo important for us sinc
	Interstate provides b tasks using awesom data, etc).
	Proxlet helps you fig a Twitter API proxy, way of browser exte per-user setting docu designed to scale ho • A Beautiful 2011)
	fiesta.cc makes crea tool that everybody a powering the websit • MongoDB a • Behind the
	Dayload provides a resource usage and mail. Storing AWS re MapReduce functior

Goo.by	Avivo is creating mo mobile, web and cus Through design and interaction. Ultimatel that build brand loya is leading us to dive best for our clients. \ primary database fo
ABUSIX The Global Reporting Project	Abusix is the networ spamfeeds(www.spa technology providers today, while handlin behavior in their net major need for a sol increase customers resources in abuse MongoDB in environ almost suits all need
idea2	Idea2 is a versatile, create and impleme projects, customer c browser-based, fully application and work transition, we also c
FamilyTies.com	FamilyTies is family families and friends children under 13, b more to come. Mon various cache objec relationship maps.
Pracanowo.pl	Pracanowo.pl is a jo formats and finds jo and in few month to
REPUBLIKA.co.id	Republika Online is Republika", a nation
Grepler	Grepler.com is a dis lists to enable single the user accounts w
thuisvergelijken	Thuisvergelijken.nl i wide range of produ all our data storage: MongoDB's flexibility
salsadb eta	salsadb.com uses M capabilities are used
CHECK24	CHECK24 is one of compare a wide ran travel and thus save link-out the custome contract. The compa Furthermore, custom

I'mOK	I'mOK is a mobile ap their location, taking exchanged for things MongoDB is used to
wheelhouse	Wheelhouse CMS is flexible to develop si is used for all data s
QWK.IO	Qwk.io is the quicke backend.
Ameba by CyberAgent	CyberAgent,Inc. has CyberAgent's major Investment Develop American office: Cyb Social Game with nc
HeiaHeia!	HeiaHeia.com provic friends, and cheer o can choose from ove HeiaHeia.com's tech provide engaging, cu
newscurve	NewsCurve is an an MongoDB as primar
A-SaaS	Accounting SaaS Ja accountants and sm services include, but necessary to meet th that is difficult to acc
THRILLIST	Thrillist is using mon Right now it's emplo click-track data. We' our continuous integ
FStructures.com online fabric structure community	Fabric Structures Co in the overall constru customers about the field. MongoDB is us

eastghost.com higher paranormal	Brenden Digital uses migrated. We apprec repeatedly exhibits a comprehension.
attachments.me	Attachments.me is a surrounding attachm
thumbtack.com	Thumbtack is an onl analytics, due to its e details can be found
chirpat.me	ChirpAt.Me enables expertise with their f sessions to message
大众点评 dianping.com	DianPing.com is a le objective and rich lo weddings and variou on businesses, user
venmo	Venmo is a free and drinks, rent, grocerie activities published b
Intelie	Intelie is a software t and machine learnin centers in real time. documents per minu new rules, in additio
SIGNAL	First, we are using M users can create a S perform a lookup of user. Second, we ar statistics for our Ema queries in tables wit anywhere from 30 - queries. This use ca http://johnpwood.net
directdialogs	Directdialogs.com is capability and a flexi powerful cache with purposes, OLAP rep MongoDB.
dc storm optimising digital marketing	DC Storm helps mar better. The intuitive t across all digital cha powers the Storm Pl redundancy.
Bouncely	Bouncely uses Mon everything and store information and run

PCLICKS	PClicks uses Mongo then adapt to meet t
MAGNETIC™	Magnetic is the lead their most relevant a month. Beyond perfo leverage the rich dat and unify a mix of hi Magnetic a competit
okezone.com Indonesian News & Entertainment Online!	Okezone.com is usir and commenting sys
OpenChime	OpenChime uses M Thousands of people instructors in Mexicc
MONOLOOP REALTIME BEHAVIOURAL TARGETING	Monoloop is a behav MongoDB is the core and deliver personal
Yeay.me	Yeay.me is a small s MongoDB is used fo
ACTIVESPHERE	Activesphere provide as well.
SOCIALCITYDEALS Tracking all local daily deals in your city	SocialCityDeals use set of fields and new architecture to store effort in half for the s
Thin PHP Framework Light weight. Lightning Fast. Easy.	Thin PHP Framewor fast, simple, scalable users scale their dat
	Nearley is a mobile uses MongoDB to st

	Newsman App is an DB for now to build for each subscriber t all our other databas
	Sizzix uses mongod
	Gazaro uses Mongo products across a ra
	Dropzone.by uses m introduced OAuth 2.
	Cairenhuiis a Financ and use Mongodb_1
	Ez Texting uses Mo outgoing text messa
	iKeepm is a cloud-b uploads in GridFS.
	Maansu.com is an re sells.
	www.motores24h.pt our data storage nee that we can easily ex
	Techunits seeks to w technology, Informat TechMVC is the bes architecture. Techun Buzzers.co.za, Trivi
	Buzzers.co.zahttp:// changing environme millions of products using MongoLantern system more intellig
	Paraimpu is a social and socially share th used for all the persi sharding, heterogen

TRIVIAN	Trivian is a location l to store quiz questio
BUSINESS.COM Buy smarter	Business.com is the the products and ser taxonomy systems. \
GF	General Flows is bui into some pre-bakec build a booking syste plugging them into th structures, because code-generation tecl (EAV) but found the Google App Engine and running; we're ju
	Art.sy is a new way t collections around th
spoondate food is love. get plenty!	Spoondate is a socia experiences. The sit in a conversation, or data and member se rapidly and consister
D SHARP MOBILE DIABETES MANAGEMENT	D Sharp uses Mongo people with diabetes and JQuery Mobile, modern smartphone desktop browsers.
uQuery the appstore search engine	uQuery Inc. uses a s we receive from App software stack from
46elks	46elks uses MongoD make sure the data c
Chalkboard	Chalkboard is a daily socially over mobile of consumers across MongoDB to archive internally. With the g future.
tisProperty We search the internet so that you don't have to.	tisProperty is a Real realestates listed on to store our crawling
mobius	Digital Media Exchan more instead of our r storage 2. We run a schema-less design. are just some of the

Daily Gourmet	Daily Gourmet provi our extensive loggin wasn't possible with
{P} THE LOCKER PROJECT	Sponsored by Singly over their personal c as a personal data s
Loc-cit	Loc. cit. is a small ne can be explosive. Tr database for everyth Mongoengine for Dja
NAVER	Naver Japan utilizes recently released ph http://itunes.apple.cc with MongoDB (Japa http://www.slideshar
My eStore app	My eStore App uses content via Mongo.
FOTOSEARCH® photos and images	Fotosearch Stock Pl imagesare served w customer is looking f results.
cloudamp	CloudAmp is a data many data sources a across different priva data integration for a platform from private Twitter, where we or
Vigilant Medical	Vigilant Medical use medical images. Mo application, ImageSl
โรงพยาบาล ปิยะเวท PIYAVATE HOSPITAL Powered by SiteMongo CMS	Piyavate Internationa new website need so search with Solr. Mu WYSIWYG form cre these features are to way. That's why Mo

	Mindvalley is using M Mongoid. The ability speed, while its docu lead. Due to our suc developed CMS, as
	Cookfollower is a Tw meet new people. M
	WuuhuuOnline.com leverages SPHINX a
	Weconext develops secured web platforr allows simple collab MongoDB stores for
	Visual.ly is building t platform for Big Data
	Trendrr provides the for media and brand curation, charting an
	Wireclub is an online are exchanged by ou
	KANO/APPS creates worldwide players ac the primary data stor
	OnePageCRM uses been using MySQL s
	Blabbermouth Socia for sweepstakes and

Персик	Persik.me uses Mon
AwkwardTurtle®.com	Kungfuters LLC is us Users must sign in u submit stories of awk Each week, the story This site was former Stories, Users, Vote:
pt PunchTab	PunchTab store all l
	Nexon Corp. is a ma
	SPARC uses Mongo happy with how simp
heyZap	Heyzap uses mongo iOS gaming apps, ac tens of millions of ou
FISHIDY	Fishidy.com uses Mo
WOKEEY.com	Wokeey.com uses M hundreds records.

	ThinktankSocial use brands such as Pric and mass storage w
	Data Publica uses M
	Hipster uses Mongo and analytics.
	Noktacom Medya us analytics. We track e custom video scorin suggestion system. second. We will exp usage of caching sy
	Searce we used mo very large set of dat
	deegr is using Mong recommendation en architecture and sha
	AppHarbor is a .NE messages generate provision MongoDB
	Pictomist is a simple written in two basic style mongo collecti off the top elements second part is web l them in memory, an spin out the bulk of t layer without muckin
	Xperantum Xemanti structures, capture t
	4Apps uses MongoD memcached.
	GLO AB uses Mong (MES) software proc data gathered durin but expecting to gro within company net

	CMI Soft uses Mong & e-commerce store social networking ap use Rails with Mong
	Fyndlr implements M posts and users are functionality. Since c MongoDB instance.
	Exceptiontail uses M exception managem them see the entire s
	AdHui.com uses mo optimization.
	Travelmap uses Mor sources. Travelmap 5 times faster and m average our databas
	Global Advertisers u outdoor advertising a Airport. Global is one
	Storeden is a cloud ecommerc mongoDB as: - stora -real time statistics.
	Melt DSP is using M
	Startup Threads Mo

	Songza uses Mongc streaming service th The schema-free de and occasionally a c
	Ciris is a new applic: MongoDB to store th
	Socialbakers Inc. Fo
	Neon Grid uses mor
	Ex Machina develop involves tens or hun and as a result we h needed a storage sc cost-efficient regardl We never looked ba
	BellyBallot is a new : MongoDB for it's prir their associated mea Rails) is also on Mor
	AdMaster uses Mon of over 40,000,000 f
	GetVega is a second memory relevant data from th list on GetVega whe GetVega.com uses I MySQL. MongoDB a
	5Searches uses Mor

	barbull.co.uk uses M reliable and the abili calculations are a br
	Power-lan uses Mor applications are : - N
	Sidebuy is a product find products and de properties. With Mor across our servers h everything in BSON
	OviPets uses Mongo very happy with the scaling beautifully wi storage and image c
	Tip or Skip uses Mo and job queues.
	Nodegrid provides a service.
	Courseoff uses Mon

	Mini Medical Record especially useful for Whether paper or el and medicolegal pro addition, even simpl be inaccessible for p Medical Record alle
	ClaimAble operates use it exclusively to
	Ponder Design uses portal for a large stu implementing new fe wishes.
	Stamped uses Mong
	Hopper uses Mongo
	Recommendly INC u
	AlisverisRobotu.com including inventory m
	ITERNOVA SL prou store real-time data networks...), and veh

	tvb.com uses Mongc MongoDB is schema processing. We crea
	Ob1b.com is one of price and find produ products which they because of its simpli
	CNN Turk uses Mon new portal's showca MongoDB on 1st Jul
	Talis uses MongoDE http://community.tali achieved an order of to simplify our stack
	Digital Flow uses Mc ease of use and dep
	Transmachin uses a sentences with loads delivering our websi
	Aggregate Knowledg make media accoun manage and exploit time at hand, Aggreg combination.

about.me	About.me is currentl About.me turned to I addition to the shard required.
DEALER.COM™	Dealer.com deploye supports configuratic on-disk size of almos
Famigo	Famigo stores all va cloud architecture is
MATERIALS PROJECT	By scaling materials properties of over 80 Project is making the sophisticated web in schedule and track c and search the resul materials properties.
MapMyFITNESS	The MapMyFitness u traditional MySQL sc the candidate for exp application deployme for live tracking, to u

	MoPub uses Mongo
	SimpleReach power SimpleReach builds
	MongoDB is used fo Stripe's in-house fra received, or a cron j aggregate totals, fee
	Tart New Media is T experience and win
	WireLoad uses Mon narrowly focused so pleasant user experi
	Illicotravel compares and more. We love M
	MKN's (MKN Web S operations. The data

	Spideo using Mongc their own tastes.
	radi8.co.uk using Mc
	PRIJSZOEKEN.nl is products from 500 c data-feeds, our cust allows us to develop
	Firebase is a cloud s built with Firebase c are inherently scalat massively scaleable which offers the built unique URL stored i
	JustReco uses Mon

	Kano Games uses M gaming web app tha MongoDB is used to redundancy and flex beat out the competi
	Gimme Bar use Mor stored directly in S3) around 70g.
	Nodex use MongoDI data store for our CN and are expanding a fast and secure web outlining how we use
	FramtidsPost AB use services. The Future messages are big ar been able to use Mo
	uses MongoDB to s we use MongoDB as

See also

- MongoDB Apps
- Use Cases
- User Feedback!Shelby.tv.png|align=left!

Contributors

- JS Benchmarking Harness
- MongoDB kernel code development rules
- Project Ideas
- Roadmap
- UI
- Source Code
- Building
- Database Internals
- Contributing to the Documentation

- 10gen Contributor Agreement

JS Benchmarking Harness

CODE:

```
db.foo.drop();
db.foo.insert( { _id : 1 } )

ops = [
    { op : "findOne" , ns : "test.foo" , query : { _id : 1 } } ,
    { op : "update" , ns : "test.foo" , query : { _id : 1 } , update : { $inc : { x : 1 } } }

]

for ( x = 1; x<=128; x*=2){
    res = benchRun( { parallel : x ,
                      seconds : 5 ,
                      ops : ops
                    } )
    print( "threads: " + x + "\t queries/sec: " + res.query )
}
```

Dynamic values

```

res = benchRun( { ops : [ { ns : t.getFullName() ,
                            op : "update" ,
                            query : { _id : { "#RAND_INT" : [ 0 , 100 ] } } ,
                            update : { $inc : { x : 1 } } } ] ,
                  parallel : 2 ,
                  seconds : 1 ,
                  totals : true } )
```

- dynamic operatos
  - RAND_INT [min , max , <multiplier >
    - [0 , 10 , 4] would produce random numbers between 0 and 10 and then multiply by 4

More info:

http://github.com/mongodb/mongo/commit/3db3cb13dc1c522db8b59745d6c74b0967f1611c

MongoDB kernel code development rules

Coding conventions for the MongoDB C++ code...

- Kernel class rules
- Kernel code style
- Kernel concurrency rules
- Kernel exception architecture
- Kernel logging
- Kernel string manipulation
- Memory management

- Writing tests

- Git Committing and Pushing
- User Facing Conventions
  - Use camelCase for about everything
  - Include units in fields

For anything not mentioned here, default to google c++ style guide

Git Committing and Pushing

- commit messages should have the case in the message SERVER-XXX
- commit messages should be descriptive enough that a glance can tell the basics
- commits should only include 1 thought.
- do **NOT** push until running the test suite

User Facing Conventions

> ⚠ These are very important as we can't change them easily – Much more than code conventions!

Anything users see – command line options, command names, command output, we need to think hard and carefully about the name to be used, and the exact format and consistency of the items output. For example, serverStatus output is a bit of a mismash of lowercase and camelCase. Let's fix that over time, starting with new things.

Anything user facing must be ran by several team members first.

- **Do NOT add a new $operator without signoff by the entire team.**

- **Do NOT add a new command without signoff by the entire team.**

Use camelCase for about everything

- `--commandLineOptions`
- { commandNames : 1, commandOption : "abc" }
- Names of fields in objects we generate - such as command responses, profiler fields.

Include units in fields

In things like serverStatus, include the units in the stat name if there is any chance of ambiguity. For example:

- writtenMB
- timeMs

We should have standards for these – i.e. megabytes should always be "MB" and not "Mb" and "Megabytes" in different places. So the standards are:

- for bytes : use "MB" and show in megabytes unless you know it will be tiny. Note you can use a float so 0.1MB is fine to show.
- for time: use millis ("Ms") for time by default. you can also use Secs and a float for times you know will be very long.
- for microseconds, use "Micros" as the suffix, e.g., `timeMicros`.

Kernel class rules

Design guidelines

- ***Never use multiple inheritance***. If you need the service of several classes, use delegation. The only possible but highly unlikely exception to this is if your class inherits from other pure abstract classes.

- Have a comment before a class that explains its purpose. Perhaps the class name is so clear that this is obvious. Then some commentary on what you are up to.

- Only add ***members and methods to a class if they make sense*** w.r.t the bullet above. If you find yourself unsure to where to hook a piece of logic, rethink the class and surrounding classes purposes.

- Class ***names and methods names are to be descriptive*** of what they do. Avoid generic overloaded names (e.g., write, add, ...) to make grep easier (and maybe reading too).

- Don't put implementation details in the header unless the user of the class needs to know them. Sometimes single line inline implementations are good "documentation". If something needs to be inline for performance, put it at the bottom of the fine using the `inline` keyword instead of in the middle of the class definition (if the implementation is more than a line or two long).

- Assume all methods can throw a DBException. If a class should never throw (e.g can be called in a destructor), that should be clear.

- *Write a unit test for each class you create.* If you can't easily write a unit test for the class, that is a strong hint it has way too many external dependencies.

- ***Do not create early hierarchies***. An early hierarchy is a one where there is only one type of derived class. If you need to separate functionality, use delegation instead. In that case, make sure to test separately.

- Avoid `friend`.

- Default to making classes ***non-assignable and non-copyable***. (Use boost::noncopyable.)

Layout guidelines

- For classes where layout matters (anything with #pragma pack), put data members together at the top of the class. You must also have a BOOST_STATIC_ASSERT(sizeof(ClassName) == EXPECTED_SIZE) either directly under the class or in the associated .cpp file

Kernel code style

- basics
- case
- comments
- inlines
- strings
- brackets
- class members
- functions
- templates
- namespaces
- start of file
- assertions
- return early
- numeric constants
- explicit constructors

basics

- Use spaces, no literal tabs.
- 4 spaces per indentation.
- Limit lines to 100 columns.

case

Use camelCase for most `varNames`

See important notes on case on the parent page for user facing names!

comments

We follow http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Comments for placement of comments

As for style, we use javadoc's in classes and methods (public or private) and simple comments for variables and inside code

```
/**
 * My class has X as a goal in life
 * Note: my class is fully synchronized
 */
class DoesX {

...
  /**
   * This methods prints something and turns off the lights.
   * @param y the something to be printed
   */
   void printAndGo(const string& y) const;

...
 private:
    // a map from a namespace into the min key of a chunk
// one entry per chunk that lives in this server
map< string , BSONObj > _chunkMap;

    /**
     * Helper that finds the light switch
     */
    Pos _findSwitch() const;

    /** @return the light switch state. */
    State _getSwitchState() const;
};
```

```
void DoX( bool y) {
    // if y is false, we do not need to do a certain action and explining
// why that is takes multiple lines.
if (! y ) {
}
```

Don't forget – even if a class's purpose is obvious, you can put a comment on it as to why it exists!

inlines

- Put long inline functions in a `-inl.h` file. *
- If your inline function is a single line long, put it and its decl on the same line e.g.:

```
int length() const { return _length; }
```

- If a function is not performance sensitive, and it isn't one (or 2) lines long, put it in the cpp file. Keep code out of headers.

strings

See

- util/mongoutils/str.h
- bson/stringdata.h

Use `str::startsWith()`, `str::endsWith()`, not `strstr()`.

Use `<< 'c'` not `<< "c"`.

Use `str[0] == '\0'` not `strlen(str) == 0`.

See Kernel string manipulation.

brackets

```
if ( 0 ) {
}
else if ( 0 ) {
}
else {
}

do {
} while ( 0 );
```

class members

```
class Foo {
   int _bar;
};
```

functions

Declaration:

```
void foo( int v, MyType myItem );
```

Avoid declarations of extern functions in source files! Instead, #include a proper .h file. Be sure to match the header filename to the source filename where the function definition appears.

Definition:

```
void foo( int v, MyType myItem ) {
}
```

Invocation:

```
foo( 1, MyType() );
```

templates

```
set<int> s;
```

namespaces

```
namespace foo {
   int foo;
   namespace bar {
      int bar;
   }
}
```

start of file

```
// @file <filename>
license
```

assertions

See Kernel exception architecture.

return early

BAD

```
int foo(){
   if ( x ){
     ...
   }
}
```

GOOD

```
int foo(){
   if ( ! x )
     return;

   ...
}
```

Keeps indentation levels down and makes more readable.

numeric constants

Large, round numeric constants should be written in multiplied form so that you never need to count digits.

```
const int tenMillion = 10*1000*1000;
const int megabyte = 1024*1024;
```

explicit constructors

To avoid implicit type conversion, use the "explicit" keyword before constructors that take a single parameter.

Kernel concurrency rules

All concurrency classes must be placed under `utils/concurrency`. You will find several helper libraries there.

- Do not add mutexes without discussion with others. Concurrency and correctness is very hard in the large. Great care is required. For example the concurrency model in replica sets is hard to understand and error prone (at least it was initially and probably still is).

If you think there is a real need for an exception to the list below let's have the group weigh in and get a concensus on the exception:

- Do not use/add recursive locks.
- Do not use rwlocks.
- Always acquire locks in a consistent order. In fact, the MutexDebugger can assist with verification of this. MutexDebugger is on for _DEBUG builds and will alert if locks are taken in opposing orders during the run.

Kernel exception architecture

There are several different types of assertions used in the MongoDB code. In brief:

- `uassert` checks for per-operation user errors. Operation-fatal.
- `massert` checks per-operation invariants. Operation-fatal.
- `verify` is a synonym for massert, that doesn't require an error code.
- `fassert` checks fatal process invariants. Process-fatal.
- `wassert` warn (log) and continue.

- Calling `assert` is not allowed. Use one of the above instead.
- `dassert` just calls `verify` but only in debug mode. Do not use!

When per-operation invariant checks fail, the current operation fails, but the process and connection persist. This means that `massert`, `uassert` and `verify` only terminate the current operation, not the whole process. Be careful not to corrupt process state by mistakenly using these assertions midway through mutating process state. Examples of this include uassert and massert inside of constructors and destructors. `fassert` failures will terminate the entire process; this is used for low-level checks where continuing might lead to corrupt data or loss of data on disk.

Both `massert` and `uassert` take error codes, so that all errors have codes associated with them. These error codes are assigned incrementally;

the numbers have no meaning other than a way to associate a log message with a line of code. scons checks for duplicates, but if you want the next available code you can run:

```
python buildscripts/errorcodes.py
```

A failed operation-fatal assertion throws an `AssertionException` or a child of that. The inheritance hierarchy is something like:

- std::exception
  - mongo::DBException
    - mongo::AssertionException
      - mongo::UserException
      - mongo::MsgAssertionException

See `util/assert_util.h`.

Generally, code in the server should be prepared to catch a DBException. UserAssertionException's are particularly common as errors and should be expected. We use resource acquisition is initialization heavily.

Gotchas to watch out for:

- generally, don't throw a assertionexception directly. Functions like uasserted() do work beyond just that. In particular, it makes sure that the getLastError structures are set up properly.
- think about where your asserts are in constructors, as the destructor wouldn't be called. (But at a minimum, use wassert a lot therein, we want to know if something is wrong.)
- don't throw in destructors of course.

Kernel logging

- Basic Rules
  - cout/cerr should never be used

- Normal Logging
  - debugging with levels of verbosity. default=0 (use just log() for that). See -v command line option.

    ```
    LOG( int x ) << ...
    ```

  - informational

    ```
    log() << ...
    ```

  - rate limited

    ```
    LOGSOME() << ...
    ```

  - warnings
    - recoverable
    - e.g. replica set node down

    ```
    warning()
    ```

  - errors
    - unexpected system state (disk full)
    - internal code errors

    ```
    error()
    ```

- Debugging Helpers
  - PRINT(x) = prints expression text and value (can also do PRINT(x.method()))
  - PRINTFL = prints file and line (good for tracing execution)
  - printStackTrace() = shows a stack trace. Alternative to using a debugger.
  - GEODEBUG, etc... = used for incredibly verbose logging for a section of code that has to be turned on at compile time

Kernel string manipulation

For string manipulation, use the `util/mongoutils/str.h` library.

`str.h`

`util/mongoutils/str.h` provides string helper functions for each manipulation. Add new functions here rather than lines and lines of code to your app that are not generic.

Typically these functions return a string and take two as parameters : string f(string,string). Thus we wrap them all in a namespace called `str`.

str::stream() is quite useful:

```
uassert(12345, str::stream() << "bad ns:" << ns, isok);
```

`StringData`

```
    /** A StringData object wraps a 'const string&' or a 'const char*' without
     * copying its contents. The most common usage is as a function argument that
     * takes any of the two forms of strings above. Fundamentally, this class tries
     * go around the fact that string literals in C++ are char[N]'s.
     *
     * Note that the object StringData wraps around must be alive while the StringData
     * is.
    */
    class StringData {
```

See also `bson/stringdata.h`.

`mongoutils`

MongoUtils has its own namespace. Its code has these basic properties:

1. are not database specific, rather, true utilities
2. are cross platform
3. may require boost headers, but not libs (header-only works with mongoutils)
4. are clean and easy to use in any c++ project without pulling in lots of other stuff
5. apache license

Memory management

Overall guidelines

- avoid using bare pointers for dynamically allocated objects. Prefer scoped_ptr, shared_ptr, or another RAII class such as BSONObj.
- do not use auto_ptr's and refactor legacy ones out whenever possible. (Careful with c++ driver and backward compatibility though.)
- If you assign the output of new/malloc() directly to a bare pointer you should document where it gets deleted/freed, who owns it along the way, and how exception safety is ensured. If you cannot answer all three questions then you probably have a leak.

Writing tests

We have three general flavors of tests currently.

General guidelines

It is important that tests can be run in parallel and still succeed.
For example, make sure that:

- try to use a unique collection name, for example named after the test

  ```
  t = db.jstests_currentop
  ```

- if checking on current operations, make sure to add an NS filter

```
db.currentOp({ns: "test.mytest"})
```

- if possible, try to avoid things that are global to mongod or the database (oplog, profiling, fsync)

If the test cannot be run in parallel, add it to the blacklist in "skipTests" in shell/utils.js.

dbtests

Tests can be written in C++. These are in the dbtests/ subfolder. See the code therein. To run:

```
scons test
./test --help
```

jstests

Many tests are written as .js scripts that are executed via the `mongo` shell. See the Smoke Tests link at the bottom for how to run comprehensive sets of tests. To run a particular test:

```
# start mongod first then run a few simple tests:
mongo jstests/basic*.js
```

Note there are several subdirectories for different test suites. slowNightly is run by the buildbots only once a night; slowWeekly only once a week. Most other tests are ran every CI cycle (all the time).

Also note that that the js tests rely on functions defined in the "shell" directory (see servers.js and utils.js in particular).

Lightweight startup test.

You can inherit from class `mongo::UnitTest` and make a test that runs at program startup. These tests run EVERY TIME the program starts. Thus, they should be minimal: the test should ideally take 1ms or less to run. Why run the tests in the general program? This gives some validation at program run time that the build is reasonable. For example, we test that pcre supports UTF8 regex in one of these tests at startup. If someone had built the server with other settings, this would be flagged upon execution, even if the test suite has not been invoked. Most tests are not of this sort.

See Also

- Smoke Tests
- http://buildbot.mongodb.org/

Project Ideas

If you're interested in getting involved in the MongoDB community (or the open source community in general) a great way to do so is by starting or contributing to a MongoDB related project. Here we've listed some project ideas for you to get started on. For some of these ideas projects are already underway, and for others nothing (that we know of) has been started yet.

A GUI

One feature that is often requested for MongoDB is a GUI, much like CouchDB's futon or phpMyAdmin. There are a couple of projects working on this sort of thing that are worth checking out:

http://github.com/sbellity/futon4mongo
http://www.mongodb.org/display/DOCS/Http+Interface
http://www.mongohq.com

We've also started to spec out the features that a tool like this should provide.

Try Mongo!

It would be neat to have a web version of the MongoDB Shell that allowed users to interact with a real MongoDB instance (for doing the tutorial, etc). A project that does something similar (using a basic MongoDB emulator) is here:

http://github.com/banker/mongulator

Real-time Full Text Search Integration

It would be interesting to try to nicely integrate a search backend like Xapian, Lucene or Sphinx with MongoDB. One idea would be to use MongoDB's oplog (which is used for master-slave replication) to keep the search engine up to date.

GridFS FUSE

There is a project working towards creating a FUSE filesystem on top of GridFS - something like this would create a bunch of interesting potential uses for MongoDB and GridFS:

http://github.com/mikejs/gridfs-fuse

Framework Adaptors

Working towards adding MongoDB support to major web frameworks is a great project, and work has been started on this for a variety of different frameworks (please use google to find out if work has already been started for your favorite framework).

Logging and Session Adaptors

MongoDB works great for storing logs and session information. There are a couple of projects working on supporting this use case directly.

Logging:
Zend: http://raphaelstolt.blogspot.com/2009/09/logging-to-mongodb-and-accessing-log.html
Python: http://github.com/andreisavu/mongodb-log
Rails: http://github.com/peburrows/mongo_db_logger

Sessions:
web.py: http://github.com/whilefalse/webpy-mongodb-sessions
Beaker: http://pypi.python.org/pypi/mongodb_beaker

Package Managers

Add support for installing MongoDB with your favorite package manager and let us know!

Locale-aware collation / sorting

MongoDB doesn't yet know how to sort query results in a locale-sensitive way. If you can think up a good way to do it and implement it, we'd like to know!

Drivers

If you use an esoteric/new/awesome programming language write a driver to support MongoDB! Again, check google to see what people have started for various languages.

Some that might be nice:

- Scheme (probably starting with PLT)
- GNU R
- Visual Basic
- Lisp (e.g, Common Lisp)
- Delphi
- Falcon

Write a killer app that uses MongoDB as the persistance layer!

Roadmap

Please see jira.

UI

Spec/requirements for a future MongoDB admin UI.

- list databases
    - repair, drop, clone?
- collections
    - validate(), datasize, indexsize, clone/copy

- indexes
- queries - explain() output
- security: view users, adjust
- see replication status of slave and master
- sharding
- system.profile viewer ; enable disable profiling
- curop / killop support

Source Code

Source for MongoDB and mongodb.org supported drivers is open source and hosted at Github .

- Mongo Database (includes C++ driver)
- Python Driver
- PHP Driver
- Ruby Driver
- Java Driver
- Perl Driver
- C# Driver
- Scala Driver
- Erlang Driver
- Haskell Driver

(Additionally, community drivers and tools also exist and will be found in other places.)

See Also

- Building
- License

Building

Note: see the Downloads page for prebuilt binaries, it's recommended to use those as all full QA occurs after those are built.

This section provides instructions on setting up your environment to write Mongo drivers or other infrastructure code. For specific instructions, go to the document that corresponds to your setup.

Sub-sections of this section:

- Building Boost
- Building for FreeBSD
- Building for Linux
- Building for OS X
- Building for Solaris
- Building for Windows
- Building Spider Monkey
- Building with V8
- scons

See Also

- The main Database Internals page

Building Boost

NOTE
This is not necessary when building mongo versions 2.1.1 or later.

MongoDB uses the [www.boost.org] C++ libraries.

Windows

See also the prebuilt libraries page.

By default c:\boost\ is checked for the boost files. Include files should be under \boost\boost, and libraries in \boost\lib.

First download the boost source. Then use the 7 Zip utility to extra the files. Place the extracted files in C:\boost.

Then we will compile the required libraries.

See buildscripts/buildboost.bat and buildscripts/buildboost64.bat for some helpers.

```
> rem set PATH for compiler:
> "C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\vcvarsall.bat"
>
> rem build the bjam make tool:
> cd \boost\tools\jam\src\
> build.bat
>
> cd \boost
> tools\jam\src\bin.ntx86\bjam --help
> rem see also mongo/buildscripts/buildboost*.bat
> rem build DEBUG libraries:
> tools\jam\src\bin.ntx86\bjam variant=debug threading=multi --with-program_options --with-filesystem
--with-date_time --with-thread
> mkdir lib
> move stage\lib\* lib\
```

Linux

It's common with linux to install boost via a package manager – see the Building for Linux page for specifics.

However one might also want to build from boost.org sources in some cases, for example to use a newer version of boost, to generate static libraries, etc.

The following – far from comprehensive, rather an example – shows manually building of boost libraries on Linux.

```

$ sudo ./bootstrap.sh

$ ./b2 --help

$ # now build it
$ ./b2
$ #or
$ ./b2 --with-program_options --with-filesystem --with-date_time --with-thread

$ sudo ./b2 install
```

Troubleshooting

Unresolved external get_system_category() when linking MongoDB
Try defining BOOST_SYSTEM_NO_DEPRECATED in the MongoDB SConstruct file.

Building for FreeBSD

On FreeBSD 8.0 and later, there is a mongodb port you can use.

For FreeBSD <= 7.2:

1. Get the database source: http://www.github.com/mongodb/mongo.
2. Update your ports tree:

   ```
   $ sudo portsnap fetch && portsnap extract
   ```

   The packages that come by default on 7.2 and older are too old, you'll get weird errors when you try to run the database)
3. Install SpiderMonkey:

   ```
   $ cd /usr/ports/lang/spidermonkey && make && make install
   ```

4. Install scons:

```
$ cd /usr/ports/devel/scons && make && make install
```

5. Install boost: (it will pop up an X "GUI", select `PYTHON`)

```
$ cd /usr/ports/devel/boost-all && make && make install
```

6. Install libexecinfo:

```
$ cd /usr/ports/devel/libexecinfo && make && make install
```

7. Change to the database source directory
8. `scons .`

See Also

- Building for Linux - many of the details there including how to clone from git apply here too.

Building for Linux

Note: Binaries are available for most platforms. Most users won't need to compile mongo themselves; in addition every prebuilt binary has been regression tested. See the Downloads page for these prebuilt binaries.

These instructions apply to the git master branch, and versions 2.1.1 and onward. At the end of the document are instructions for older versions.

Build Prerequisites

You'll need SCons, the gnu C++ toolchain, and glibc-devel. To get the code from github, you'll probably also want git.

Fedora

```
sudo yum -y install git-core scons gcc-c++ glibc-devel
```

Ubuntu

```
sudo apt-get install git-core build-essential scons
```

Building

1. Install any needed prerequisites (see above).
2. Get the source code

```
git clone git://github.com/mongodb/mongo.git
cd mongo
```

3. Pick a version to build (only use "master" if you're doing development work).
   a. List all tagged versions

```
git tag -l
```

b. Check out a tagged release, e.g. 2.0.4

```
git checkout r2.0.4
```

4. Compile

```
scons all
```

5. Install. Use --prefix to specify where you want to install your binaries. Defaults to /usr/local.

```
scons --prefix=/opt/mongo  install
```

Building Older Versions

Fedora

The following steps have been reported to work for Fedora versions from 11 to 16. (If they don't work on newer versions, please report this to mongodb-user@googlegroups.com.) Fedora versions 10 and earlier ship with a version of SCons that is too old to build MongoDB, but may work if you manually install SCons.

```
sudo yum -y install git-core scons gcc-c++ glibc-devel
sudo yum -y install boost-devel boost-devel-static
```

Ubuntu

Note: See SpiderMonkey note above.

Use cat /etc/lsb-release to see your version.

Ubuntu 8.04 and 8.10

```
apt-get -y install git-core build-essential
apt-get -y install libboost-dev libboost-program-options-dev libboost-thread-dev
libboost-filesystem-dev
```

Ubuntu 8.04 and 8.10 ship with a version of SCons that is too old to build MongoDB, but may work if you manually install SCons.

Ubuntu 9.04 and Newer

```
aptitude install -y git-core build-essential scons
aptitude install -y libboost-dev libboost-program-options-dev libboost-thread-dev
libboost-filesystem-dev
```

Building

1. Install prerequisites
2. get source

```
git clone git://github.com/mongodb/mongo.git
cd mongo
# pick a stable version unless doing true dev
git tag -l
# Switch to a stable branch (unless doing development) --
# an even second number indicates "stable". For example:
git checkout r2.0.4
```

3. build

```
scons all
```

4. install --prefix can be anywhere you want to contain your binaries, e.g., /usr/local or /opt/mongo.

```
scons --prefix=/opt/mongo install
```

Troubleshooting

- Link errors. If link errors occur, the `-t` gcc option is useful for troubleshooting. Try adding `-t` to the SConstruct file's `LINKFLAGS`.
- Static libraries. The `--release` scons option will build a binary using static libraries. You may need to install static boost libraries when using this option.

Building for OS X

To set up your OS X computer for MongoDB development:

Prerequisites

Xcode

Available in the App Store. You only need to get the command line tools, if you don't want to install the whole IDE.

SCons

SCons is the build tool used to compile mongo binaries. It is available from http://www.scons.org.

If you have easy_install or pip already installed, you can use them to install scons.

```
easy_install scons
```

```
pip install scons
```

git

An installer package is available from http://git-scm.com/

Building

1. Install any needed prerequisites (see above).
2. Get the source code

```
git clone git://github.com/mongodb/mongo.git
cd mongo
```

3. Pick a version to build (only use "master" if you're doing development work).
   a. List all tagged versions

   ```
   git tag -l
   ```

   b. Check out a tagged release, e.g. 2.0.4

   ```
   git checkout r2.0.4
   ```

4. Compile

```
scons all
```

5. Install. Use --prefix to specify where you want to install your binaries. Defaults to /usr/local.

```
scons --prefix=/opt/mongo  install
```

Older Versions of Mongo

Upgrading to Snow Leopard

If you have installed Snow Leopard, the builds will be 64-bit -- so if moving from a previous OS release, a bit more setup may be required than one might first expect.

Setup

Sources

The mongodb source is on github. To get sources first download the git client and install it.

- Then git clone git://github.com/mongodb/mongo.git (more info)
  Note If you do not wish to install git you can instead get the source code from the Downloads page.

Prerequisites

- Install gcc. Install XCode tools for Snow Leopard. gcc version 4.0.1 (from older XCode Tools install) works, but you will receive compiler warnings. One way to get a newer gcc is to install Command Line Tools for XCode from developer.apple.com.

Package Manager Setup

1. Install Homebrew
2. Update/install dependencies:

```
brew install boost
```

3. Install SCons:

```
brew install scons
```

Manual Setup

Install Apple developer tools

Install libraries (32-bit option)

1. Download boost 1.37.0. Apply the following patch:

```
diff -u -r a/configure b/configure
--- a/configure 2009-01-26 14:10:42.000000000 -0500
+++ b/configure 2009-01-26 10:21:29.000000000 -0500
@@ -9,9 +9,9 @@

 BJAM=""
 TOOLSET=""
-BJAM_CONFIG=""
+BJAM_CONFIG="--layout=system"
 BUILD=""
 PREFIX=/usr/local
 EPREFIX=
diff -u -r a/tools/build/v2/tools/darwin.jam b/tools/build/v2/tools/darwin.jam
--- a/tools/build/v2/tools/darwin.jam   2009-01-26 14:22:08.000000000 -0500
+++ b/tools/build/v2/tools/darwin.jam   2009-01-26 10:22:08.000000000 -0500
@@ -367,5 +367,5 @@

 actions link.dll bind LIBRARIES
 {
-    "$(CONFIG_COMMAND)" -dynamiclib -Wl,-single_module -install_name "$(<:B)$(<:S)" -L
"$(LINKPATH)" -o "$(<)" "$(>)" "$(LIBRARIES)" -l$(FINDLIBS-SA) -l$(FINDLIBS-ST)
$(FRAMEWORK_PATH) -framework$(_)$(FRAMEWORK:D=:S=) $(OPTIONS) $(USER_OPTIONS)
+    "$(CONFIG_COMMAND)" -dynamiclib -Wl,-single_module -install_name
"/usr/local/lib/$(<:B)$(<:S)" -L"$(LINKPATH)" -o "$(<)" "$(>)" "$(LIBRARIES)" -l$(FINDLIBS-SA)
-l$(FINDLIBS-ST) $(FRAMEWORK_PATH) -framework$(_)$(FRAMEWORK:D=:S=) $(OPTIONS) $(USER_OPTIONS)
 }
```

   then,

```
./configure; make; sudo make install
```

2. Install pcre (must enable UTF8)

```
./configure --enable-utf8 --enable-unicode-properties  --with-match-limit=200000
--with-match-limit-recursion=4000; make; sudo make install
```

3. Install c++ unit test framework http://unittest.red-bean.com/ (optional)

```
./configure; make; sudo make install
```

Install libraries (64-bit option)

(The 64-bit libraries will be installed in /usr/64/{include,lib}.)

1. Download SpiderMonkey: ftp://ftp.mozilla.org/pub/mozilla.org/js/js-1.7.0.tar.gz

Apply the following patch:

```
diff -u  -r js/src/config/Darwin.mk js-1.7.0/src/config/Darwin.mk
--- js/src/config/Darwin.mk 2007-02-05 11:24:49.000000000 -0500
+++ js-1.7.0/src/config/Darwin.mk 2009-05-11 10:18:37.000000000 -0400
@@ -43,7 +43,7 @@
 # Just ripped from Linux config
 #

-CC = cc
+CC = cc -m64
 CCC = g++
 CFLAGS +=  -Wall -Wno-format
 OS_CFLAGS = -DXP_UNIX -DSVR4 -DSYSV -D_BSD_SOURCE -DPOSIX_SOURCE -DDARWIN
@@ -56,9 +56,9 @@
 #.c.o:
 #      $(CC) -c -MD $*.d $(CFLAGS) $<

-CPU_ARCH = $(shell uname -m)
+CPU_ARCH = "X86_64"
 ifeq (86,$(findstring 86,$(CPU_ARCH)))
-CPU_ARCH = x86
+CPU_ARCH = x86_64
 OS_CFLAGS+= -DX86_LINUX
 endif
 GFX_ARCH = x
@@ -81,3 +81,14 @@
 # Don't allow Makefile.ref to use libmath
 NO_LIBM = 1

+ifeq ($(CPU_ARCH),x86_64)
+# Use VA_COPY() standard macro on x86-64
+# FIXME: better use it everywhere
+OS_CFLAGS += -DHAVE_VA_COPY -DVA_COPY=va_copy
+endif
+
+ifeq ($(CPU_ARCH),x86_64)
+# We need PIC code for shared libraries
+# FIXME: better patch rules.mk & fdlibm/Makefile*
+OS_CFLAGS += -DPIC -fPIC
+endif
```

compile and install

```
cd src
make -f Makefile.ref
sudo JS_DIST=/usr/64 make -f Makefile.ref export
```

remove the dynamic library

```
sudo rm /usr/64/lib64/libjs.dylib
```

Download boost 1.37.0 Apply the following patch:

```
diff -u -r a/configure b/configure
--- a/configure 2009-01-26 14:10:42.000000000 -0500
+++ b/configure 2009-01-26 10:21:29.000000000 -0500
@@ -9,9 +9,9 @@

 BJAM=""
 TOOLSET=""
-BJAM_CONFIG=""
+BJAM_CONFIG="architecture=x86 address-model=64 --layout=system"
 BUILD=""
-PREFIX=/usr/local
+PREFIX=/usr/64
 EPREFIX=
 LIBDIR=
 INCLUDEDIR=
diff -u -r a/tools/build/v2/tools/darwin.jam b/tools/build/v2/tools/darwin.jam
--- a/tools/build/v2/tools/darwin.jam   2009-01-26 14:22:08.000000000 -0500
+++ b/tools/build/v2/tools/darwin.jam   2009-01-26 10:22:08.000000000 -0500
@@ -367,5 +367,5 @@

 actions link.dll bind LIBRARIES
 {
-    "$(CONFIG_COMMAND)" -dynamiclib -Wl,-single_module -install_name "$(<:B)$(<:S)" -L"$(LINKPATH)"
-o "$(<)" "$(>)" "$(LIBRARIES)" -l$(FINDLIBS-SA) -l$(FINDLIBS-ST) $(FRAMEWORK_PATH)
-framework$(_)$(FRAMEWORK:D=:S=) $(OPTIONS) $(USER_OPTIONS)
+    "$(CONFIG_COMMAND)" -dynamiclib -Wl,-single_module -install_name "/usr/64/lib/$(<:B)$(<:S)" -L
"$(LINKPATH)" -o "$(<)" "$(>)" "$(LIBRARIES)" -l$(FINDLIBS-SA) -l$(FINDLIBS-ST) $(FRAMEWORK_PATH)
-framework$(_)$(FRAMEWORK:D=:S=) $(OPTIONS) $(USER_OPTIONS)
 }
```

then,

```
./configure; make; sudo make install
```

Install pcre (must enable UTF8)

```
CFLAGS="-m64" CXXFLAGS="-m64" LDFLAGS="-m64" ./configure --enable-utf8 --with-match-limit=200000
--with-match-limit-recursion=4000 --enable-unicode-properties --prefix /usr/64; make; sudo make
install
```

Install unit test framework http://unittest.red-bean.com/ (optional)

```
CFLAGS="-m64" CXXFLAGS="-m64" LDFLAGS="-m64" ./configure --prefix /usr/64; make; sudo make install
```

Compiling

To compile 32-bit, just run:

```
scons
```

To compile 64-bit on 10.5 (64 is default on 10.6), run:

```
scons  --64
```

See the, MongoDB scons page for more details/compile options.

Troubleshooting

- Undefined symbols: "_PR_NewLock", referenced from: _JS_Init in libjs.a.
    - Try not using the scons `--release` option (if you are using it). That option attempts to use static libraries.

Building for Solaris

MongoDB server currently supports little endian Solaris operation. (Although most drivers – not the database server – work on both.)

Community: Help us make this rough page better please! (And help us add support for big endian please...)

Prerequisites:

- g++ 4.x (SUNWgcc)
- scons (need to install from source)
- spider monkey Building Spider Monkey
- pcre (SUNWpcre)
- boost (need to install from source)

See Also

- Joyent
- Building for Linux - many of the details there including how to clone from git apply here too

Building for Windows

> Binaries are available for most platforms. Most users won't need to compile mongo themselves; in addition every prebuilt binary has been regression tested. See the Downloads page for these prebuilt binaries.

MongoDB can be compiled for Windows (32 and 64 bit); You will need to have the platform sdk installed. The platform sdk can be installed manually, and it comes with Visual (Studio) C++ as well. SCons is the make mechanism, although several .vcxprojs and a .sln solution file are also included in the project for convenience when using the Visual Studio 2010 IDE.

There are several dependencies exist which are listed below; you may find it easier to simply download a pre-built binary.

- Building with Visual Studio 2008
- Building with Visual Studio 2010
- Building the Shell

See Also

- Prebuilt Boost Libraries
- Prebuilt SpiderMonkey for VS2010
- Building Boost
- Building SpiderMonkey
- Windows Quick Links
- scons

Boost 1.41.0 Visual Studio 2010 Binary

> This is OLD and was for the VS2010 BETA. See the new Boost and Windows page instead.

The following is a prebuilt boost binary (libraries) for Visual Studio 2010 beta 2.

The MongoDB vcxproj files assume this package is unzipped under c:\Program Files\boost\boost_1_41_0\.

- http://downloads.mongodb.org/misc/boost_1_41_0_binary_vs10beta2.zipx

Note: we're not boost build gurus please let us know if there are things wrong with the build.

See also the prebuilt boost binaries at http://www.boostpro.com/download.

Boost and Windows

- Visual Studio 2010
  - Prebuilt from mongodb.org
  - Building Yourself
- Visual Studio 2008
  - Prebuilt from mongodb.org

Visual Studio 2010

Prebuilt from mongodb.org

Click here for a prebuilt boost library for Visual Studio 2010. 7zip format.

Building Yourself

- Download the boost source from boost.org. Move it to C:\boost\.
    - We have successfully compiled version **1.42** – you might want to try that version or higher, but not 1.45 or later. 1.45 changed the interface to the boost::filesystem library and we've yet to catch up. See additional notes section at end of this page too.
- Run `C:\Program Files (x86)\Microsoft Visual Studio 10.0\vc\vcvarsall.bat`.
- From the MongoDB source project, run buildscripts\buildboost.bat. Or, buildboost64.bat for the 64 bit version.

Visual Studio 2008

Prebuilt from mongodb.org

Click here for a prebuilt boost library for Visual Studio 2008. 7zip format. This file has what you need to build MongoDB, but not some other boost libs, so it's partial.

Prebuilt from boostpro.com

Or, you can download a complete prebuilt boost library for 32 bit VS2008 at http://www.boostpro.com/products/free. Install the prebuilt libraries for Boost version 1.35.0 (or higher - generally newer is better). During installation, for release builds choose `static multithread libraries for installation`. The Debug version of the project uses the DLL libraries; choose all multithread libraries if you plan to do development. From the BoostPro installer, be sure to select all relevant libraries that mongodb uses -- for example, you need Filesystem, Regex, Threads, and ProgramOptions (and perhaps others).

Building Yourself

- Download the boost source from boost.org. Move it to C:\boost\.
- From the Visual Studio 2008 IDE, choose Tools.Visual Studio Command Prompt to get a command prompt with all PATH variables set nicely for the C++ compiler.
- From the MongoDB source project, run buildscripts\buildboost.bat. Or, buildboost64.bat for the 64 bit version.

Additional Notes

When using bjam, MongoDB expects

- `variant=debug` for debug builds, and `variant=release` for release builds
- `threading=multi`
- `link=static runtime-link=static` for release builds
- `address-model=64` for 64 bit

Building the Mongo Shell on Windows

You can build the mongo shell with either scons or a Visual Studio 2010 project file.

Scons

```
scons mongo.exe
```

Visual Studio 2010 Project File

A VS2010 vcxproj file is availabe for building the shell. From the mongo directory open shell/msvc/mongo.vcxproj.

For versions of the shell prior to version 1.9, the project file assumes that GNU readline is installed in ..\readline\ relative to the mongo project. If you would prefer to build without having to install readline, remove the definition of USE_READLINE in the preprocessor definitions section of the project file, and exclude readline.lib from the project.

The project file currently only supports 32 bit builds of the shell (scons can do 32 and 64 bit). However this seems sufficient given there is no real need for a 64 bit version of the shell.

Readline Library

Versions of the shell prior to 1.9 used the GNU readline library to facilitate command line editing and history. For these older versions, you can build the shell without readline but would then lose that functionality. USE_READLINE is defined when building with readline. SCons will look for readline and if not found build without it.

See Also

- Prebuilt readline for Windows 32 bit at SourceForge (DLL version)

Building with Visual Studio 2008

NOTE

These instructions are for versions of mongo prior to 2.1.1. For version 2.1.1 and newer, the instructions for Visual Studio 2010 and Visual Studo 2008 are the same.

- Get the MongoDB Source Code
- Get Boost Libraries
- Get SpiderMonkey
- Install SCons
- Building MongoDB with SCons
- Troubleshooting

MongoDB can be compiled for Windows (32 and 64 bit) using Visual C++. SCons is the make mechanism we use with VS2008. (Although it is possible to build from a sln file with vs2010.)

There are several dependencies exist which are listed below; you may find it easier to simply download a pre-built binary.

Get the MongoDB Source Code

Download the source code from Downloads.

Or install Git. Then:

- `git clone git://github.com/mongodb/mongo.git` (more info)
- `git tag -l` to see tagged version numbers
- Switch to a stable branch (unless doing development) -- an even second number indicates "stable". (Although with sharding you will want the latest if the latest is less than 1.6.0.) For example:
  - `git checkout r1.4.1`

Get Boost Libraries

- Click here for a prebuilt boost library for Visual Studio. 7zip format. This file has what you need to build MongoDB, but not some other boost libs, so it's partial.
- See the Boost and Windows page for other options.

The Visual Studio project files are setup to look for boost in the following locations:

- c:\program files\boost\latest
- c:\boost
- \boost

You can unzip boost to c:\boost, or use an NTFS junction point to create a junction point to one of the above locations. Some versions of windows come with linkd.exe, but others require you to download Sysinternal's junction.exe to accomplish this task. For example, if you installed boost 1.42 via the installer to the default location of **c:\Program Files\boost\boost_1_42**, You can create a junction point with the following command:

```
junction "c:\Program Files\boost\latest" "c:\Program Files\boost\boost_1_42"
```

This should return the following output:

```
Junction v1.05 - Windows junction creator and reparse point viewer
Copyright (C) 2000-2007 Mark Russinovich
Systems Internals - http://www.sysinternals.com

Created: c:\Program Files\boost\latest
Targetted at: c:\Program Files\boost\boost_1_42
```

Get SpiderMonkey

Build a SpiderMonkey js engine library (js.lib) – details here.

Install SCons

If building with scons, install SCons:

- First install Python: http://www.python.org/download/releases/2.6.4/.
- Then SCons itself: http://sourceforge.net/projects/scons/files/scons/1.2.0/scons-1.2.0.win32.exe/download.
- Add the python scripts directory (e.g., C:\Python26\Scripts) to your PATH.

Building MongoDB with SCons

The SConstruct file from the MongoDB project is the preferred way to perform production builds. Run scons in the mongo project directory to build.

If scons does not automatically find Visual Studio, preset your path for it by running the VS2010 vcvars*.bat file.

To build:

```
scons                    // build mongod
scons mongoclient.lib    // build C++ client driver library
scons all                // build all end user components
scons .                  // build all including unit test
```

Troubleshooting

If you are using `scons`, check the file `config.log` which is generated.

- **Can't find jstypes.h when compiling.** This file is generated when building SpiderMonkey. See the Building SpiderMonkey page for more info.
- **Can't find / run cl.exe when building with scons.** See troubleshooting note on the Building SpiderMonkey page.
- **Error building program database.** (VS2008.) Try installing the Visual Studio 2008 Service Pack 1.

Building with Visual Studio 2010

Binaries are available for most platforms. Most users won't need to compile mongo themselves; in addition every prebuilt binary has been regression tested. See the Downloads page for these prebuilt binaries.

v2.1.1+

MongoDB can be compiled for Windows (32 and 64 bit) using Visual C++. SCons is the make mechanism, although a solution file is also included in the project for convenience when using the Visual Studio IDE. (These instructions don't work using Visual Studio Express, which must be uninstalled to get Visual Studio Professional/Ultimate to work properly; VSE can only do 32 bit builds.)

These instructions are for mongo versions 2.1.1 and later.

Get the MongoDB Source Code

Download the source code from Downloads.

Or install Git. Then:

- `git clone git://github.com/mongodb/mongo.git` (more info)
- `git tag -l` to see tagged version numbers
- Switch to a stable branch (unless doing development) -- an even second number indicates "stable". For example:
  - `git checkout r1.4.1`

Building MongoDB from the IDE

Open the db\db_10.sln solution file.

Note: a separate project file exists for the `mongo` shell. Currently the C++ client libraries must be built from scons (this obviously needs to be fixed...)

Building with SCons

1. Install SCons:
   a. First install Python: http://www.python.org/download/releases/2.7.2/.
      - **Note** Make sure to install the 32 bit version of python and not the 64 bit as the scons binaries below are 32 bit.
   b. It is recommended you install pywin32 if you want to do parallel builds (scons -j). http://sourceforge.net/projects/pywin32/files/pywin32/Build216/pywin32-216.win32-py2.7.exe/download
   c. Then install SCons itself: http://sourceforge.net/projects/scons/files/scons/2.1.0/scons-2.1.0.win32.exe/download.
   d. Add the python scripts directory (e.g., C:\Python27\Scripts) to your PATH.

1. Build:

```
scons                    // build mongod
scons --release mongoclient.lib   // build C++ client driver library
scons --release core               // build all end user components
```

Add --64 or --32 to get the 64 and 32-bit versions, respectively. Replace --release with --dd to build a debug build.

Troubleshooting

> If you are using `scons`, check the file `config.log` which is generated.

If scons does not automatically find Visual Studio, try using the Visual Studio Command Prompt, which will set your path for you. Alternatively, set your path manually by running the VS2010 vcvars*.bat files. Location may vary with install but usually it is something like:

- C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\Tools\vsvars32.bat
- C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\amd64\vcvars64.bat

Older versions

The following instructions are for versions of mongo prior to 2.1.1.

There are several dependencies exist which are listed below; you may find it easier to simply download a pre-built binary.

Get the MongoDB Source Code

Download the source code from Downloads.

Or install Git. Then:

- `git clone git://github.com/mongodb/mongo.git` (more info)
- `git tag -l` to see tagged version numbers

- Switch to a stable branch (unless doing development) -- an even second number indicates "stable". (Although with sharding you will want the latest if the latest is less than 1.6.0.) For example:
    - `git checkout r1.4.1`

Get Boost Libraries

- Click here for a prebuilt boost library for Visual Studio. 7zip format. This file has what you need to build MongoDB, but not some other boost libs, so it's partial. Uncompress this to the c:\boost directory. Your actual files are in c:\boost\boost
- See the Boost and Windows page for other options. Use v1.42 or higher with VS2010.

Get SpiderMonkey

- Download prebuilt libraries and headers here for VS2010. Place these files in ..\js\ relative to your mongo project directory.
- Or (more work) build SpiderMonkey js.lib yourself – details here.

Building MongoDB from the IDE

Open the db\db_10.sln solution file.

Note: a separate project file exists for the `mongo` shell. Currently the C++ client libraries must be built from scons (this obviously needs to be fixed...)

Install SCons

If building with scons, install SCons:

- First install Python: http://www.python.org/download/releases/2.7.2/.
    - **Note** Make sure to install the 32 bit version of python and not the 64 bit as the scons binaries below are 32 bit.
- Its recommended you install pywin32 if you want to do parallel builds (scons -j). http://sourceforge.net/projects/pywin32/files/pywin32/Build216/pywin32-216.win32-py2.7.exe/download
- Then install SCons itself: http://sourceforge.net/projects/scons/files/scons/2.1.0/scons-2.1.0.win32.exe/download.
- Add the python scripts directory (e.g., C:\Python27\Scripts) to your PATH.

The SConstruct file from the MongoDB project is the preferred way to perform production builds. Run scons in the mongo project directory to build.

If scons does not automatically find Visual Studio, preset your path for it by running the VS2010 vcvars*.bat files. Location may vary with install but usually it is something like:

- C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\Tools\vsvars32.bat
- C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\amd64\vcvars64.bat

To build:

```
scons                    // build mongod
scons mongoclient.lib    // build C++ client driver library
scons all                // build all end user components
scons .                  // build all including unit tests and C++ driver zip file
scons --64               // build the 64 bit version
scons --dd               // build with debug symbols
scons -jX                // perform X steps in parallel (e.g. -j16 will compile 16 files at once)
```

Troubleshooting

If you are using `scons`, check the file `config.log` which is generated.

- Can't find jstypes.h when compiling.
    - This file is generated when building SpiderMonkey. See the Building SpiderMonkey page for more info.
- Can't find / run cl.exe when building with scons.
    - Be sure to use Visual Studio Command Prompt so that your path is set correctly.
- LINK : fatal error LNK1104: cannot open file js64d.lib js64r.lib js32d.lib js32r.lib
    - Get the prebuilt spidermonkey libraries -- or copy your self-built js.lib to the above name.
    - You can also see this if you're using the wrong compiler; this is the result if you try to use Visual Studio Express instead of Visual Studio Professional/Ultimate, which is a different product.

Building Spider Monkey

MongoDB uses SpiderMonkey for server-side Javascript execution.

Pre v2.0: MongoDB requires a js.lib file when linking. This page details how to build js.lib.

v2.0+: this is handled automatically by the Mongo build scripts via files under third_party/ in the MongoDB project directory.

Note: V8 Javascript support is under development.

Building js.lib - Unix

Remove any existing xulrunner

First find out what has been installed

```
dpkg -l | grep xulrunner
```

e.g.

```
ubuntu910-server64:mongo$ sudo dpkg -l | grep xul
ii  xulrunner-1.9.1                   1.9.1.13+build1+nobinonly-0ubuntu0.9.10.1 XUL + XPCOM
application runner
ii  xulrunner-1.9.1-dev               1.9.1.13+build1+nobinonly-0ubuntu0.9.10.1 XUL + XPCOM
development files
```

Next remove the two installed packages

```
sudo apt-get remove xulrunner-1.9.1-dev xulrunner-1.9.1
```

Download

```
curl -O ftp://ftp.mozilla.org/pub/mozilla.org/js/js-1.7.0.tar.gz
tar zxvf js-1.7.0.tar.gz
```

Build

```
cd js/src
export CFLAGS="-DJS_C_STRINGS_ARE_UTF8"
make -f Makefile.ref
```

SpiderMonkey does not use UTF-8 by default, so we enable before building.

An experimental SConstruct build file is available here.

Install

```
JS_DIST=/usr make -f Makefile.ref export
```

By default, the mongo scons project expects spidermonkey to be located at ../js/.

Building js.lib - Windows

Prebuilt

- VS2008: a prebuilt SpiderMonkey library and headers for Win32 is attached to this document (this file may or may not work depending on your compile settings and compiler version).
- VS2010 prebuilt libraries (js64d.lib, etc.)

Alternatively, follow the steps below to build yourself.

Download

From an msysgit or cygwin shell, run:

```
curl -O ftp://ftp.mozilla.org/pub/mozilla.org/js/js-1.7.0.tar.gz
tar zxvf js-1.7.0.tar.gz
```

Build

cd js/src

```
export CFLAGS="-DJS_C_STRINGS_ARE_UTF8"
make -f Makefile.ref
```

If cl.exe is not found, launch Tools...Visual Studio Command Prompt from inside Visual Studio -- your path should then be correct for make.

If you do not have a suitable make utility installed, you may prefer to build using scons. An experimental SConstruct file to build the js.lib is available in the mongodb/snippets project. For example:

```
cd
git clone git://github.com/mongodb/mongo-snippets.git
cp mongo-snippets/jslib-sconstruct js/src/SConstruct
cd js/src
scons
```

Troubleshooting scons

Note that scons does not use your PATH to find Visual Studio. If you get an error running cl.exe, try changing the following line in the msvc.py scons source file from:

```
MVSdir = os.getenv('ProgramFiles') + r'\Microsoft Visual Studio 8'
```

to

```
MVSdir = os.getenv('ProgramFiles') + r'\Microsoft Visual Studio ' + version
```

See Also

- Building MongoDB

Building with V8

Linux or OSX

```
$ pwd
~/mongo
$ cd ..
$ svn checkout http://v8.googlecode.com/svn/trunk/ v8
$ cd v8
$ scons arch=x64 debuggersupport=off snapshot=off profilingsupport=off
$ cd ../mongo
$ scons --usev8
```

Windows

```
$ pwd
~/mongo
$ cd ..
$ svn checkout http://v8.googlecode.com/svn/trunk/ v8
$ cd v8
$ scons env="PATH:%PATH%,INCLUDE:%INCLUDE%,LIB:%LIB%" arch=x64
$ cd ../mongo
$ scons --usev8 --release --64
```

scons

Use scons to build MongoDB and related utilities and libraries. See the SConstruct file for details.

Run `scons --help` to see all options.

Targets

Run `scons <target>`.

- `scons .`
- `scons all`
- `scons mongod` build mongod
- `scons mongo` build the shell
- `scons shell` generate (just) the shell `.cpp` files (from .js files)
- `scons mongoclient` build just the client library (builds libmongoclient.a on unix)
- `scons test` build the unit test binary `test`

Options

- `--d` debug build; all this does is turns optimization off
- `--dd` debug build with _DEBUG defined (extra asserts, checks, etc.)
- `--release`
- `--32` force 32 bit
- `--64` force 64 bit
- `--clean`
- `--mute` suppress compile and link command lines

Troubleshooting

scons generates a `config.log` file. See this file when there are problems building.

See Also

Smoke Tests

Database Internals

This section provides information for developers who want to write drivers or tools for MongoDB, \ contribute code to the MongoDB codebase itself, and for those who are just curious how it works internally.

Sub-sections of this section:

- Caching
- Durability Internals
- Parsing Stack Traces
- Cursors
- Error Codes
- Internal Commands
- Replication Internals
- Smoke Tests
- Pairing Internals

Caching

Memory Mapped Storage Engine

This is the current storage engine for MongoDB, and it uses memory-mapped files for all disk I/O. Using this strategy, the operating system's virtual memory manager is in charge of caching. This has several implications:

- There is no redundancy between file system cache and database cache: they are one and the same.
- MongoDB can use all free memory on the server for cache space automatically without any configuration of a cache size.
- Virtual memory size and resident size will appear to be very large for the mongod process. This is benign: virtual memory space will be just larger than the size of the datafiles open and mapped; resident size will vary depending on the amount of memory not used by other processes on the machine.
- Caching behavior such as *Least Recently Used* (LRU) discarding of pages, and laziness of page writes is controlled by the operating system (the quality of the VMM implementation will vary by OS.)

To monitor or check memory usage see: Checking Server Memory Usage

Memory per connection

A thread is associated with each connection from clients to the database. Each thread has a stack that has a size of a few MB. The unused portions of these stacks can be swapped out by the OS as long as the connections live for a while (would not if connect / operation / disconnect in 2 seconds).

Binary footprint

You can get a feel for the “inherent” memory footprint of Mongo by starting it fresh, with no connections, with an empty /data/db directory and looking at the resident bytes.

See Also

- Checking Server Memory Usage
- The Linux Out of Memory OOM Killer

Durability Internals

> ⚠ The main durability page (not the internals page) is the Journaling page.

- Files
- Running
- Declaring Write Intent
- Tests
- Administrative
- Diagrams

Files

The data file format is unchanged.

Journal files are placed in */data/db/journal/*.

Running

Run with `--journal` to enable journaling/durable storage. Both `mongod` and `test` support this option.

Declaring Write Intent

When writing `mongod` kernel code, one must now declare an intention to write. Declaration of the intent occurs before the actual write. See `db/dur.h`. The actual write must occur before releasing the write lock.

When you do your actual writing, use the pointer that dur::writing() returns, rather than the original pointer.

```
Foo *foo;
getDur().writing(thing)->bar = something;

int *x;
getDur().writingInt(x) += 3;

DiskLoc &loc;
loc.writing() = newLoc;

void *p;
unsigned len;
memcpy( getDur().writingPtr(p,len), src, len );
```

Try to declare intent on as small a region as possible. That way less information is journalled. For example

```
BigStruct *b;

dur::writing(b)->x = 3; // less efficient

*dur::writing(&b->x) = 3; // more efficient
```

However, there is some overhead for each intent declaration, so if many members of a struct will be written, it is likely better to just declare intent on the whole struct.

Tests

`jstests/dur/` contains tests for durability.

```
mongo --nodb jstests/dur/<testname>.js
```

Administrative

```
# dump journal entries during any recover, and then start normally
mongod --journal --durOptions 1

# recover and terminate
mongod --journal --durOptions 4

# dump journal entries (doing nothing else) and then terminate
mongod --journal --durOptions 7

# extra checks that everything is correct (slow but good for qa)
mongod --journal --durOptions 8
```

Diagrams

- diagram 1 - process steps
- diagram 2 - journal file structure

Parsing Stack Traces

addr2line

```
addr2line -e mongod -ifC <offset>
```

c++filt

You can use `c++filt` to demangle function names by pasting the whole stack trace to stdin.

Finding the right binary

To find the binary you need:

- Get the commit at the header of any of our logs.
- Use git to locate that commit and check for the surrounding "version bump" commit

Download and open the binary:

```
curl -O http:
//s3.amazonaws.com/downloads.mongodb.org/linux/mongodb-linux-x86_64-debugsymbols-1.x.x.tgz
```

You can also find debugsymbols for official builds by clicking "list" on the Downloads page.

Example 1

Then, the log has lines like this:

```
/home/abc/mongod(_ZN5mongo15printStackTraceERSo+0x27) [0x689280]
```

You want the address in between the brackets [0x689280]

Note you will get more than one stack frame for the address if the code is inlined.

Example 2

Actual example from a v1.8.1 64 bit linux build:

```
$ curl http://downloads.mongodb.org/linux/mongodb-linux-x86_64-debugsymbols-1.8.1.tgz > out.tgz
$ tar -xzf out.tgz
$ cd mongodb-linux-x86_64-debugsymbols-1.8.1/
$ cd bin
$ addr2line --help
$ addr2line -i -e mongod 0x6d6a74
/mnt/home/buildbot/slave/Linux_64bit_V1.8/mongo/db/repl/health.cpp:394
$ addr2line -i -e mongod 0x6d0694
/mnt/home/buildbot/slave/Linux_64bit_V1.8/mongo/db/repl/rs.h:385
/mnt/home/buildbot/slave/Linux_64bit_V1.8/mongo/db/repl/replset_commands.cpp:111
```

Cursors

Redirection Notice
This page should redirect to Internals.

Error Codes

If you have and error event and it isn't obvious what the error is, query for that error code on Jira. If still nothing please post to support forums.

This list is HIGHLY incomplete. This page is a stub.

Error Code	Description	Comments
10003	objects in a capped ns cannot grow	
11000	duplicate key error	_id values must be unique in a collection
11001	duplicate key on update	
12000	idxNo fails	an internal error
12001	can't sort with $snapshot	the $snapshot feature does not support sorting yet
12010, 12011, 12012	can't $inc/$set an indexed field	
13312	replSet error : logOp() but not primary?	Fixed in v2.0. Report if seen v2.0+
13440	bad offset accessing a datafile	Run a database --repairr. If journaling is on this shouldn't happen.

Internal Commands

Most commands have helper functions and do not require the `$cmd.findOne()` syntax. These are primarily internal and administrative.

```
> db.$cmd.findOne({assertinfo:1})
{
    "dbasserted" : false , // boolean: db asserted
"asserted" : false , // boolean: db asserted or a user assert have happend
"assert" : "" ,  // regular assert
"assertw" : "" , // "warning" assert
"assertmsg" : "" , // assert with a message in the db log
"assertuser" : "" , // user assert - benign, generally a request that was not meaningful
"ok" : 1.0
}

> db.$cmd.findOne({serverStatus:1})
{
  "uptime" : 6 ,
  "globalLock" : {
                  "totalTime" : 6765166 ,
                  "lockTime" : 2131 ,
                  "ratio" : 0.00031499596610046226
               } ,
   "mem" : {
                  "resident" : 3 ,
                  "virtual" : 111 ,
                  "mapped" : 32
   } ,
  "ok" : 1
}



> admindb.$cmd.findOne({replacepeer:1})
{
    "info" : "adjust local.sources hostname; db restart now required" ,
    "ok" : 1.0
}

// close all databases.  a subsequent request will reopen a db.
> admindb.$cmd.findOne({closeAllDatabases:1});
```

Replication Internals

On the *master* mongod instance, the `local` database will contain a collection, `oplog.$main`, which stores a high-level transaction log. The transaction log essentially describes all actions performed by the user, such as "insert this object into this collection." Note that the oplog is not a low-level redo log, so it does not record operations on the byte/disk level.

The *slave* mongod instance polls the `oplog.$main` collection from *master*. The actual query looks like this:

```
local.oplog.$main.find({ ts: { $gte: ''last_op_processed_time'' } }).sort({$natural:1});
```

where 'local' is the master instance's `local` database. `oplog.$main` collection is a capped collection, allowing the oldest data to be aged out automatically.

See the Replication section of the Mongo Developers' Guide for more information.

`OpTime`

An `OpTime` is a 64-bit timestamp that we use to timestamp operations. These are stored as Javascript `Date` datatypes but are *not* JavaScript Date objects. Implementation details can be found in the `OpTime` class in *repl.h*.

Applying OpTime Operations

Operations from the oplog are applied on the slave by reexecuting the operation. Naturally, the log includes write operations only.

Note that inserts are transformed into upserts to ensure consistency on repeated operations. For example, if the slave crashes, we won't know exactly which operations have been applied. So if we're left with operations 1, 2, 3, 4, and 5, and if we then apply 1, 2, 3, 2, 3, 4, 5, we should achieve the same results. This repeatability property is also used for the initial cloning of the replica.

Tailing

After applying operations, we want to wait a moment and then poll again for new data with our `$gte`operation. We want this operation to be fast, quickly skipping past old data we have already processed. However, we do not want to build an index on `ts`, as indexing can be somewhat expensive, and the oplog is write-heavy. Instead, we use a table scan in natural order, but use a tailable cursor to "remember" our position. Thus, we only scan once, and then when we poll again, we know where to begin.

Initiation

To create a new replica, we do the following:

```
t = now();
cloneDatabase();
end = now();
applyOperations(t..end);
```

`cloneDatabase`effectively exports/imports all the data in the database. Note the actual "image" we will get may or may not include data modifications in the time range (`t..end`). Thus, we apply all logged operations from that range when the cloning is complete. Because of our repeatability property, this is safe.

See class `Cloner` for more information.

Smoke Tests

Test Organization

1. `dbtests/.*cpp` has C++ unit tests
2. `jstests/*.js` has core tests
3. `jstests/repl*/*.js` has replication tests
4. `jstests/sharding/*.js` has sharding tests

5. `slowNightly/*js` has tests that take longer and run only at night
6. `slowWeekly/*.js` has tests that take even longer and run only once a week

Running all the tests

```
scons smoke smokeDisk smokeTool smokeAuth startMongod smokeClient smokeJs
scons startMongodSmallOplog smokeJs
scons startMongod smokeJsSlowNightly
scons smokeTool
scons smokeReplSets
scons smokeDur
scons mongosTest smokeSharding
scons smokeRepl smokeClone
scons startMongod smokeParallel
```

smoke.py

smoke.py lets you run a subsets of the tests in *jstests/*. When it is running tests, it starts up an instance of mongod, runs the tests, and then shuts it down again. You can run it while running other instances of MongoDB on the same machine: it uses ports in the 30000 range and its own data directories.

For the moment, *smoke.py* must be run from the top-level directory of a MongoDB source repository. This directory must contain at least the `mongo` and `mongod` binaries. To run certain tests, you'll also need to build the tools and `mongos`. It's a good idea to run `scons .` before running the tests.

To run *smoke.py* you'll need a recent version of PyMongo.

To see the possible options, run:

```
$ python buildscripts/smoke.py --help
Usage: smoke.py [OPTIONS] ARGS*

Options:
  -h, --help            show this help message and exit
  --mode=MODE           If "files", ARGS are filenames; if "suite", ARGS are
                        sets of tests (suite)
  --test-path=TEST_PATH
                        Path to the test executables to run, currently only
                        used for 'client' (none)
  --mongod=MONGOD_EXECUTABLE
                        Path to mongod to run (/Users/mike/10gen/mongo/mongod)
  --port=MONGOD_PORT    Port the mongod will bind to (32000)
  --mongo=SHELL_EXECUTABLE
                        Path to mongo, for .js test files
                        (/Users/mike/10gen/mongo/mongo)
  --continue-on-failure
                        If supplied, continue testing even after a test fails
  --from-file=FILE      Run tests/suites named in FILE, one test per line, '-'
                        means stdin
  --smoke-db-prefix=SMOKE_DB_PREFIX
                        Prefix to use for the mongods' dbpaths ('')
  --small-oplog         Run tests with master/slave replication & use a small
                        oplog
```

By default, `smoke.py` will run tests that create data in `/data/db`, which may interfere with other MongoDB instances you are running. To change the directory in which the smoke tests create databases, use `--smoke-db-prefix=/some/other/path`

To run specific tests, use the *--mode=files* option:

```
python buildscripts/smoke.py --mode=files jstests/find1.js
```

You can specify as many files as you want.

You can also run a suite of tests. Suites are predefined and include:

- *test*
- *all*
- *perf*
- *js*
- *quota*
- *jsPerf*
- *disk*
- *jsSlowNightly*
- *jsSlowWeekly*
- *parallel*
- *clone*
- *repl* (master/slave replication tests)
- *replSets* (replica set tests)
- *auth*
- *sharding*
- *tool*
- *client*
- *mongosTest*

To run a suite, specify the suite's name:

```
python buildscripts/smoke.py js
```

Running a jstest manually

You can run a jstest directly from the shell, for example:

```
mongo --nodb jstests/replsets/replsetarb3.js
```

Running the C++ unit tests

The tests under jstests/ folder are written in mongo shell javascript. However there are a set of C++ unit tests also. To run them:

```

scons test
./test
```

See Also

- scons

Pairing Internals

Policy for reconciling divergent oplogs

pairing is deprecated

In a paired environment, a situation may arise in which each member of a pair has logged operations as master that have not been applied to the other server. In such a situation, the following procedure will be used to ensure consistency between the two servers:

1. The new master will scan through its own oplog from the point at which it last applied an operation from it's peer's oplog to the end. It will create a set C of object ids for which changes were made. It will create a set M of object ids for which only modifier changes were made. The values of C and M will be updated as client operations are applied to the new master.
2. The new master will iterate through its peer's oplog, applying only operations that will not affect an object having an id in C.
3. For any operation in the peer's oplog that may not be applied due to the constraint in the previous step, if the id of the of the object in question is in M, the value of the whole object on the new master is logged to the new master's oplog.
4. The new slave applies all operations from the new master's oplog.

Contributing to the Documentation

Qualified volunteers are welcome to assist in editing the wiki documentation. Contact us for more information.

Emacs tips for MongoDB work

You can edit confluence directly from emacs:

First, follow the basic instructions on http://code.google.com/p/confluence-el/

Change the confluence-url in their sample setup to http://mongodb.onconfluence.com/rpc/xmlrpc

Might also want to change the default space to DOCS or DOCS-ES or whatever space you edit the most.

etags setup (suggested by mstearn)

First, install "exuberant ctags", which has nicer features than GNU etags.

http://ctags.sourceforge.net/

Then, run something like this in the top-level mongo directory to make an emacs-style TAGS file:

```
ctags -e --extra=+qf --fields=+iasnfSKtm --c++-kinds=+p  --recurse .
```

Then you can use M-x visit-tags-table, M-., M-* as normal.

Mongo Documentation Style Guide

This page provides information for everyone adding to the Mongo documentation on Confluence. It covers:

- General Notes on Writing Style
- Guide to Confluence markup for specific situations
- Some general notes about doc production

General Notes on Writing Style

Voice

Active voice is almost always preferred to passive voice.

To make this work, however, you may find yourself anthropromorphizing components of the system - that is, treating the driver or the database as an agent that actually does something. ("The dbms writes the new record to the collection" is better than "the new record is written to the database", but some purists may argue that the dbms doesn't do anything - it's just code that directs the actions of the processor - but then someone else says "yes, but does the processor really do anything?" and so on and on.) It is simpler and more economical to write as if these components are actually doing things, although you as the infrastructure developers might have to stop and think about which component is actually performing the action you are describing.

Tense

Technical writers in general prefer to keep descriptions of processes in the present tense: "The dbms writes the new collection to disk" rather than "the dbms will write the new collection to disk." You save a few words that way.

MongoDB Terminology

It would be good to spell out precise definitions of technical words and phrases you are likely to use often, like the following:

- Mongo
- database (do you want "a Mongo database"? Or a Mongo database instance?)
- dbms (I have't seen this term often - is it correct to talk about "the Mongo DBMS"?)
- Document
- Record
- Transaction (I stopped myself from using this term because my understanding is the Mongo doesn't support "transactions" in the sense of operations that are logged and can be rolled back - is this right?)

These are just a few I noted while I was editing. More should be added. It would be good to define these terms clearly among yourselves, and then post the definitions for outsiders.

Markup for terms

It's important to be consistent in the way you treat words that refer to certain types of objects. The following table lists the types you will deal with most often, describes how they should look, and (to cut to the chase) gives you the Confluence markup that will achieve that appearance.

Type	Appearance	Markup
Object name (the type of "object" that "object-oriented programming" deals with)	monospace	{{ `term` }}
short code fragment inline	monospace	{{ `term` }}
file path/name, extension	italic	_term_
programming command, statement or expression	monospace	{{ `term` }}
variable or "replaceable item"	monospace italic	`_term_`
Placeholders in paths, directories, or other text that would be italic anyway	angle brackets around <item>	<item>
GUI element (menus menu items, buttons	bold	*term*
First instance of a technical term	italic	_term_
tag (in HTML or XML, for example)	monospace	{{ `term` }}
Extended code sample	code block	{code} program code {code}

In specifying these, I have relied on the O'Reilly Style Guide, which is at:

http://oreilly.com/oreilly/author/stylesheet.html

This guide is a good reference for situations not covered here.

I should mention that for the names of GUI objects I followed the specification in the Microsoft Guide to Technical Publications.

Other Confluence markup

If you are editing a page using Confluence's RTF editor, you don't have to worry about markup. Even if you are editing markup directly, Confluence displays a guide on the right that shows you most of the markup you will need.

References and Links

Confluence also provides you with a nice little utility that allows you to insert a link to another Confluence page by searching for the page by title or by text and choosing it from a list. Confluence handles the linking markup. You can even use it for external URLs.

The one thing this mechanism does NOT handle is links to specific locations within a wiki page. Here is what you have to know if you want to insert these kinds of links:

- Every heading you put in a Confluence page ("h2.Title", "h3.OtherTitle", etc.) becomes an accessible "anchor" for linking.
- You can also insert an anchor anywhere else in the page by inserting "{anchor\:*anchorname}" where _anchorname* is the unique name you will use in the link.
- To insert a link to one of these anchors, you must go into wiki markup and add the anchor name preceded by a "#". Example: if the page `MyPage` contains a heading or an ad-hoc anchor named `GoHere`, the link to that anchor from within the same page would look like `[#GoHere]`, and a link to that anchor from a different page would look like `[MyPage#GoHere]`. (See the sidebar for information about adding other text to the body of the link.)

Special Characters

- You will often need to insert code samples that contain curly braces. As Dwight has pointed out, Confluence gets confused by this unless you "escape" them by preceding them with a backslash, thusly:

  ```
  \{  \}
  ```

  You must do the same for "[", "]", "_" and some others.
  Within a {code} block you don't have to worry about this. If you are inserting code fragments inline using {{ and }}, however, you still need to escape these characters. Further notes about this:

  - If you are enclosing a complex code expression with {{ and }}, do NOT leave a space between the last character of the expression and the }}. This confuses Confluence.
  - Confluence also gets confused (at least sometimes) if you use {{ and }}, to enclose a code sample that includes escaped curly brackets.

About MongoDB's Confluence wiki

Confluence has this idea of "spaces". Each person has a private space, and there are also group spaces as well.

The MongoDB Confluence wiki has three group spaces defined currently:

- Mongo Documentation - The publicly accessible area for most Mongo documentation
- Contributor - Looks like, the publicly accessible space for information for "Contributors"
- Private - a space open to MongoDB developers, but not to the public at large.
  As I said in my email on Friday, all of the (relevant) info from the old wiki now lives in the "Mongo Documentation"

Standard elements of Wiki pages

You shouldn't have to spend a lot of time worrying about this kind of thing, but I do have just a few suggestions:

- Since these wiki pages are (or can be) arranged hierarchically, you may have "landing pages" that do little more than list their child pages. I think Confluence actually adds a list of children automatically, but it only goes down to the next hierarchical level. To insert a hierarchical list of a page's children, all you have to do is insert the following Confluence "macro":

  ```
  {children:all=true}
  ```

  See the Confluence documentation for more options and switches for this macro.

- For pages with actual text, I tried to follow these guidelines:
  - For top-level headings, I used "h2" not "h1"
  - I never began a page with a heading. I figured the title of the page served as one.
  - I always tried to include a "See Also" section that listed links to other Mongo docs.
  - I usually tried to include a link to the "Talk to us about Mongo" page.

Community

- Technical Support
- Bug/Feature Tracker (Jira)
- Blog
- Mailing List
- Events
- Job Board
- Twitter etc.
- Store
- Resources for Driver and Database Developers
  - Source
  - Developer List
  - Project Ideas
- Contribute!
  - Write a book
  - Write a driver, framework, and other tools
  - Help with Free Support
  - Work on the DB

Technical Support

See the Support page.

Bug/Feature Tracker (Jira)

File, track, and vote on bugs and feature requests. There is issue tracking for MongoDB and all supported drivers.

Blog

http://blog.mongodb.org/

Mailing List

http://groups.google.com/group/mongodb-announce - for release announcement and important bug fixes.

Events

The events page includes information about MongoDB conferences, webcasts, users groups, local meetups, and open office hours.

Job Board

- Click Here to access the Job Board. The Board is a community resource for all employers to post MongoDB-related jobs. Please feel free to post/investigate positions!
- See also the Indeed MongoDB jobs list

Twitter etc.

- @mongodb
- facebook
- linkedin

Store

- Visit the MongoDB store on Cafepress.

Resources for Driver and Database Developers

Source

The source code for the database and drivers is available at the http://github.com/mongodb.

Developer List

This mongodb-dev mailing list is for people developing drivers and tools, or who are contributing to the MongoDB codebase itself.

Project Ideas

Start or contribute to a MongoDB-related project.

Contribute!

Write a book

If interested contact info@10gen.com we'll try to get you in touch with publishers.

Write a driver, framework, and other tools

Writing Drivers and Tools

Help with Free Support

Jump in with answers on http://groups.google.com/group/mongodb-user and IRC (freenode.net#mongodb) .

Work on the DB

http://groups.google.com/group/mongodb-dev

Technical Support

- Free Support Forum - http://groups.google.com/group/mongodb-user

- IRC Chat and Support - irc://irc.freenode.net/#mongodb

- Commercial Support

MongoDB Commercial Services Providers

10gen is the initiator, contributor and continual sponsor of the MongoDB project. 10gen offers subscriptions that include production support, commercial licensing, and MongoDB Monitoring Service. In addition, 10gen offers advisory consulting and training.

Ready to learn more about 10gen? Fill out our contact form, and a 10gen representative will get in touch to learn about how 10gen services can help your organization.	**Need support right away?** If you are having a production issue and need support, please contact us, call us at (866) 237-8815, or visit the community Technical Support page.	**Want to get listed here?** If you provide consultative or support services for MongoDB and wish to be listed here, just let us know.

Hosting and Cloud

See the MongoDB Hosting Center.

Official 10gen Partners

10gen provides training to its systems integrator partners. To learn more about the partner program and what 10gen provides, or for information about joining the program, visit 10gen.com.

Company	More Information
CIGNEX DATAMATICS Making Open Source Work™	CIGNEX Datamatics (a subsidiary of Datamatics Global Services Ltd.) is the global leader in Commercial Open Source enterprise solutions and a global partner of 10gen (MongoDB) offering advisory consulting, implementation, and other services around the MongoDB application.
comSysto sense and respond	comSysto is a Munich based software company specialized in lean business and technology development. While supporting all three steps of a well known Build-Measure-Learn lean feedback loop, comSysto focuses on open source frameworks and software as major enablers of short, agile Build-Measure-Learn iterations and fast gains in validated learning. Powerful MongoDB technology provides the needed flexibility and agility for turning ideas into products as well as performance for handling Big Data while turning data into knowledge. We also enjoy developing with Spring framework and its sub-projects, Apache Wicket, Gradle, Git, Oracle DB and Oracle BI. comSysto is dedicated to eliminating waste in both business and technology since 2005.
codecentric	codecentric AG specialises in developing customised IT-solutions. The company is one of the leading German providers in the areas of agility, architecture, Java performance, Java and Enterprise Content Management. codecentric offers development, IT-consulting and services throughout the complete life cycle of applications and infrastructures: from individual software solutions to performance optimisation of Java-applications, to support of organisational processes within the company. The more than 120 codecentric employees work at various locations in Germany and Europe.
<epam>	EPAM's core competencies include complex software product engineering for leading global software and technology vendors as well as development, testing, maintenance, and support of mission critical business applications and vertically oriented IT consulting services for global Fortune 2000 corporations.

	Since its inception in 2005, Equal Experts has been providing bespoke application development services to blue chip clients, including major investment banks and leading Telecom companies. Equal Experts has defined a unique approach to Agile software development that is both pragmatic and Enterprise-friendly. By harnessing the potential of modern application development techniques, our goal is to radically improve our customer's efficiency.
	iTechArt Group Since 2003 iTechArt Group, a leading consulting firm, has been providing R&D Services for emerging technology companies with heavy focus on web and mobile development. iTechArt offers a traditional IT outsourcing model complete with project-based services giving our clients the ability to build and manage their own dedicated team in one of our development centers in Eastern Europe. iTechArt specializes in product research and conceptualization, helping our clients select the correct architecture design & platform, implementation & customization, testing & QA, product installation, migration & porting, as well as modernization & product extension.
	NewWave Telecom & Technologies Inc. is SEI CMMI® Maturity Level 2 appraised; a GSA Schedule 70 IT Service Provider and 8(a) STARS II holder, service Information Technology and Business Services firm who is a leading implementer of business solutions using service-oriented technologies. We specialize in providing end-to-end, mission critical information technology solutions for healthcare, finance, telecoms, government agencies and private sector clients.
	OpenCredo is a team of highly experienced consultants who are amongst the best in the business. Dedicated to maximizing the ability to deliver value from any enterprise software development and delivery capability. They help their clients use leading-edge technologies and practices to drive waste from their software delivery projects.
	PalominoDB is a boutique consultancy, known for being top MySQL experts, and specializing in open source DB technologies including MongoDB, MySQL, Postgres, Cassandra and HBase.
	Thermopylae Sciences + Technology (TST) is a Service-Disabled Veteran-Owned Small Business (SDVOSB) that focuses on excellence in all we do for our government and private sector customer base. We particularly excel in software development, cyber security, geospatial engineering, cloud/grid computing in cleared environments, and mobile application development.
	Thumbtack is a full service application development partner specializing in highly scalable and cloud-based solutions. We provide strategic advice, technical architecture, software development, as well as quality assurance and deployment support to companies in Media, Publishing, and Financial Services. With a core competency in NoSQL, Thumbtack has brought robust MongoDB solutions to production for a variety of clients and industries.
	Rothbury Software is a very experienced and successful boutique software consulting company. Our impressive customer list says more about us than we can state here.

	Xebia is an international IT consultancy and project organization focused on Enterprise Java technology, Agile development methods and outsourcing services. Xebia consists of over 250 professionals, all committed to be the best in their field of expertise. Passion for in depth technology, in combination with Lean, Agile and Scrum practices are Xebia's driving factors and competitive edge.
	FastConnect FastConnect is a french consulting and engineering company specialised in distributed architectures. Our expertise spans the domains of SOA and EDA, Process and Decision Management, Cloud, Big Data and Analytics for business-critical applications. We bring to our clients our significant and strategic investments in specific technologies and development methodologies. Our consultants are helping our clients in designing and delivering flexible and linearly performing applications at the best price/performance ratio using state of the art architecture and enterprise-ready tools. For more information visit http://www.fastconnect.fr/.
	Silpion IT-Solutions GmbH is an expanding consultancy in Germany. Founded in 2000, today 97 highly qualified employees work for us. Of these, 57 are employed directly and 40 as freelancers. Our focus is safety and functionality, while we focus on innovative concepts and advanced technologies.
	Orange11 is a leading full service supplier of high-quality custom-built applications. Our specialist teams provide end-to-end project delivery services to support the full project life-cycle: from the first design stages to ongoing maintenance. In addition, Orange11 can provide high-end consulting and training services.

Other Resources

- Hashrocket is a full-service design and development firm that builds successful web businesses. Hashrocket continually creates and follows best practices and surround themselves with passionate and talented craftsmen to ensure the best results for you and your business.
- LightCube Solutions provides PHP development and consulting services, as well as a lightweight PHP framework designed for MongoDB called 'photon'.
- Squeejee builds web applications on top of MongoDB with multiple sites already in production.

Visit the 10gen Offices

Bay Area

10gen's West Coast office is located in downtown Palo Alto.

555 University Ave.
Palo Alto, CA 94301

View Larger Map

Directions:

From 101:

- Take 101 to the University Ave Exit.

- Drive West to downtown Palo Alto. The 10gen office is at the northeast corner of the intersection of Tasso St. and University Ave.

From 280:

- Driving southbound, take 280 to the Sand Hill Road exit. Driving northbound, take 280 to the Alpine Road exit; make a right onto Sand Hill Road from Alpine Road.

- Take Sand Hill Road east to the end at El Camino Real
- Make a right onto El Camino Real, then veer right towards University Ave.
- Make a left onto University Ave. The 10gen office is at the northeast corner of the intersection of Tasso St. and University Ave.

New York City

10gen's East Coast office is located in the SOHO neighborhood of NYC.

578 Broadway
7th Floor
New York, NY 10012

View Larger Map

User Feedback

"I just have to get my head around that mongodb is really _this_ good"
-muckster, #mongodb

"Guys at Redmond should get a long course from you about what is the software development and support "
-kunthar@gmail.com, mongodb-user list

"#mongoDB keep me up all night. I think I have found the 'perfect' storage for my app "
-elpargo, Twitter

"Dude, you guys are legends!"
-Stii, mongodb-user list

"Times I've been wowed using MongoDB this week: 7."
-tpitale, Twitter

Community Blog Posts

B is for Billion
-Wordnik (July 9, 2010)

[Reflections on MongoDB]
-Brandon Keepers, Collective Idea (June 15, 2010)

Building a Better Submission Form
-New York Times Open Blog (May 25, 2010)

Notes from a Production MongoDB Deployment
-Boxed Ice (February 28, 2010)

NoSQL in the Real World
-CNET (February 10, 2010)

Why I Think Mongo is to Databases what Rails was to Frameworks
-John Nunemaker, Ordered List (December 18, 2009)

MongoDB a Light in the Darkness...
-EngineYard (September 24, 2009)

Introducing MongoDB
-Linux Magazine (September 21, 2009)

Choosing a non-relational database; why we migrated from MySQL to MongoDB
-Boxed Ice (July 7, 2010)

The Other Blog - The Holy Grail of the Funky Data Model
-Tom Smith (June 6, 2009)

GIS Solved - Populating a MongoDb with POIs
-Samuel

Community Presentations

Scalable Event Analytics with MongoDb and Ruby on Rails
Jared Rosoff at RubyConfChina (June 2010)

How Python, TurboGears, and MongoDB are Transforming SourceForge.net
Rick Copeland at PyCon 2010

MongoDB
Adrian Madrid at Mountain West Ruby Conference 2009, video

MongoDB - Ruby friendly document storage that doesn't rhyme with ouch
Wynn Netherland at Dallas.rb Ruby Group, slides

MongoDB
jnunemaker at Grand Rapids RUG, slides

Developing Joomla! 1.5 Extensions, Explained (slide 37)
Mitch Pirtle at Joomla!Day New England 2009, slides

Drop Acid (slide 31) (video)
Bob Ippolito at Pycon 2009

Python and Non-SQL Databases (in French, slide 21)
Benoit Chesneau at Pycon France 2009, slides

Massimiliano Dessì at the Spring Framework Italian User Group

- MongoDB (in Italian)
- MongoDB and Scala (in Italian)

Presentations and Screencasts at Learnivore
Frequently-updated set of presentations and screencasts on MongoDB.

Benchmarking

We keep track of user benchmarks on the Benchmarks page.

Job Board

Redirecting...

Redirection Notice
This page should redirect to http://jobs.mongodb.org/ in about 2 seconds.

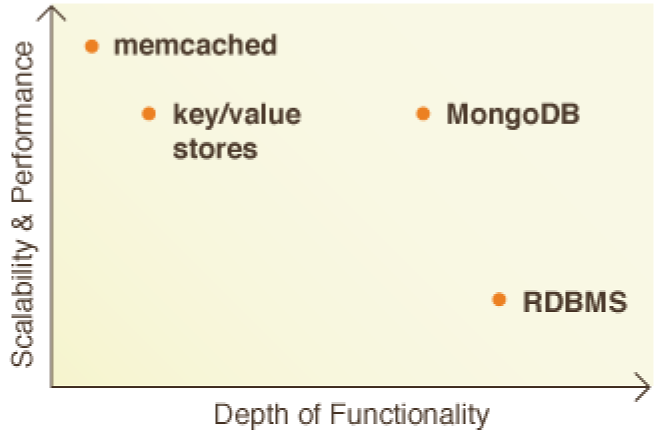
About

- Philosophy
- Use Cases
- Mongo-Based Applications
- Events
- Articles
- Benchmarks
- FAQ
- Misc
- Licensing

Philosophy

Design Philosophy

- New database technologies are needed to facilitate horizontal scaling of the data layer, easier development, and the ability to store order(s) of magnitude more data than was used in the past.
- A non-relational approach is the best path to database solutions which scale horizontally to many machines.
- It is unacceptable if these new technologies make writing applications harder. Writing code should be faster, easier, and more agile.
- The document data model (JSON/BSON) is easy to code to, easy to manage(schemaless), and yields excellent performance by grouping relevant data together internally.
- It is important to keep deep functionality to keep programming fast and simple. While some things must be left out, keep as much as possible – for example secondaries indexes, unique key constraints, atomic operations, multi-document updates.



- Database technology should run anywhere, being available both for running on your own servers or VMs, and also as a cloud pay-for-what-you-use service.

Focus

MongoDB focuses on four main things: flexibility, power, speed, and ease of use. To that end, it sometimes sacrifices things like fine grained control and tuning, overly powerful functionality like MVCC that require a lot of complicated code and logic in the application layer, and certain ACID features like multi-document transactions.

Flexibility

MongoDB stores data in JSON documents (which we serialize to BSON). JSON provides us a rich data model that seamlessly maps to native programming language types, and since its schema-less, makes it much easier to evolve your data model than with a system with enforced schemas such as a RDBMS.

Power

MongoDB provides a lot of the features of a traditional RDBMS such as secondary indexes, dynamic queries, sorting, rich updates, upserts (update if document exists, insert if it doesn't), and easy aggregation. This gives you the breadth of functionality that you are used to from an RDBMS, with the flexibility and scaling capability that the non-relational model allows.

Speed/Scaling

By keeping related data together in documents, queries can be much faster than in a relational database where related data is separated into multiple tables and then needs to be joined later. MongoDB also makes it easy to scale out your database. Autosharding allows you to scale your cluster linearly by adding more machines. It is possible to increase capacity without any downtime, which is very important on the web when load can increase suddenly and bringing down the website for extended maintenance can cost your business large amounts of revenue.

Ease of use

MongoDB works hard to be very easy to install, configure, maintain, and use. To this end, MongoDB provides few configuration options, and instead tries to automatically do the "right thing" whenever possible. This means that MongoDB works right out of the box, and you can dive right into developing your application, instead of spending a lot of time fine-tuning obscure database configurations.

See also:

- Introduction

Use Cases

See also the Production Deployments page for a discussion of how companies like Craigslist, Shutterfly, foursquare, bit.ly, SourceForge, etc. use MongoDB.

Well Suited

- Archiving and event logging
- Document and Content Management Systems - as a document-oriented (JSON) database, MongoDB's flexible schemas are a good fit

for this.
- ECommerce. Several sites are using MongoDB as the core of their ecommerce infrastructure (often in combination with an RDBMS for the final order processing and accounting).
- Gaming. High performance small read/writes are a good fit for MongoDB; also for certain games geospatial indexes can be helpful.
- High volume problems. Problems where a traditional DBMS might be too expensive for the data in question. In many cases developers would traditionally write custom code to a filesystem instead using flat files or other methodologies.
- Mobile. Specifically, the server-side infrastructure of mobile systems. Geospatial key here.
- Operational data store of a web site MongoDB is very good at real-time inserts, updates, and queries. Scalability and replication are provided which are necessary functions for large web sites' real-time data stores. Specific web use case examples:
    - content management
    - comment storage, management, voting
    - user registration, profile, session data
- Projects using iterative/agile development methodologies. Mongo's BSON data format makes it very easy to store and retrieve data in a document-style / "schemaless" format. Addition of new properties to existing objects is easy and does not generally require blocking "ALTER TABLE" style operations.
- Real-time stats/analytics

Less Well Suited

- Systems with a heavy emphasis on complex transactions such as banking systems and accounting. These systems typically require multi-object transactions, which MongoDB doesn't support. It's worth noting that, unlike many "NoSQL" solutions, MongoDB does support atomic operations on single documents. As documents can be rich entities; for many use cases, this is sufficient.
- Traditional Non-Realtime Data Warehousing. Traditional relational data warehouses and variants (columnar relational) are well suited for certain business intelligence problems – especially if you need SQL (see below) to use client tools (e.g. MicroStrategy) with the database. For cases where the analytics are realtime, the data very complicated to model in relational, or where the data volume is huge, MongoDB may be a fit.
- Problems requiring SQL.

Use Case Articles

> *Using MongoDB for Real-time Analytics*
> *Using MongoDB for Logging*
> *MongoDB and E-Commerce*
> *Archiving*

Use Case Videos

> *Analytics*
> *Content Management*
> *Ecommerce*
> *Finance*
> *Gaming*
> *Government*
> *Media*

How MongoDB is Used in Media and Publishing

We see growing usage of MongoDB in both traditional and new media organizations. In these areas, the challenges for application developers include effectively managing rich content (including user-generated content) at scale, deriving insight into how content is consumed and shared in real-time, weaving personalization and social features into their applications and delivering content to a wide variety of browsers and devices.

From a data storage perspective, there is a need for databases that make it easy to rapidly develop and deploy interactive, content-rich web and mobile application, while cost-effectively meeting performance and scale requirements. Specifically, MongoDB is good fit for application across the media and publishing world for the following reasons:

- The document model is natural fit content-rich data
- Schema-free JSON-like data structures make it easy to import, store, query, and deliver structured and semi-structured content
- High-performance and horizontal scalability for both read and write intensive applications

MongoDB for Content management

MongoDB’s document model makes it easy to model content and associated metadata in flexible ways. While the relational model encourages dividing up related data into multiple tables, the document model (with its support for data structures such as arrays and embedded documents) encourages grouping related pieces of data within a single document. This leads to a data representation that is both efficient and closely matches objects in your application
As an example, the following document represents how a blog post (including its tags and comments) can be modeled with MongoDB:

```
{
_id:
author: “Nosh”
title: “Why MongoDB is good for CMS”
date:11/29/2010
body: “MongoDB’s document model makes it easy to model....”
tags: [“media”, “CMS”, “web”, “use case”]
comments: [{author:bob, date:1/1/2011, comment:"I agree"},{..}, {..}]
}
```

Modeling content elements with these patterns also simplifies queries. For example, we can retrieve all blog posts by the author ‘nosh’ which have the tag mongodb with the query,

```
find({author:”nosh”, tags:”mongodb”)
```

Flexible document-based representation, efficient and simple queries and scalability makes MongoDB a well suited as a datastore for content management systems. The Business Insider has built their content management system from the ground up using MongoDB and PHP, which serves over 2 million visits/month. For sites based on Drupal, Drupal 7 now makes it easier to use MongoDB as a datastore. Examiner.com, ported their legacy CMS (based on ColdFusion and Microsoft SQLServer) to Drupal 7 and a hybrid of MySQL and MongoDB. You can read a case study about the how the examiner.com (a top 100 website, and one of most trafficked Drupal deployments) made the transition.

MongoDB can also be used to augment existing content management systems with new functionality. One area that we see MongoDB used increasingly is as a metadata store for rich media. MongoDB’s document model makes it simple to represent the attributes for an asset (e.g. author, dates, categories, versions, etc) and a pointer to the asset (e.g. on a filesystem or on S3) as document and then efficiently search or query the metadata for display. Additionally, because MongoDB is schema-free, new metadata attributes can be added without having to touch existing records. IGN uses MongoDB as a the metadata store for all videos on IGN Videos and serves up millions of video views per month. Another similar use case for MongoDB is in storing user-submitted content. The New York Times, uses MongoDB as the backend for ‘Stuffy’, their tool for allowing users and editors to collaborate on features driven by user-submitted photos. A brief overview on the tool is here.

How-to Guides

Modeling content, comments, and tags with MongoDB (coming soon)
Modelling image and video metadata with MongoDB (coming soon)
Using MongoDB with Drupal 7 (coming soon)

Roadmap tip: Watch the full-text search ticket

MongoDB for Real-time analytics

The ability to track and change content based on up-to-minute statistics is becoming increasingly important. MongoDB’s fast write performance and features such as upsert and the $inc operator, make it well suited for capturing real time analytics on how content is being consumed. This blog post outlines some basic patterns you can use to capture real-time pageview statistics for your content.

A number of companies are using MongoDB, either internally to track their own content in real-time, or are building platforms based on MongoDB to help other companies get real time statistics on their content: Chartbeat provides analytics for publishers, with live dashboards and APIs showing how users are engaging with their content in real-time. BuzzFeed uses MongoDB to help understand how and when content will go viral, while ShareThis uses MongoDB to power its API that gives publishers insight into how content is shared across the web and social media

How-to guides

Real-time analytics with MongoDB (coming soon)

MongoDB for Social Graphs & Personalization

While systems such as graph databases excel at complex graph traversal problems, MongoDB’s document structure is well suited for building certain types of social and personalization features. Most often, this involves building user profile documents that include a list of friends, either imported from external social networks or in site. For example,

```
{
_id:
user: nosh
email: nosh@10gen.com
friendIDs: [....., ….., …..]
}
```

In this case the friendIDs field is an array with a list of IDs corresponding to profiles of users that are my friends. This data can then be used to generate personalized feeds based on content that my friends have viewed or liked. IGN's social network, MY IGN, uses MongoDB to store profiles of users and generate personalized fields. Users have the ability to import their friends from facebook, 'follow' IGN authors, or follow specific game titles they are interested in. When they log in, they are presented with a personalized feed composed from this data.

How-to guides:

Storing user profiles with MongoDB (coming soon)
Importing social graphs into MongoDB (coming soon)
Generating personalized feeds with MongoDB (coming soon)

MongoDB for Mobile/Tablet Apps

Serving similar content across desktop browsers, mobile browsers, as well as mobile apps is driving developers to build standardized API layers that can be accessed by traditional web application servers, mobile client applications, as well as 3rd party applications. Typically these are RESTful APIs that serve JSON data. With MongoDB's JSON-like data format, building these APIs on top of MongoDB is simplified as minimal code is necessary to translate MongoDB documents and query to JSON representation. Additionally, features such as in-built two-dimensional geospatial indexing allow developers to easily incorporate location-based functionality into their applications.

MongoDB for Data-driven journalism

One of the strengths of MongoDB is dealing with semi-structured data. Data sources such as those produced by governments and other organizations are often denormalized and distributed in formats like CSV files. MongoDB, with its schema-free JSON-like documents is an ideal store for processing and storing these sparse datasets.

The Chicago Tribune uses MongoDB in its Illinois School Report Cards application, which is generated from a nearly 9,000 column denormalized database dump produced annually by the State Board of Education. The application allows readers to search by school name, city, county, or district and to view demographic, economic, and performance data for both schools and districts.

How-to guides:

Importing and Exporting data from MongoDB (coming soon)
Reporting and Visualization with MongoDB (coming soon)

Presentations

- How MTV Networks leverages MongoDB for CMS - MongoNYC Presentation (June 2011)
- Schema Design for Content Management: eHow on MongoDB - MongoSF Presentation (May 2011)
- More Presentations

Use Case - Session Objects

MongoDB is a good tool for storing HTTP session objects.

One implementation model is to have a sessions collection, and store the session object's _id value in a browser cookie.

With its update-in-place design and general optimization to make updates fast, the database is efficient at receiving an update to the session object on every single app server page view.

Aging Out Old Sessions

The best way to age out old sessions is to use the auto-LRU facility of capped collections. The one complication is that objects in capped collections may not grow beyond their initial allocation size. To handle this, we can "pre-pad" the objects to some maximum size on initial addition, and then on further updates we are fine if we do not go above the limit. The following mongo shell javascript example demonstrates padding.

(Note: a clean padding mechanism should be added to the db so the steps below are not necessary.)

```
> db.createCollection('sessions', { capped: true, size : 1000000 } )
{"ok" : 1}
> p = "";
> for( x = 0; x < 100; x++ ) p += 'x';
> s1 = { info: 'example', _padding : p };
{"info" : "example" , "_padding" :
"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
}
> db.sessions.save(s1)
> s1
{"info" : "example" , "_padding" :
"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
, "_id" :  ObjectId( "4aafb74a5761d147677233b0") }

> // when updating later
> s1 = db.sessions.find( { _id : ObjectId( "4aafb74a5761d147677233b0") } )
{"_id" :  ObjectId( "4aafb74a5761d147677233b0")  , "info" : "example" , "_padding" :
"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
}
> delete s._padding;
true
>  s.x = 3; // add a new field
3
> db.sessions.save(s);
> s
{"_id" :  ObjectId( "4aafb5a25761d147677233af")  , "info" : "example" , "x" : 3}
```

Mongo-Based Applications

Please list applications that leverage MongoDB here. If you're using MongoDB for your application, we'd love to list you here! Email meghan@10gen.com.

See Also

- Production Deployments - Companies and Sites using MongoDB
- Hosting Center

Applications Using MongoDB

c5t

Content-management using TurboGears and Mongo

Calipso

Content management system built using NodeJS and MongoDB

Cube

Cube is an open-source system for visualizing time series data, built on MongoDB, Node and D3.

ErrorApp

ErrorApp tracks errors from your apps. It reports them to you and gathers all information and make reports available to you.

Graylog2

Graylog2 is an open source syslog server implementation that stores logs in MongoDB and provides a Rails frontend.

HarmonyApp

Harmony is a powerful web-based platform for creating and managing websites. It helps connect developers with content editors, for unprecedented flexibility and simplicity. For more information, view Steve Smith's presentation on Harmony at MongoSF (April 2010).

Hummingbird

Hummingbird is a real-time web traffic visualization tool developed by Gilt Groupe

Mogade

Mogade offers a free and simple to use leaderboard and achievement services for mobile game developers.

MongoLantern

MongoLantern is an open source full text search server using MongoDB as index storage, which allows MongoLantern to migrate any changes very easily into account using MongoDB API. It's written originally written in PHP can be migrated to any desired language as required using it's future APIs.

MongoPress

A flexible CMS that uses MongoDB and PHP.

Mongs

A simple, web-based data browser for MongoDB.

Mongeez

Mongeez is an opensource solution allowing you to manage your mongo document changes in a manner that is easy to synchronize with your code changes. Check out mongeez.org.

NewsBlur

NewsBlur is an open source visual feed reader that powers http://newsblur.com. NewsBlur is built with Django, MongoDB, Postgres and RabbitMQ

phpMyEngine

A free, open source CMS licensed under the GPL v.3. For storage, the default database is MongoDB.

Quantum GIS

Plugin for Quantum GIS that lets you plot geographical data stored in MongoDB

Scribe

Open source image transcription tool

Shapado

Free and open source Q&A software, open source stackoverflow style app written in ruby, rails, mongomapper and mongodb.

Websko

Websko is a content management system designed for individual Web developers and cooperative teams.

Events

There are MongoDB User Groups (MUGs) all over the world. Please check out the full listing.

Upcoming Meetups and Conferences

Date	Event	What/Where
May 9-10, 2012	**Progressive NoSQL Tutorials** Document Databases: Scaling with MongoDB Chris Harris, 10gen	London, UK
May 9, 2012	**NoSQL in the Real World** Jeff Yemin MongoDB from the Vendor's Perspective	Cambridge, MA
May 15, 2012	**NoSQL and Big Data: Scaling the Enterprise in the New Age** Max Schireson, President, 10gen MIT/Stanford Venture Lab at Stanford Graduate School of Business	Stanford, CA
May 15, 2012	**MongoDB Pittsburgh** MongoDB Pittsburgh is a free evening event dedicated to the open source, non-relational database, MongoDB	Pittsburgh, PA
May 18-20, 2012	**NoSQL data modeling with Mongo and Neo4j** Peter Bell No Fluff Just Stuff	Dallas, TX
May 19-20	**Stanford Datafest** Look out for Paul Pederson's Demo of MongoDB	Stanford, CA
May 21, 2012	**Integrating Hadoop with MongoDB** Brendan McAdams, 10gen NoSQL Netherlands	Amsterdam, NL
May 23, 2012	**MongoNYC** MongoNYC is a one-day conference in New York City dedicated to development with the open source, non-relational database MongoDB.	New York, NY
May 23-24 2012	**Glue Conference** Max Schireson Re-Inventing The Database	Broomfield, CO
May 24, 2012	**Operating MongoDB** Michael Fiedler, 10gen NYC DevOps User Group	New York, NY
May 29-30, 2012	**NoSQL Matters** MongoDB Sharding - Muharem Hrnjadovic Building Hybrid Applications with MongoDB, RDBMS & Hadoop - Chris Harris, 10gen	Kölon, DE
June 2-6, 2012	**International PHP Conference** Profiling PHP Applications Advanced Date/Time Handling Derick Rethans 10gen	Berlin, DE
June 6-9, 2012	**Symfony Live Paris 2012** Using MongoDB Responsibly Jeremy Mikola	Paris, France
June 6-8, 2012	**Hybrid Applications with MongoDB and RDBMS** Norweigan Developers Conference Chris Harris	Oslo, Norway
June 9, 2012	**PASS SQL Saturday** Justin Dearing, MongoDB Master	Philadelphia, PA
June 13-14, 2012	**Hadoop Summit** Hadoop Plugin for MongoDB: The Elephant in the Room Brendan McAdams, 10gen	San Jose, CA
June 14, 2012	**MongoDB Paris** MongoDB Paris is a one-day conference in Paris, dedicated to development with the open source, non-relational database MongoDB.	Paris, France
June 20, 2012	**MongoDB UK 2012** MongoDB UK is a one-day conference in London, dedicated to development with the open source, non-relational database MongoDB.	London, United Kingdom

June 26, 2012	**MongoDB DC 2012** MongoDB DC is a one-day conference in Washington, DC, dedicated to development with the open source, non-relational database MongoDB.	Washington, DC
July 13, 2012	**MongoDB São Paulo 2012** MongoDB DC is a one-day conference in São Paulo, Brazil, dedicated to development with the open source, non-relational database MongoDB.	São Paulo, Brazil
July 16-20, 2012	**OSCON** Steve Francia, 10gen Hybrid Application with MongoDB and RDBMS Meghan Gill, 10gen Scaling Your Community By Nurturing Leaders Visit the MongoDB Booth	Portland, Oregon
July 25, 2012	**MongoDB San Diego** is a Little MongoDB Day - a free evening meetup dedicated to the open source, non-relational database MongoDB. Come join us for talks on MongoDB, food and drinks!	San Diego, CA
August 20-24	**DrupalCon Munich** Stop by the MongoDB Booth	München, DE
Sept 24	**MongoDB Labs at DataWeek**	San Francisco, CA

Webinars

Webinar Date	Sessions	Topic	Description
May 10, 2012	1:00 PM EST 10:00 AM PST	**MongoDB's New Aggregation Framework**	We're working on a new aggregation framework for MongoDB that will introduce a new aggregation system that will make it a lot easier to do simple tasks like counting, averaging, and finding minima or maxima while grouping by keys in a collection. The new aggregation features are not a replacement for map-reduce, but will make it possible to do a number of things much more easily, without having to resort to the big hammer that is map-reduce. After introducing the syntax and usage patterns for the new aggregation system, we will give some demonstrations of aggregation using the new system.
May 17, 2012	1:00 PM EST 10:00 AM PST	**MongoDB for Content Management**	MongoDB's flexible schema makes it a great fit for your next content management application. MongoDB's data model makes it easy to catalog multiple content types with diverse meta data. In this session, we'll review schema design for content management, using GridFS for storing binary files, and how you can leverage MongoDB's auto-sharding to partition your content across multiple servers.
May 24, 2012	1:00 PM EST 10:00 AM PST	**Building Web Services with MongoDB, Node.JS, and Openshift**	MongoDB and Node.JS are an excellent match. Server-side javascript FTW! We will discuss how to model a basic "to do list" application in MongoDB, and after modeling the data, we will map that to an application we build together in Node.JS. Finally, we'll deploy the application to OpenShift, Red Hat's Platform as a Service. We'll be left with a a full fledged application running in the cloud!
June 7, 2012	1:00 PM EST 10:00 AM PST	**Product Data Management with MongoDB**	Customers are leveraging MongoDB to build rich product catalogs for online and offline applications. In this session, we will review flexible schema design, indexing and query language of MongoDB and how it is well suited to scalable management of diverse product catalogs and inventory in real time.
June 21, 2012	2:00 PM EST 11:00 AM PST	**MongoDB + Hadoop: Taming the Elephant in the Room**	10gen released the Hadoop plugin for MongoDB v1.0.0. In this session, Brendan will go through how to integrate MongoDB with Hadoop for large-scale distributed data processing. Using tools like MapReduce, Pig and Streaming you will learn how to do analytics and ETL on large datasets with the ability to load and save data against MongoDB. With Hadoop MapReduce, Java and Scala programmers will find a native solution for using MapReduce to process their data with MongoDB. Programmers of all kinds will find a new way to work with ETL using Pig to extract and analyze large datasets and persist the results to MongoDB. Python and Ruby Programmers can rejoice as well in a new way to write native Mongo MapReduce using the Hadoop Streaming interfaces.

MongoDB User Groups

Your Go-To Guide to Running A MongoDB User Group

North America	***MEETUPS***

Atlanta	
Boston	
Chicago	
DC	
Hawaii	**April 19** First MongoDB User Group
Huntsville	**March 7** Intro to MongoDB
Houston	*None currently scheduled. Join us to be notified!*
Los Angeles	
Madison, WI	Monday March 12 Databases at Scale
Mexico	*None currently scheduled. Join us to be notified!*
New York	
New Jersey	_None currently scheduled. Join us to be notified.
Philadelphia	*None currently scheduled. Join us to be notified!*
Phoenix, AZ	**April 24** Replication, Sharding and Multi-Datacenters Deployment
Raleigh	
San Francisco Bay Area	
Seattle	April 17 Intro To MongoDB and Schema Design Kevin Hanson, 10gen
St. Louis	*None currently scheduled. Please check back*
Toronto	
South America	***MEETUPS***
Sao Palo	
Belo Horizonte	
EUROPE	***MEETUPS***
Barcelona	**April 12** First Ever MongoDB Barcelona Meetup
Bristol	
Finland	
London	
Minsk	_none currently scheduled. Join to get updates
München	
Paris	
Thames Valley, UK	
Saint Petersburg	Join us to get updates
Sophia-Antipolis, FR	Join us to get updates
Stockholm	
Switzerland	

Middle East	MEETUPS
Israel	**March 7, 2012** MongoDB-IL
Asia and Pacific	***MEETUPS***
*Hanoi, Vietnam	_No meetups currently scheduled. Like the facebook page to receive updates!
Melbourne	
Kuala Lumpur	**May 2** Learn about setting-up MongoDB on Linode from Gareth and searching MongoDB with richer context from Tristan

If you're interested in having someone present MongoDB at your conference or meetup, or if you would like to list your MongoDB event on this page, contact meghan at 10gen dot com. Want some MongoDB stickers to give out at your talk? Complete the Swag Request Form.

Training

MongoDB training from 10gen is the best way to get you and your team up to speed quickly. Our MongoDB Developer Training and MongoDB Administrator Training are comprehensive, two day courses.

Contact us at training@10gen.com if you would like to buy training credits for multiple students or classes, schedule an on-site training, or discuss a custom training option.

MongoDB for Administrators	New York, NY	May 15-16, 2012	Register
MongoNYC Training Workshops	New York, NY	May 22, 2012	Register
Xebia Presents MongoDB for Administrators	Paris, FR	June 12-13, 2012	Register
MongoDB Paris Training Workshops	Paris, FR	June 13, 2012	Register
MongoDB for Developers	New York, NY	June 19-20, 2012	Register
MongoDB UK Training Workshops	London, England	June 19, 2012	Register
MongoDC Training Workshops	Washington, DC	June 25, 2012	Register
Trainologic presents MongoDB for Developers	Tel Aviv, Israel	June 25-26	Register
Xebia Presents MongoDB for Developers	Amsterdam, NL	June 26-27, 2012	Register
Trainologic presents MongoDB for Administrators	Tel Aviv, Israel	June 27-28	Register
MongoDB Essentials	London, UK	July 03-05, 2012	Register
MongoDB Essentials	Palo Alto, CA	July 17-19, 2012	Register
MongoDB Essentials	Sydney, Australia	July 25-27, 2012	Register
MongoDB Essentials	New York, NY	July 31- August 2, 2012	Register
MongoDB for Developers	Palo Alto, CA	August 7-8, 2012	Register
MongoDB for Administrators	New York, NY	August 21-22, 2012	Register
MongoDB Essentials	Palo Alto, CA	September 11-13, 2012	Register
MongoDB Essentials	New York, NY	September 25-27, 2012	Register
MongoDB for Administrators	London, UK	September 27-28, 2012	Register
Xebia Presents MongoDB for Developers	Amsterdam, NL	October 2-3, 2012	Register
Xebia Presents MongoDB for Developers	Paris, FR	October 04-05, 2012	Register
MongoDB for Administrators	Palo Alto, CA	October 9-10, 2012	Register
MongoDB for Developers	New York, NY	October 23-24, 2012	Register
MongoDB Essentials	New York, NY	November 6-8, 2012	Register
MongoDB Essentials	New York, NY	November 27-29, 2012	Register
MongoDB for Developers	London, UK	November 28-29, 2012	Register

Xebia Presents MongoDB for Administrators	Paris, FR	December 06-07, 2012	Register
MongoDB for Developers	Palo Alto, CA	December 11-12, 2012	Register
MongoDB for Administrators	New York, NY	December 18-19, 2012	Register
Xebia Presents MongoDB for Developers	Amsterdam, NL	December 18-19, 2012	Register

"Office Hours"

Atlanta Office Hours will be cancelled the week of April 22nd, 2012

City	Date	Time	Location	Look For
Atlanta, GA		4-6pm	Please check the Atlanta MongoDB User Group page for upcoming office hours	Look for a MongoDB logo!
New York, NY	Wednesdays	4-6:30pm	10gen Headquarters, 578 Broadway, 7th Floor	10gen holds weekly open "office hours" with whiteboarding and hack sessions at 10gen headquarters.
Palo Alto, CA	Thursdays	4-6pm	10gen CA office, 555 University Avenue, Palo Alto, CA 94301	Have questions about MongoDB? Visit the 10gen office in Palo Alto to speak directly with the MongoDB engineers (or just come say hi!).
San Francisco, CA	Every other Monday	5-7pm	Epicenter Cafe, 764 Harrison St, Between 4th St & Lapu St, San Francisco, CA	Stop by the Epicenter Cafe in San Francisco to meet 10gen Software Engineers Sridhar Nanjundeswaran. Ask questions, hack, have some coffee. Look for a laptop with a "Powered by MongoDB" sticker. **Click here for more info and signup.**
London, UK	Every other Wednesday	5-7pm	Look Mum No Hands!, 49 Old Street London EC1V 9HX	Stop by Look Mum No Hands! near Old Street & Barbican to meet 10gen Engineers Ross Lawley, Chris Harris and Dan Roberts. Ask questions, hack, have some coffee. Look for a laptop with a MongoDB leaf sticker. **Click here for more info and signup.**

See Also

- http://www.10gen.com/presentations
- http://www.10gen.com/events
- http://lanyrd.com/topics/mongodb/

MongoDB User Groups (MUGs)

MongoDB User Groups (MUGs) are a great way for the community to learn from one another about MongoDB best practices, to network, and to have fun. Interested in starting a user group in your city? Submit a proposal and check out our guide to starting a MongoDB User Group to get started!

MongoDB User Groups

North America

- Atlanta, GA
- Boston, MA
- Chicago, IL
- Los Angeles, CA
- Hawaii
- Mexico City, Mexico
- New Jersey
- New York, NY
- Philadelphia, PA
- Raleigh, NC
- San Diego
- San Francisco, CA
- Seattle, WA
- St. Louis, MO
- Toronto, Ontario
- Washington, DC

South America

- Belo Horizonte, Brazil
- Sao Paulo, Brazil

Europe

- Barcelona, ESP
- Berlin, DE
- Bristol, UK
- Sofia, Bulgaria
- Dublin
- Düsseldorf
- Helsinki, Finland
- Paris, France
- London, UK
- Mallorca, ESP
- Minsk, Belarus
- München, Germany
- Sophia-Antipolis, France
- St. Petersburg
- Stockholm, Sweden
- Thames Valley, UK
- Zurich, Switzerland

Middle East

- Israel

Africa

- Cape Town

Asia

- China
- Hanoi
- Tokyo, Japan
- Thailand
- Turkey
- Kuala Lumpur, Malaysia
- South Korea

Australia

- Melbourne

Video & Slides from Recent Events and Presentations

Table of Contents:

MongoDB Conferences

One-day conferences hosted by 10gen. 10gen develops and supports MongoDB.

MongoUK Video (June 2010)
MongoFR Video (June 2010)
MongoNYC (May 2010) and MongoSF (April 2010) Video
MongoSF (April 2010) Slides & Video

Ruby/Rails

Practical Ruby Projects with MongoDB
Alex Sharp, OptimisCorp
Ruby Midwest - June 2010

Scalable Event Analytics with MongoDB and Ruby

Jared Rosoff, Yottaa
RubyConfChina - June 26, 2010

The MongoDB Metamorphosis (Kyle Banker, 10gen)
Million Dollar Mongo (Obie Fernandez & Durran Jordan, Hashrocket)
Analyze This! (Blythe Dunham)
RailsConf
Baltimore, MD
June 7-10

MongoDB
Seth Edwards
London Ruby Users Group
London, UK
Wednesday April 14
Video & Slides

MongoDB: The Way and its Power
Kyle Banker, Software Engineer, 10gen
RubyNation
Friday April 9 & Saturday April 10
Reston, VA
Slides | Video

MongoDB Rules
Kyle Banker, Software Engineer, 10gen
Mountain West Ruby Conference
Salt Lake City, UT
Thursday March 11 & Friday March 12
Slides

MongoDB Isn't Water
Kyle Banker, Software Engineer, 10gen
Chicago Ruby
February 2, 2010
Video | Slides | Photos

Introduction to Mongo DB
Joon Yu, RubyHead
teachmetocode.com
Nov-Dec, 2009
Screencasts

Python

How Python, TurboGears, and MongoDB are Transforming SourceForge.net
Rick Copeland, SourceForge.net
PyCon - Atlanta, GA
February 21, 2010
Slides

Alt.NET

.NET and MongoDB - Building Applications with NoRM and MongoDB
Alex Hung
July 28, 2010

User Experiences

The Future of Content Technologies
Scaling Web Applications with NonSQL Databases: Business Insider Case Study
Ian White, Lead Developer, Business Insider
Gilbane Conference
San Francisco, CA
Thursday, May 20
Slides

Chartbeat and MongoDb - a perfect marriage
Kushal Dave, CTO, Chartbeat & Mike Dirolf, Software Engineer, 10gen
New York City Cloud Computing Meetup
New York, NY
May 18
Slides

Why MongoDB is Awesome
John Nunemaker, CTO, Ordered List
DevNation Chicago
May 15
Slides

Humongous Data at Server Density: Approaching 1 Billion Documents in MongoDB
David Mytton, Founder, Boxed Ice
Webinar
Wednesday May 5
Recording & Slides

Humongous Drupal
DrupalCon San Francisco
Karoly Negyesi, Examiner.com
Saturday April 17
Slides | Video

MongoDB: huMONGOus Data at SourceForge
Mark Ramm, Web Developer, SourceForge
QCon London
Thursday March 11
Slides

Migrating to MongoDB
Bruno Morency, DokDok
Confoo.ca
March 10 - 12
Slides

More about MongoDB

Recording of Michael Dirolf on MongoDB @ E-VAN 07 June 2010

NoSQL-Channeling the Data Explosion
Dwight Merriman, CEO, 10gen
Inside MongoDB: the Internals of an Open-Source
Mike Dirolf, Software Engineer, 10gen
Gluecon
Denver, CO
Wednesday May 26 & Thursday May 27

Schema Design with MongoDB
Kyle Banker, Software Engineer, 10gen
Webinar
Tuesday April 27
Recording and Slides

Dropping ACID with MongoDB
Kristina Chodorow, Software Engineer, 10gen
San Francisco MySQL Meetup
San Francisco, CA
Monday, April 12
Video

Introduction to MongoDB
Mike Dirolf, Software Engineer, 10gen
Emerging Technologies for the Enterprise Conference
Philadelphia, PA
Friday, April 9
Slides

Indexing with MongoDB
Aaron Staple, Software Engineer, 10gen
Webinar
Tuesday April 6, 2010
Video | Slides

TechZing Interview with Mike Dirolf, Software Engineer, 10gen
Monday, April 5
Podcast

Hot Potato and MongoDB
New York Tech Talks Meetup
Justin Shaffer and Lincoln Hochberg

New York, NY
Tuesday March 30
Video

MongoDB Day
Geek Austin Data Series
Austin, TX
Saturday March 27
Photo

Mongo Scale!
Kristina Chodorow, Software Engineer, 10gen
Webcast
Friday March 26
Webcast

NoSQL Live Boston
Boston, MA
Thursday March 11
Recap with slides and MP3

MongoDB: How it Works
Mike Dirolf, Software Engineer, 10gen
Monday March 8, 12:30 PM Eastern Time
Slides

Intro to MongoDB
Alex Sharp, Founder / Lead Software Architect, FrothLogic
LA WebDev Meetup
February 23, 2010
Slides

Introduction to MongoDB
Kristina Chodorow, Software Engineer, 10gen
FOSDEM - Brussels, Belgium
February 7, 2010
Video | Slides | Photos

If you're interested in having someone present MongoDB at your conference or meetup, or if you would like to list your MongoDB event on this page, contact meghan at 10gen dot com.

Your Go-to Resource for Running a MongoDB User Group

Tips for Running a Successful User Group

Interested in Starting a User Group?

Organizing a user group is a fantastic way to meet and learn from other MongoDB fans in your local community. Interested in starting up a user group in your city? Submit a proposal!

Logistics

Use Meetup.com: This one may be obvious but important to emphasize. Not only does meetup have a lot of great tools for event organizers, but they do a really good job of making it easy for meetup members to find relevant groups. Make sure to tag your group and include a description with keywords so that your group appears in meetup searches!

Consistency is important: It's important to establish a routine early on. If you consistently meet on, the second Tuesday of every month, your members will come to expect the meetup. The first few meetings of any user group will be small, but at every meetup, new members will join your group. So meeting at least on a monthly basis is very important. We have all the NY MUG meetups listed far in advance, even if we don't have a speaker lined up. This also makes your life easier when you are approaching speakers and hosts. It's much easier to ask a speaker "Can you present at the May 19 NY MUG?" than going back and forth coordinating dates. And hosts will appreciate having the events reserved far in advance.

Cross promote: Consider partnering with other technology meetups. This is a great way for communities to learn from one another and gain exposure to new technologies. It could be as simple as ocassionally posting on other meetup lists. For example, when we had a presentation on

Scalable Event Analytics with Ruby on Rails and MongoDB, I cross-posted the meetup on the NYC Ruby mailing list and we soon had a dozen new members in the group. I also typically list our events in Startup Digest, LinkedListNYC, Gary's Guide, Charlie O'Donnell's newsletter, Mashable, and more.

Once you Meet Up...

Get great speakers: Make a wish list of speakers, and then just start asking people! After organizing dozens of MongoDB events, I've been amazed at how willing people are to present. Most of the time, it's just a matter of asking. And if the person says no, ask them to refer someone else.

Get Clever About Getting Great Speakers: Are the speakers on your wish list 3,000 miles away from you? That's okay. We live in a wired world. Use Skype or WebX to bring your speakers to you. You can do screen shares to see demos, and provide your meetup members with a great learning experience.

Host Lightning Talks: Sometimes your user group members will not have the chance to pull together a full 30 minute presentation on a product or feature, but some of them may be interested in giving a 5-10 minute lightning talk on something they've designed, an issue they're having with MongoDB, or their dream project. Offer the opportunity when you send out the Meetup invite. If you don't get any submissions, don't worry. Deliver the first lightning talk at the event. Someone might step up to the plate.

Start a Study Group: Does your meetup have a lot of inexperienced MongoDB Users? Start a study group and learn together. Learning in a group is a great experience, and study groups are very common in different developer communities--particularly Ruby on Rails and Node.js

Host a Helpathon: The NYC On Rails Meetup is famous for innovating on the Hackathon to produce the Helpathon , a four hour session for helping one another get through programming hurdles.

Start a Book Club: If your group is at a loss for topics to discuss you could start a reading group. This is similar to a study group and can be a great way to create a more tightly knit User Group for the future.

Raffle off prizes: Prizes are an excellent way to get people to come to your meetup. An easy and free way to get great prizes is to join the O'Reilly User Group program. They will send you free books to give out at your group, as well as discounts on upcoming conferences.

Host a Coding Contest: Google is planning a coding competition for tickets to Google I/O, their largest developer contest. Why not do the same before your first user group? Present a small problem to your members and ask them to bring their files the day of to show off their solution. You can then have judges or the audience choose the winning solution(s).

Stay Engaged

Social media: Consider creating a twitter handle or hashtag for the group. Ask the presenter to tweet or blog about the event, and ask the key members of the group to do the same. Post the hashtag or twitter handle at the event so that members know to use it.

Continue the discussion after the meetup: You can easily record videos and take photos at sessions to share information after the meetup. Encourage the presenter to send a message the group with the slides to start a conversation among the entire meetup.

More Tips and Tricks

Agenda Tips for User Groups by Nathen Harvey , organizer of the MongoDC User Group, 2011 Community Champion

Tips and Tricks for Running a Successful Tech Meetup by Meghan Gill , Community and Events Manager for MongoDB at 10gen

How to Run a Successful Tech Meetup by Chris Westin , Software Engineer at 10gen

Articles

See also the User Feedback page for community presentations, blog posts, and more.

Best of the MongoDB Blog

- What is the Right Data Model? - (for non-relational databases)
- Why Schemaless is Good
- The Importance of Predictability of Performance
- Capped Collections - one of MongoDB's coolest features
- Using MongoDB for Real-time Analytics
- Using MongoDB for Logging
- http://blog.mongodb.org/tagged/best+of

Articles / Key Doc Pages

- On Atomic Operations
- Reaching into Objects - how to do sophisticated query operations on nested JSON-style objects
- Schema Design

* Full Text Search in Mongo
* MongoDB Production Deployments

Presentations

* Presentations from MongoDB Conferences
* MongoDB for Rubyists (February 2010 Chicago Ruby Meetup)
* Introduction to MongoDB (FOSDEM February 2010)
* NY MySql Meetup - NoSQL, Scaling, MongoDB
* Teach Me To Code - Introduction to MongoDB
* DCVIE

Benchmarks

MongoDB does not publish any official benchmarks.

We recommend running application performance tests on **your** application's work-load to find bottleneck and for performance tuning.

For more information about benchmarks in general please see the internet oracle.

FAQ

This FAQ answers basic questions for new evaluators of MongoDB.

* What kind of database is the Mongo database?
* What languages can I use to work with the Mongo database?
* Does it support SQL?
* Does it use Javascript?
* What are the use cases?
* Does it support transactions?
* Does MongoDB require lots of RAM?
* How do I configure the cache size?
* Are writes written to disk immediately, or lazily?
* Is caching handled by the database?
* What language is MongoDB written in?
* What are the 32-bit limitations?

See also:

* Developer FAQ
* Sharding FAQ
* Replica Set FAQ
* Indexing Advice and FAQ
* Production Notes

What kind of database is the Mongo database?

MongoDB is an document-oriented DBMS. Think of it as MySQL but JSON (actually, BSON) as the data model, not relational. There are no joins. If you have used object-relational mapping layers before in your programs, you will find the Mongo interface similar to use, but faster, more powerful, and less work to set up.

What languages can I use to work with the Mongo database?

Lots! See the drivers page.

Does it support SQL?

No, but MongoDB does support ad hoc queries via a JSON-style query language. See the Tour and Advanced Queries pages for more information on how one performs operations.

Does it use Javascript?

MongoDB core does not use Javascript, although it comes into play in a few places:

* the administrative shell is javascript based;
* the built-in map/reduce functionality expects javascript for the user-provided map and reduce functions;
* ancillary things such as $where and $eval use javascript. However the general direction would be towards less usage of these.

What are the use cases?

MongoDB is designed to be fairly general purpose so a large number of use cases are good fits. Content management systems, mobile, gaming, ecommerce, realtime report stats, archiving, logging are some examples.

Does it support transactions?

MongoDB does not provide fully generalized transactions but it does provide some transactional capabilities. Atomic operations are possible within the scope of a single document : that is, we can debit a and credit b as a transaction if they are fields within the same document. Given that documents can be rich, sometimes containing thousands of fields with lots of nesting, for some use cases this proves to be quite powerful.

Users have built e-commerce systems using MongoDB (and are happy with the results).

A general ledger application would be a good example of something you wouldn't build in Mongo, given the highly transactional nature of the problem.

Writes in MongoDB are durable (the 'D' in acid). Use journaling (which is on by default) and the j:true parameter when requesting getLastError result acknowledgements.

Does MongoDB require lots of RAM?

No; in fact it is possible to run MongoDB on a machine with a tiny amount of free RAM.

MongoDB automatically uses all free memory on the machine as its cache. Thus it will look like it is using a lot of memory - and it is for the cache - but that usage is dynamic; for example if another process suddenly needed half the server's RAM, Mongo will yield that memory to it (and then have a smaller cache).

How do I configure the cache size?

There is no configuration, rather all free memory on the system is automatically used – much like operating systems do with their file system caches. In fact, mongo actually is using the file system cache : the mongodb storage engine uses memory-mapped files.

Are writes written to disk immediately, or lazily?

Writes are physically written to the journal within approximately 50 milliseconds. At that point, the write is "durable" in the sense that after a pull-plug-from-wall event, the data should still be there on restart.

While the journal write is almost immediate, writes to the datafiles are done lazily and may not flush for up to one minute. However this does not effect durability as the journal will have already recorded sufficient information for a crash recovery.

Is caching handled by the database?

For simple queries (with an index) Mongo should be fast enough that you can query the database directly without needing the equivalent of memcached. The goal is for Mongo to be an alternative to an `ORM/memcached/mysql` stack. Some MongoDB users do like to mix it with memcached though.

What language is MongoDB written in?

The database is written in C++. Drivers are usually written in their respective languages, although some use C extensions for speed.

What are the 32-bit limitations?

MongoDB uses memory-mapped files. When running on a 32-bit operating system, the total storage size for the server (data, indexes, everything) is 2gb. If you are running on a 64-bit os, there is virtually no limit to storage size. Thus 64 bit production deployments are recommended. See the blog post for more information. One other note: journaling is not on by default in the 32 bit binaries as journaling uses extra memory-mapped views.

Misc

Demo App in Python

From an Interop 2009 presentation

Code: http://github.com/mdirolf/simple-messaging-service/tree/master

MongoDB, CouchDB, MySQL Compare Grid

pending...

	CouchDB	MongoDB	MySQL
Data Model	Document-Oriented (JSON)	Document-Oriented (BSON)	Relational
Data Types	string,number,boolean,array,object	string, int, double, boolean, date, bytearray, object, array, others	link
Large Objects (Files)	Yes (attachments)	Yes (GridFS)	Blobs
Horizontal partitioning scheme	CouchDB Lounge	Auto-sharding	Partitioning
Replication	Master-master (with developer supplied conflict resolution)	Master-slave and replica sets	Master-slave, multi-master, and circular replication
Object(row) Storage	One large repository	Collection-based	Table-based
Query Method	Map/reduce of javascript functions to lazily build an index per query	Dynamic; object-based query language	Dynamic; SQL
Secondary Indexes	Yes	Yes	Yes
Atomicity	Single document	Single document	Yes - advanced
Interface	REST	Native drivers ; REST add-on	Native drivers
Server-side batch data manipulation	?	Map/Reduce, server-side javascript	Yes (SQL)
Written in	Erlang	C++	C++
Concurrency Control	MVCC	Update in Place	
Geospatial Indexes	GeoCouch	Yes	Spatial extensions
Distributed Consistency Model	Eventually consistent (master-master replication with versioning and version reconciliation)	Strong consistency. Eventually consistent reads from secondaries are available.	Strong consistency. Eventually consistent reads from secondaries are available.

See Also

- Comparing Mongo DB and Couch DB

Comparing Mongo DB and Couch DB

We are getting a lot of questions "how are mongo db and couch different?" It's a good question: both are document-oriented databases with schemaless JSON-style object data storage. Both products have their place -- we are big believers that databases are specializing and "one size fits all" no longer applies.

We are not CouchDB gurus so please let us know in the forums if we have something wrong.

MVCC

One big difference is that CouchDB is MVCC based, and MongoDB is more of a traditional update-in-place store. MVCC is very good for certain classes of problems: problems which need intense versioning; problems with offline databases that resync later; problems where you want a large amount of master-master replication happening. Along with MVCC comes some work too: first, the database must be compacted periodically, if there are many updates. Second, when conflicts occur on transactions, they must be handled by the programmer manually (unless the db also does conventional locking -- although then master-master replication is likely lost).

MongoDB updates an object in-place when possible. Problems requiring high update rates of objects are a great fit; compaction is not necessary. Mongo's replication works great but, without the MVCC model, it is more oriented towards master/slave and auto failover configurations than to complex master-master setups. With MongoDB you should see high write performance, especially for updates.

Horizontal Scalability

One fundamental difference is that a number of Couch users use replication as a way to scale. With Mongo, we tend to think of replication as a way to gain reliability/failover rather than scalability. Mongo uses (auto) sharding as our path to scalabity (sharding is GA as of 1.6). In this sense MongoDB is more like Google BigTable. (We hear that Couch might one day add partitioning too.)

Query Expression

Couch uses a clever index building scheme to generate indexes which support particular queries. There is an elegance to the approach, although one must predeclare these structures for each query one wants to execute. One can think of them as materialized views.

Mongo uses traditional dynamic queries. As with, say, MySQL, we can do queries where an index does not exist, or where an index is helpful but only partially so. Mongo includes a query optimizer which makes these determinations. We find this is very nice for inspecting the data administratively, and this method is also good when we *don't* want an index: such as insert-intensive collections. When an index corresponds perfectly to the query, the Couch and Mongo approaches are then conceptually similar. We find expressing queries as JSON-style objects in MongoDB to be quick and painless though.

Update Aug2011: Couch is adding a new query language "UNQL".

Atomicity

Both MongoDB and CouchDB support concurrent modifications of single documents. Both forego complex transactions involving large numbers of objects.

Durability

CouchDB is a "crash-only" design where the db can terminate at any time and remain consistent.

Previous versions of MongoDB used a storage engine that would require a repairDatabase() operation when starting up after a hard crash (similar to MySQL's MyISAM). Version 1.7.5 and higher offer durability via journaling; specify the `--journal` command line option

Map Reduce

Both CouchDB and MongoDB support map/reduce operations. For CouchDB map/reduce is inherent to the building of all views. With MongoDB, map/reduce is only for data processing jobs but not for traditional queries.

Javascript

Both CouchDB and MongoDB make use of Javascript. CouchDB uses Javascript extensively including in the building of views .

MongoDB supports the use of Javascript but more as an adjunct. In MongoDB, query expressions are typically expressed as JSON-style query objects; however one may also specify a javascript expression as part of the query. MongoDB also supports running arbitrary javascript functions server-side and uses javascript for map/reduce operations.

REST

Couch uses REST as its interface to the database. With its focus on performance, MongoDB relies on language-specific database drivers for access to the database over a custom binary protocol. Of course, one could add a REST interface atop an existing MongoDB driver at any time -- that would be a very nice community project. Some early stage REST implementations exist for MongoDB.

Performance

Philosophically, Mongo is very oriented toward performance, at the expense of features that would impede performance. We see Mongo DB being useful for many problems where databases have not been used in the past because databases are too "heavy". Features that give MongoDB good performance are:

- client driver per language: native socket protocol for client/server interface (not REST)
- use of memory mapped files for data storage
- collection-oriented storage (objects from the same collection are stored contiguously)
- update-in-place (not MVCC)
- written in C++

Use Cases

It may be helpful to look at some particular problems and consider how we could solve them.

- if we were building Lotus Notes, we would use Couch as its programmer versioning reconciliation/MVCC model fits perfectly. Any problem where data is offline for hours then back online would fit this. In general, if we need several eventually consistent master-master replica databases, geographically distributed, often offline, we would use Couch.
- mobile
    - Couch is better as a mobile embedded database on phones, primarily because of its online/offine replication/sync capabilities.

- we like Mongo server-side; one reason is its geospatial indexes.
- if we had very high performance requirements we would use Mongo. For example, web site user profile object storage and caching of data from other sources.
- for a problem with very high update rates, we would use Mongo as it is good at that because of its "update-in-place" design. For example see updating real time analytics counters
- in contrast to the above, couch is better when lots of snapshotting is a requirement because of its MVCC design.

Generally, we find MongoDB to be a very good fit for building web infrastructure.

Licensing

- Database:
    - Free Software Foundation's GNU AGPL v3.0.
    - Commercial licenses are also available from 10gen, including free evaluation licenses.
- Drivers:
    - mongodb.org supported drivers: Apache License v2.0.
    - Third parties have created drivers too; licenses will vary there.
- Documentation: Creative Commons.

The goal of the server license is to require that enhancements to MongoDB be released to the community. Traditional GPL often does not achieve this anymore as a huge amount of software runs in the cloud. For example, Google has no obligation to release their improvements to the MySQL kernel – if they do they are being nice.

To make the above practical, *we promise that your client application which uses the database is a separate work*. To facilitate this, the mongodb.org supported drivers (the part you link with your application) are released under Apache license, which is copyleft free. Note: if you would like a signed letter asserting the above promise please request via email.

If the above isn't enough to satisfy your organization's vast legal department (some will not approve GPL in any form), please contact us – commercial licenses are available including free evaluation licenses. We will try hard to make the situation work for everyone.

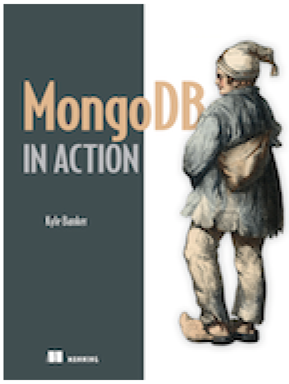
International Docs

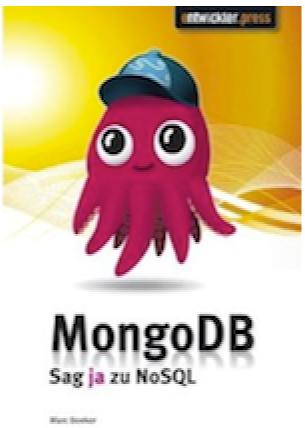
Most documentation for MongoDB is currently written in English. We are looking for volunteers to contribute documentation in other languages. If you're interested in contributing to documentation in another language please email **docs at 10gen dot com** .

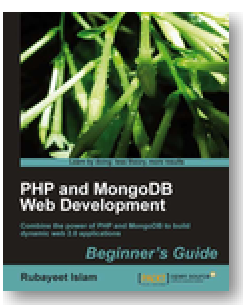
Language Homepages

- Deutsch
- Español
- Français
-
- Italiano
- [!hu.png! Magyar]
-
- Português
-
- Svenska
-
-
-

Books







You can download samples at 10gen.com/books.

MongoDB: The Definitive Guide
Kristina Chodorow and Mike Dirolf

The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing
Peter Membrey

Scaling MongoDB

Kristina Chodorow

MongoDB in Action
Read the **Early Access Edition**
Kyle Banker

MongoDB and Python: Patterns and processes for the popular document-oriented database
Niall O'Higgins

MongoDB and PHP
Steve Francia

MongoDB for Web Development
Mitch Pirtle

MongoDB: Sag Ja zu NoSQL
Marc Boeker

The Little MongoDB Book
Karl Seguin
Free ebook

50 Tips and Tricks for MongoDB Developers
Kristina Chodorow

PHP and MongoDB Web Development Beginner's Guide book and eBook
Rubayeet Islam

Slides and Video

Upcoming Events

- http://www.10gen.com/events

Presentations from past conferences

- http://www.10gen.com/presentations

Some past webinars

- Analytics MongoDB Analytics for Online Advertising at Magnetic
- DevOps Data Center Awareness with MongoDB
- DevOps Deployment Best Practices
- DevOps Diagnostics and Performance Tuning
- DevOps Keeping your data safe: replication, backup/restore, mongodump
- DevOps MongoDB and Amazon Web Services
- DevOps MongoDB on Amazon EC2
- DevOps Scaling with MongoDB
- DevOps Tips for a Successful, Scalable MongoDB Deployment
- Geo Geospatial Indexing with MongoDB
- Hadoop Hadoop plugin for MongoDB: The Elephant in the Room
- Indexing Indexing Matters: A MongoDB Optimization Primer
- Indexing Indexing With MongoDB
- Intro Developing Applications with MongoDB: A Quickstart Tutorial
- Intro Introduction to MongoDB
- Java What's New in Java Driver Version 2.5
- Java Zero to MongoDB - Introduction for MongoDB with Java
- Journaling Journaling in MongoDB version 1.8
- Node.js Node.js and MongoDB, a Panel Discussion
- Ruby MongoDB and Ruby
- Scala MongoDB and Scala
- Schema Schema Design with MongoDB
- Use Cases How MongoDB helps Visibiz tackle social CRM
- Use Cases Humongous Data at Server Density: Approaching 1 Billion Documents in MongoDB
- Hybrid Applications with MongoDB and RDBMS
- v1.8 What's new in MongoDB v1.8.0
- v2.0 What You Need to Know About MongoDB 2.0

Doc Index

Space Index

0-9

1.0 Changelist
Wrote MongoDB. See documentation

1.1 Development Cycle

1.2.x Release Notes
New Features More indexes per collection Faster index creation Map/Reduce Stored JavaScript functions Configurable fsync time Several small features and fixes DB Upgrade Required There are some changes that will require doing an upgrade ...

1.4 Release Notes
We're pleased to announce the 1.4 release of MongoDB. 1.4 is a drop in replacement for 1.2. To upgrade you just need to shutdown mongod, then restart with the new binaries. (Users upgrading from release 1.0 should review the 1.2 release notes 1.2.x ...

1.6 Release Notes
MongoDB 1.6 is a dropin replacement for 1.4. To upgrade, simply shutdown {{mongod}} then restart with the new binaries. MongoDB v1.8 DOCS:1.8 Release Notes is now available. \ Please note that you should upgrade to the latest version of whichever driver you're ...

1.8 Release Notes
MongoDB 1.8 is a dropin replacement for 1.6, except: replica set nodes should be upgraded in a particular order Upgrading to 1.8. The options to the Map/Reduce command have changed in 1.8, causing incompatibility with previous releases. If you use MapReduce, please ...

2.0 Release Notes
Upgrading Although the major version number has changed, MongoDB 2.0 is a standard, incremental production release and works as a dropin replacement for MongoDB 1.8. However, there are a few changes you must be aware of before attempting to upgrade: # If you create new ...

A

A Sample Config
following exa
on a single te
single ...

About

About the local c
mongod}} res
authenticatio
against the {{

Adding a New S
Adding a new
recent copy o

Adding an Arbite
Arbiters are r
the primary n
has two mem

Admin UIs
MongoDB do
such as the r
focused on a

Admin Zone
Community A
http://blog.bo
http://blog.tin
http://tag1co

Advanced Queri
MongoDB, ju
Indexes page
features ...

Aggregation
Mongo incluc
advanced ag
Aggregation

Aggregation Fra
Aggregation
MongoDB Ma
http://docs.m
http://docs.m

Aggregation Fra
Overview In i
value based
each web pa

Aggregation Fra
Overview Th
a single num
Here's a simp

Aggregation Fra
Overview $m
progress furt
$match predi

Aggregation Fra

Overview $pr
Specification
is my title" , a

Aggregation Fra
Overview Th
Specification
rest are pass

Aggregation Fra
Overview Th
$sort require
Indexes#Inde

Aggregation Fra
Overview $u
document, fo
one of its ele

Aggregation Fra
Overview Ex
as JSON fiel
$add:\5, 6, 7\

Alerts
page lists crit
http://jira.mo
missing on a

Amazon EC2
MongoDB ru
Started on E
production in

Amazon EC2 Qu
guide is inter
(AWS) EC2 i
then we'll set

Architecture and
list of the cor
Stopping Mo
(uses interac

Articles
See also the
MongoDB Bl
nonrelational

Atomic operation
key goal of M
found that a g
Below are so

Atomic Operatio
MongoDB su
complex tran
slow.

Automating Depl
CloudFormat
and manage
resources in

AWS Marketplac
you installed
you are inter
interested in

Azure Configura
following are
configuration
1 would run a

Azure Deployme
development
using Visual
development

B

Backing Up Sharded Cluster
See also Backups DOCS:Backups and Import Export Tools DOCS:Import Export Tools for more information on

C

C Language Cen
C Driver {}Th

backups, particularly on the step of backing up each individual server. If you are on EC2 you should look at EC2 Backup & Restore ...

Backups
Several strategies exist for backing up MongoDB databases. A word of warning: if you do not have journaling enabled, then it's not safe to simply copy the mongod data files (by default in /data/db/) while the database is running and writes are occurring; such a backup ...

Benchmarks
MongoDB does not publish any official benchmarks. We recommend running application performance tests on your application's workload to find bottleneck and for performance tuning. For more information about benchmarks in general please see the internet oracle http://en.wikipedia.org ...

Books
mongodbinaction.png! http://manning.com/banker !sagya.png! https://www.amazon.de/MongoDBSagJazuNoSQL/dp/3868020578/ref=sr13?ie=UTF8&qid=1291985616&sr=83 !mongodbcover.png! http://openmymind.net/2011/3/28/TheLittleMongoDBBook !packt.jpg! http://link.packtpub.com/2qxi08 You can download samples at 10gen.com/books ...

Boost 1.41.0 Visual Studio 2010 Binary
OLD and was for the VS2010 BETA. See the new Boost and Windows page instead. The following is a prebuilt boost http://www.boost.org/ binary (libraries) for Visual Studio 2010 beta 2. The MongoDB vcxproj files assume this package is unzipped under c:\Program ...

Boost and Windows
Visual Studio 2010 Prebuilt from mongodb.org Click here http://www.mongodb.org/pages/viewpageattachments.action?pageId=12157032 for a prebuilt boost library for Visual Studio 2010. 7zip http://www.7zip.org/ format. Building Yourself Download the boost source ...

BSON
bsonspec.org http://www.bsonspec.org/ BSON is a bin-aryen-coded seri-al-iz-a-tion of JSONlike doc-u-ments. BSON designed to be lightweight, traversable, and efficient. BSON, like JSON, supports the embedding of objects and arrays within other objects ...

BSON Arrays in C++
examples using namespace mongo; using namespace bson; bo anobj; / transform a BSON array into a vector of BSONElements. we match array # positions with their vector position, and ignore any fields with nonnumeric field names. / vector<be> a = anobj"x".Array ...

Building
Note: see the Downloads DOCS:Downloads page for prebuilt binaries, it's recommended to use those as all full QA occurs after those are built. This section provides instructions on setting up your environment to write Mongo drivers or other infrastructure ...

Building Boost
necessary when building mongo versions 2.1.1 or later. MongoDB uses the www.boost.org Boost C\ libraries. Windows See also the prebuilt libraries http://www.mongodb.org/pages/viewpageattachments.action?pageId=12157032 page. By default c:\boost\ is checked for the boost ...

Building for FreeBSD
FreeBSD 8.0 and later, there is a mongodb port you can use. For FreeBSD <= 7.2: # Get the database source: http://www.github.com/mongodb/mongo. # Update your ports tree: $ sudo portsnap fetch && portsnap extract The packages that come by default on 7.2 ...

Building for Linux
Note: Binaries are available for most platforms. Most users won't need to compile mongo themselves; in addition every prebuilt binary has been regression tested. See the Downloads page for these prebuilt binaries. These instructions apply to the git master branch ...

Building for OS X
set up your OS X computer for MongoDB development: Prerequisites Xcode Available in the App Store. You only need to get the command line tools, if you don't want to install the whole IDE. SCons SCons is the build tool ...

Building for Solaris
MongoDB server currently supports little endian Solaris operation. (Although most drivers not the database server work on both.) Community: Help us make this rough page better please\! (And help us add support for big ...

Building for Windows
Binaries are available for most platforms. Most users won't need to compile mongo themselves; in addition every prebuilt binary has been regression tested. See the Downloads page for these prebuilt binaries. MongoDB can be compiled for Windows (32 and 64 ...

Building indexes with replica sets
Version 2.1.0 and later Indexes can be built in the foreground or background. Background indexes builds on the primary will result in background index builds on the secondaries. Index built on primary Index built on secondary Index built on recovering member Foreground Foreground Foreground ...

Building Spider Monkey
MongoDB uses SpiderMonkey http://www.mozilla.org/js/spidermonkey/ for serverside Javascript execution. Pre v2.0: MongoDB requires a js.lib file when linking. This page details how to build js.lib. v2.0: this is handled automatically by the Mongo build scripts via ...

stable as of t
http://api.mor

C Sharp Langua

C++ BSON Libra
MongoDB C
This library c
Include {{bso

C++ Driver Com
C\ driver is in
tarball (see D
of the driver s

C++ Driver Dowr
Driver tarball
filenames, th
code ...

C++ getLastErro
string mongo
http://api.mor
Get error res
DBClientWith

C++ Language C
C\ driver is a
some core M
successfully

C++ Tutorial
document is
for details. N
language ind

Caching
Memory Map
disk I/O.&nbs
several impli

Capped Collectio
Capped colle
on insertion o
automatically

CentOS and Fec
10gen publis
mongo10gen
mongo10gen

Changing a Shar
automatic su
operation mo
choose the ri

Changing Config
Sections: The
maintenance
...

Checking Server
How Caching
memory in th
process mem

Choosing a Shar
important to c
in doubt plea
system. Docu

cloneCollection C
Copy a single
not use on a
copy of the n

Collections
MongoDB co
relational dat
usually have

collStats Comma
Statistics on
db.command

Building SpiderMonkey

Building the Mongo Shell on Windows
You can build the mongo shell with either scons or a Visual Studio 2010 project file. Scons scons mongo.exe Visual Studio 2010 Project File A VS2010 vcxproj file is availabe for building the shell. From the mongo directory open ...

Building with V8
Linux or OSX $ pwd /mongo $ cd .. $ svn checkout http://v8.googlecode.com/svn/trunk/ v8 $ cd v8 $ scons arch=x64 debuggersupport=off snapshot=off profilingsupport=off $ cd ../mongo $ scons usev8 Windows $ pwd /mongo ...

Building with Visual Studio 2008
instructions are for versions of mongo prior to 2.1.1. For version 2.1.1 and newer, the instructions for Visual Studio 2010 DOCS:Building with Visual Studio 2010 and Visual Studo 2008 are the same. MongoDB can be compiled for Windows (32 and 64 bit) using Visual ...

Building with Visual Studio 2010
Binaries are available for most platforms. Most users won't need to compile mongo themselves; in addition every prebuilt binary has been regression tested. See the Downloads page for these prebuilt binaries. v2.1.1 MongoDB can be compiled for Windows (32 ...

db.foo.stats f

Command Line I
MongoDB ca
supported se
Information c

Commands
Introduction
database to
Commands T

Community
Technical Su
feature reque
http://blog.m

Community Info

Compact Comm
v2.0\ The co
similar to {{re
shell): > db.r

Comparing Mon
We are gettir
documentori
data storage.

Components

Configuring Sha
document de
One to 1000

Connecting
C\ driver incl
our normal c
for connectin

Connecting Driv
Ideally a Mo
automatically
general step

Connecting to R
Most drivers
support conn
list of host ...

Connections
MongoDB is
you start Mo

Contributing to t
Qualified vol

Contributing to t
easiest way t
help code the

Contributors
10gen Contri

Conventions for
Interface Co
and when the
drivers is des

cookbook.mong

Copy Database
MongoDB in
alternative of
representatio

createCollection
Use the crea
collections. >
create collec

Creating and De

CSharp Commu
Community S
mongodbcsh

http://code.g
CSharp Driver L
Introduction
have read at
Quickstart Fi
CSharp Driver Q
Introduction
started you c
CSharp Driver S
Introduction
(and deserial
object to a B
CSharp Driver T
C# Driver ve
C# Driver for
be used inde
CSharp getLast
C# driver Sa
"mongodb://
database lev
CSharp Langua
MongoDB C
QuickStart C
Cursors

D

Data Center Awareness
Examples One primary data center, one disaster recovery site Multiple set members can be primary at the main data center. Have a member at a remote site that is never primary (at least, not without human intervention). { id: 'myset', members ...

Data Processing Manual
data processing", we generally mean operations performed on large sets of data, rather than small interactive operations. Import One can always write a program to load data of course, but the mongoimport DOCS:Import Export Tools utility ...

Data Types and Conventions
MongoDB (BSON) Data Types Mongo uses special data types in addition to the basic JSON types of string, integer, boolean, double, null, array, and object. These types include date, object id Object IDs, binary data, regular ...

Database Internals
section provides information for developers who want to write drivers or tools for MongoDB, \ contribute code to the MongoDB codebase itself, and for those who are just curious how it works internally. Subsections of this section

Database Profiler
Mongo includes a profiling tool to analyze the performance of database operations. See also the currentOp DOCS:Viewing and Terminating Current Operation command. Enabling Profiling Through the profile command You can enable and disable profiling from the mongo shell ...

Database References
MongoDB is nonrelational (no joins), references ("foreign keys") between documents are generally resolved clientside by additional queries to the server ("linking"). These links are always resolved clientside. Doing this directly/manually can be quite easy and is recommended. There is also a DBRef mechanism which ...

Databases
Each MongoDB server can support multiple databases. Each database is independent, and the data for each database is stored separately, for security and ease of management. A database consists of one or more collections DOCS:Collections, the documents DOCS:Documents (objects) in those ...

Dates
BSON Date/Time data type is referred to as "UTC DateTime" in the BSON spec http://bsonspec.org. Note There is a Timestamp data type but that is a special internal type for MongoDB that typically should not be used. A BSON Date value stores the number of milliseconds since ...

DBA Operations from the Shell
page lists common DBAclass operations that one might perform from the MongoDB shell DOCS:mongo The Interactive Shell. Note one may also create .js scripts to run in the shell for administrative purposes. help show help show ...

dbshell Reference
Command Line {{\help}} Show command line options {{\nodb}} Start without a db, you can connect later with {{new Mongo()}} or {{connect()}} {{\shell}} After running a .js file from the command line, stay in the shell rather than ...

Demo App in Python
From an Interop 2009 presentation Code: http://github.com/mdirolf/simplemessagingservice/tree/master

E

EC2 Backup & R
Overview Th
http://aws.a
1.8 (or above

Emacs tips for M
You can edit
Change the
the default s

Erlang Languag
Driver Downl
Driver http://
Frameworks

Error Codes
you have an
nothing plea
Comments \\

Error Handling i
an error occ
first field gua

Events
MongoDB U
http://www.m
Event What/

Excessive Disk
Understandi
the data set i
DOCS:Replic

Explain
Basics A gre
This will disp
Interactive S

Design Overview

Developer FAQ

See also: Intro FAQ FAQ DOCS:Sharding FAQ DOCS:Replica Set FAQ DOCS:Indexing Advice and FAQ Markus Gattol's excellent FAQ on his website http://www.markusgattol.name/ws/mongodb.html What's a "namespace"? MongoDB stores ...

Developer Zone

Tutorial Shell mongo The Interactive Shell Manual Databases Collections Indexes Data Types and Conventions GridFS Inserting Updating Querying Removing Optimization Developer FAQ Cookbook http://cookbook.mongodb.org If you have a comment or question about anything, please ...

Diagnostic Tools

Django and MongoDB

Do I Have to Worry About SQL Injection

See link at bottom for driver specific security notes; specifically one for PHP http://us.php.net/manual/en/mongo.security.php. Instructions for the MongoDB server cluster are represented in two forms: # BSON # Javascript With BSON being the main mechanism. BSON As a client ...

Doc Index

Document-Oriented Datastore

Documentation

Documents

MongoDB can be thought of as a documentoriented database. By 'document', we mean structured documents, not freeform text documents. These documents can be thought of as objects http://blog.10gen.com/post/437029788/jsondbvsodbms but only the data of an object, not the code, methods or class hierarchy. Additionally, there is much ...

Dot Notation

Dot Notation

Dot Notation (Reaching into Objects)

MongoDB is designed for storing JSONstyle objects. The database understands the structure of these objects and can reach into them to evaluate query expressions. Let's suppose we have some objects of the form: > db.persons.findOne() { name: "Joe", address: , likes: 'scuba', 'math ...

dotCloud

Running MongoDB on dotCloud MongoDB can run on dotCloud http://www.dotcloud.com/. It supports replica sets, and has alpha support for sharding. The whole point of dotCloud is to run your apps and your databases in the same place, to optimize for latency and reliability. However ...

Downloads

table id="downloads"> <thead> <tr> <td></td> <td class="tabletop tableleft odd">OS X 32bit
note</td> <td class="tabletop even">OS X 64bit</td ...

Driver and Integration Center

Driver Syntax Table

wiki generally gives examples in JavaScript, so this chart can be used to convert those examples to any language. JavaScript Python PHP Ruby Java C\ C# Perl \ \ \\ {{array()}} \\ {{BasicDBList}} {{BSONObj or bson::bo}} {{BsonArray}} \ \ {} {} {{new stdClass ...

Driver Testing Tools

Object IDs driverOIDTest for testing toString > db.runCommand

Drivers

MongoDB currently has client support for the following programming languages: mongodb.org Supported C C Language Center C\ C Language Center Erlang Erlang Language Center Haskell Haskell Language Center Java Java Language Center ...

Durability and Repair

MongoDB (specifically, the mongod process) is normally ran with journaling enabled. This makes it crashsafe. Journaling is on by default in v2.0 for 64 bit builds. (Use journaling, leave it enabled.) Journaling Enabled If you are running with Journaling you should not do a repair to recover ...

Durability Internals

main durability page (not the internals page) is the Journaling page. Files The data file format is unchanged. Journal files are placed in /data/db/journal/. Running Run with {{journal}} to enable journaling/durable storage. Both {{mongod}} and {{test}} support this option ...

F

FAQ

FAQ answers basic questions for new evaluators of MongoDB. See also: Developer FAQ DOCS:Sharding FAQ DOCS:Replica Set FAQ DOCS:Indexing Advice and FAQ DOCS:Production Notes What kind of database is the Mongo database ...

Feature Checklist for Mongo Drivers

Functionality Checklist This section lists tasks the driver author might handle. Essential BSON

G

Geospatial Hays

addition to or
indexing", the
For example,

Geospatial Index

v1.4\ MongoI

serialization/deserialization Basic operations: {{query}}, {{insert}}, {{update}}, {{remove}}, {{ensureIndex}}, {{findOne}}, {{limit}}, {{sort}} Fetch more data from a cursor when necessary ...

File Based Configuration
addition to accepting command line parameters, MongoDB can also be configured using a configuration file. A configuration file can be specified using the {{\f}} or {{\config}} command line options. On some packaged installs of MongoDB (for example Ubuntu & Debian), the default ...

findAndModify Command
Find and Modify (or Remove) v1.3.0 and higher MongoDB 1.3\ supports a "find, modify, and return" command. This command can be used to atomically modify a document (at most one) and return it. Note that, by default, the document returned will not include the modifications made on the update ...

flushRouterConfig command
flushRouterConfig This command will clear the current cluster information that a mongos DOCS:Architecture and Components process has cached and load the latest settings from the config db. This can be used to force an update when the config db and the data cached in mongos ...

Forcing a Member to be Primary
Replica sets automatically negotiate which member of the set is primary and which are secondaries. If you want a certain member to be primary, there are a couple ways to force this. v2.0 In v2.0, you can set the priority of the preferred primary to be higher than the priorities of the other ...

Frequently Asked Questions - Ruby
list of frequently asked questions about using Ruby with MongoDB. If you have a question you'd like to have answered here, please add it in the comments. Can I run \insert command name here\ from the Ruby driver? Yes ...

fsync Command
v1.4 fsync Command The fsync command allows us to flush all pending writes to datafiles. More importantly, it also provides a lock option that makes backups easier. The fsync command forces the database to flush all datafiles: > use ...

Full Text Search in Mongo
Introduction Mongo provides some functionality that is useful for text search and tagging. Multikeys (Indexing Values in an Array) The Mongo multikey feature can automatically index arrays of values. Tagging is a good example of where this feature is useful. Suppose you ...

me the close
museums to

getCmdLineOpts
getCmdLine(
use admin db
"512", "nojou

getLastError Cor
MongoDB dc
that operation
operation. Er

getLastError_old

getLog Comman
LogRelated (
messages fo
possible log (

Getting Started

Getting the Softw
Placeholder :

GridFS
GridFS is a s
spec DOCS:(
objects.&nbs

GridFS in Ruby
GridFS, whic
manageable
this: one coll

GridFS Specifica
Introduction (
chunks, usua
including the

GridFS Tools
File Tools {{n
./mongofiles
connected to

H

Hadoop
https://github.com/mongodb/mongohadoop
http://blog.10gen.com/post/20840407875/mongodbhadoopconnectorannounced

Halted Replication
instructions are for master/slave replication. For replica sets, see DOCS:Resyncing a Very Stale Replica Set Member instead. If you're running mongod with masterslave replication DOCS:Master Slave, there are certain scenarios where the slave will halt replication because ...

Haskell Language Center
Haskell driver and its API documentation reside on Hackage http://hackage.haskell.org/package/mongoDB

Home
Getting Started Quickstart \ how to install Tutorial \ quick tutorial showing examples of manipulation of a database from the mongo shell SQL to Mongo Mapping Chart Development Developer Manual DOCS:Manual See the documentation below for the driver for your application's ...

Hosting Center
DatabaseasaService MongoOd.com http://mongood.com/ MongoLab https://mongolab.com/home/ HostedMongo.com http://hostedmongo.com/ MongoGrid http://nodegrid.com/resources/mongogrid/ InfrastructureasaService Amazon EC2 DOCS:Amazon EC2 Joyent DOCS:Joyent PlatformasaService alwaysdata http://blog.alwaysdata.com/en/2011/04/26/officialsupportofmongodb ...

How does concurrency work
documentation below covers for MongoDB v2.0. v2.0 typically achieves significantly more concurrency in disk activity than v1.8 did. v2.2 will include substantial enhancements with concurrency work being by far the #1 item on the v2.2 road map. mongos ...

How MongoDB is Used in Media and Publishing
We see growing usage of MongoDB in both traditional and new media organizations. In these areas, the challenges for application developers include effectively managing rich content (including usergenerated content) at scale, deriving insight into how content is consumed and shared in realtime ...

How to do Snapshotted Queries in the Mongo Database
document refers to query snapshots. For backup snapshots of the database's datafiles, see the fsync lock page fsync Command. MongoDB does not support full pointintime snapshotting. However, some functionality is available which is detailed below. Cursors A MongoDB query ...

I

Implementing Au
current versio
context of a p
question. The

Import Export To
you just want
instead of the
they do not s

Index Versions
MongoDB v2
old {{\{v:0\}}}

Index-Related C
Operations: (
info to the {{s
db.system.in

Indexes
Indexes enha
need so that
Indexes in M

Indexes in Mong

Indexing Advice
We get a lot (
in mind, thou
building effici

Indexing as a Ba
Prior to 2.1.0
slave/second
completely b

Inserting

[How to Make an Auto Incrementing Field]
Generally in MongoDB, one does not use an autoincrement pattern for \id's (or other fields), as this does not scale up well on large database clusters. Instead one typically uses Object IDs. Side counter method One can keep ...

[HowTo]

[Http Interface]
REST Interfaces Sleepy Mongoose (Python) Sleepy Mongoose http://www.snailinaturtleneck.com/blog/2010/02/22/sleepymongooseamongodbrestinterface/ is a full featured REST interface for MongoDB which is available as a separate project. MongoDB Rest (Node.js) MongoDB Rest http://github.com/tdegrunt/mongodbrest is an alpha ...

When we ins
JSON, Pytho
and describe

[Installing the PH

[Internal Comma
Most comma
internal and a
admindb.$cm

[Internals
Cursors Taila
OptionCurso
when ...

[International Doc
Most docume
other langua
com. Langua

[International Doc

[Internationalized
MongoDB su
UTF8.) Gene
serializing an

[Introduction
MongoDB wa
systems. We

[Introduction - Ho

[iostat
Linux, use th
work well. (U

J

[Java - Saving Objects Using DBObject]
Java driver provides a DBObject interface to save custom objects to the database. For example, suppose one had a class called Tweet that they wanted to save: public class Tweet implements DBObject Then you can say: Tweet myTweet = new Tweet ...

[Java Driver Concurrency]
Java MongoDB driver is thread safe. If you are using in a web serving environment, for example, you should create a single Mongo instance, and you can use it in every request. The Mongo object maintains an internal pool of connections to the database ...

[Java Language Center]
Java Driver Basics Download the Java Driver http://github.com/mongodb/mongojavadriver/downloads Tutorial http://www.mongodb.org/display/DOCS/JavaTutorial API Documentation http://api.mongodb.org/java/index.html Release Notes https://github.com/mongodb/mongojavadriver/wiki ...

[Java Tutorial]
Introduction This page is a brief overview of working with the MongoDB Java Driver. For more information about the Java API, please refer to the online API Documentation for Java Driver http://api.mongodb.org/java/index.html A Quick Tour Using the Java driver is very ...

[Java Types]
Object Ids {{com.mongodb.ObjectId}} http://api.mongodb.org/java/0.11/com/mongodb/ObjectId.html is used to autogenerate unique ids. ObjectId id = new ObjectId(); ObjectId copy = new ObjectId(id); Regular Expressions The Java driver uses {{java.util.regex.Pattern}} http://java.sun.com ...

[Javascript Language Center]
MongoDB can be Used by clients written in Javascript; Uses Javascript internally serverside for certain options such as map/reduce; Has a shell DOCS:mongo The Interactive Shell that is based on Javascript for administrative purposes. node.JS and V8 See the node.JS page. node.JS ...

[Job Board]
Redirecting

[Journaling]
MongoDB v1.7.5 supports writeahead journaling of operations to facilitate fast crash recovery and durability in the storage engine. Disabling/Enabling In version 1.9.2, journaling is enabled by default for 64bit platforms. You can disable journaling with the mongod {{\\nojournal}} command line ...

[Journaling Administration Notes]
Journal Files (e.g. journal/j.0) Journal files are appendonly and are written to the {{journal/}} directory under the dbpath directory (which is {{/data/db/}} by default). Journal files are named {{j.0}}, {{j.1}}, etc. When a journal file reached 1GB in size ...

[Joyent]

K

[Kernel class rule
Design guide
possible but
a class ...

[Kernel code styl
basics Use s
{{varNames}

[Kernel concurre
All concurren
mutexes with
example the

[Kernel exceptio
several differ
Operationfat
require an er

[Kernel logging
Basic Rules
that). See v

[Kernel string ma
string manipu
each manipu

quickest start, you can use the Joyent SmartMachine for MongoDB Appliance http://wiki.joyent.com/display/jpc2/JoyentMongoDBSmartMachine For installing MongoDB on a Joyent Node Smart Machine, see this article http://wiki.joyent.com/display/node/InstallingMongoDBonaNodeSmartMachine The prebuilt DOCS:Downloads MongoDB Solaris ...

JS Benchmarking Harness
CODE: db.foo.drop(); db.foo.insert( ) ops = { op : "findOne" , ns : "test.foo" , query : } , { op : "update" , ns : "test.foo" , query : , update : { $inc : } } for (x = 1; x<=128; x=2){ res = benchRun( ) print("threads: " x "\t queries/sec: " res.query ...

JVM Languages

L

Language Support

Legal Key Names
Key names in inserted documents are limited as follows: The '$' character must not be the first character in the key name. The '.' character must not appear anywhere in the key name

Licensing
Database: Free Software Foundation's GNU AGPL v3.0 http://www.fsf.org/licensing/licenses/agpl3.0.html. Commercial licenses are also available from 10gen mailto:info@10gen.com, including free evaluation licenses. Drivers: mongodb.org supported drivers: Apache License ...

List of Database Commands
iframe src ="http://api.mongodb.org/internal/current/commands.html" width="100%" height="1000px" frameborder="0"> List of MongoDB Commands </iframe> See the Commands page for details on how to invoke a command ...

List of helper functions
nonexhaustive list of helper functions for use in the C stream syntax. An exhaustive list is here: bsonmisc.h https://github.com/mongodb/mongo/blob/master/src/mongo/bson/bsonmisc.h Typical example of stream syntax: BSONObj p = BSON ...

Locking

Locking in Mongo

Logging
MongoDB outputs some important information to stdout while its running. There are a number of things you can do to control this Command Line Options {{\quiet}} less verbose output (more details quiet) {{v}} more verbose output. use ...

M

Manual
MongoDB ma
The Interacti

MapReduce
Map/reduce i
something lik
would have u

Master Slave
Use Replica
more robust
configuration

Memory manage
Overall guide
class such as
backward co

min and max Qu
min()}} and {{
keys betwee
conjunction.

Misc

mongo - The Inte
MongoDB dis
allows you to
https://develo

Mongo Administ

Mongo Concepts
See the Man
files and very
environment

Mongo Database

Mongo Develope

Mongo Documen
page provide
Writing Style
Writing Style

Mongo Driver Re
highlevel list
should be tak
to learn abou

Mongo Extended
Mongo's RES
types that do
interface sup

Mongo Metadata
dbname>.sys
collections in
Additional na

Mongo Query La
Queries in M
clause: > db.

Mongo Usage Ba

Mongo Wire Pro
Introduction
database ser

should conne
Mongo-Based A
Please list ap
here\!
Center Applic
MongoDB - A De
MongoDB Comm
10gen.png! h
subscriptions
http://www.10
http://www.10
MongoDB Data I
tutorial discu
mapper.&nbs
end, we'll be
MongoDB kerne
Coding conve
http://googles
the case in th
MongoDB Langu
MongoDB Maste
sorted alphab
joined the tea
pages of Sou
MongoDB Monit
MongoDB Mo
interface feat
setup and co
MongoDB on Az
MongoDB Re
http://groups.
#mongodb ar
MongoDB User G
MongoDB Us
to network, a
http://www.10
MongoDB, Couc
pending... Co
DocumentOr
string,numbe
mongoexport
mongoexport
for the query
mongoperf
mongoperf is
disk i/o's. The
echo ...
mongosniff
Unix releases
fairly low leve
mongosniff h
mongostat
Use the mon
mongostat he
of updates ...
Monitoring
Monitoring and D
Admin UIs m
issues it is a
http://docs.m
movePrimary Co
movePrimary
collections in
existing data
Moving Chunks
any given tim

Parsing Stack Traces
: addr2line addr2line e mongod ifC <offset> cfilt You can use {{cfilt}} to demangle function names by pasting the whole stack trace to stdin. Finding the right binary To find the binary you need: Get the commit at the header of any ...

Perl Language Center
: Installing Start a MongoDB server instance ({{mongod}}) before installing so that the tests will pass. The {{mongod}} cannot be running as a slave for the tests to pass. Some tests may be skipped, depending on the version of the database you are running. CPAN $ sudo cpan MongoDB ...

Perl Tutorial

Philosophy
: Design Philosophy !featuresPerformace.png align=right! New database technologies are needed to facilitate horizontal scaling of the data layer, easier development, and the ability to store order(s) of magnitude more data than was used in the past. A nonrelational approach is the best path to database ...

PHP - Storing Files and Big Data

PHP Language Center
: Using MongoDB in PHP To access MongoDB from PHP you will need: The MongoDB server running the server is the "mongo{}d" file, not the "mongo" client (note the "d" at the end) The MongoDB PHP driver installed Installing the PHP Driver \NIX Run ...

PHP Libraries, Frameworks, and Tools
: PHP community has created a huge number of libraries to make working with MongoDB easier and integrate it with existing frameworks. Libraries and Frameworks CakePHP MongoDB datasource https://github.com/ichikaway/cakephpmongodb for CakePHP. There's also an introductory blog post http ...

Production Deployments
: you're using MongoDB in production, we'd love to list you here\! Please complete this web form https://www.10gen.com/weusemongodb and we will add you. Archiving <DIV mcestyle="textalign:center;margin:0" style="margin: 0pt ...

Production Notes
: Backups Backups Doc Page DOCS:Backups Import Export Tools TCP Port Numbers Default TCP port numbers for MongoDB processes: Standalone {{mongod}} : 27017 {{mongos}} : 27017 shard server DOCS:Architecture and Components ({{mongod \shardsvr ...

Project Ideas
: you're interested in getting involved in the MongoDB community (or the open source community in general) a great way to do so is by starting or contributing to a MongoDB related project. Here we've listed some project ideas for you to get started on. For some of these ideas ...

PyMongo and mod_wsgi

Python Language Center
: an overview of the available tools for using Python with MongoDB. Those wishing to skip to more detailed discussion should check out the Python Driver Tutorial http://api.mongodb.org/python/current/tutorial.html. Python Driver PyMongo is the recommended way to work ...

Python Tutorial

Querying
: One of Mong
require any s

Querying and nu
: null}} value ir
the following
Query #2 > d

Quickstart
: Quickstart O:
Guides Cons
MongoDB or

Quickstart OS X
: Install Mongc
and on the M
Consider all

Quickstart Unix
: Consider one
for getting st
http://docs.m

Quickstart Windo
: Download Tr
recommende
to run that ve

R

Rails - Getting Started
: Using Rails 3? See Rails 3 Getting Started This tutorial describes how to set up a simple Rails application with MongoDB, using MongoMapper as an object mapper. We assume you're using Rails versions prior to 3.0 ...

Rails 3 - Getting Started
: difficult to use MongoDB with Rails 3. Most of it comes down to making sure that you're not loading ActiveRecord and understanding how to use Bundler http://github.com/carlhuda/bundler/blob/master/README.markdown, the new Ruby dependency manager. Install the Rails ...

Recommended Configurations
: Replica sets support quite a few options, and it can be confusing determining the best configuration. Here following a few suggestions. One data center If you have just one data center, then the most economical setup is a threenode replica set ...

Recommended Production Architectures

Reconfiguring a replica set when members are down
: One can easily reconfigure a replica set when some members are down as long as a majority is still up. In that case, simply send the reconfig command to the current primary. DOCS:Reconfiguring when Members are Up If there is no majority (i.e. a majority of nodes ...

Reconfiguring when Members are Up
: v1.8 Use the rs.reconfig() helper in the shell. (Run "rs.reconfig" in the shell with no parenthesis to see what it does.) $ mongo > // example : give 1st member priority 2 > cfg = rs.conf() > cfg.members0.priority = 2 > rs.reconfig(cfg) Earlier versions (1.6) You can ...

RedHat OpenShift
: OpenShift https://openshift.redhat.com/app/ is a Platform as a Service (PaaS) offering from RedHat, which provides support for rapid deployment and automatic scalability support for web applications developed with Java

S

Scala Language
: Casbah http:
and extensio
supports seri

Schema Design
: Schema desi
the first step
relationship r

scons
: Use scons ht
details. Run
...

Scripting the she
: MongoDB sh
(.js) you can

Searching and R

Security and Aut
: MongoDB in
runs the data
Older sharde

Server-side Cod
: Mongo supp
map/reduce

EE, Node.js, Python, PHP, Perl and Ruby, and several databases ...

removeshard command
Removing a Shard The {{removeshard}} command will remove a shard from an existing cluster. It has two phases which are described below. Starting Before a shard can be removed, we have to make sure that all the chunks and databases that once lived ...

Removing
Removing Objects from a Collection To remove objects from a collection, use the {{remove()}} function in the mongo shell mongo The Interactive Shell. (Other drivers offer a similar function, but may call the function "delete". Please check your driver's documentation ...

renameCollection Command
renameCollection command cannot be used with sharded collections. This command can be used to rename an existing collection. Shell: > db.oldname.renameCollection("newname") From drivers without a helper method one can invoke the generic command syntax: > db.runCommand(This command is executed ...

Replica Pairs
Replica pairs should be migrated to their replacement, Replica Sets. Setup of Replica Pairs Mongo supports a concept of replica pairs. These databases automatically coordinate which is the master and which is the slave at a given point in time. At startup, the databases will negotiate which is master ...

Replica Set Admin UI
mongod}} process includes a simple administrative UI for checking the status of a replica set. To use, first enable {{\rest}} from the {{mongod}} command line. The rest port is the db port plus 1000 (thus, the default is 28017). Be sure this port is secure ...

Replica Set Authentication
Authentication was added in 1.7.5 Replica set authentication works a little differently from singleserver authentication, so that each member can automatically authenticate itself to the other members of the set. See the main docs on authentication Security and Authentication#Replica Set ...

Replica Set Commands
Shell Helpers rs.help() show help rs.status() rs.initiate() initiate with default settings rs.initiate(cfg) rs.add(hostportstr) add a new member to the set rs.add(membercfgobj) add a new member to the set rs.addArb(hostportstr) add a new member which ...

Replica Set Configuration
Command Line Each {{mongod}} participating in the set should have a {{\replSet}} parameter on its command line. The syntax is mongod replSet setname rest {}setname is the logical name of the set. Use the {{rest}} command line parameter when using replica ...

Replica Set Design Concepts
replica set has at most one primary at a given time. If a majority of the set is up, the most uptodate secondary will be elected primary. If a majority of the set is not up or reachable, no member will be elected primary. There is no way to tell (from the set's point of view) the difference ...

Replica Set FAQ
See Also: Intro FAQ FAQ DOCS:Developer FAQ DOCS:Sharding FAQ How long does replica set failover take? It may take 1030 seconds for the primary to be declared down by the other members and a new primary elected ...

Replica Set Internals
page contains notes on the original MongoDB replica set design. While the concepts still apply, this page is not kept perfectly uptodate; consider this page historical rather than definitive. Design Concepts Check out the Replica Set Design Concepts ...

Replica Set Semantics
MongoDB Java driver handles failover in replicated setups with tunable levels of transparency to the user. By default, a {{Mongo}} connection object will ignore failures of secondaries, and only reads will throw {{MongoExceptions}} when the primary node is unreachable. The level of exception reporting ...

Replica Set Tutorial
v1.6 This tutorial will guide you through a basic replica set initial setup. Given the tutorial is an example and should be easy to try, it runs several mongod processes on a single machine (in the real world one would use several machines ...

Replica Set Versions and Compatibility
Features Feature Version Slave delay v1.6.3 Hidden v1.7 replSetFreeze and replSetStepDown v1.7.3 Replicated ops in {{mongostat}} v1.7.3 Syncing from secondaries v1.8.0 Authentication v1.8.0 Replication from nearest server (by ping time) v2.0.0 Syncing 1.8.x ...

Replica Sets
Overview Replica sets are a form of asynchronous master/slave replication DOCS:Master Slave, adding automatic failover and automatic recovery of member nodes. A replica set consists of two or more nodes that are copies of each other. (i.e.: replicas) The replica ...

Replica Sets - Basics
production use you will want a minimum of three nodes in the replica set. Either: 2 full nodes and 1 arbiter 3 full nodes To avoid a single point of failure, these nodes must be on different computers. It is standard to have at least 2 ...

Replica Sets - Oplog
Replication of data between nodes is done using a special collection known as the {{oplog}}. See also : DOCS:Replication Oplog Length. The basic replication process # All write operations are sent to the server (Insert,

{{db.eval()}} l

Server-Side Proc

serverStatus Co
serverStatus
db.command
Field Exampl

setParameter Co
Setting Runti
the Comman
pay attention

Shard Ownershi
shard owners
copy of the o
shard ...

Sharding
MongoDB sc
balancing for
nodes No sin

Sharding Admini
Here we pres
see the docs
mongos proc

Sharding and Fa
properlyconfi
failure scena
routing proce

Sharding Config
Sharding cor
current meta
(chunkSize/b

Sharding Design
concepts cor
be either a si
region ...

Sharding FAQ
See also: Intr
nonsharded l

Sharding Interna
section incluc
documentatio

Sharding Introdu
MongoDB su
applications t
managing fai

Sharding Limits
Security Auth
https://jira.mo
must be run i

Sharding Use Ca
What specific
here for discu
videos ...

Simple Initial Sha
Overview Thi
(East=primar

Slide Gallery

Slides and Video
Upcoming Ev
Some past w
http://www.1
Awareness ..

Smoke Tests
Test Organiz
{{jstests/shar

Sorting and Natu
Natural order

Update, Remove, DB/Collection/Index ...

Replica Sets - Priority
default, all full nodes in a replica set have equal priority. If a primary steps down, all other members are electable as the new primary. That is, each node by default has {{priority:1}}. Arbiters DOCS:Adding an Arbiter have ...

Replica Sets - Rollbacks
Overview The classic example of a rollback occurs when you have two replicas (a primaryA and secondaryB) and the secondary (B) is not up to date (replication is behind). If the primary (A) fails (or is shutdown) before B is uptodate and B becomes primary, then there is data which B ...

Replica Sets - Voting
Each replica sets contains only one primary node. This is the only node in the set that can accept write commands (insert/update/delete). The primary node is elected by a consensus vote of all reachable nodes. Consensus Vote For a node to be elected ...

Replica Sets in Ruby
Here follow a few considerations for those using the Ruby driver Ruby Tutorial with MongoDB and replica sets DOCS:Replica Sets. Setup First, make sure that you've configured and initialized a replica set. Connecting to a replica set from the Ruby ...

Replica Sets Limits
set can contain A maximum of 12 members A maximum of 7 members that can vote Typically the set configuration can be changed only when a majority can be established. Limits on config changes to sets at first. Especially when a lot of set ...

Replica Sets slaveDelay
Overview A replica set member can be configured with a {{slaveDelay:<secs>}} option. When {{slaveDelay}} is specified, the member will intentionally remain behind current in its application of replicated writes. This is typically used to keep a "rolling backup" that is a certain amount of time in the past ...

Replica Sets Troubleshooting
can't get local.system.replset config from self or any seed (EMPTYCONFIG) Set needs to be initiated. Run {{rs.initiate()}} from the shell. If the set is already initiated and this is a new node, verify it is present in the replica set's configuration and there are no typos in the host names: > // send ...

Replication
MongoDB supports asynchronous replication of data between servers for failover and redundancy. Only one server (in the set/shard) is active for writes (the primary, or master) at a given time this is to allow strong consistent (atomic) operations. One can optionally send read ...

Replication Internals
master mongod instance, the {{local}} database will contain a collection, {{oplog.$main}}, which stores a highlevel transaction log. The transaction log essentially describes all actions performed by the user, such as "insert this object into this collection." Note that the oplog is not a lowlevel redo log ...

Replication Oplog Length
Replication uses an operation log ("oplog") to store write operations. These operations replay asynchronously on other nodes. The length of the oplog is important if a secondary is down. The larger the log, the longer the secondary can be down and still recover. Once the oplog has ...

Resyncing a Very Stale Replica Set Member
Error RS102 MongoDB writes operations to an oplog. For replica sets this data is stored in collection local.oplog.rs. This is a capped collection and wraps when full "RRD"style. Thus, it is important that the oplog collection is large enough to buffer ...

Retrieving a Subset of Fields
default on a find operation, the entire object is returned. However we may also request that only certain fields be returned. This is somewhat analogous to the list of \column specifiers in a SQL SELECT statement (projection). // select z from things where x ...

Roadmap
Please see jira
https://jira.mongodb.org/browse/SERVER#selectedTab=com.atlassian.jira.plugin.system.project%3Aroadmappanel

Ruby External Resources
number of good resources appearing all over the web for learning about MongoDB and Ruby. A useful selection is listed below. If you know of others, do let us know. Screencasts Introduction to MongoDB Part I http://www.teachmetocode.com/screencasts ...

Ruby Language Center
an overview of the available tools and suggested practices for using Ruby with MongoDB. Those wishing to skip to more detailed discussion should check out the Ruby Driver Tutorial http://api.mongodb.org/ruby/current/file.TUTORIAL.html, Getting started with Rails Rails ...

Ruby Tutorial
tutorial gives many common examples of using MongoDB with the Ruby driver. If you're looking for information on data modeling, see MongoDB Data Modeling and Rails. Links to the various object mappers are listed on our object mappers page http://www.mongodb.org ...

parameters,
because, alth

Source Code
Source for M
Mongo Datal
http://github.

Spec, Notes and
Assume that
time. See Als

Splitting Shard C
MongoDB us
key range) to
no one chunl

SQL to Aggregat
SQL term Mo
can try the ag
...

SQL to Mongo M
MySQL exec
Components
table collectio

SQL to Shell to C
MongoDB qu
as SQL, Mon
index key ...

SSD
We are not e
welcome. Mu
Endurance ..

Starting and Stop
MongoDB is
on those opti
{{mongod}} is

Storing Data

Storing Files

Structuring Data

T

Tailable Cursors

U

Ubuntu and Deb

Tailable cursors are only allowed on capped collections and can only return objects in natural order http://www.mongodb.org/display/DOCS/SortingandNaturalOrder. Tailable queries never use indexes. A tailable cursor "tails" the end of a capped collection, much like ...

Technical Support
Free Support Forum http://groups.google.com/group/mongodbuser IRC Chat and Support irc://irc.freenode.net/#mongodb Commercial Support http://www.10gen.com/support

The Database and Caching
relational databases, object caching is usually a separate facility (such as memcached), which makes sense as even a RAM page cache hit is a fairly expensive operation with a relational database (joins may be required, and the data must be transformed into an object representation ...

The Linux Out of Memory OOM Killer
Linux out of memory killer kills processes using too much memory. On a kill event you will see a line such as the following in the system log file: Feb 13 04:33:23 hostm1 kernel: 279318.262555 mongod invoked oomkiller ...

Timestamp data type
normal Date Dates datatype. This is a special type for internal MongoDB use. BSON http://bsonspec.org/ includes a timestamp data type with special semantics in MongoDB. Timestamps are stored as 64 bit values which, on a single {{mongod}}, are guaranteed unique. The first ...

Too Many Open Files
you receive the error "too many open files" or "too many open connections" in the mongod log, there are a couple of possible reasons for this. First, to check what file descriptors are in use, run lsof (some variations shown below): lsof grep ...

Tools and Libraries
Talend Adapters https://github.com/adrienmogenet

TreeNavigation
Follow @mongodb http://www.twitter.com/mongodb MongoDB Pittsburgh http://www.10gen.com/events/mongodbpgh May 15 MongoNYC http://www.10gen.com/events/mongonyc May 23 MongoDB Paris http://www.10gen.com/events/mongodbparis Jun 14 MongoDB UK ...

Trees in MongoDB
best way to store a tree usually depends on the operations you want to perform; see below for some different options. In practice, most developers find that one of the "Full Tree in Single Document", "Parent Links", and "Array of Ancestors" patterns ...

Troubleshooting
mongod process "disappeared" Scenario here is the log ending suddenly with no error or shutdown messages logged. On Unix, check /var/log/messages: $ sudo grep mongod /var/log/messages $ sudo grep score /var/log/messages Socket ...

Troubleshooting MapReduce
Tips on troubleshooting map/reduce. Troubleshooting the {{map}} function We can troubleshoot the map function in the shell by defining a test {{emit}} function in the shell and having it print out trace information. For example suppose we have some data: > db.articles.find ...

Troubleshooting the PHP Driver

Tutorial
Running MongoDB First, run through the Quickstart guide for your platform to get Mongo installed. Getting A Database Connection Let's now try manipulating the database with the database shell DOCS:mongo The Interactive Shell . (We could perform similar ...

Tweaking performance by document bundling during schema design
Note: this page discusses performance tuning aspects if you are just getting started skip this for later. If you have a giant collection of small documents that will require significant tuning, read on. During schema design DOCS:Schema Design one consideration ...

two-phase commit
common problem with nonrelational database is that it is not possible to do transactions across several documents. When executing a transaction composed of several sequential operations, some issues arise: Atomicity: it is difficult to rollback changes by previous operations if one fails. Isolation: changes ...

Please read
generally fres
the name mo

UI
Spec/require
indexsize, clo
...

Updates

Updating
MongoDB su
{{update()}} r
use ...

Updating Data in
Updating a D
to save a nev
with the exar

Upgrading from a
mongod}} pro
so yet, feel fr
familiarize yo

Upgrading to 1.8
First, upgrad
primaries. 1.

Upgrading to Re
Upgrading Fr
idea\!). First,

Use Case - Sess
MongoDB is
store the ses
updates fast,

Use Cases
See also the
Shutterfly, fo
http://blog.m

User Feedback
I just have to
get a long co
mongodbuse

Using a Large N
technique on
single collect
that key may

Using Multikeys
One way to v
feature where
db.foo.insert(

V

v0.8 Details
Existing Core Functionality Basic Mongo database functionality: inserts, deletes, queries, indexing. Master / Slave Replication Replica Pairs Serverside javascript code execution New to v0.8 Drivers for Java, C, Python, Ruby. db shell utility ...

Validate Command
Use this command to check that a collection is valid (not corrupt) and to get various statistics. This command scans the entire collection and its indexes and will be very slow on large datasets. option description full Validates everything new in 2.0.0 scandata Validates basics (index ...

Verifying Propagation of Writes with getLastError
Please read the getLastError command page first. A client can await a write operation's replication to N servers (v1.6). Use the getlasterror command with the parameter {{w}}: // examples: db.runCommand( ) db.runCommand( ) db.runCommand( ) db.runCommand( ) If {{w}} is not set, or equals one, the command ...

W

What is the Com
MongoDB all
from differen
...

What's New by V
summary of I
details. 1.4 G
directoryperd

When to use Gri
page is unde
better than m

Why are my data

Version Numbers MongoDB uses the oddnumbered versions for development releases http://en.wikipedia.org/wiki/Softwareversioning#Oddnumberedversionsfordevelopmentreleases. There are 3 numbers in a MongoDB version: A.B.C A is the major version. This will rarely change and signify very large changes B is the release number. This will include many changes ... Video & Slides from Recent Events and Presentations Table of Contents: MongoDB Conferences Oneday conferences hosted by 10gen http://www.10gen.com/. 10gen develops and supports MongoDB. MongoUK Video (June 2010) http://skillsmatter.com/event/cloudgrid/mongouk MongoFR Video (June 2010) http://lacantine.ubicast.eu/categories ... Viewing and Terminating Current Operation View Operation(s) in Progress > db.currentOp(); { inprog: { "opid" : 18 , "op" : "query" , "ns" : "mydb.votes" , "query" : " " , "inLock" : 1 } } > > // to include idle connections in report: > db.currentOp(true); Fields: opid an incrementing operation number. Use with killOp(). active ... Virtualization Generally MongoDB works very well in virtualized environments, with the exception of OpenVZ. EC2 Compatible. No special configuration requirements. VMWare Some suggest not using overcommit as they may cause issues. Otherwise compatible. Cloning a VM is possible. For example you might ... Visit the 10gen Offices Bay Area 10gen's West Coast office is located in downtown Palo Alto. 555 University Ave. Palo Alto, CA 94301 \\ <iframe width="495" height="395" frameborder="0" scrolling="no" marginheight="0" marginwidth="0" src ... VMware CloudFoundry MongoDB is a supported service on VMware's Cloud Foundry http://www.cloudfoundry.com/. Starting a MongoDB service {{vmc createservice mongodb name MyMongoDB}} Once you create a MongoDB service, you can bind and use it inside of Cloud Foundry applications. Developing ...	Why Replica Set Replica sets features shou Why so many "C Windows Windows Qu Windows for page. The M Windows Servic mongod.exe mongos.exe Windows ser Wireshark Supp Wireshark ht http://wiki.wir on specific v Working with Mo Writing Drivers a See Also DO Writing tests We have thre succeed. For db.jstestscur
X	**Y** Your Go-to Reso Tips for Runr meet and lea
Z	**!@#$**

Alerts

This page lists critical alerts and advisories for MongoDB. This page is a work in progress and will be enhanced over time.

See http://jira.mongodb.org/ for a comprehensive list of bugs and feature requests.

Data Integrity Related

- Documents may be missing on a replication secondary after initial sync if a high number of updates occur during the sync that move the document (i.e., the documents are growing).
    - https://jira.mongodb.org/browse/SERVER-3956 Replica set case. Fixed: 1.8.4, 2.0.1
    - https://jira.mongodb.org/browse/SERVER-4270 Master/slave case. Fixed: 1.8.5, 2.0.2

- When stepping down a replica set primary, the primary's connection to clients is not automatically closed in 2.0.1. This means that if the primary steps down while an application is issuing fire-and-forget write, the application won't necessarily fail over until it issues a query. Applications that only issue safe writes are not affected by this. The issue is fixed in 2.0.2. See https://jira.mongodb.org/browse/SERVER-4405 for details.

Security Related

- Limit access of __system when running --auth without replica sets.
    - https://jira.mongodb.org/browse/SERVER-3666 Fixed: 1.8.4

Make sure to subscribe to http://groups.google.com/group/mongodb-announce for important announcements.