

.bookmarks	6
1.1 Development Cycle	6
Creating and Deleting Indexes	6
C Sharp Language Center	6
Diagnostic Tools	6
Django and MongoDB	6
Getting Started	7
International Documentation	7
Monitoring	7
Older Downloads	7
PyMongo and mod_wsgi	7
Python Tutorial	7
Recommended Production Architectures	8
v0.8 Details	8
Building SpiderMonkey	8
Documentation	8
Dot Notation	9
Dot Notation	
Getting the Software	9
Language Support	9
Mongo Administration Guide	9
Working with Mongo Objects and Classes in Ruby	9
MongoDB Language Support	10
Community Info	10
Internals	10
TreeNavigation	10
Old Pages	11
Storing Data	11
Indexes in Mongo	11
HowTo	11
Searching and Retrieving	11
Locking	11
Mongo Developers' Guide	11
Locking in Mongo	11
Mongo Database Administration	12
Mongo Concepts and Terminology	12
MongoDB - A Developer's Tour	12
Updates	12
Structuring Data for Mongo	12
Design Overview	12
Document-Oriented Datastore	12
Why so many "Connection Accepted" messages logged?	
Why are my datafiles so large?	13
Storing Files	13
Introduction - How Mongo Works	13
Optimizing Mongo Performance	13
Mongo Usage Basics	13
Server-Side Processing	13
Home	14
Introduction	14
Quickstart	16
Quickstart OS X	16
Quickstart Unix	17
Quickstart Windows	18
Downloads	20
1.0 Changelist	21
1.2.x Release Notes	21
1.4 Release Notes	22
1.6 Release Notes	23
1.8 Release Notes	24
Upgrading to 1.8.0	25
CentOS and Fedora Packages	27
Ubuntu and Debian packages	28
Version Numbers	29
Drivers	29
C Language Center	31
C Tutorial	31
CSharp Language Center	37
CSharp API Documentation	38
CSharp Community Projects	38
CSharp Driver Serialization Tutorial	38
CSharp Driver Tutorial	49
Tools and Libraries	69
Driver Syntax Table	69
Javascript Language Center	70

node.JS	70
JVM Languages	70
Python Language Center	71
PHP Language Center	71
Installing the PHP Driver	72
PHP Libraries, Frameworks, and Tools	72
PHP - Storing Files and Big Data	74
Troubleshooting the PHP Driver	74
Ruby Language Center	74
Ruby Tutorial	76
Replica Sets in Ruby	81
GridFS in Ruby	83
Rails - Getting Started	85
Rails 3 - Getting Started	87
MongoDB Data Modeling and Rails	89
Ruby External Resources	93
Frequently Asked Questions - Ruby	94
Java Language Center	97
Java Driver Concurrency	98
Java - Saving Objects UsingDBObject	99
Java Tutorial	99
Java Types	104
C++ Language Center	106
BSON Arrays in C++	106
C++ BSON Library	107
C++ Driver Compiling and Linking	109
C++ Tutorial	110
Connecting	114
Perl Language Center	114
Contributing to the Perl Driver	115
Perl Tutorial	116
Online API Documentation	117
Writing Drivers and Tools	117
Overview - Writing Drivers and Tools	117
bsonspec.org	117
Mongo Driver Requirements	117
Spec, Notes and Suggestions for Mongo Drivers	121
Feature Checklist for Mongo Drivers	121
Conventions for Mongo Drivers	121
Driver Testing Tools	122
Mongo Wire Protocol	122
BSON	127
Mongo Extended JSON	129
GridFS Specification	131
Implementing Authentication in a Driver	133
Notes on Pooling for Mongo Drivers	134
Driver and Integration Center	137
Connecting Drivers to Replica Sets	137
Error Handling in Mongo Drivers	137
Developer Zone	138
cookbook.mongodb.org	139
Tutorial	139
Manual	145
Connections	146
Databases	148
Commands	148
Mongo Metadata	157
Collections	157
Capped Collections	158
Using a Large Number of Collections	160
Data Types and Conventions	160
Timestamp Data Type	161
Internationalized Strings	161
Object IDs	161
Database References	163
GridFS	165
When to use GridFS	165
Indexes	165
Using Multikeys to Simulate a Large Number of Indexes	170
Geospatial Indexing	170
Indexing as a Background Operation	174
Multikeys	174
Indexing Advice and FAQ	176
Inserting	179
Legal Key Names	180

Schema Design	180
Trees in MongoDB	182
Optimization	185
Optimizing Object IDs	188
Optimizing Storage of Small Objects	188
Query Optimizer	189
Querying	189
Mongo Query Language	191
Retrieving a Subset of Fields	191
Advanced Queries	192
Dot Notation (Reaching into Objects)	200
Full Text Search in Mongo	202
min and max Query Specifiers	203
OR operations in query expressions	204
Queries and Cursors	205
Server-side Code Execution	207
Sorting and Natural Order	210
Aggregation	210
Removing	214
Updating	214
Atomic Operations	219
findAndModify Command	221
two-phase commit	224
Updating Data in Mongo	227
MapReduce	228
Troubleshooting MapReduce	233
Data Processing Manual	236
mongo - The Interactive Shell	236
Scripting the shell	238
Overview - The MongoDB Interactive Shell	239
dbshell Reference	241
Developer FAQ	244
Do I Have to Worry About SQL Injection	245
How does concurrency work	246
SQL to Mongo Mapping Chart	247
What is the Compare Order for BSON Types	249
Admin Zone	250
Journaling	251
Journaling Administration Notes	252
Production Notes	253
iostat	254
Replication	255
About the local database	255
Replica Set Authentication	256
Verifying Propagation of Writes with getLastError	257
Replica Sets	257
Moving or Replacing a Member	258
Replica Set Design Concepts	258
Replica Set Tutorial	259
Replica Set Configuration	263
Upgrading to Replica Sets	267
Replica Set Admin UI	270
Replica Set Commands	270
Connecting to Replica Sets from Clients	273
Replica Set FAQ	273
Replica Sets Troubleshooting	274
Replica Sets Limits	275
Replica Set Internals	275
Master Slave	278
Replica Pairs	282
Replication Oplog Length	284
Halted Replication	284
Sharding	286
Sharding Introduction	286
Configuring Sharding	291
A Sample Configuration Session	294
Choosing a Shard Key	295
Upgrading from a Non-Sharded System	297
Sharding Administration	298
Backing Up Sharded Cluster	301
Sharding and Failover	302
Sharding Limits	302
Sharding Internals	303
Moving Chunks	304
Sharding Config Schema	305

Sharding Design	307
Sharding Use Cases	308
Shard Ownership	308
Splitting Chunks	309
Sharding FAQ	310
Hosting Center	311
Amazon EC2	312
Joyent	313
Monitoring and Diagnostics	314
Checking Server Memory Usage	315
Database Profiler	317
Munin configuration examples	319
serverStatus	322
Http Interface	324
mongostat	327
mongosniff	328
Backups	328
EC2 Backup & Restore	329
How to do Snapshotted Queries in the Mongo Database	334
Import Export Tools	335
Durability and Repair	339
Security and Authentication	341
Admin UIs	343
Starting and Stopping Mongo	349
Logging	350
Command Line Parameters	351
File Based Configuration	352
GridFS Tools	354
DBA Operations from the Shell	355
Architecture and Components	356
Windows	356
Troubleshooting	357
Excessive Disk Space	357
Out of Memory OOM Killer	359
Too Many Open Files	359
Contributors	360
JS Benchmarking Harness	360
MongoDB kernel code development rules	361
Git Commit Rules	361
Kernel class rules	361
Kernel code style	362
Kernel concurrency rules	364
Kernel exception architecture	364
Kernel Logging	365
Kernel string manipulation	365
Memory Management	366
Writing Tests	366
Project Ideas	366
UI	368
Source Code	368
Building	368
Building Boost	368
Building for FreeBSD	369
Building for Linux	370
Building for OS X	371
Building for Solaris	375
Building for Windows	375
Boost 1.41.0 Visual Studio 2010 Binary	375
Boost and Windows	375
Building the Mongo Shell on Windows	376
Building with Visual Studio 2008	377
Building with Visual Studio 2010	378
Building Spider Monkey	379
scons	381
Database Internals	382
Caching	382
Durability Internals	382
Parsing Stack Traces	383
Cursors	384
Error Codes	384
Internal Commands	384
Replication Internals	385
Smoke Tests	386
Pairing Internals	388
Contributing to the Documentation	388

Emacs tips for MongoDB work	388
Mongo Documentation Style Guide	388
Community	390
MongoDB Commercial Services Providers	391
Visit the 10gen Offices	393
User Feedback	393
Job Board	395
About	395
Philosophy	395
Use Cases	395
How MongoDB is Used in Media and Publishing	396
Use Case - Session Objects	398
Production Deployments	399
Mongo-Based Applications	422
Events	423
Video & Slides from Recent Events and Presentations	425
Slide Gallery	428
Articles	428
Benchmarks	429
FAQ	429
Product Comparisons	430
Interop Demo (Product Comparisons)	430
MongoDB, CouchDB, MySQL Compare Grid	430
Comparing Mongo DB and Couch DB	431
Licensing	432
International Docs	433
Books	433
Slides and Video	434
Doc Index	435

.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

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

**Redirection Notice**

This page should redirect to [Manual](#).

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](#) | [Downloads](#) | [Tutorial](#) | [Features](#)
[SQL to Mongo Mapping Chart](#)

Development

- [Manual](#)
- [C](#) | [C#](#) | [C++](#) | [C# & .NET](#) | [ColdFusion](#) | [Erlang](#) | [Factor](#) | [Java](#) | [Javascript](#) | [PHP](#) | [Python](#) | [Ruby](#) | [Perl](#) | [More...](#)

Production

[Production Notes](#) | [Replication](#) | [Sharding](#) | [Security](#) | [Backup](#)

Support

[Forum](#) | [IRC](#) | [Bug tracker](#) | [Commercial support](#) | [Training](#) | [Consulting](#) | [Hosting](#)

Community

[Blog](#) | [Articles](#) | [Twitter](#) | [Forum](#) | [Facebook](#) | [LinkedIn](#) | [Job Board](#) | [User groups: NY, SF, DC, Chicago, London, Stockholm](#)

Meta

[Use Cases](#) | [Philosophy](#) | [License](#)

Translations

[Deutsch](#) | [Español](#) | [Français](#) | [Italiano](#) | [Português](#) | | |

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 asynchronous writes
- **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.

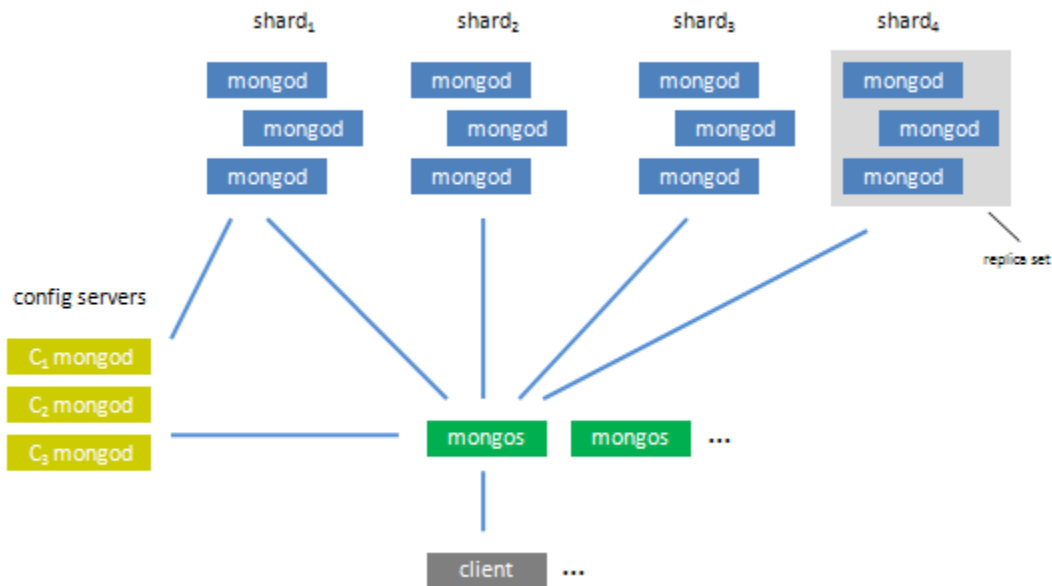
A replica set is one or more servers, each holding copies of the same data. At any given time one is primary and the rest are secondaries. If the primary goes down one of the secondaries takes over automatically as primary. All writes and consistent reads go to the primary, and all eventually consistent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

See Also

- [SQL to Mongo Mapping Chart](#)
- [Tutorial](#)

Quickstart OS X

Install MongoDB

The easiest way to install MongoDB is to use a package manager or the pre-built binaries:

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:

```
$ mkdir -p /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

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!

Learn more

Once you have MongoDB installed and running, [head over to the Tutorial](#).

Quickstart Windows

- [Download](#)
 - [32-bit binaries](#)
 - [64-bit binaries](#)
- [Unzip](#)
- [Create a data directory](#)

- [Run and connect to the server](#)
- [Learn more](#)

Download

The easiest (and recommended) way to install MongoDB is to use the pre-built binaries.

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.

Note: [64-bit is recommended](#), although you must have a 64-bit version of Windows to run that version.

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.

Run and connect to the server

The important binaries for a first run are:

- `mongod.exe` - the database server
- `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
```

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 the 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
> 3+3
6
> db
test
> // the first write will create the db:
> db.foo.insert( { a : 1 } )
> db.foo.find()
{ _id : ..., a : 1 }
```

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

Learn more

- [Tutorial](#)
- [Windows quick links](#)
- [\[Mongo Shell\]](#)

Mongo Shell Info

Downloads

See also [Packages](#).

Version	OS X 32 bit	OS X 64 bit	Linux 32 bit	Linux 64 bit	Windows 32 bit	Windows 64-bit	Solaris i86pc	Solaris 64	Source	Date
Production (Recommended)										
1.4.3	os x 10.5+ os x 10.4	download	download * legacy-static	download * legacy-static	download	download	download	download	tgz zip	5/24/2010
nightly	os x 10.5+ os x 10.4	download	download * legacy-static	download * legacy-static	download	download	download	download	tgz zip	Daily
Previous Release										
1.2.5	os x 10.5+ os x 10.4	download	download * legacy-static	download * legacy-static	download	download	download	download	tgz zip	4/7/2010
nightly	os x 10.5+ os x 10.4	download	download * legacy-static	download * legacy-static	download	download	download	download	tgz zip	Daily
Dev (unstable)										
1.5.3	os x 10.5+ os x 10.4	download	download * legacy-static	download * legacy-static	download	download	download	download	tgz zip	6/17/2010
1.5.x nightly	os x 10.5+ os x 10.4	download	download * legacy-static	download * legacy-static	download	download	download	download	tgz zip	Daily
Archived Releases	list	list	list	list	list	list	list	list	list	

- See [Version Numbers](#)
- The linux legacy-static builds are only recommended for older systems. If you try to run and get a floating point exception, try the legacy-static builds. Otherwise you should use the regular ones.
- Currently the mongod server must run on little-endian cpu (intel) so if you are using a ppc os x, mongod will not work.
- 32-bit builds are limited 2gb of data. See <http://blog.mongodb.org/post/137788967/32-bit-limitations> for more info
- See <http://buildbot.mongodb.org/waterfall> for details of builds and completion times.

Included in Distributions

- The MongoDB database server
- The MongoDB shell
- Backup and restore tools
- Import and export tools
- GridFS tool
- The MongoDB C++ client

Drivers

Information on how to separately download or install the drivers and tools can be found on the [Drivers](#) page.

Language	Packages	Source	API Reference
Python	bundles	github	api
PHP	pecl	github	api
Ruby	gemcutter	github	api
Java	jar	github	api
Perl	cpan	github	api
C++	included in database	github	api

See [Drivers](#) for more information and other languages.

Source Code

[Source code for MongoDB and all drivers](#)

Packages

MongoDB is included in several different package managers:

- For [MacPorts](#), see the **mongodb** and **mongodb-devel** packages.
- For [FreeBSD](#), see the **mongodb** and **mongodb-devel** packages.
- For [Homebrew](#), see the **mongodb** formula.
- For [ArchLinux](#), see the **mongodb** package in the AUR.
- For [Debian](#) and [Ubuntu](#), see [Ubuntu](#) and [Debian](#) packages.
- For [Fedora](#) and [CentOS](#), see [CentOS](#) and [Fedora Packages](#).

Documentation

[Pre-Exported](#)

You can export yourself: [HTML](#), [PDF](#), or [XML](#).

Logos

MongoDB logos are available for download as attachments on this page.

Powered By MongoDB Badges

We've made badges in beige, brown, blue and green for use on your sites that are powered by MongoDB. They are available below and in multiple sizes as [attachments](#) on this page.



Training

If you're just getting started with MongoDB, consider registering for an upcoming [training course](#).

<#comment><#comment><#comment>

1.0 Changelist

Wrote MongoDB. See [documentation](#).

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 $\leq 1.0.x$. If you're already using a version $\geq 1.1.x$ 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 $\leq 1.1.2$ 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>

1.4 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

1.6 Release Notes

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

** 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.

Please 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

1.5.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!

1.8 Release Notes

MongoDB 1.8 is a drop-in replacement for 1.6. To upgrade, simply shutdown mongod and the restart with the new binaries.*

* Please note that you should upgrade to the latest version of whichever driver you're using



The options to the Map/Reduce command have changed in 1.8, breaking compatibility with previous releases. If you use MapReduce, please see [Map Reduce Output Options](#) for more information.

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](#)

Support for 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 now 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](#).

Other Improvements

1.7.6 enhancements

- bug fixes

1.7.5 enhancements

- [journaling](#)
- extent allocation improvements
- improved replica set connectivity for mongos
- getLastError improvements for sharding

1.7.4 enhancements

- mongos will route SLAVE_OK queries to secondaries in replica sets
- [new map/reduce output options](#)
- [sparse indexes](#)

1.7.3 enhancements

- [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 enhancements

- [\\$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 enhancements

- 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 enhancements

- 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

1.7.x Release Notes

- [1.7.6](#)
- [1.7.5](#)
- [1.7.4](#)
- [1.7.3](#)
- [1.7.2](#)
- [1.7.1](#)
- [1.7.0](#)

You can see the full list of changes on [Jira](#)

Upgrading to 1.8.0

First, upgrade your shell (mongo) to the 1.8.0 shell.

Replica Sets

1.8.x slaves **can** replicate from 1.6.x masters.

1.6.x slaves **cannot** replicate from 1.8.x masters.

Thus, the trick is to replace all of your slaves, then the master.

For example, suppose we have a typical replica set with 1 master, 1 slave, 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.

Sharding

1. Turn off the balancer. Wait until any currently occurring migrates have stopped.

```
> 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 server, shut it down and start it back up with the 1.8 binary.
3. For each mongos:
 - Shut down the server
 - Restart with the 1.8 binary
4. For each config server:
 - Shut down the config server
 - 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.

Durability

Returning to 1.6 after using 1.8 durability works fine, as durability does not change anything about the data file format. Suppose you are running 1.8.0 with durability 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 1.8.0, restart with the 1.6 binary.
2. If 1.8.0 shut down uncleanly, start 1.8.0 up again and let the transaction log run to fix any damage that may have been done in the crash. Then shut down 1.8.0 and restart with the 1.6 binary.

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 packages.

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

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 Debian or Ubuntu. We publish 2 distinct packages, named **mongodb-10gen**, **mongodb-10gen-unstable** corresponding to our latest stable release, our latest development release. Each of these packages conflicts with the others, and with the **mongodb** package in Debian/Ubuntu.

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

Installing

To use the packages, add a line to your `/etc/apt/sources.list`, then 'aptitude update' and one of 'aptitude install mongodb-10gen', 'aptitude install mongodb-10gen-unstable'. 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 Ubuntu), add this line 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 Ubuntu), 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/ei0IxyjVEp6EEtbEbWk1Q4dKa0NtiC0DwB8di+L8tlUd
Ra5QYxeyV90C+dqdh34o79enXxT6idHfYYqDdob2/kAPE6vFi4sLmrWIVGCRY++7
RPlZuezPm1sxG1TRAYEsW0VZUE9ofdoQ8xLUZDYN2BSjG80CT2e4orRg1pHgzw2
n3hnWqJNuJS4jxcRJ0xI049THIGUtqBfF8bQoZw8C3Wg/R6pGghUfNjpA6uF9KAH
gnqrC0swZ1/vwIJt9fnvAlzkqLrssYtKH0rMdn5n4g5tJLqY5q/NruHMq2rhoy3r
4MC1w8GTbP7qR83wAyaLJ7xACOKqxBSrDFJABEBAAG0I1JpY2hhcmQgS3JldXRl
ciA8cm1jaGFyZEAXMGdlbi5jb20+iQE4BBMBAGAiBQJLbEHvAhsDBgsJCAcDAgYV
CAIJCgsEFgIDAQIeAQIXgAAKCRCEy+xGfwzrEGXbB/4nrmf/2rEnztReImup3duI
eepzEtW1cv3uHg2oZXGS6S7o5Fsk+amngaWelWkfkSw5La7aH5vL4tKFKUfuaME1
avInDIU/0IEs8jLrdSwq601HowLQcxAhqNPdaGONDtHw56Qhs0Ba8GA6329vLWgZ
ODnXweiNSCDrv3xbIN6IjPyY05AoUkxmJfD0mVtp3u5Ar7kfIw7ieGGxokaHewNL
Xzqcp9rPiUR6dFw2uRvDdVRrXFUPlgVugaHKytm15JpHmQfyzQIMdYXnIz0oofJO
WM/PY1iw+QJZ2M7PnfbtJJeADXIc/Eo0AJDRggh533SjhiCaT6FdPMMk6rCZ5cgL
uQENBEtsQe8BCADlNPIJZVSL2i6H9X19YK4CpEqsjIUGISMB1cDT311WFSnhfuMs
GL9xYRb8dlbyeJFFOyHNKIBmH5ekCvGRfS6qJYpcUQZZcwsJEMqBYQV5cw1efd0B
ek64jfvrsLz8+YhKzn+NI803nyGvpEEWv0hN4hNjwkDhYbXLvAlsqagbnSMf+Htf
3lgCGYa2gLiNiQnKWCsEVAan/Er6KS39WANGXi6ih0yjReBiU8WR6Qh2ylMi2xKw
yHnT0sbWxP0hqALUa7N4AEGCXS/qn+vUz/hcIbt+eUNy45qoZcTT3dZswGfJqknh
RFMIuP1ej7/WY4Ugzes5NG02ecDkDkpJvrSNABEBAAGJAR8EGAECaAKFAktsQe8C
GwwACGkQnsvsRn8M6xABeggaLnkqbqa12L1bgaCgnGGdCiuXB3F6/VFmSQdUKpts
EuqWH6rSp30r67PupzneX++ouh+9WD507gJ0kP3VQJpmXjT/QnN5ANjI4kAtRZUW
qCXlX0xVAeXHL5oiKz0NM23Xc2rNAYfBQY8+SUYrKBaLNBq5m68g8oogX8QD5u2F
x+6C+QK9G2EBDD/NWgkKN3G0xpQ5DTdPHI5/fjwYFs1leIaQjjijJwAifxB/1+w0
VChE2LDVpRXY5uBTefF2guhVYisKY6n5wNdaQpBmA8w17it5Yp8ge0HMMN1A+aZ+6
L6MsuHbG20YDZgAk8eKhvyd0y/pAhZpNuQ82MMGBmcueSA==
=74Cu
-----END PGP PUBLIC KEY BLOCK-----
```

Install

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

```
sudo apt-get update
```

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

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.

Drivers

MongoDB currently has client support for the following programming languages:

mongodb.org Supported

- C
- C#
- C++
- Erlang
- Haskell
- Java
- Javascript
- Perl
- PHP
- Python
- Ruby
- Scala (via Casbah)

Community Supported

- REST
- ActionScript3
 - <http://github.com/argoncloud>
- C
 - [libmongo-client](#)
- C# and .NET
- Clojure
- ColdFusion
 - [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](#)
- Delphi
 - [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
- 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](#)
 - [mongogo](#)
 - [go-mongo](#)
 - [mgo](#)
- Groovy
 - [See Java Language Center](#)
- Javascript
- Lisp
 - <https://github.com/fons/cl-mongo>
- Lua
 - [LuaMongo](#)
- [node.js](#)
- Objective C
 - [NuMongoDB](#)
- PHP
 - [Asynchronous PHP driver using libevent](#)
- PowerShell
 - [Blog post](#)
- Python
- R
 - [RMongo](#) - R client to interface with MongoDB
- 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 JVM Languages](#)
- 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](#)

C Language Center

- [C Driver](#)
 - [Download](#)
 - [Build](#)
- [Notable Projects](#)

C Driver

The **MongoDB C Driver** is the 10gen-supported driver for MongoDB. It's written in pure C. The goal is to be super strict for ultimate portability, no dependencies, and very embeddable anywhere.

- [Tutorial](#)
- [C Driver README](#)
- [Source Code](#)

Download

The C driver is hosted at [GitHub.com](#). Check out the latest version with git.

```
$ git clone git://github.com/mongodb/mongo-c-driver.git
```

Build

Building with gcc:

```
$ gcc --std=c99 -Isrc /path/to/mongo-c-driver/src/*.c YOUR_APP.c
```

Building with scon:

```
$ scon # this will produce libbson.a and libmongoc.a
$ scon --c99 # this will use c99 mode in gcc (recommended)
$ scon test # this will compile and run the unit tests (optional)
$ scon test --test-server=123.4.5.67 # use remote server for tests
```

Notable Projects

NuMongodb – An Objective-C wrapper around the MongoDB C driver. It is intended for use with Nu but may be useful in other Objective-C programming applications.

If you're working on a project that you'd like to have included, let us know.

C Tutorial

- [Writing Client Code](#)
 - [Connecting](#)
 - [BSON](#)
 - [Inserting](#)
 - [Single](#)
 - [Batch](#)
 - [Querying](#)
 - [Simple Queries](#)

- [Complex Queries](#)
 - [Sorting](#)
 - [Hints](#)
 - [Explain](#)
- [Indexing](#)
- [Updating](#)
- [Further Reading](#)

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.

A working C program complete with examples from this tutorial can be found [here](#).

Writing Client Code



For brevity, the examples below are simply inline code.

Connecting

Let's make a tutorial.c file that connects to the database:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "bson.h"
#include "mongo.h"

int main() {
    mongo_connection conn[1]; /* ptr */
    mongo_connection_options opts[1];
    mongo_conn_return status;

    strcpy( opts->host , "127.0.0.1" );
    opts->port = 27017;

    status = mongo_connect( conn, opts );

    switch (status) {
        case mongo_conn_success: printf( "connection succeeded\n" ); break;
        case mongo_conn_bad_arg: printf( "bad arguments\n" ); return 1;
        case mongo_conn_no_socket: printf( "no socket\n" ); return 1;
        case mongo_conn_fail: printf( "connection failed\n" ); return 1;
        case mongo_conn_not_master: printf( "not master\n" ); return 1;
    }

    /* CODE WILL GO HERE */

    mongo_destroy( conn );
    printf( "\nconnection closed\n" );

    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:


```
$ gcc -Isrc --std=c99 /path/to/mongo-c-driver/src/*.c -I /path/to/mongo-c-driver/src/ tutorial.c -o tutorial
$ ./tutorial
connection succeeded
connection closed
$
```

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 bson objects. We use `bson_buffer` to make bson objects, and `bson_iterator` to enumerate bson objects.

Let's now create a BSON "person" object which contains name and age. We might invoke:

```
bson b[1];
bson_buffer buf[1];

bson_buffer_init( buf )
bson_append_string( buf, "name", "Joe" );
bson_append_int( buf, "age", 33 );
bson_from_buffer( b, buf );
```

Use the `bson_append_new_oid()` helper to add an object id to your object. The server will add an `_id` automatically if it is not included explicitly.

```
bson b[1];
bson_buffer buf[1];

bson_buffer_init( buf );
bson_append_new_oid( buf, "_id" );
bson_append_string( buf, "name", "Joe" );
bson_append_int( buf, "age", 33 );
bson_from_buffer( b, buf );
```

`bson_buffer_new_oid( ..., "_id" )` should be at the beginning of the generated object.

When you are done using the object remember to use `bson_destroy()` to free up the memory allocated by the buffer.

```
bson_destroy( b )
```

Inserting

Single

We now save our person object in a persons collection in the database:

```
mongo_insert( conn, "tutorial.persons", b );
```

The first parameter to `mongo_insert` is the pointer to the `mongo_connection` object. The second parameter is the namespace. *tutorial* is the database and *persons* is the collection name. The third parameter is a pointer to the bson "person" object that we created before.

Batch

We can do batch inserts as well:

```

static void tutorial_insert_batch( mongo_connection *conn ) {
    bson *p, **ps;
    bson_buffer *p_buf;
    char *names[4];
    int ages[] = { 29, 24, 24, 32 };
    int i, n = 4;
    names[0] = "Eliot"; names[1] = "Mike"; names[2] = "Mathias"; names[3] = "Richard";

    ps = ( bson ** )malloc( sizeof( bson * ) * n );

    for ( i = 0; i < n; i++ ) {
        p = ( bson * )malloc( sizeof( bson ) );
        p_buf = ( bson_buffer * )malloc( sizeof( bson_buffer ) );
        bson_buffer_init( p_buf );
        bson_append_new_oid( p_buf, "_id" );
        bson_append_string( p_buf, "name", names[i] );
        bson_append_int( p_buf, "age", ages[i] );
        bson_from_buffer( p, p_buf );
        ps[i] = p;
        free( p_buf );
    }

    mongo_insert_batch( conn, "tutorial.persons", ps, n );

    for ( i = 0; i < n; i++ ) {
        bson_destroy( ps[i] );
        free( ps[i] );
    }
}

```

Querying

Simple Queries

Let's now fetch all objects from the persons collection, and display them.

```

static void tutorial_empty_query( mongo_connection *conn ) {
    mongo_cursor *cursor;
    bson empty[1];
    bson_empty( empty );

    cursor = mongo_find( conn, "tutorial.persons", empty, empty, 0, 0, 0 );
    while( mongo_cursor_next( cursor ) ) {
        bson_print( &cursor->current );
    }

    mongo_cursor_destroy( cursor );
    bson_destroy( empty );
}

```

empty is the empty BSON object -- we use it to represent {} which indicates an empty query pattern (an empty query is a query for all objects).

We use bson_print() above to print out information about each object retrieved. bson_print() is a diagnostic function which prints an abbreviated JSON string representation of the object.



mongo_find() returns a mongo_cursor which must be destroyed after use.

Let's now write a function which prints out the name (only) of all persons in the collection whose age is a given value:

```
static void tutorial_simple_query( mongo_connection *conn ) {
    bson query[1];
    bson_buffer query_buf[1];
    mongo_cursor *cursor;

    bson_buffer_init( query_buf );
    bson_append_int( query_buf, "age", 24 );
    bson_from_buffer( query, query_buf );

    cursor = mongo_find( conn, "tutorial.persons", query, NULL, 0, 0, 0 );
    while( mongo_cursor_next( cursor ) ) {
        bson_iterator it[1];
        if ( bson_find( it, &cursor->current, "name" ) ) {
            printf( "name: %s\n", bson_iterator_string( it ) );
        }
    }

    bson_destroy( query );
}
```

Our query above, written as JSON, is of the form

```
{ age : <agevalue> }
```

Queries are BSON objects of a particular format.

In the mongo shell (which uses javascript), we could invoke:

```
use tutorial;
db.persons.find( { age : 24 } );
```

Complex Queries

Sometimes we want to do more than a simple query. We may want the results to be sorted in a special way, or what the query to use a certain index.

Sorting

Let's now make the results from previous query be sorted alphabetically by name. To do this, we change the query statement from:

```
bson_buffer_init( query_buf );
bson_append_int( query_buf, "age", 24 );
bson_from_buffer( query, query_buf );
```

to:

```
bson_buffer_init( query_buf );
bson_append_start_object( query_buf, "$query" );
bson_append_int( query_buf, "age", 24 );
bson_append_finish_object( query_buf );
bson_append_start_object( query_buf, "$orderby" );
bson_append_int( query_buf, "name", 1);
bson_append_finish_object( query_buf );
bson_from_buffer( query, query_buf );
```

Hints

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:

```
bson_buffer_init( query_buf );
bson_append_start_object( query_buf, "$query" );
bson_append_int( query_buf, "age", 24 );
bson_append_string( query_buf, "name", "Mathias" );
bson_append_finish_object( query_buf );
bson_append_start_object( query_buf, "$hint" );
bson_append_int( query_buf, "name", 1);
bson_append_finish_object( query_buf );
bson_from_buffer( query, query_buf );
```

Explain

A great way to get more information on the performance of your database queries is to use the \$explain feature. This will return "explain plan" type info about a query from the database.

```
bson_buffer_init( query_buf );
bson_append_start_object( query_buf, "$query" );
bson_append_int( query_buf, "age", 33 );
bson_append_finish_object( query_buf );
bson_append_bool( query_buf, "$explain", 1);
bson_from_buffer( query, query_buf );
```

Indexing

Let's suppose we want to have an index on age so that our queries are fast. We would use:

```
static void tutorial_index( mongo_connection * conn ) {
    bson key[1];
    bson_buffer key_buf[1];

    bson_buffer_init( key_buf );
    bson_append_int( key_buf, "name", 1 );
    bson_from_buffer( key, key_buf );

    mongo_create_index( conn, "tutorial.persons", key, 0, NULL );

    bson_destroy( key );

    printf( "simple index created on \"name\"\n" );

    bson_buffer_init( key_buf );
    bson_append_int( key_buf, "age", 1 );
    bson_append_int( key_buf, "name", 1 );
    bson_from_buffer( key, key_buf );

    mongo_create_index( conn, "tutorial.persons", key, 0, NULL );

    bson_destroy( key );

    printf( "compound index created on \"age\", \"name\"\n" );
}
```

Updating

Use the `mongo_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 running the following C function:

```
static void tutorial_update( mongo_connection *conn ) {
    bson cond[1], op[1];
    bson_buffer cond_buf[1], op_buf[1];

    bson_buffer_init( cond_buf );
    bson_append_string( cond_buf, "name", "Joe");
    bson_append_int( cond_buf, "age", 33);
    bson_from_buffer( cond, cond_buf );

    bson_buffer_init( op_buf );
    bson_append_start_object( op_buf, "$inc" );
    bson_append_int( op_buf, "visits", 1 );
    bson_append_finish_object( op_buf );
    bson_from_buffer( op, op_buf );

    mongo_update(conn, "tutorial.persons", cond, op, 0);

    bson_destroy( cond );
    bson_destroy( op );
}
```

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](#);
- Consider getting involved to make the product (either C driver, tools, or the database itself) better!

CSharp Language Center

- [C# Driver](#)
 - [Download](#)
 - [Build](#)

C# Driver

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

- [C# Driver Tutorial](#)
- [C# Driver Serialization Tutorial](#)
- [API Documentation](#)
- [C# Driver README](#)
- [Source Code](#)

Download

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>.

Build

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

- Visual Studio 2010
- Visual Studio 2008

CSharp API Documentation

Under construction

The API Documentation will be generated automatically from the documentation comments in the source code (also yet to be written).

Any suggestions for a good tool to process the documentation comments would be appreciated.

In the meantime, please refer to the [C# Driver Tutorial](#).

CSharp Community Projects

Community Supported C# Drivers

- [mongodb-csharp driver](#)
- [simple-mongodb driver](#)
- [NoRM](#)

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](#)

Tools

- [MongoDB.Emitter Document Wrapper](#)
- [log4net appender](#)

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 Serialization Tutorial

- [Introduction](#)
- [Creating a class map](#)
- [Conventions](#)
- [Field or property level serialization options](#)
 - [Element name](#)
 - [Element order](#)
 - [Identifying the Id field or property](#)
 - [Ignoring a field or property](#)
 - [Ignoring null values](#)
 - [Default values](#)
 - [Representation](#)
 - [Other serialization options](#)
- [Class level serialization options](#)
 - [Ignoring extra elements](#)
 - [Supporting extra elements](#)
 - [Polymorphic classes and discriminators](#)
 - [Setting the discriminator value](#)
 - [Specifying known types](#)

- Scalar and hierarchical discriminators
- Customizing serialization
 - Supplementing the default serializer
 - Make a class responsible for its own serialization
 - Write a custom serializer
 - Write a custom Id generator
 - Write a custom convention

Introduction

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 needed 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 initialization code write:

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

which results in MyClass being automapped. 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 lambda 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.

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
});
```

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", SerializedDefaultValue = 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);
});
```

Representation

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);
});
```

Other serialization options

Representation is just one specialized type of serialization option, and is only used for simple types for which the choice of representation is the only option you can specify. In other cases, specific attributes and classes are used to hold options appropriate to that particular class. For example, serialization of DateTime values can be controlled using either the BsonDateTimeOptionsAttribute or the DateTimeSerializationOptions class. 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.

This same configuration could be achieved using initialization code:

```

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 });
});

```

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 possible 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 do nothing. 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 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 Deserializer
    // implement Serialize
}

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

```

You also must implement GetDocumentId and SetDocumentId, but these methods can be stubs that do nothing if your class does not have an Id member.

Write a custom Id generator

An Id generator is responsible for generating a unique value for an Id. The default serializer has Id generators for these types:

1. ObjectId (ObjectIdGenerator)
2. Guid (GuidGenerator and CombGuidGenerator)

You do not need to do anything to use the standard Id generators for ObjectId and Guid. If you want to use the CombGuidGenerator you can either register it as the Id generator for all Guids or select it on an individual field or property level.

To use the CombGuidGenerator for all Guids:

```

BsonSerializer.RegisterIdGenerator(
    typeof(Guid),
    new CombGuidGenerator()
);

```

To choose the CombGuidGenerator for just one field or property you can use the IdGenerator named parameter of the BsonId attribute:

```

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

```

Or using initialization code instead of attributes:

```

BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.SetIdMember(
        cm.GetMember(c => c.Id).SetIdGenerator(new CombGuidGenerator())
    );
});

```

You can also write your own Id generator. 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.SetIdMember(
        cm.GetMember(c => c.Id).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 v1.0](#)
- [Introduction](#)
- [Downloading](#)
- [Building](#)
 - [Dependencies](#)
 - [Running unit tests](#)
- [Installing](#)
- [References and namespaces](#)
- [The BSON Library](#)
 - [BsonType](#)
 - [BsonValue and subclasses](#)
 - [BsonType property](#)
 - [As\[Type\] Properties](#)
 - [Is\[Type\] Properties](#)
 - [To\[Type\] conversion methods](#)
 - [Static Create methods](#)
 - [Implicit conversions](#)
 - [BsonMaxKey, BsonMinKey, BsonNull and BsonUndefined](#)
 - [ObjectId and BsonObjectId](#)
 - [BsonElement](#)
 - [BsonDocument](#)
 - [BsonDocument constructor](#)
 - [Create a new document and call Add and Set methods](#)
 - [Create a new document and use the fluent interface Add and Set methods](#)
 - [Create a new document and use C#'s collection initializer syntax \(recommended\)](#)
 - [Add methods](#)
 - [Accessing BsonDocument elements](#)
 - [BsonArray](#)
 - [Constructors](#)
 - [Add and AddRange methods](#)
 - [Indexer](#)
- [The C# Driver](#)
 - [Thread safety](#)
 - [MongoServer class](#)
 - [Connection strings](#)
 - [Create method](#)
 - [GetDatabase method](#)
 - [RequestStart/RequestDone methods](#)
 - [Other properties and methods](#)
 - [MongoDatabase class](#)
 - [GetCollection method](#)
 - [Other properties and methods](#)
 - [MongoCollection<TDefaultDocument> class](#)
 - [Insert<TDocument> method](#)
 - [InsertBatch method](#)
 - [FindOne and FindOneAs methods](#)
 - [Find and FindAs methods](#)
 - [Save<TDocument> method](#)
 - [Update method](#)
 - [FindAndModify method](#)
 - [MapReduce method](#)
 - [Other properties and methods](#)
 - [MongoCursor<TDocument> class](#)
 - [Enumerating a cursor](#)
 - [Modifying a cursor before enumerating it](#)
 - [Modifiable properties of a cursor](#)
 - [Other methods](#)
 - [SafeMode class](#)

C# Driver version v1.0

This tutorial is for v1.0 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 2008 and Visual Studio 2010. There are two solution files, one for each version of Visual Studio. The names of the solution files are CSharpDriver-2008.sln and CSharpDriver-2010.sln. The project files are shared by both solutions.

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 three projects containing unit tests:

1. BsonUnitTests
2. DriverUnitTests
3. DriverOnlineUnitTests

The first two do not connect to a MongoDB server. DriverOnlineUnitTests connects 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 arguments to: BsonUnitTests.csproj /run
 4. Set working directory to: the directory where BsonUnitTest.csproj is located

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

To run the DriverUnitTests or DriverOnlineUnitTests 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\CSsharpDriver 1.0
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.

References and namespaces

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

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

You also will likely want to add the following using statements to your source files:

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

You might need to 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.Builders;  
using MongoDB.Driver.GridFS;  
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.BsonType 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
- AsBsonDocument
- AsBsonJavaScript // also works if BsonType == JavaScriptWithScope
- AsBsonJavaScriptWithScope
- AsBsonMaxKey
- AsBsonMinKey
- AsBsonNull
- AsBsonRegularExpression
- AsBsonSymbol
- AsBsonTimestamp
- AsBsonUndefined
- 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
- 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
- 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.

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(Hashtable hashtable)
- BsonDocument(Hashtable hashtable, IEnumerable<string> keys)
- BsonDocument(IDictionary<string, object> dictionary)
- BsonDocument(IDictionary<string, object> dictionary, IEnumerable<string> keys)
- BsonDocument(IEnumerable<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 corresponding 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();
document.Add("author", "Ernest Hemingway");
document.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 methods.

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.

Add methods

BsonDocument has the following overloaded Add methods:

- Add(BsonElement element)
- Add(Hashtable hashtable)
- Add(Hashtable hashtable, 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 becomes 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 = books.FindOne();
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<object> values)`
- `BsonArray(IEnumerable<ObjectId> values)`
- `BsonArray(IEnumerable<string> 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<object> values)`
- `BsonArray AddRange(IEnumerable<ObjectId> values)`
- `BsonArray AddRange(IEnumerable<string> 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).

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 apply to a single database if the database name is present, otherwise they will be the default credentials for all databases. To authenticate against the admin database append "(admin)" to the username.

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 as 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
```

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.

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 dbName)`
- `MongoDatabase GetDatabase(string dbName, MongoCredentials credentials)`
- `MongoDatabase GetDatabase(string dbName, MongoCredentials credentials, SafeMode safeMode)`
- `MongoDatabase GetDatabase(string dbName, SafeMode safeMode)`
- `MongoDatabase this[MongoDatabaseSettings settings]`
- `MongoDatabase this[string dbName]`
- `MongoDatabase this[string dbName, MongoCredentials credentials]`
- `MongoDatabase this[string dbName, MongoCredentials credentials, SafeMode safeMode]`
- `MongoDatabase this[string dbName, 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. 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:

- `AdminDatabase`

- `ConnectionPool`
- `EndPoints`
- `IndexCache`
- `MaxDocumentSize`
- `MaxMessageLength`
- `ReplicaSet`
- `SecondaryConnectionPools`
- `Settings`
- `State`

MongoServer has the following additional methods:

- `Connect`
- `CopyDatabase`
- `DatabaseExists`
- `Disconnect`
- `DropDatabase`
- `FetchDBRef`
- `FetchDBRefAs`
- `GetAdminDatabase`
- `GetDatabaseNames`
- `GetLastError`
- `Reconnect`
- `RunAdminCommand`
- `RunAdminCommandAs`

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
- 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 Cradle")
);
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 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` (`SetOptions`)
- `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`
- `SetSortOrder`

Other methods

`MongoCursor` has a few methods used for some special purpose operations:

- `Count`
- `Explain`
- `Size`

SafeMode class

There are various level 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 has static properties and methods that let you easily access common modes or create your own:

- `SafeMode.False`
- `SafeMode.True`
- `SafeMode.W2`
- `SafeMode.Create(int w)`

The value for "w" indicates how many replica set members must have completed the write before returning, and this includes the primary, so typically you want `w >= 2`.

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
<code>[]</code>	<code>[]</code>	<code>array()</code>	<code>[]</code>
<code>{}</code>	<code>{}</code>	<code>new stdClass</code>	<code>{}</code>

<code>{x:1}</code>	<code>{"x":1}</code>	<code>array('x' => 1)</code>	<code>{'x'=>1}</code>
<code>{x:1}</code>			
<code>connect("www.example.net")</code>	<code>Connection("www.example.net")</code>	<code>new Mongo("www.example.net")</code>	<code>Connection.new("www.exa</code>
<code>cursor.next()</code>	<code>cursor.next()</code>	<code>\$cursor->getNext()</code>	<code>cursor.next_document()</code>
<code>cursor.hasNext()</code>	<code>*</code>	<code>\$cursor->hasNext()</code>	<code>cursor.has_next?</code>
<code>collection.findOne()</code>	<code>collection.find_one()</code>	<code>\$collection->findOne()</code>	<code>collection.find_one()</code>
<code>db.eval()</code>	<code>db.eval()</code>	<code>\$db->execute()</code>	<code>db.eval()</code>

* 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.

3rd Party ORM/ODM

- [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)
- [node-mongodb-native](#) - Native async Node interface to MongoDB.
- [mongo-v8-driver](#) - V8 MongoDB driver (experimental, written in C++).

JVM Languages

moved to [Java Language Center](#)

Python Language Center



Redirection Notice

This page should redirect to <http://api.mongodb.org/python>.

PHP Language Center

Using MongoDB in PHP

To access MongoDB from PHP you will need:

- The MongoDB server running - the server is the "mongod" 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>
 - VC6 is for Apache (VC9 is for IIS)
 - 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](#)

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

The PHP community has created a huge number of libraries to make working with MongoDB easier and integrate it with existing 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.

Memcached

- [MongoNode](#)
PHP script that replicates MongoDB objects to Memcached.

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.

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 embedded 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

Mondongo

[Mondongo](#) is a simple, powerful and ultrafast Object Document Mapper (ODM) for PHP and MongoDB.

Mondongo is to the ODM what Mongo is to databases

- Simple: Mondongo is developed in a simple way. This makes it very easy to learn, use, and avoid bugs.
- Powerful: Mondongo is very flexible thanks to Mondator, so you'll be able to use it to develop any type of application.
- Ultrafast: Mondongo has been designed to be extremely light in memory consumption and processing cost.

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 code.google.com

simplemongophp

Very simple layer for using data objects see [blog post](#)

- [simplemongophp](#) at github

Uniform Server 6-Carbo with MongoDB and phpMoAdmin

The Uniform Server is a lightweight WAMP server solution for running a web server under Windows without having anything to install; just unpack and run it. Uniform Server 6-Carbo includes the latest versions of Apache2, Perl5, PHP5, MySQL5 and phpMyAdmin. The Uniform Server MongoDB plugin adds the MongoDB server, phpMoAdmin browser administration interface, the MongoDB PHP driver and a Windows interface to start and stop both Apache and MongoDB servers. From this interface you can also start either the Mongo-client or phpMoAdmin to administer MongoDB databases.

- [Uniform Server 6-Carbo and MongoDB plugin](#) at SourceForge
- [Uniform Server](#) web site

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](#)

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 1.5.2 as of Feb 2011):

```
$ 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, of which MonogMapper and Mongoid seem to have the most traction in the community.

- [MongoMapper](#) from John Nunemaker
- [Mongoid](#) from Durran Jordan
- [Mongomatic](#) from Ben Myles
- [MongoODM](#) from Carlos Paramio

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.

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](#)
- [A Quick Tour](#)
 - [Using the RubyGem](#)
 - [Making a Connection](#)
 - [Listing All Databases](#)
 - [Dropping a Database](#)
 - [Authentication \(Optional\)](#)
 - [Getting a List Of Collections](#)
 - [Getting a Collection](#)
 - [Inserting a Document](#)
 - [Updating a Document](#)
 - [Finding the First Document In a Collection using find_one\(\)](#)
 - [Adding Multiple Documents](#)
 - [Counting Documents in a Collection](#)
 - [Using a Cursor to get all of the Documents](#)
 - [Getting a Single Document with a Query](#)
 - [Getting a Set of Documents With a Query](#)
 - [Selecting a subset of fields for a query](#)
 - [Querying with Regular Expressions](#)
 - [Creating An Index](#)
 - [Creating and querying on a geospatial index](#)
 - [Getting a List of Indexes on a Collection](#)
 - [Database Administration](#)
- [See Also](#)

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]}}).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  
  
# 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 ($\geq 1.0.6$) 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](#)
- [\[Replicas Set Configuration\]](#)

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](#)
 - [Saving files](#)
 - [File metadata](#)
 - [Safe mode](#)
 - [Deleting files](#)
- [The GridFileSystem class](#)
 - [Saving files](#)
 - [Deleting files](#)
 - [Metadata and safe mode](#)
- [Advanced Users](#)

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, an 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:

```
image = File.open("me.jpg")
@fs.open("me.jpg", "w") do |f|
  f.write image
end
```

We can then retrieve the file by filename:

```
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:

```
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:

```
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:

```
@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:

```
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.

- [Configuration](#)
- [Testing](#)
- [Coding](#)

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
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](#)
- [Configure your application](#)
- [Bundle and Initialize](#)
 - [Bundling](#)
 - [Initializing](#)
- [Running Tests](#)
- [Conclusion](#)
- [See also](#)

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 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.

See also

- [Rails 3 App skeleton with MongoMapper](#)
- [Rails 3 Release Notes](#)

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.

- [Modeling Stories](#)
 - [Caching to Avoid N+1](#)
 - [A Note on Denormalization](#)
 - [Fields as arrays](#)
 - [Atomic Updates](#)
- [Modeling Comments](#)
 - [Linear, Embedded Comments](#)
 - [Nested, Embedded Comments](#)
 - [Comment collections](#)
- [Unfinished business](#)

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
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

Basics

- [Download the Java Driver](#)
- [Tutorial](#)
- [API Documentation](#)
- [Release Notes](#)

Specific Topics and How-To

- [Concurrency](#)
- [Saving Objects](#)
- [Data Types](#)

Hadoop Support

- There is a nascent and experimental [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

Third Party Frameworks and Libs

POJO Mappers

- [Morphia](#) - Type-Safe Wrapper with DAO/Datastore abstractions
- [Mungbean](#) (w/closure support)

Code Generation

- [Sculptor](#) - mongodb-based DSL -> Java (code generator)
- [GuicyData](#) - DSL -> Java generator with Guice integration
 - [Blog Entries](#)

Misc

- [log4mongo](#) a log4j appender
- (Experimental, Type4) JDBC driver

Scala

- [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.
 - There is [API documentation](#), a [detailed setup guide](#) and a [tutorial for new users](#) available.

Third Party

- [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](#)

Other JVM-based Languages

Clojure

- [Congo Mongo](#)

Groovy

- [Groovy Tutorial for MongoDB](#)
- [MongoDB made more Groovy](#)
- [GMongo](#), a Groovy wrapper to the mongodb Java driver
 - [GMongo 0.5 Release Writeup](#)
 - [Mailing List](#)
 - [GitHub Project Page](#)

JavaScript

- [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.

Presentations

- [Using MongoDB with Scala](#) - Brendan McAdams' Presentation at the New York Scala Enthusiasts (August 2010)
- [Java Development](#) - Brendan McAdams' Presentation from MongoNYC (May 2010)
- [Java Development](#) - James Williams' Presentation from MongoSF (April 2010)
- [Building a Mongo DSL in Scala at Hot Potato](#) - Lincoln Hochberg's Presentation from MongoSF (April 2010)

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).



This is needed less since the 2.2 version of the driver. The connection pool is now thread-aware and will use the same connection (in the thread) as long as the pool isn't depleted between operations.

However, if you want to ensure complete consistency in a "session" (maybe an http request), you probably want the driver to use the same socket for that session (which isn't necessarily the case since Mongo instances have built-in connection pooling). This is only necessary for a write heavy environment, where you might read data that you wrote.

To do that, you would do something like:

```
DB db...;
db.requestStart();

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

By using a `WriteConcern.SAFE` you will get the same behavior for a single write + `getLastError`.

```

DBCollection coll...;
coll.insert(..., WriteConcern.SAFE);

// is the same as
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);

Tweet myTweet = (Tweet)collection.findOne();

```

Java Tutorial

- [Introduction](#)
- [A Quick Tour](#)
 - [Making A Connection](#)
 - [Authentication \(Optional\)](#)
 - [Getting A List Of Collections](#)
 - [Getting A Collection](#)
 - [Inserting a Document](#)
 - [Finding the First Document In A Collection using findOne\(\)](#)
 - [Adding Multiple Documents](#)
 - [Counting Documents in A Collection](#)
 - [Using a Cursor to Get All the Documents](#)
 - [Getting A Single Document with A Query](#)
 - [Getting A Set of Documents With a Query](#)
 - [Creating An Index](#)
 - [Getting a List of Indexes on a Collection](#)
- [Quick Tour of the Administrative Functions](#)
 - [Getting A List of Databases](#)
 - [Dropping A Database](#)

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();
Mongo m = new Mongo( "localhost" );
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.

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} , "_ns" : "testCollection"}
```

Note the `_id` and `_ns` elements have 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 , "_ns" : "testCollection" }
```

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 $20 < i \leq 30$:

```
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} , "_ns" : "system.indexes" }
```

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
- 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[]`) can be used for binary data.

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);
```

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.

- [API Documentation](#)
- [C++ Driver Compiling and Linking](#)
- [MongoDB C++ Client Tutorial](#)
- [Using BSON from C++](#)
- HOWTO
 - [Connecting](#)
 - [Tailable Cursors](#)
 - [BSON Arrays in C++](#)
- [Mongo Database and C++ Driver Source Code](#) (at github). See the client subdirectory for client driver related files.
- [Linux 64 bit C++ client download](#)

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

- [Overview](#)
- [Examples](#)
- [API Docs](#)
- [Short Class Names](#)

Overview

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:

- `BSONObj` a BSON object

- `BSONElement` a single element in a bson object. This is a key and a value.
- `BSONObjBuilder` used to make BSON objects
- `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();
```

Examples

- <http://github.com/mongodb/mongo/blob/master/bson/bsondemo/bsondemo.cpp>

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

- [Linux](#)
 - [Using the prebuilt library](#)
 - [Using mongo_client_lib.cpp instead of a library](#)
- [Windows](#)
 - [Windows Troubleshooting](#)

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 \(v1.9+\)](#)
- [Prebuilt Boost Libraries](#)
- [Building with Visual Studio 2010](#)

Other notes for Windows:

- Add the following to your project's VC++ Include Directories:
 - the mongo project's top directory location
 - pcre-7.4/ under the mongo project's top directory location
 - the location of your [boost](#) installation. For example c:\boost\.
- 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++ Tutorial

- [Installing the Driver Library and Headers](#)
 - [Unix](#)
 - [Full Database Source Driver Build](#)
 - [Driver Build](#)
 - [Windows](#)
- [Compiling](#)
- [Writing Client Code](#)
 - [Connecting](#)
 - [BSON](#)
 - [Inserting](#)
 - [Querying](#)
 - [Indexing](#)
 - [Sorting](#)
 - [Updating](#)
- [Further Reading](#)

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-mt -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**. Also, you may need to append "-mt" to boost_filesystem and boost_program_options. And, 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();
```

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", emptyObj);  
while( cursor->more() )  
    cout << cursor->next().toString() << endl;
```

emptyObj is the empty BSON object -- we use it to represent {} 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 ) ) );
```

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!

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](#)
 - [CPAN](#)
 - [Manual \(Non-CPAN\) Installation](#)
 - [Big-Endian Systems](#)
- [Next Steps](#)
- [MongoDB Perl Tools](#)
 - [Entities::Backend::MongoDB](#)
 - [MojoX::Session::Store::MongoDB](#)
 - [MongoDB::Admin](#)
 - [Mongoose](#)
 - [MongoDBx-Class](#)
 - [Mongrel](#)
 - [MongoX](#)

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.

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{MONGODB} 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:

```
MONGODB=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

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.

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.

Mongrel

Mongrel provides a simple database abstraction layer for MongoDB. **Mongrel** uses the Oogly data validation framework to provide you with a simple way to create codebased schemas that have data validation built-in, etc.

MongoX

MongoX - DSL sugar for MongoDB

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.

bsonspec.org

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 4MB. This size may change over time but for now the limit is 4MB per object. We recommend you test your driver with 4MB objects.

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, save, 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 efficiency these are sent in batches)
- Convert all strings to utf8
- [Authentication](#)

Recommended

- automatic doc_id generation (important when using replication)
- 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 for objects up to 4MB in size

More Recommended

- [lasterror](#) helper functions
- count() helper function
- \$where clause
- eval()
- File chunking
- [hint](#) fields
- [explain](#) helper
- Automatic _id index creation (maybe the db should just do this???)

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
- Messages Types and Formats
 - Standard Message Header
 - Request Opcodes
- Client Request Messages
 - `OP_UPDATE`
 - `OP_INSERT`
 - `OP_QUERY`
 - `OP_GETMORE`
 - `OP_DELETE`
 - `OP_KILL_CURSORS`
 - `OP_MSG`
- Database Response Messages
 - `OP_REPLY`

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 responseTo 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_GET_MORE	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    ZERO;              // 0 - reserved for future use
    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.

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 cursor ID 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. 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 messages 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 response. 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 and MongoDB](#)
- [Language-Specific Examples](#)
 - [C](#)
 - [C++](#)
 - [Java](#)
 - [PHP](#)
 - [Python](#)
 - [Ruby](#)
- [MongoDB Document Types](#)

[bsonspec.org](#)

BSON is a binary-encoded serialization of JSON-like documents. 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](#).

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 MongoDB::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 pymongo.bson import BSON
>>> bson_string = BSON.from_dict({"hello": "world"})
>>> bson_string
'\x16\x00\x00\x00\x02hello\x00\x06\x00\x00\x00world\x00\x00'
>>> bson_string.to_dict()
{'hello': 'world'}
```

PyMongo also supports "ordered dictionaries" through the `pymongo.son` module. The `BSON` class can handle `SON` instances using the same methods you would use for regular dictionaries.

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	Explanation
data_binary	<pre>{ "\$binary" : : "<bindata>" , "\$type" : "<t>" }</pre>	<pre>{ "\$binary" : "<bindata>" , "\$type" : "<t>" }</pre>	<pre>{ "\$binary" : "<bindata>" , "\$type" : "<t>" }</pre>	<bindata> is the base64 representation of a binary string. <t> is the hexadecimal representation of a single byte indicating the data type.
data_date	<pre>{ "\$date" : <date> }</pre>	<pre>Date(<date>)</pre>	<pre>Date(<date>)</pre>	<date> is the JSON representation of a 64 bit unsigned integer for milliseconds since epoch.
data_regex	<pre>{ "\$regex" : : "<sRegex>" , "\$options" : "<sOptions>" }</pre>	<pre>/<jRegex>/<jOptions></pre>	<pre>/<jRegex>/<jOptions></pre>	<sRegex> is a string of valid JSON characters. <jRegex> is a string that may contain valid JSON characters and unescaped "" characters, but may not contain unescaped '/' characters. <sOptions> is a string containing letters of the alphabet. <jOptions> is a string that may contain only the characters 'g', 'i', and 'm'. Because the JS and TenGen representation support a limited range of options, any nonconforming options will be dropped when converting to this representation.

data_oid	<pre>{ "\$oid" : "<id>" }</pre>	<pre>{ "\$oid" : "<id>" }</pre>	<pre>ObjectId("<id>")</pre>	<id> is a 24 character hexadecimal string. Note that these representation: require a data_oid value to have an associated field name "_id".
data_ref	<pre>{ "\$ref" : "<name>", "\$id" : "<id>" }</pre>	<pre>{ "\$ref" : "<name>" , "\$id" : "<id>" }</pre>	<pre>Dbref("<name>", "<id>")</pre>	<name> is a string of valid JSON characters. <id> is a 24 character hexadecimal string.

GridFS Specification

- [Introduction](#)
- [Specification](#)
 - [Storage Collections](#)
 - [files](#)
 - [chunks](#)
 - [Indexes](#)

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 its `files_id` and `n` values. Note that GridFS implementations should use `findOne` operations to get chunks individually, and should **not** leave open a cursor to query for all chunks. So to get the first chunk, we could do:

```
db.fs.chunks.findOne({files_id: myFileID, n: 0});
```

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 Pairs and Authentication

For drivers that support replica pairs, extra care with replication is required.

When switching from one server in a pair to another (on 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 half a pair, 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.
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

- [Getting the Database](#)
- [Getting A Database Connection](#)
- [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](#)

Getting the Database

First, run through the [Quickstart](#) guide for your platform to get up and running.

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
```

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.

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.

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 any "structure" - 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. However, 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.
- 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, let's 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.

Let's 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)`.

Limiting the Result Set via limit()

You may limit the size of a query's result set by specifying 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.whatever 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 [manual](#) for more details.
- See also [SQL to Mongo Mapping Chart](#)

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](#)
 - [Clone Database](#)
 - [fsync Command](#)
 - [Index-Related Commands](#)
 - [Last Error Commands](#)
 - [Windows Service](#)
 - [Viewing and Terminating Current Operation](#)
 - [Validate Command](#)
 - [getLastError](#)
 - [List of Database Commands](#)
 - [Mongo Metadata](#)
- [Collections](#)
 - [Capped Collections](#)
 - [createCollection Command](#)
 - [Using a Large Number of Collections](#)
- [Data Types and Conventions](#)
 - [Timestamp Data Type](#)
 - [Internationalized Strings](#)
 - [Object IDs](#)
 - [Database References](#)
- [GridFS](#)
 - [When to use GridFS](#)
- [Indexes](#)
 - [Using Multikeys to Simulate a Large Number of Indexes](#)
 - [Geospatial Indexing](#)
 - [Indexing as a Background Operation](#)
 - [Multikeys](#)
 - [Indexing Advice and FAQ](#)
- [Inserting](#)
 - [Legal Key Names](#)
 - [Schema Design](#)
 - [Trees in MongoDB](#)
- [Optimization](#)
 - [Optimizing Object IDs](#)
 - [Optimizing Storage of Small Objects](#)
 - [Query Optimizer](#)
- [Querying](#)
 - [Mongo Query Language](#)
 - [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](#)

- Removing
- Updating
 - Atomic Operations
 - findAndModify Command
 - 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:

- `connect=direct|replicaSet`
 - `direct`: a direct connection will be made to one server. If more than one host is provided they will be tried in order until a connection can be established. `direct` is the default value when only one host is specified.
 - `replicaSet`: connect with replica set semantics (even if only one host is provided). The hosts specified are used as a seed list to find the full replica set. `replicaSet` is the default value when multiple hosts are specified.
- `replicaSet=name`
 - The driver verifies that the name of the replica set it connects to matches this name. Implies `connect=replicaSet`.
- `slaveOk=true|false`
 - `true`: In `connect=direct` mode the driver will connect to the first server in the list to respond even if it is not a primary. In `connect=replicaSet` mode the driver will send all writes to the primary and will distribute reads to the secondaries in round robin order.
 - `false`: In `connect=direct` mode the driver will try all hosts in order until a primary is found. In `connect=replicaSet` mode the driver will connect only to the primary and send all reads and writes to the primary.
- `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 add `fsync` to the `getLastError` command.
- `maxPoolSize=n`
 - The connection pool is not allowed to grow beyond this size. When a thread requests a connection it either: receives one immediately from the connection pool if one is available, receives a newly created connection if the pool has not reached the maximum size, or waits for a connection to be released by another thread (see `waitQueueTimeoutMS` and `waitQueueMultiple`).
- `minPoolSize=n`
 - Some drivers will close connections that haven't been used for awhile. However, they won't close an idle connection if that would cause the number of connections to fall below the minimum pool size. Note that connections are only created on demand, so `minPoolSize` does not imply that the connection pool will be pre-populated with this many connections.
- `waitQueueTimeoutMS=ms`
 - The amount of time a thread can wait for a connection to become available before timing out. This only applies if the connection pool has reached the maximum size and all connections are already in use.
- `waitQueueMultiple=n`
 - The drivers impose a limit on how many threads can be waiting for a connection at the same time. This limit is expressed as a multiple of `maxPoolSize`.
- `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.

Connection strings 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 the first server to respond, whether or not it is part of a replica set or primary or secondary:

```
mongodb://host1,host2,host3/?connect=direct;slaveOk=true
```

This type of connection string can be used when you have a preference for which server to connect to but want to list some alternatives.

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 recommended 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.

- [Commands](#)
 - [Clone Database](#)
 - [fsync Command](#)
 - [Index-Related Commands](#)
 - [Last Error Commands](#)
 - [Windows Service](#)
 - [Viewing and Terminating Current Operation](#)
 - [Validate Command](#)
 - [getLastError](#)
 - [List of Database Commands](#)
- [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](#)

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` 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

- [Clone Database](#)
- [fsync Command](#)
- [Index-Related Commands](#)
- [Last Error Commands](#)
- [Windows Service](#)
- [Viewing and Terminating Current Operation](#)
- [Validate Command](#)
- [getLastError](#)
- [List of Database Commands](#)

- [Commands Quick Reference Card](#)

Clone Database

MongoDB includes commands for copying a database from one server to another.

```
// copy 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. Must be run on the "destination" server.
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>);
// in "command" syntax (runnable from any driver):
db.runCommand( { copydb : 1, fromdb : ..., todb : ..., fromhost : ... } );
// 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 > } );

// clone the current database (implied by 'db') from another host
var fromhost = ...;
print("about to get a copy of database " + db + " from " + fromhost);
db.cloneDatabase(fromhost);
// in "command" syntax (runnable from any driver):
db.runCommand( { clone : fromhost } );
```

Notes

- copyDatabase may be run against a slave/secondary (i.e., the source server can be a slave/secondary).
- copyDatabase is **not** an atomic operation 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.
- 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

- [fsync Command](#)
- [Lock, Snapshot and Unlock](#)
 - [Caveats](#)
 - [Snapshotting Slaves](#)
- [See Also](#)

fsync Command



Version 1.3.1 and higher

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-file-system-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 : <http://jira.mongodb.org/browse/SERVER-1423>

Snapshotting Slaves

The above procedure works on replica slaves. The slave will not apply operations while locked. However, see the above caveats section.

See Also

- [Backups](#)

Index-Related Commands

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. However, beginning with version 1.3.2, you can specify that indexing happen in the background. See the [background indexing docs](#) for details.

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
```

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.

Last Error Commands

Since MongoDB doesn't wait for a response by default when writing to the database, a couple commands exist for ensuring that these operations have succeeded. These commands can be invoked automatically with many of the drivers when saving and updating in "safe" mode. But what's really happening is that a special command called `getLastError` is being invoked. Here, we explain how this works.

- `getLastError`
 - Drivers
 - Use Cases
 - options
 - `fsync`
 - `w`
 - Mongo Shell REPL Behavior
- `getPrevError`

getLastError

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

```
> db.runCommand("getLastError")
```

Or you can use the helper:

```
> db.getLastError()
```

For more about commands, see the [command documentation](#).

Drivers

The drivers support `getLastError` in the command form and many also offer a "safe" mode for operations. If you're using Python, for example,

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. For more on "safe" mode, see your driver's documentation.

Use Cases

`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.

Note: The special [mongo wire protocol](#) `killCursors` operation does not support `getLastError`. (This is really only of significant to [driver developers](#).)

options

`fsync`

When running `mongod` without `--dur`, the `fsync` option forces the database to `fsync` all files before returning.

When running with `--dur`, the `fsync` option awaits the next group commit before returning.

```
> db.runCommand({getLastError:1,fsync:true})
{ "err" : null, "n" : 0, "fsyncFiles" : 2, "ok" : 1 }
```

`w`

A client can block until a write operation has been replicated to `N` servers.

`wtimeout` may be used in conjunction with `w`.

```
> db.getLastError(2, 5000) // w=2, timeout 5000ms
```

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



`getPrevError` may be deprecated in the future.

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!");
```

Windows Service

On windows mongod.exe has native support for installing and running as a windows service.

Service Related Commands

The service related commands are:

```
mongod --install
mongod --service
mongod --remove
mongod --reinstall
```

You may also option pass the following to --install and --reinstall

```
--serviceName {arg}
--serviceUser {arg}
--servicePassword {arg}
```

The --install and --remove options install and remove the mongo daemon as a windows service respectively. The --service option starts the service. --reinstall will attempt to remove the service, and then install it. If the service is not already installed, --reinstall will still work.

Both --remove and --reinstall will stop the service if it is currently running.

To change the name of the service use --serviceName. To make mongo execute as a local or domain user, as opposed to the Local System account, use --serviceUser and --servicePassword.

Whatever other arguments you pass to mongod.exe on the command line alongside --install are the arguments that the service is configured to execute mongod.exe with. Take for example the following command line:

```
mongod --bind_ip 127.0.0.1 --logpath d:\mongo\logs --logappend --dbpath d:\mongo\data --directoryperdb
--install
```

Will cause a service to be created called Mongo that will execute the following command:

```
mongod --bind_ip 127.0.0.1 --logpath d:\mongo\logs --logappend --dbpath d:\mongo\data --directoryperdb
```

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.

Viewing and Terminating Current Operation

- [View Current Operation\(s\) in Progress](#)
- [Terminate \(Kill\) an Operation in Progress](#)
- [See Also](#)

View Current Operation(s) in Progress

```
> db.currentOp();
> // same as: db.$cmd.sys.inprog.findOne()
{ inprog: [ { "opid" : 18 , "op" : "query" , "ns" : "mydb.votes" ,
              "query" : "{ score : 1.0 }" , "inLock" : 1 }
            ]
}
```

Fields:

- opid - an incrementing operation number. Use with killOp().
- op - the operation type (query, update, etc.)
- ns - namespace for the operation (database + collection name)
- 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](#).
- waitingForLock - if true, lock has been requested but not yet granted
- 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.

NOTE: currentOp's output format varies from version 1.0 and version 1.1 of MongoDB. The format above is for 1.1 and higher.

You can also do

```
db.$cmd.sys.inprog.find()
```

or this version which prints all connections

```
db.$cmd.sys.inprog.find( { $all : 1 } )
```

Terminate (Kill) an Operation in Progress

```
// <= v1.2
> db.killOp()
> // same as: db.$cmd.sys.killop.findOne()
{"info" : "no op in progress/not locked"}

// v>= 1.3
> db.killOp(1234/*opid*/)
> // same as: db.$cmd.sys.killop.findOne({op:1234})
```

Note: be careful about terminating internal operations, for example from a replication sync thread. Typically you only kill operations from external clients.

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.

From the mongo shell:

```

> db.foo.validate()
{"ns" : "test.foo" , "result" : "
validate
  details: 08D03C9C ofs:963c9c
  firstExtent:0:156800 ns:test.foo
  lastExtent:0:156800 ns:test.foo
  # extents:1
  datasize?:144 nrecords?:3 lastExtentSize:2816
  padding:1
  first extent:
    loc:0:156800 xnext:null xprev:null
    ns:test.foo
    size:2816 firstRecord:0:1568b0 lastRecord:0:156930
  3 objects found, nobj:3
  192 bytes data w/headers
  144 bytes data w/out/headers
  deletedList: 00000001000000000000
  deleted: n: 1 size: 2448
  nIndexes:1
    test.foo.$x_1 keys:3
  " , "ok" : 1 , "valid" : true , "lastExtentSize" : 2816}

```

From a driver one might invoke the driver's equivalent of:

```

> db.$cmd.findOne({validate:"foo" } );

```

validate takes an optional scandata parameter which skips the scan of the base collection (but still scans indexes).

```

> db.$cmd.findOne({validate:"foo", scandata:true});

```

getLastError



Redirection Notice

This page should redirect to [Last Error Commands](#) in about 3 seconds.

Most drivers, and the db shell, support a getlasterror capability. This lets one check the error code on the last operation.

Database [commands](#), as well as queries, have a direct return code.

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.

```

> db.runCommand("getlasterror")
> db.getLastError()

```

Note: The special [mongo wire protocol](#) killCursors operation does not support getlasterror. (This is really only of significant to [driver developers](#).)

getPrevError

Note: getPrevError may be deprecated in the future.

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!");
```

Last Error in the Shell

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"
```

FSync with GetLastError

Include the `fsync` option to force the database to fsync all files before returning (v1.3+):

```
> db.runCommand({getLastError:1,fsync:true})
{ "err" : null, "n" : 0, "fsyncFiles" : 2, "ok" : 1 }
```

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:

- `system.namespaces` lists all namespaces.
- `system.indexes` lists all indexes.
- Additional namespace / index metadata exists in the database.ns files, and is opaque.
- `system.profile` stores database profiling information.
- `system.users` lists users who may access the database.
- `local.sources` stores replica slave configuration data and state.
- Information on the structure of a stored object is stored within the object 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.

Details

A MongoDB collection is a collection of [BSON](#) documents. These documents are usually have the same structure, but this is not a requirement since MongoDB is a *schema-free* 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.

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.");
```

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.

See also:

- [Capped Collections](#)
- [Using a Large Number of Collections](#)

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 objects in the collection; this is very powerful for certain use cases such as logging.

Creating a Fixed Size (capped) Collection

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})
```

Usage and Restrictions

- You may insert new objects in the capped collection.
- You may update the existing objects in the collection. However, the objects must not grow in size. If they do, the update will fail. (There are some possible workarounds which involve pre-padding objects; contact us in the support forums for more information, if help is needed.)
- The database does not allow deleting objects from a capped collection. Use the `drop()` method to remove all rows from the collection. Note: After the drop you must explicitly recreate the collection.
- Maximum size for a capped collection is currently 1e9 bytes on a thirty-two bit machine. The maximum size of a capped collection on a sixty-four bit machine is constrained only by system resources.

Behavior

- Once the space is fully utilized, newly added objects will replace the oldest objects in the collection.
- If you perform a `find()` on the collection with no ordering specified, the objects will always be returned in insertion order. Reverse order is always retrievable with `find().sort({$natural:-1})`.

Applications

- **Logging.** Capped collections provide a high-performance means for storing logging documents in the database. Inserting objects in an unindexed capped collection will be close to the speed of logging to a filesystem. Additionally, with the built-in LRU mechanism, you are not at risk of using excessive disk space for the logging.
- **Caching.** If you wish to cache a small number of objects 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.
- **Auto Archiving.** 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 archival cron scripts.

Recommendations

- For maximum performance, 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.

Capping the Number of Objects

You may also cap the number of objects in the collection. Once the limit is reached, items roll out on a least recently inserted basis.

To cap on number of objects, specify a `max:` parameter on the `createCollection()` call.

Note: When specifying a cap on the number of objects, you must also cap on size. Be sure to leave enough room for your chosen number of objects 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.

Note: Capped collections are always capped by size and hence also limiting by number of documents is an overhead. Limiting by just size is more performant.

```
db.createCollection("mycoll", {capped:true, size:100000, max:100});
db.mycoll.validate();
```

Tip: When programming, a handy way to store the most recently generated version of an object can be a collection capped with `max=1`.

Preallocating space for a normal collection

The `createCollection` command may be used for non capped collections as well. For example:

```
db.createCollection("mycoll", {size:10000000});
db.createCollection("mycoll", {size:10000000, autoIndexId:false});
```

Explicitly creating a non capped collection via `createCollection` allows parameters of the new collection to be specified. For example, specification of a collection size causes the corresponding amount of disk space to be preallocated for use by the collection. 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` object field. By default, such an index is created for non capped collections but is not created for capped collections.



An index is not automatically created on `_id` for capped collections by default

Sharding

Capped collections are not shardable.

Check if a collection is capped

You can check if a collection is capped by using the `isCapped()` function. `db.foo.isCapped()`

See Also

- The [Sorting and Natural Order](#) section of this Guide

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})
> show collections
```

Most drivers also have a create collection helper method. You can manually issue any command also.

```
> db.runCommand( {createCollection:"mycoll", capped:true, size:100000} )
```

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.

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](#)

Timestamp Data Type

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 }
>
```

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 are required to have a key, _id, which uniquely identifies them.

- Document IDs: _id
- The BSON ObjectId Datatype
 - BSON ObjectId Specification
 - TimeStamp
 - Machine
 - Pid
 - Increment

- Document Timestamps
- Sequence Numbers

Document IDs: `_id`

Almost every MongoDB document has an `_id` field as its first attribute (there are a few exceptions for system collections and some capped collections). This value usually a BSON ObjectId. Such an id must be unique for each member of a collection; this is enforced if the collection has an index on `_id`, which is the case by default.

If a user tries to insert a document without providing an `id`, the database will automatically generate an `_object id` and store it the `_id` field.

Users are welcome to use their own conventions for creating ids; the `_id` value may be of any type, other than arrays, so long as it is a unique. Arrays are not allowed 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. Here's the schema:

0	1	2	3	4	5	6	7	8	9	10	11
time	machine			pid	inc						

Here is a breakdown of the sections:

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.

Document Timestamps

One useful consequence of this specification is that it provides documents with a creation timestamp for free. All of the drivers implement methods for extracting these timestamps; see the relevant api docs for details.

Sequence Numbers

Traditional databases often use monotonically increasing sequence numbers for primary keys. In MongoDB, the preferred approach is to use

Object IDs instead. Object IDs are more synergistic with sharding and distribution.

However, sometimes you may want a sequence number. The Insert if Not Present section of the [Atomic Operations](#) page shows an example of how to do this.

Database References

- [Simple Manual References](#)
- [DBRef](#)
- [DBRef in Different Languages / Drivers](#)
 - [C#](#)
 - [C++](#)
 - [Java](#)
 - [Javascript \(mongo shell\)](#)
 - [PHP](#)
 - [Python](#)
 - [Ruby](#)
- [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. Two conventions are common for references in MongoDB: first simple manual references, and second, the DBRef standard, which many drivers support explicitly.

Note: Often embedding of objects eliminates the need for references, but sometimes references are still appropriate.

Simple Manual References

Generally, manually coded references work just fine. We simply store the value that is present in `_id` in some other document in the database. For example:

```
> p = db.postings.findOne();
{
  "_id" : ObjectId("4b866f08234ae01d21d89604"),
  "author" : "jim",
  "title" : "Brewing Methods"
}
> // get more info on author
> db.users.findOne( { _id : p.author } )
{ "_id" : "jim", "email" : "jim@gmail.com" }
```

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, although more features may be added later. 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" }
>
```

PHP

PHP supports DB references with the [MongoDBRef class](#), as well as creation and dereferencing 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 [pymongo.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 officially 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
 - Default Indexes
 - Embedded Keys
 - Documents as Keys
 - Arrays
- Compound Keys Indexes
- Sparse Indexes
- Unique Indexes
 - Missing Keys
 - Duplicate Values
- Background Index Building
- Dropping Indexes
- ReIndex
- Additional Notes on Indexes
 - Index Performance
 - Using `sort()` without an Index
- Geospatial
- Additional Resources

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()`; to see the existing indexes on the collection.

Default Indexes

An index is always created on `_id`. This index is special and cannot be deleted. The `_id` index enforces uniqueness for its keys. For [Capped Collections](#) no index is created.

Embedded Keys

With MongoDB you can even index on a key inside of an embedded document. For example:

```
db.things.ensureIndex({"address.city": 1})
```

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" } } );
//As well as this
db.factories.find( { metro: { $gte : { city: "New York" } } } );

// But this query will not return the document because the order of the fields is significant and
doesn't match in this case
db.factories.find( { metro: { state: "NY" , city: "New York" } } );

```

An alternative to documents as keys it to create a compound index such as:

```

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.

Arrays

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.

Compound Keys Indexes

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

```

**New in 1.6+**

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.

Sparse Indexes

**Current Limitations**

A sparse index can only have one field. [SERVER-2193](#)

New in 1.7.4.

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 possibly 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({title:{$ne:null}}).sort({title:1}) // returns only Sarah
```

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 Indexes

MongoDB supports unique indexes, which guarantee that no documents are inserted whose values for the indexed keys match those of an existing document. To create an index that guarantees that no two documents have the same values for both `firstname` and `lastname` you would do:

```
db.things.ensureIndex({firstname: 1, lastname: 1}, {unique: true});
```

Missing Keys

When a document is saved to a collection with unique indexes, any missing indexed keys will be inserted with null values. Thus, it won't be possible to insert multiple documents missing the same indexed key.

```
db.things.ensureIndex({firstname: 1}, {unique: true});
db.things.save({lastname: "Smith"});

// Next operation will fail because of the unique index on firstname.
db.things.save({lastname: "Jones"});
```

Duplicate Values

A unique index cannot be created on a key that has 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})
```


Background Index Building

By default, building an index blocks other database operations. v1.3.2 and higher has a [background index build option](#) .

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()
// 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.

Repair database recreates all indexes in the database.

Additional Notes on Indexes

- MongoDB indexes (and string equality tests in general) are case sensitive.
- 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.
- Index information is kept in the system.indexes collection, run `db.system.indexes.find()` to see example data.

Index Performance

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 four megabytes). For these cases it is best to use `limit()` and `sort()` together.

Geospatial

- See [Geospatial Indexing](#) page.

Additional Resources

- [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](#)

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.

Geospatial Indexing

- [Creating the Index](#)
- [Querying](#)
- [Compound Indexes](#)
- [geoNear Command](#)
- [Bounds Queries](#)
- [The Earth is Round but Maps are Flat](#)
 - [New Spherical Model](#)
 - [Spherical Example](#)
- [Multi-location Documents](#)
- [Sharded Environments](#)
- [Implementation](#)



v1.3.3+

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 is either a sub-object or 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). Some examples:

```
{ loc : [ 50 , 30 ] }
{ loc : { x : 50 , y : 30 } }
{ loc : { foo : 50 , y : 30 } }
{ loc : { long : 40.739037, lat: 73.992964 } }
```

Creating the Index

```
db.places.ensureIndex( { loc : "2d" } )
```

By default, the index assumes you are indexing latitude/longitude and is thus configured for a [-180..180] value range.

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. Currently geo indexing is limited to indexing squares with no "wrapping" at the outer boundaries. You cannot insert values on the boundaries, for example, using the code above, the point (-500, -500) could not be inserted.



you can only have 1 geo2d index per collection right now

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)
```

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' } );
```

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" : "1100110000001111110000001111110000001111110000001111110000001111",
  "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,
    "btrellocs" : 1,
    "btrellocs" : 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:

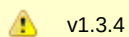
```

> db.runCommand( { geoNear : "places" , near : [ 50 , 50 ], num : 10,
... query : { type : "museum" } } );

```

query can be any regular mongo query.

Bounds Queries



v1.3.4

`$within` can be used instead of `$near` to find items within a shape. At the moment, `$box` (rectangles) and `$center` (circles) are supported.

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]}}})
```

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

In 1.7.0 we added support for correctly using spherical distances by adding "Sphere" to the name of the query. For example, use \$nearSphere or \$centerSphere (\$boxSphere doesn't really make sense so it isn'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:

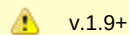
1. The code assumes that you are using [decimal degrees](#) in (X,Y) / (longitude, latitude) order. This is the same order used for the [GeoJSON spec](#).
2. All distances use [radians](#). This allows you to easily multiply by the [radius of the earth](#) (about 6371 km or 3959 miles) to get the distance in your choice of units. Conversely, divide by the radius of the earth when doing queries.
3. 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.

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 : { long : 30, lat : 30 } })
> db.points.insert({ pos : { long : -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" : {
        "long" : -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], { long : 44.2, lat : 38.2 } ] })
> db.places.ensureIndex({ "lastSeenAt" : "2d" })
```

When performing \$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 will not return duplicate documents,

Sharded Environments

Support for geospatial in sharded collections is coming; please watch this ticket: <http://jira.mongodb.org/browse/SHARDING-83>.

In the meantime sharded clusters can use geospatial indexes for unsharded collections within the cluster.

Implementation

The current implementation encodes geographic hash codes atop standard MongoDB b-trees. Results of \$near queries are exact. The problem with geohashing is that prefix lookups don't give you exact results, especially around bit flip areas. MongoDB solves this by doing a grid by grid search after the initial prefix scan. This guarantees performance remains very high while providing correct results.

Indexing as a Background Operation



Slaves and replica secondaries build all indexes in the foreground in certain releases (including the latest). Thus even when using background:true on the primary, the slave/secondary will be unavailable to service queries while the index builds there.

By default the `ensureIndex()` operation is blocking, and will stop other operations on the database from proceeding until completed. However, in v1.3.2+, a background indexing option is available.

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});
```

With background mode enabled, other operations, including writes, will not be obstructed during index creation. The index is not used for queries until the build is complete.

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.

While the 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 build process.

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 is permitted per collection.
- Some administrative operations, such as `repairDatabase`, are disallowed while a background indexing job is in progress.
- v1.4 and higher (for production usage)

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 }
```

Embedded object fields in an array

Additionally the same technique can be used for fields in embedded objects:

```
> 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
```

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.

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](#).

- [Indexing Strategies](#)
 - [Create indexes to match your queries.](#)
 - [One index per query.](#)
 - [Make sure your indexes can fit in RAM.](#)
 - [Be careful about single-key indexes with low selectivity.](#)
 - [Use explain.](#)
 - [Understanding explain's output.](#)
 - [Pay attention to the read/write ratio of your application.](#)
 - [Indexing Properties](#)
 - [1. The sort column must be the last column used in the index.](#)
 - [2. The range query must also be the last column in an index. This is an axiom of 1 above.](#)
 - [3. Only use a range query or sort on one column.](#)
 - [4. Conserve indexes by re-ordering columns used on equality \(non-range\) queries.](#)
 - [5. MongoDB's \\$ne or \\$nin operator's aren't efficient with indexes.](#)
- [FAQ](#)
 - [I've started building an index, and the database has stopped responding. What's going on? What do I do?](#)
 - [I'm using \\$ne or \\$nin in a query, and while it uses the index, it's still slow. What's happening?](#)
- [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 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 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`: the 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 required 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 will 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](#)
- [JSON](#)
- [Mongo-Friendly Schema](#)
 - [Store Example](#)

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 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 subobject 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 works 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 relational 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.

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.

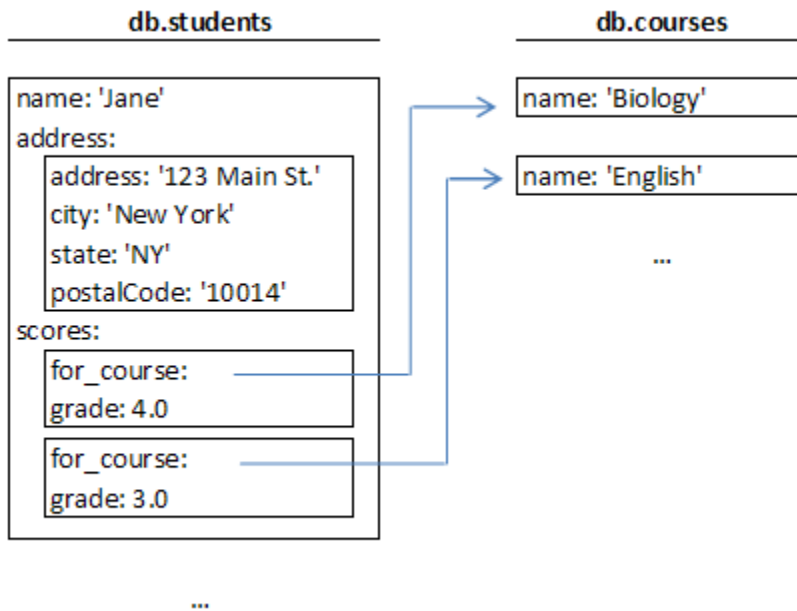
Schema Design

- [Introduction](#)
- [Embed vs. Reference](#)
- [Use Cases](#)
- [Index Selection](#)
- [How Many Collections?](#)
- [See Also](#)

Introduction

With Mongo, you do less "normalization" than you would perform designing a relational schema because there are no server-side joins. Generally, you will want one database collection for each of your top level objects.

You do not want a collection for every "class" - instead, embed objects. For example, in the diagram below, we have two collections, students and courses. The student documents embed address documents and the "score" documents, which have references to the courses.



Compare this with a relational schema, where you would almost certainly put the scores in a separate table, and have a foreign-key relationship back to the students.

Embed vs. Reference

The key question in Mongo schema design is "does this object merit its own collection, or rather should it embed in objects in other collections?" In relational databases, each sub-item of interest typically becomes a separate table (unless denormalizing for performance). In Mongo, this is not recommended - embedding objects is much more efficient. Data is then colocated on disk; client-server turnarounds to the database are eliminated. So in general the question to ask is, "why would I not want to embed this object?"

So why are references slow? Let's consider our students example. If we have a student object and perform:

```
print( student.address.city );
```

This operation will always be fast as address is an embedded object, and is always in RAM if student is in RAM. However for

```
print( student.scores[0].for_course.name );
```

if this is the first access to this course, the shell or your driver must execute the query

```
// pseudocode for driver or framework, not user code
student.scores[0].for_course = db.courses.findOne({_id:_course_id_to_find});
```

Thus, each reference traversal is a query to the database. Typically, the collection in question is indexed on `_id`. The query will then be reasonably fast. However, even if all data is in RAM, there is a certain latency given the client/server communication from appserver to database. In general, expect 1ms of time for such a query on a ram cache hit. Thus if we were iterating 1,000 students, looking up one reference per student would be quite slow - over 1 second to perform even if cached. However, if we only need to look up a single item, the time is on the order of 1ms, and completely acceptable for a web page load. (Note that if already in db cache, pulling the 1,000 students might actually take much less than 1 second, as the results return from the database in large batches.)

Some general rules on when to embed, and when to reference:

- "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 by reference.
- 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 harder to reference than "top level" objects in collections, as you cannot have a DBRef to an embedded object (at least not yet).
- It is more difficult to get a system-level view for embedded objects. For example, it would be easier to query the top 100 scores across all students if Scores were not embedded.
- If the amount of data to embed is huge (many megabytes), you may reach the limit on size of a single object.
- If performance is an issue, embed.

Use Cases

Let's consider a few use cases now.

1. Customer / Order / Order Line-Item

- orders should be a collection. customers a collection. line-items should be an array of line-items embedded in the order object.

1. Blogging system.

- posts should be a collection. post author might be a separate collection, or simply a field within posts if only an email address. comments should be embedded objects within a post for performance.

Index Selection

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 very expensive.

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. For performance reasons, we do not recommend this approach. Data within a Mongo collection tends to be contiguous on disk. Thus, table scans of the collection are possible, and efficient. Collections are very important for high throughput batch processing.

See Also

- [Video Schema Design: Data as Documents - MongoSV Presentation \(December 2010\)](#)
- [DBRef](#)
- [Trees in MongoDB](#)
- [MongoDB Data Modeling and Rails](#)
Next: Advanced Queries
- [Document Design for MongoDB - O'Reilly Ebook](#)

Trees in MongoDB

- [Patterns](#)
 - [Full Tree in Single Document](#)
 - [Parent Links](#)
 - [Child Links](#)
 - [Array of Ancestors](#)
 - [Materialized Paths \(Full Path in Each Node\)](#)
 - [acts_as_nested_set](#)
- [See Also](#)

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 (there is a 4MB per doc limit)

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," }

// or if its path not already known:
> b = t.findOne( { _id : "b" } )
{ "_id" : "b", "path" : "a,b," }
> t.find( { path : new RegExp("^" + b.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://api.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.

See Also

- Sean Cribbs [blog post](#) (source of several ideas on this page).

Optimization

- Additional Articles
- Optimizing A Simple Example
 - Optimization #1: Create an index
 - Optimization #2: Limit results
 - Optimization #3: Select only relevant fields
- Using the Profiler
- Optimizing Statements that Use count ()
- Increment Operations
- Circular Fixed Size Collections
- Server Side Code Execution
- Explain
- Hint
- See Also

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` feature. This will display "explain plan" type info about a query from the database.

When using the [mongo - The Interactive Shell](#), you can find out this "explain plan" via the `explain()` function called on a cursor. The result will be a document that contains the "explain plan".

```
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
}
```

This will tell you the type of cursor used (`BtreeCursor` is another type – which will include a lower & upper bound), the number of records the DB had to examine as part of this query, the number of records returned by the query, and the time in milliseconds the query took to execute.

- `nscanned` - number of items examined. Items might be objects or index keys. If a "covered index" is involved, `nscanned` may be higher than `nscannedObjects`.
- `nscannedObjects` - number of objects examined
- `nYields` - number of times this query yielded the read lock to let writes in
- `indexOnly` - if the query could be resolved by a covered index or not

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](#)

Optimizing Object IDs

The `_id` field in MongoDB objects is very important and is always indexed. This page lists some recommendations.

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.

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](#) are allocated in a manner such that they have this property.

Store GUIDs as `BinData`, rather than as 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()
```

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 is 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 submitted by a client. These plans are executed to return results. Thus, MongoDB supports ad hoc queries much like say, MySQL.

The database uses an interesting approach to query optimization though. Traditional approaches (which tend to be cost-based and statistical) are not used, as these approaches have a couple of problems.

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 Mongo 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.

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.

- [Query Expression Objects](#)
- [Query Options](#)
 - [Field Selection](#)
 - [Sorting](#)
 - [Skip and Limit](#)

- [slaveOk](#)
- [Cursors](#)
- [More info](#)
- [Quick Reference Card](#)
- [See Also](#)

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

When querying a replica pair or replica set, drivers route their requests to the master mongod by default; to perform a query against an (arbitrarily-selected) slave, the query can be run with the `slaveOk` option. Here's how to do so in the shell:

```
db.getMongo().setSlaveOk(); // enable querying a slave
db.users.find(...)
```

Note: some language drivers permit specifying the `slaveOk` option on each `find()`, others make this a connection-wide setting. See your language's driver for details.

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"}
>
```

More info

This was just an introduction to querying in Mongo. For the full details please look in at the pages in the "Querying" sub-section to the right of your screen.

Quick Reference Card

[Download the Query and Update Modifier Quick Reference Card](#)

See Also

- [Queries and Cursors](#)
- [Advanced Queries](#)
- [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 specifiers 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](#)"

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.

```
// 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.

Covered Indexes



New in 1.7.4

You can now return fields with data from the index used in the query. This will cover both the query and field selection from the index so there is no need to access the actual documents. You must be careful to exclude the `_id` field if it not in the index and you wish to use this feature.

```
// do a login with a covered index, returning the users roles/groups
db.users.ensureIndex( { username : 1, password : 1, roles : 1 } );
db.users.find( { username : "<username>", password : "<password>" }, { roles : 1, _id : 0 } );
```

See [Explain](#) to determine if the covered index will be used by the query or not.

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("4c23f0486dad1c3a68457d20"), "x" : { "y" : 1 } }
```

Retrieving a Subrange of Array Elements

You can use the `$slice` operator to retrieve a subrange of elements in an array.



New in MongoDB 1.5.1

```
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
```

See Also

- [example slice1](#)

Advanced Queries

- Introduction
- Retrieving a Subset of Fields
- Conditional Operators
 - <, <=, >, >=
 - \$all
 - \$exists
 - \$mod
 - \$ne
 - \$in
 - \$nin
 - \$nor
 - \$or
 - \$size
 - \$type
- Regular Expressions
- Value in an Array
 - \$elemMatch
- Value in an Embedded Object
- Meta operator: \$not
- Javascript Expressions and \$where
- Cursor Methods
 - count()
 - limit()
 - skip()
 - snapshot()
 - sort()
- Special operators
- group()
- See Also

Introduction

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} });
```

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
```



Currently \$exists is not able to use an index. Indexes on other fields are still used.

\$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 might 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);
```

\$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

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.



New in MongoDB 1.5.3

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. A number of \$or optimizations are planned for 1.8. See [this thread](#) for details.



\$or cannot be nested.

\$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.

\$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 } );
```

If you need to combine the regex with another operator, you can 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'] } } );
```

For simple prefix queries (also called rooted regexps) like `/^prefix/`, the database will use an index when available and appropriate (much like most SQL databases that use indexes for a `LIKE 'prefix%'` expression). This only works if you don't have `i` (case-insensitivity) in the flags.



While `/^a/`, `/^a./`, and `/^a.$/` are equivalent, they will have different performance characteristics. They will all use an index in the same way, but 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** 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.

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

Version 1.3.1 and higher.

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("4b5783300334000000000000aa9"),
  "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 1 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

Version 1.3.3 and higher.

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 ] } } } );
```



\$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 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);
```

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 always effectively snapshotted.

Currently, snapshot mode may not be used with sorting or explicit hints.

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 index, a full in-memory sort must be done but by using a limit() it reduces the memory and increases the speed of the operation by using an optimized sorting algorithm.

Special operators

Only return the index key:

```
db.foo.find()._addSpecial("$returnKey" , true )
```

Limit the number of items to scan:

```
db.foo.find()._addSpecial( "$maxScan" , 50 )
```

Set the query:

```
db.foo.find()._addSpecial( "$query" : {x : {$lt : 5}} )
// same as
db.foo.find({x : {$lt : 5}})
```

Sort results:

```
db.foo.find()._addSpecial( "$orderby", {x : -1} )
// same as
db.foo.find().sort({x:-1})
```

Explain the query instead of actually returning the results:

```
db.foo.find()._addSpecial( "$explain", true )
// same as
db.foo.find().explain()
```

Snapshot query:

```
db.foo.find()._addSpecial( "$snapshot", true )
// same as
db.foo.find().snapshot()
```

Set index bounds (see [min and max Query Specifiers](#) for details):

```
db.foo.find()._addSpecial("$min" , {x: -20})._addSpecial("$max" , { x : 200 })
```

Show disk location of results:

```
db.foo.find()._addSpecial("$showDiskLoc" , true)
```

Force query to use the given index:

```
db.foo.find()._addSpecial("$hint", {_id : 1})
```

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

- [Optimizing Queries](#) (including `explain()` and `hint()`)

Dot Notation (Reaching into Objects)

- [Dot Notation vs. Subobjects](#)
- [Array Element by Position](#)
- [Matching with \\$elemMatch](#)
- [See Also](#)

MongoDB is designed to store 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.

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. Before 1.5.3 MongoDB did not include an "\$or" operator for such queries, however there are ways to express such queries.

`$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 ] } }
```

`$where`

We can provide arbitrary Javascript expressions 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; } );
```

`$or`

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.



New in MongoDB 1.5.3

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.

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); } );
```

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.

Mongo DB 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.

Auditing allocated cursors

Information on allocated cursors may be obtained using the {cursorInfo:1} command.

```
db.runCommand({cursorInfo:1})
```

See Also

- [Advanced Queries](#)
- [Multikeys](#) in the [HowTo](#)

Tailable Cursors



Tailable cursors are only allowed on capped collections and can only return objects in [natural order](#).



If the field you wish to "tail" is indexed, simply requerying for { *field* : { \$gt : *value* } } is already quite efficient. Tailable will be slightly faster in situations such as that. However, if the field is not indexed, tailable provides a huge improvement in performance. Situations without indexes are the real use case for a tailable cursor.

MongoDB has a feature known as tailable cursors which are similar to the Unix "tail -f" command.

Tailable means the cursor is not closed once all data is retrieved. Rather, the cursor marks the last known object's position and you can resume using the cursor later, from where that object was located, provided more data is available.

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 may be dead upon creation if the initial query returns no matches. In this case a requery is required to create a persistent tailable cursor.

MongoDB replication uses this feature to follow the end of the master server's replication op log collection -- the tailable feature eliminates the need to create an index for the oplog at the master, which would slow log writes.

C++ example:

```

#include "client/dbclient.h"

using namespace mongo;

/* "tail" the namespace, outputting elements as they are added.
   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();
    // { $natural : 1 } means in forward capped collection insertion order
    Query query = Query().sort("$natural");
    while( 1 ) {
        auto_ptr<DBClientCursor> c =
            conn.query(ns, query, 0, 0, 0, QueryOption_CursorTailable);
        while( 1 ) {
            if( !c->more() ) {
                if( c->isDead() ) {
                    // we need to requery
                    break;
                }
                sleepsecs(1); // all data (so far) exhausted, wait for more
                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");
    }
}

```

See Also

- http://github.com/mongodb/mongo-snippets/blob/master/cpp-examples/tailable_cursor.cpp

Server-side Code Execution

- \$where Clauses and Functions in Queries
 - Restrictions
- Map/Reduce
- Using db.eval()
 - Examples
 - Limitations of eval
 - Write locks
 - Sharding
- Storing functions server-side
- Notes on Concurrency
- Running .js files via a mongo shell instance on the server

Mongo supports the execution of code inside the database process.

\$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.

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 at least slightly 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:

```
db.myCollection.find( { active: true, $where: function() { return obj.credits - obj.debits < 0; } } );
```

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. For example:

```
db.myCollection.find( { active: true, $where: "this.credits - this.debits < 0" } );
```

Restrictions

Do not write to the collection being inspected from the \$where expression.

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:ObjectId()}).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 `db.eval`

Sharding

Note also that `eval` doesn't work with sharding. If you expect your system to be sharded eventually, it's probably best to avoid `eval` altogether.

Storing functions server-side

There is a special system collection called `system.js` that can store JavaScript function to be re-used. 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`)

See <http://github.com/mongodb/mongo/tree/master/jstests/storefunc.js> for a full example

Notes on Concurrency

`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

Mongo includes utility functions which provide server-side `count`, `distinct`, and `group by` operations. More advanced aggregate functions can be crafted using [MapReduce](#).

- [Count](#)

- Distinct
- Group
 - Examples
 - Using Group from Various Languages
- Map/Reduce
- See Also

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 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 an existing member of the object (or, to access embedded members). 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](#)

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
```

If you have a document in memory and wish to delete it, the most efficient method is to specify the item's document `_id` value as a criteria:

```
db.things.remove({_id: myobject._id});
```

You may be tempted to simply pass the document you wish to delete as the selector, and this will work, but it's inefficient.



References

If a document is deleted, any existing [references](#) to the document will still exist in the database. These references will return null when evaluated.

Concurrency and Remove

v1.3+ supports concurrent operations while a remove runs. 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 undesirable, 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.

Updating

MongoDB supports atomic, in-place updates as well as more traditional updates for replacing an entire document.

- [update\(\)](#)
- [save\(\)](#) in the mongo shell
- [Modifier Operations](#)
 - `$inc`

- \$set
- \$unset
- \$push
- \$pushAll
- \$addToSet
- \$pop
- \$pull
- \$pullAll
- \$rename
- \$bit
- The \$ positional operator
- Upserts with Modifiers
- Pushing a Unique Value
- Checking the Outcome of an Update Request
- Notes
 - Object Padding
 - Blocking
 - Field (re)order
- See Also

update()

`update()` replaces the document matching criteria entirely with `objNew`. If you only want to modify some fields, you should use the atomic 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"; that is, if the record does not exist, insert it
- *multi* - if all documents matching *criteria* should be updated



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

save() in the mongo shell

The `save()` command in the [mongo shell](#) provides a shorthand syntax to perform a single object update with upsert:

```
// x is some JSON style object
db.mycollection.save(x); // updates if exists; inserts if new
```

`save()` does an upsert if `x` has an `_id` field and an insert if it does not. Thus, normally, you will not need to explicitly request upserts, just use `save()`.

Upsert means "update if present; insert if missing".

```
myColl.update( { _id: X }, { _id: X, name: "Joe", age: 20 }, 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*.

\$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.

\$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

```
{ $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 many values

```
{ $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

Version 1.7.5+ only.

```
{ $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

Version 1.3.4+ only.

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 like this "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.

```
> 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 ] }
```

\$pull can now do much of this so this example is now mostly historical (depending on your version).

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.

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 like this:

```
db.students.update({score: {$gt: 60}, $atomic: true}, {$set: {pass: true}})
```

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. ATM 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 - set a particular value (since 1.3.0)
- \$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. The following example shows how to insert monotonically increasing `_id` values into a collection using optimistic concurrency:

```
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.getLastErrorMessage();
    if( err && err.code ) {
      if( err.code == 11000 /* dup key */ )
        continue;
      else
        print("unexpected error inserting data: " + toJson(err));
    }
    break;
  }
}
```

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 fully isolated you can use the `$atomic` modifier:

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 that you are the only one writing when the update is done; this means each update is done without any interference from any other.

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
<i>query</i>	a filter for the query	{}
<i>sort</i>	if multiple docs match, choose the first one in the specified sort order as the object to manipulate	{}
<i>remove</i>	set to a true to remove the object before returning	N/A
<i>update</i>	a modifier object	N/A
<i>new</i>	set to true if you want to return the modified object rather than the original. Ignored for remove.	false
<i>fields</i>	see Retrieving a Subset of Fields (1.5.0+)	All fields
<i>upsert</i>	create object if it doesn't exist. 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. This is the same as regular sharded updates.

See Also

- [Atomic Operations](#)

two-phase commit

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 subtracted 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 "cancelling":

```
foo:PRIMARY> db.transactions.update({_id: t._id}, {$set: {state: "pending"}})
```

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.findOne();
{"_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.findOne();
{"_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.



Indexing and standard queries in MongoDB are separate from map/reduce. If you have used CouchDB in the past, note this is a big difference: MongoDB is more like MySQL for basic querying and indexing. See the [queries](#) and [indexing](#) documentation for those operations.

- [Overview](#)
 - [Output options](#)
 - [Result object](#)
- [Map Function](#)
- [Reduce Function](#)
 - [A more technical explanation](#)
- [Finalize Function](#)
- [Sharded Environments](#)
- [Examples](#)
 - [Shell Example 1](#)
 - [Shell Example 2](#)

- [More Examples](#)
- [Note on Permanent Collections](#)
- [Parallelism](#)
- [Presentations](#)
- [Troubleshooting](#)
- [See Also](#)

Overview

map/reduce is invoked via a database [command](#). The database creates a temporary collection to hold output of the operation. The collection is cleaned up when the client connection closes, or when explicitly dropped (see output options below). Alternatively, one can specify a permanent output collection name. 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 : <sort the query. useful for optimization>]
    [, limit : <number of objects to return from collection>]
    [, out : <see output options below>]
    [, keepTemp : <true|false>]
    [, finalize : <finalizefunction>]
    [, scope : <object where fields go into javascript global scope >]
    [, verbose : true]
  }
);
```

* 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)

- finalize - function to apply to all the results when finished
- verbose - provide statistics on job execution time
- scope - can pass in variables that can be access from map/reduce/finalize [example mr5](#)

Output options

If you're running **MongoDB 1.7.3 or below**, then there are two possible output options. If you don't 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.

For **versions of MongoDB greater than 1.7.4**, the output options have changed. Map-reduce will no longer generate temporary collections (so the keepTemp has been removed). Now, you must always supply a value for out. The possible values are these:

- "collectionName" - If you pass a string indicating the name of a collection, then the output will replace any existing output 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.

For example:

```
db.users.mapReduce(map, reduce, {out: { inline : 1}});
```

Result object

```
{
  [results : <document_array>],
  result : <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>]
}
```

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.

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.

Sharded Environments

In sharded environments, data processing of map/reduce operations runs in parallel on all shards.

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'} });
> 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);
> res
{"timeMillis.emit" : 9 , "result" : "mr.things.1254430454.3" ,
  "numObjects" : 4 , "timeMillis" : 9 , "errmsg" : "" , "ok" : 0}

> db[res.result].find()
{"_id" : "cat" , "value" : {"count" : 3}}
{"_id" : "dog" , "value" : {"count" : 2}}
{"_id" : "mouse" , "value" : {"count" : 1}}

> db[res.result].drop()

```

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

As of right now, MapReduce jobs on a single mongod process are single threaded. This is due to a design limitation in current JavaScript engines. We are looking into alternatives to solve this issue, but for now if you want to parallelize your MapReduce jobs, you will need to either use sharding or do the aggregation client-side in your code.

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](#)

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

DRAFT - TO BE COMPLETED.

This guide provides instructions for using MongoDB batch data processing oriented features including [map/reduce](#).

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](#)
- [Server-Side Code Execution](#)
- [Map/Reduce](#)

mongo - The Interactive Shell

- [Introduction](#)
- [More Information](#)
- [Some Notes on Datatypes in the Shell](#)
 - [Numbers](#)
 - [Dates](#)
 - [BinData](#)
- [Presentations](#)

Introduction

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.

More Information

- [Shell Overview](#)
- [Scripting the shell](#)
- [Shell Reference](#)
- [Shell API Docs](#)

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")

```

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")

```

Presentations

- [CRUD and the JavaScript Shell](#) - Presentation by Mike Dirolf at MongoSF (April 2010)

Scripting the shell

Introduction

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

Basic Syntax

```
./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

--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

There are three key differences:

1. **printing**
2. **use dbname**
3. **it**

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:

```
db2 = connect("server:27017/otherdbname")
```

it

The iterator command `it` does not work outside of the interactive scripting environment.

Overview - The MongoDB Interactive Shell

Starting 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:

<code>./mongo foo</code>	connects to the <i>foo</i> database on your local machine
<code>./mongo 192.168.13.7/foo</code>	connects to the <i>foo</i> database on 192.168.13.7
<code>./mongo dbserver.mydomain.com/foo</code>	connects to the <i>foo</i> database on dbserver.mydomain.com
<code>./mongo 192.168.13.7:9999/foo</code>	connects to the <i>foo</i> database on 192.168.13.7 on port 9999

Connecting

If you have not connected via the command line, you can use the following commands:

```
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 is 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 #
```

Basics Commands

The following are three basic commands that provide information about the available databases, and collections in a given database.

<code>show dbs</code>	displays all the databases on the server you are connected to
<code>use db_name</code>	switches to <i>db_name</i> on the same server
<code>show collections</code>	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 Data

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.

Modifying Data

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 Data

<code>db.foo.drop()</code>	drop the entire <i>foo</i> collection
<code>db.foo.remove()</code>	remove all objects from the collection
<code>db.foo.remove({ name : "sara" })</code>	remove objects from the collection where name is sara

Indexes

<code>db.foo.getIndexKeys()</code>	get all fields that have indexes on them
<code>db.foo.ensureIndex({ _field_ : 1 })</code>	create an index on <i>field</i> if it doesn't exist

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.

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 <code>Mongo()</code> or <code>connect()</code>
--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 <i>dbname</i>	Set the db variable to represent usage of <i>dbname</i> 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 <i>collection</i> within the database.
cursor = coll.find();	Find all objects in the collection. See queries .
coll.remove(objpattern);	Remove matching objects from the collection. <i>objpattern</i> is an object specifying fields to match. E.g.: <code>coll.remove({ name: "Joe" });</code>
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 <i>tab.name</i> . 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: <code>db.getSisterDB('production').getCollectionNames()</code>

Queries

coll.find()	Find all.

<code>it</code>	Continue iterating the last cursor returned from <code>find()</code> .
<code>coll.find(criteria);</code>	Find objects matching <i>criteria</i> in the collection. E.g.: <code>coll.find({ name: "Joe" });</code>
<code>coll.findOne(criteria);</code>	Find and return a single object. Returns null if not found. If you want only one object returned, this is more efficient than just <code>find()</code> as <code>limit(1)</code> is implied. You may use regular expressions if the element type is a string, number, or date: <code>coll.find({ name: /joe/i });</code>
<code>coll.find(criteria, fields);</code>	Get just specific fields from the object. E.g.: <code>coll.find({}, {name:true});</code>
<code>coll.find().sort({field:1[, field:1]});</code>	Return results in the specified order (field ASC). Use -1 for DESC.
<code>coll.find(criteria).sort({ field : 1 })</code>	Return the objects matching <i>criteria</i> , sorted by <i>field</i> .
<code>coll.find(...).limit(n)</code>	Limit result to <i>n</i> rows. Highly recommended if you need only a certain number of rows for best performance.
<code>coll.find(...).skip(n)</code>	Skip <i>n</i> results.
<code>coll.count()</code>	Returns total number of objects in the collection.
<code>coll.find(...).count()</code>	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, <code>count()</code> will return 100. This will be faster than iterating yourself, but still take time.

More information: see [queries](#).

Error Checking

<code>db.getLastError()</code>	Returns error from the last operation.
<code>db.getPrevError()</code>	Returns error from previous operations.
<code>db.resetError()</code>	Clear error memory.

Administrative Command Helpers

<code>db.cloneDatabase(fromhost)</code>	Clone the current database from the other host specified. fromhost database must be in noauth mode.
<code>db.copyDatabase(fromdb, todb, fromhost)</code>	Copy fromhost/fromdb to todb on this server. fromhost must be in noauth mode.
<code>db.repairDatabase()</code>	Repair and compact the current database. This operation can be very slow on large databases.
<code>db.addUser(user, pwd)</code>	Add user to current database.
<code>db.getCollectionNames()</code>	get list of all collections.
<code>db.dropDatabase()</code>	Drops the current database.

Opening Additional Connections

<code>db = connect("<host>:<port>/<dbname>")</code>	Open a new database connection. One may have multiple connections within a single shell, however, automatic <code>getLastError</code> reporting by the shell is done for the 'db' variable only.
<code>conn = new Mongo("hostname")</code>	Open a connection to a new server. Use <code>getDB()</code> to select a database thereafter.
<code>db = conn.getDB("dbname")</code>	Select a specific database for a connection

Miscellaneous

<code>Object.bsonsize(db.foo.findOne())</code>	prints the bson size of a db object (mongo version 1.3 and greater)
<code>db.foo.findOne().bsonsize()</code>	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?](#)
- [What RAID should I use?](#)
- [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](#)

Also check out 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.save(x)} );
```

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](#).)

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 [Atomic 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.

What RAID should I use?

We recommend not using RAID-5, but rather, RAID-10 or the like. Both will work of course.

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

Generally, with MongoDB we are not building queries from strings, so 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.

User-Generated Keys

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+FF0E ()

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 themselves 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

- http://groups.google.com/group/mongodb-user/browse_thread/thread/b4ef57912cbf09d7

How does concurrency work

- mongos
- mongod
 - Viewing Operations in Progress
 - Read/Write Lock
 - Operations
 - On Javascript
 - Multicore
 - Future

mongos

For [sharded](#) environments, mongos 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 original mongod architecture is concurrency friendly; however, some work with respect to granular locking and latching is not yet done. This means that some operations can block others. This is particular true in versions < 1.3. Version 1.3+ has improvements to concurrency, although future work will make things even better.

Viewing Operations in Progress

Use `db.currentOp()` to view operations in progress, and `db.killOp()` to terminate an operation.

You can also see operations in progress from the administrative [Http Interface](#).

Read/Write Lock

mongod uses a read/write lock for many operations. Any number of concurrent read operations are allowed, but typically only one write operation (although some write operations *yield* and in the future more concurrency will be added). The write lock acquisition is greedy: a pending write lock acquisition will prevent further read lock acquisitions until fulfilled.

The read/write lock is currently global, but [collection-level locking is coming soon](#).

Operations

Operation	Lock type	Notes
OP_QUERY (query)	Acquires read lock	see also: SERVER-517
OP_GETMORE (get more from cursor)	Acquires read lock	
OP_INSERT (insert)	Acquires write lock	Inserts are normally fast and short-lived operations
OP_DELETE (remove)	Acquires write lock	Yields while running to allow other operations to interleave.
OP_UPDATE (update)	Acquires write lock	Will yield for interleave (1.5.2+)
map/reduce	At times locked	Allows substantial concurrent operation.
create index	See notes	Batch build acquires write lock. But a background build option is available.

<code>db.eval()</code>	Acquires write lock	
<code>getLastError</code> command	Non-blocking	
<code>ismaster</code> command	Non-blocking	
<code>serverStatus</code> command	Non-blocking	

On Javascript

Only one thread in the mongod process executes Javascript at a time (other database operations are often possible concurrent with this).

Multicore

With read operations, it is easy for mongod 1.3+ to saturate all cores. However, because of the read/write lock above, write operations will not yet fully utilize all cores. This will be improved in the future.

Future

In versions 2.0 and 2.2 there will a great deal of work done on increasing concurrency. Please check this page or jira for more information.

SQL to Mongo Mapping Chart

This page not done. Please help us finish it!

MySQL Program	Oracle Program	Mongo Program
mysqld	oracle	mongod
mysql	sqlplus	mongo

MongoDB queries are expressed as JSON ([BSON](#)) objects. This quick reference 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. However, the actual verb (e.g. "find") is done in one's regular programming language. 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 Query Language Statement
CREATE TABLE USERS (a Number , b Number)	implicit; can be done explicitly
INSERT INTO USERS VALUES(1,1)	<code>db.users.insert({a:1,b:1})</code>
SELECT a,b FROM users	<code>db.users.find({}, {a:1,b:1})</code>
SELECT * FROM users	<code>db.users.find()</code>
SELECT * FROM users WHERE age=33	<code>db.users.find({age:33})</code>

<pre>SELECT a,b FROM users WHERE age=33</pre>	<pre>db.users.find({age:33}, {a:1,b:1})</pre>
<pre>SELECT * FROM users WHERE age=33 ORDER BY name</pre>	<pre>db.users.find({age:33}).sort({name:1})</pre>
<pre>SELECT * FROM users WHERE age>33</pre>	<pre>db.users.find({'age':{\$gt:33}}))</pre>
<pre>SELECT * FROM users WHERE age<33</pre>	<pre>db.users.find({'age':{\$lt:33}}))</pre>
<pre>SELECT * FROM users WHERE name LIKE "%Joe%"</pre>	<pre>db.users.find({name:/Joe/})</pre>
<pre>SELECT * FROM users WHERE name LIKE "Joe%"</pre>	<pre>db.users.find({name:/^Joe/})</pre>
<pre>SELECT * FROM users WHERE age>33 AND age<=40</pre>	<pre>db.users.find({'age':{\$gt:33,\$lte:40}}))</pre>
<pre>SELECT * FROM users ORDER BY name DESC</pre>	<pre>db.users.find().sort({name:-1})</pre>
<pre>CREATE INDEX myindexname ON users(name)</pre>	<pre>db.users.ensureIndex({name:1})</pre>
<pre>CREATE INDEX myindexname ON users(name,ts DESC)</pre>	<pre>db.users.ensureIndex({name:1,ts:-1})</pre>
<pre>SELECT * FROM users WHERE a=1 and b='q'</pre>	<pre>db.users.find({a:1,b:'q'})</pre>
<pre>SELECT * FROM users LIMIT 10 SKIP 20</pre>	<pre>db.users.find().limit(10).skip(20)</pre>

<pre>SELECT * FROM users WHERE a=1 or b=2</pre>	<pre>db.users.find({ \$or : [{ a : 1 } , { b : 2 }] })</pre>
<pre>SELECT * FROM users LIMIT 1</pre>	<pre>db.users.findOne()</pre>
<pre>EXPLAIN SELECT * FROM users WHERE z=3</pre>	<pre>db.users.find({z:3}).explain()</pre>
<pre>SELECT DISTINCT last_name FROM users</pre>	<pre>db.users.distinct('last_name')</pre>
<pre>SELECT COUNT(*y) FROM users</pre>	<pre>db.users.count()</pre>
<pre>SELECT COUNT(*y) FROM users where AGE > 30</pre>	<pre>db.users.find({age: {'\$gt': 30}}).count()</pre>
<pre>SELECT COUNT(AGE) from users</pre>	<pre>db.users.find({age: {'\$exists': true}}).count()</pre>
<pre>UPDATE users SET a=1 WHERE b='q'</pre>	<pre>db.users.update({b:'q'}, {\$set:{a:1}}, false, true)</pre>
<pre>UPDATE users SET a=a+2 WHERE b='q'</pre>	<pre>db.users.update({b:'q'}, {\$inc:{a:2}}, false, true)</pre>
<pre>DELETE FROM users WHERE z="abc"</pre>	<pre>db.users.remove({z:'abc'});</pre>

The [MongoDB Manual Pages](#) are a good place to learn more.

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 arbitrary 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](#).

Admin Zone

- [Journaling](#)
- [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

Journaling

MongoDB v1.7.5 and higher supports write-ahead journaling of operations to facilitate fast crash recovery and durability in the storage engine.

Enabling

To enable, use the `mongod --journal` command line option.

```
# mongod --journal
```



Before 1.8 the command line option was `--dur`

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.

Disabling

If journaling has been on, and you would like to run without it, simply (1) shut down `mongod` cleanly; (2) restart without the `--journal` option.

Journal Files

With `--journal` 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 using the `--journal` option. 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 in v1.8.0. In future versions, the commit frequency will be more frequent.

Commit Acknowledgement

You can wait for group commit acknowledgement with the `getLastError` command. When running with `--journal`, the `fsync:true` option returns after the data is physically written to the journal (rather than actually `fsync`'ing all the data files). Note that the group commit interval (see above) is considerable: you may prefer to call `getLastError` without `fsync`, or with a `w:` parameter instead with replication. In releases after 1.8.0 the delay for commit acknowledgement will be shorter.

FAQ

If I am using replication, can some members use --journal 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 `--journal`, 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?

Not yet, as the journal files are rotated out (unlinked) after data is safely in the data files.

32 bit nuances?

There is extra memory mapped file activity with the `--journal` option. This will further constrain the limited db size of 32 bit builds. Thus, for now it is recommended to use `--journal` with 64 bit systems.

Why isn't --journal the default?

Mainly, just to be conservative for now. It will be the default at some point in the future.

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.

See Also

- [Journaling Administration Notes](#)
- [Durability Internals](#)

Journaling Administration Notes

Journal Files

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.

prealloc Files

`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 `--journal`.

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
```

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).

The exact journal latency is not very important in 1.8.0, as in this release the group commit latency is a large number (100ms). However in future releases it will be much lower, and the limiting factor will be this time, which is the time to physically do a new write to the disk.

This command 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.)

Production Notes

- [Backups](#)
- [Recommended Unix System Settings](#)
- [TCP Port Numbers](#)
- [Linux File Systems](#)
- [Linux Kernel Versions](#)
- [Checking Disk IO](#)
- [Solid State Disks \(SSDs\)](#)
- [Tips](#)

Backups

- [Import Export Tools](#) | [Backups Doc Page](#)

Recommended Unix System Settings

- Turn off `atime`
- Set file descriptor limit to 4k+ (see `etc/limits` and `ulimit`)
- **Do not** use large VM pages with Linux ([more info](#))

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)

Linux File Systems

MongoDB uses large files for storing data, and preallocates these. Some filesystems are much better at this

- ext4
- xfs

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.

Checking Disk IO

```
iostat -x 2
```

Solid State Disks (SSDs)

Multiple MongoDB users have reported good success running MongoDB databases on solid state drives.

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

Real-world results should be lower, but the numbers are still impressive.

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.

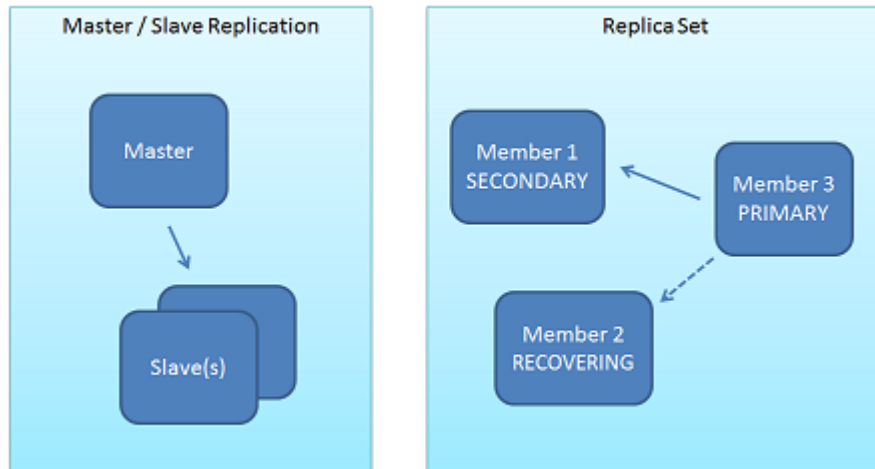
Fields

- %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 is near 100%, your server is disk I/O bound.
- r/s - reads per second
- w/s - writes per second
- rMB/s - read megabytes per second
- wMB/s - write megabytes per second

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. With a single active master at any point in time, strong consistency semantics are available. One can optionally send read operations to the slaves/secondaries when [eventual consistency](#) semantics are acceptable.

- [Master-Slave Replication](#)
- [Replica Sets](#)



Which should I use?

- if using <v1.6 : master/slave
- if need automatic fail-over and recovery (easy administration): replica sets
- if using `--auth` (security) : [for now](#), master/slave
- if using sharding : either, but replica sets are best for clusters that are not small
- if risk averse : master/slave (replica sets are new to v1.6.0)

Verifying propagation of writes with `getLastError`

A client can block until a write operation has been replicated to N servers -- [read more here](#) .

Presentations

- [Replication Video](#)
- [Replication Slides Only](#)

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)

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.

To set it up:

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 member of the set 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, `--keyFile` implies `--auth`.

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.

To convert an existing set, shut down all members and restart them with the `--keyFile` option.

You can turn off authentication by restarting the set without the `--keyFile` option.

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>
```

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, 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.

Permissions

Group and everyone must have 0 permissions. At the moment, permissions are not checked by *mongod* on Windows.

Verifying Propagation of Writes with getLastError

 v1.5+.

A client can block until a write operation has been replicated to N servers. Use the [getLastError](#) command with a new parameter w:

```
db.runCommand( { getLastError : 1 , w : 2 } )
```

If w is not set, or equals 1, the command returns immediately, implying the data is on 1 server (itself). If w is 2, then the data is on the current server and 1 other server (a secondary).

The higher w is, the longer acknowledgement may take. A recommended way of using this feature in a web context is to do all the write operations for a page, then call this once if needed. That way you're only paying the cost once.

There is an optional wtimeout parameter that allows you to timeout after a certain number of milliseconds and perhaps return an error or warning to a user. For example, the following will wait for 3 seconds before giving up:

```
> db.runCommand({getLastError : 1, w : 40, wtimeout : 3000})
{
  "err" : null,
  "n" : 0,
  "wtimeout" : true,
  "waited" : 3006,
  "errmsg" : "timed out waiting for slaves",
  "ok" : 0
}
```

Note: the current implementation returns when the data has been delivered to w servers. Future versions will provide more options for delivery vs. say, physical fsync at the server.

See also [replica set configuration](#) for information on how to change the getLastError default parameters.

Replica Sets

Replica sets are basically master/slave [replication](#), adding automatic failover and automatic recovery of member nodes. The set automatically elects a primary (master) if there is no primary currently available. Drivers can automatically detect when a replica set primary changes and will begin sending writes to the new primary.

Replica sets can be used to distribute read load and can make creating backups easier. Sets can have passive members that cannot become primary (for reading and backup) and delayed nodes (in case of user error).

 **Replica Pairs are deprecated.** (Replica Sets are effectively Replica Pairs version 2)

Docs

To get started:

- [Learn what a replica set is and is not](#)
- [Tutorial: try it out](#)
- [Learn how to configure your set](#)
- [Check out some recommended configurations](#)

If you would like to start using replica sets with an existing system:

- [Learn how to migrate your existing setup](#)
- [Upgrade your client code to use replica set connections](#) (see also your driver's documentation for details)

When running replica sets, it is important to know about:

- [The admin UI](#)
- [Administrative commands](#)

How to:

- [Adding a New Set Member](#)
- [Adding an Arbiter](#)
- [Forcing a Member to be Primary](#)
- [Moving or Replacing a Member](#)
- [Resyncing a Very Stale Replica Set Member](#)
- [Reconfiguring when members are up](#)
- [Reconfiguring when members are down](#)

More Docs

- [Sample Replica Set Config Session \(PDF\)](#)
- [Replica Sets Limits](#)
- [About the local database](#)
- [Data Center Awareness](#)
- [Troubleshooting](#)

See Also

- [Replication Video](#)
- [Replica Sets Slides](#)
- [Webcast Demo of Replica Sets](#)

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.)

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 only truly 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.

2. Writes which are committed at the primary of the set may be visible before the true cluster-wide commit has occurred.

Thus we have "READ UNCOMMITTED" read semantics. These more relaxed read semantics 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 has not replicated from the primary, the writes are un-written (thus the use of `getLastError` in #1 above).

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.

Replica Set Tutorial

This tutorial will guide you through the basic configuration of a replica set on a single machine. If you're attempting to deploy replica sets in production, be sure to read the [comprehensive replica set documentation](#). Also, do keep in mind that **replica sets are production-ready as of MongoDB 1.6**.

- [Introduction](#)
- [Starting the nodes](#)
- [Initializing the set](#)
- [Replication](#)
- [Failing Over](#)
- [Changing the replica set configuration](#)
- [Running with two nodes](#)
- [Drivers](#)

Introduction

A replica set is group of *N* mongod nodes that work together to provide automated failover.

Setting up a replica set is a two-step process that requires starting each node and then formally initiating the set. Here, we'll be configuring a set of three nodes, which is standard.

Once the mongod nodes are started, we'll issue a command to properly 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:

```
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 the name of the 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
```

The second node gets launched on port 27018:

Let's start the second node on port 27018:

```
mongod --replSet foo --port 27018 --dbpath /data/r1
```

And, finally, we'll start a third node on port 27019.

```
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 initialized it, which we'll do next.

Initializing the set

We can initiate the replica set by connecting to one of the members and running the `replSetInitiate` command. This command takes a configuration object that specifies the name of the set and each of the members.

```
mongo localhost:27017
[kyle@arete ~]$ mongo localhost:27017
MongoDB shell version: 1.5.7
connecting to: localhost:27017/test
> 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 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" : "arete.local:27017",
      "self" : true,
    },
    {
      "name" : "localhost:27019",
      "health" : 1,
      "uptime" : 101,
      "lastHeartbeat" : "Mon Aug 02 2010 11:39:07 GMT-0400",
    },
    {
      "name" : "localhost:27018",
      "health" : 1,
      "uptime" : 107,
      "lastHeartbeat" : "Mon Aug 02 2010 11:39:07 GMT-0400",
    }
  ],
  "ok" : 1
}
```

You'll see that both of 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 master node; a value of 2 indicates a slave.

You can also check the set's status in the [HTTP Admin UI](#).

Replication

Go ahead and write something to the master node:

```
db.messages.save({name: "ReplSet Tutorial"});
```

If you pay attention to the logs on the slave nodes, you'll see the write being replicated. This initial replication is essential for failover; **the system won't fail over to a new master until an initial sync between nodes is complete.**

Failing Over

Now, the purpose of a replica set is to provide automated failover. This means that, if the master node is killed, a slave node can take over. To see how this works in practice, go ahead and kill the master node with `^C`:

```
^C Mon 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 slave nodes, you'll see a series of messages indicating failover. On our first slave, we see this:

```
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 second, 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 master has gone down and, 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 up as a slave.

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 adjust the number of votes available to each node. To do this, you need to pass a new configuration object to the database's `replSetReconfig` command. Here's how.

First, define the new configuration:

```
new_config = {_id: 'foo', members: [
    {_id: 0, host: 'localhost:27017', votes: 1},
    {_id: 1, host: 'localhost:27018', votes: 2},
    {_id: 2, host: 'localhost:27019', votes: 3}]
}
```

Then, add the version to the config object. To do this, you'll need to increment the old config version.

```
use local
old_config = db.system.replset.findOne();
new_config.version = old_config.version + 1;
```

Finally, reconfigure:

```
use admin
db.runCommand({replSetReconfig: new_config});
```

Running with two nodes

Suppose you want to run replica sets with just two database servers. This is possible as long as you also use an arbiter on a separate node; most likely, running the arbiter on one or more application servers will be ideal. With an arbiter in place, the replica set will behave appropriately, recovering automatically during both network partitions and node failures (e.g., machine crashes).

You start up an arbiter just as you would a standard replica set node, 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 instead:

```
config = {_id: 'foo', members: [
    {_id: 0, host: 'localhost:27017'},
    {_id: 1, host: 'localhost:27018'},
    {_id: 2, host: 'localhost:27019', arbiterOnly: true}]
}
```

The other requirement here is that the total number of votes for the database nodes needs to consist of a majority. This means that if you have two database nodes and ten arbiters, there's a total of twelve votes. So the best bet in this case it to give each database node enough votes so that even if all but a single arbiter goes down, the master still has enough votes to stay up. In that situation, each database node would need at least three votes.

For more information on arbiters and other interesting config options, see the [replica set configuration docs](#).

Drivers

Most of the MongoDB drivers are designed to take any number of replica set seed hosts from a replica set and then cache the hosts of any other known members.

With this complete set of potential master nodes, the driver can automatically find the new master if the current master 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](#).

Replica Set Configuration

- [Command Line](#)
- [Initial Setup](#)
- [The Replica Set Config Object](#)
 - [Required 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`).

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> },
    ...
  ]
}
```

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]>,
      [, priority: <priority>]
      [, arbiterOnly : true]
      [, votes : <n>]
      [, hidden : true]
      [, slaveDelay : <n>]
      [, buildIndexes : <bool>]
      [, initialSync : {
        [state : 1|2,]
        [_id : <n>,]
        [name : <host>,]
        [optime : <date>]]
      }
    , ...
  ],

  [settings: {
    [getLastErrorDefaults: <lasterrdefaults>]
  }]
}
```

Required arguments

Every replica set configuration must contain an `_id` field and a `members` field with one or more hosts listed.

- `_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

Member options

Each member can be configured to have any of the following options.

- `arbiterOnly` - if true, this member will participate in consensus (voting) but receive no data. Defaults to false.
- `votes` - number of votes this member gets in elections. Defaults to 1. Normally you do not need to set this parameter. Sometimes useful when the number of nodes are even or for biasing towards a particular data center.
- `priority` - priority a server has for potential election as primary. The highest priority member which is up will become primary. Default 1.0. Priority zero means server can never be primary (0 and 1 are the only priorities currently supported).
- `hidden` - when true, do not advertise the member's existence to clients in `isMaster` command responses. (v1.7+)
- `slaveDelay` - how far behind this slave's replication should be (in seconds). Defaults to 0 (as up-to-date as possible). Can be used to recover from human errors (accidentally dropping a database, etc.). This option can only be set on passive members. (v1.6.3+)
- `buildIndexes` - boolean, defaults to true. If the priority is 0, you can set `buildIndexes` to false to prevent indexes from being created on this member. This could be useful on a machine which is only used for backup as there is less overhead on writes if there are no secondary indexes. Note: the `_id` index is always created.
- `initialSync` (1.7.4+) - allows you to specify where this server should initially sync from. If not specified, the server will choose the first primary or secondary it finds that is healthy. The default should usually work fine, but you can change this by specifying any of the following options:

- `state` : 1 forces the server to clone from the primary, 2 from a secondary.
- `_id` : the `_id` of the member to clone from.
- `name` : the host to clone from.
- `optime` : finds a server that is at least this up-to-date to clone from. Can be a Date or Timestamp type.

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.

- `getLastErrorDefaults` specifies defaults for the `getLastError` command. If the client calls `getLastError` with no parameters, the default object specified here is used. (v1.6.2+)

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).

To add an arbiter, bring up a new node as a replica set member (`--replicaSet` on the command line) - just like when [Adding a New Set Member](#).



For version 1.7.1 and earlier: it is best to also use `--oplogSize 1` so that 5% of your avail. disk space isn't allocated to the oplog, when it isn't needed. Version 1.7.2+ does not create an oplog for arbiters.

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 }
```

See Also

- [Adding a New Set Member](#)

Adding a New Set Member

Adding a new node to an existing replica set is easy. The new node should either have an empty data directory or a recent copy of the data from another set member. When we start the new node, we only need to 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
MongoDB shell version: ...
connecting to: test
> rs.add("broadway:27017");
{ "ok" : 1 }
```

After adding the node it will perform a full resync and come online as a secondary.

Starting with an existing copy of the dataset (`--fastsync`)

If the node is started with a recent copy of data from another node in the set (including the local database) it then does not need to do a full initial sync. You let the new member know to skip the resync by running the `mongod` with `--fastsync`:

```
$ mongod --replSet foo --fastsync
```

See Also

- [Adding an Arbiter](#)

Reconfiguring when Members are Up

Version 1.7.1+

Use the `rs.reconfig()` helper in the shell (version 1.7.1+). (Run `"rs.reconfig"` in the shell with no parenthesis to see what it does.)

```
$ mongo
> // shell v1.7.x:
> // example : give 1st set member 2 votes
> cfg = rs.conf()
> cfg.members[0].votes = 2
> rs.reconfig(cfg)
```

Earlier versions (1.6.x)

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 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 good 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.

Data Center Awareness

The 1.6.0 build of replica sets does not support much in terms of data center awareness. However additional functionality will be added in the future. Below are some suggestions configurations which work today.

Primary plus DR site

Use one site, with one or more set members, as primary. Have a member at a remote site with priority=0. For example:

```
{ _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

Another configuration would be to have one member in each of three data centers. One node arbitrarily becomes primary, the others though are secondaries and can process reads locally.

```
{ _id: 'myset',
  members: [
    { _id:0, host:'sf1', priority:1 },
    { _id:1, host:'ny1', priority:1 },
    { _id:2, host:'uk1', priority:1 }
  ]
}
```

Upgrading to Replica Sets

- [Upgrading From a Single Server](#)
- [Upgrading From Replica Pairs or Master/Slave](#)
 - [Resyncing the Slaves](#)
 - [Using a Slave's Existing Data](#)
 - [Adding the arbiter](#)

- [Upgrading Drivers](#)

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 `--repSet` 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
```

Using a Slave's Existing Data

Start up the slave with `--replSet` and `--fastsync` options.

```
s$ mongod --rest --replSet mysetname --fastsync
```

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 the arbiter

We should now add an arbiter so break ties on elections and know who is up in a network partition. An arbiter is very lightweight and can run on about 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 light in content.

```
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

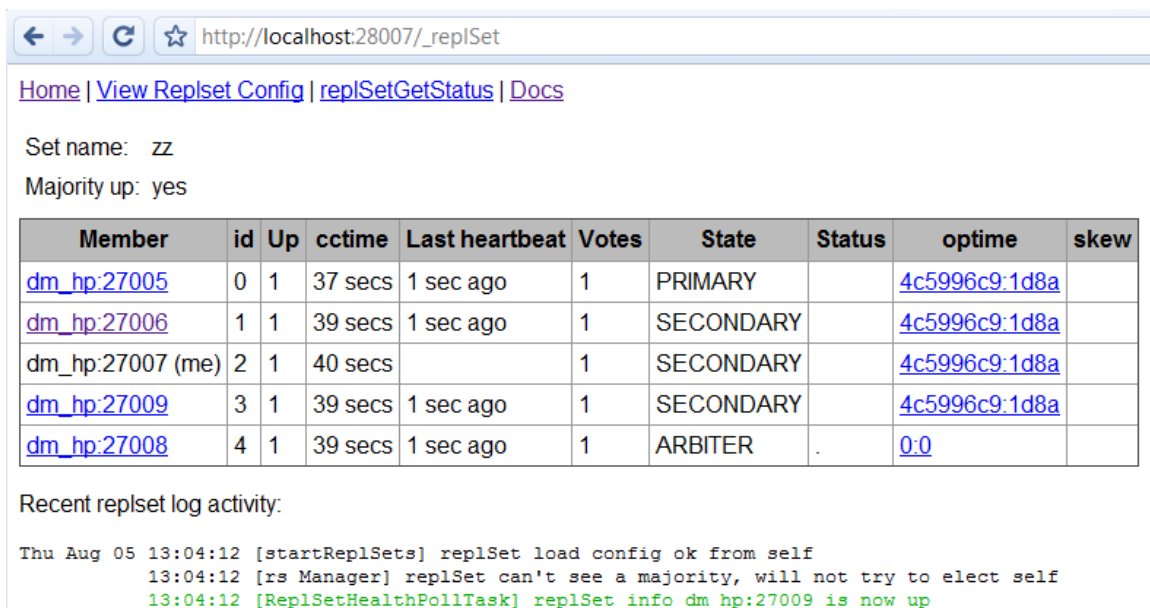
There are new versions of most MongoDB [Drivers](#) which support replica sets elegantly. See the documentation pages for the specific driver of interest.

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.



The screenshot shows a web browser window with the address bar displaying `http://localhost:28007/_replSet`. The page has navigation links: [Home](#), [View Replset Config](#), [replSetGetStatus](#), and [Docs](#). Below the links, it shows "Set name: zz" and "Majority up: yes". A table displays the status of five replica set members. The table has columns: Member, id, Up, cctime, Last heartbeat, Votes, State, Status, optime, and skew. The members are: dm_hp:27005 (PRIMARY), dm_hp:27006 (SECONDARY), dm_hp:27007 (me) (SECONDARY), dm_hp:27009 (SECONDARY), and dm_hp:27008 (ARBITER). Below the table, it shows "Recent replset log activity:" with three log entries: "Thu Aug 05 13:04:12 [startReplSets] replSet load config ok from self", "13:04:12 [rs Manager] replSet can't see a majority, will not try to elect self", and "13:04:12 [ReplSetHealthPollTask] replSet info dm_hp:27009 is now up".

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	

Recent replset log activity:

```
Thu Aug 05 13:04:12 [startReplSets] replSet load config ok from self
13:04:12 [rs Manager] replSet can't see a majority, will not try to elect self
13:04:12 [ReplSetHealthPollTask] replSet info dm_hp:27009 is now up
```

See Also

- [Http Interface](#)

Replica Set Commands

- [Shell Helpers](#)
- [Commands](#)
 - `{ isMaster : 1 }`
 - `{ replSetGetStatus : 1 }`
 - `{ replSetInitiate : <config> }`
 - `{ replSetReconfig: <config> }`
 - `{ replSetStepDown : <seconds> }`
 - `{ replSetFreeze : <seconds> }`

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 from the set
rs.stepDown()	{ replSetStepDown : true }
rs.conf()	return configuration from local.system.replset
db.isMaster()	check who is primary

Commands

{ isMaster : 1 }

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:

```
{
  "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",
  "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 : 1 }

Status information on the replica set from this node's point of view.

The output looks like:

```
{
  "set" : "florble",
  "date" : "Wed Jul 28 2010 15:01:01 GMT-0400 (EST)",
  "myState" : 1,
  "members" : [
    {
      "name" : "dev1.example.com",
      "self" : true,
      "errmsg" : ""
    },
    {
      "name" : "dev2.example.com",
      "health" : 1,
      "uptime" : 13777,
      "lastHeartbeat" : "Wed Jul 28 2010 15:01:01 GMT-0400 (EST)",
      "errmsg" : "initial sync done"
    }
  ],
  "ok" : 1
}
```

The *myState* field indicates the state of this server. Valid states are:

0	Starting up, phase 1 (parsing configuration)
1	Primary
2	Secondary
3	Recovering
4	Fatal error
5	Starting up, phase 2 (forking threads)
6	Unknown state
7	Arbiter
8	Down
9	Rollback

The *health* field indicates the health of this server. Valid states are:

0	Server is down
1	Server is up

The *errmsg* field can contain informational messages, as shown above.

{ 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. See also [Configuration](#).

{ replSetReconfig: <config> }

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.

{ 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.)

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.

`{ replSetFreeze : <seconds> }`

v1.7.3+ (also in currently nightly).

'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.

However, if for administrative reasons you want to force a node to be primary at a given point in time, use the `replSetFreeze` and `replSetStepdown` commands. 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.



`replSetFreeze` is new to v1.7.2

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.

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

How long does failover take?

Failover thresholds are configurable. With the defaults, it may take 20-30 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, including during this window.

Should I use replica sets or replica pairs?

After 1.6, use [Replica Sets](#).

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()
```

Reconfiguring a replica set when members are down

One may modify a set when some members are down as long as a majority is established. In that case, simply send the `reconfig` command to the [current primary](#).

If there is no primary (and this condition is not transient), no majority is available. Reconfiguring a minority partition would be dangerous as two sides of a network partition won't both be aware of the reconfiguration. Thus, this is not allowed.

However, in some administrative circumstances we will want to take action even though there is no majority. Suggestions on how to deal with this are outlined below.

Example 1

A replica set has three members, which in the past were healthy. Two of the servers are permanently destroyed. We wish to bring the remaining member online immediately.

One option is to make the last standing mongod a standalone server and not a set member:

1. stop the 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.

Example 2

A replica set has three members, which in the past were healthy. Two of the servers are permanently destroyed. We wish to bring the remaining member online and add a new member to its set.

We cannot reconfigure the existing set with only 1 of 3 members available. However, we can "break the mirror" and start a new set:

1. stop the surviving mongod
2. consider doing a backup...
3. delete the `local.*` datafiles in the data directory.
4. restart the mongod with a new replica set name
5. initiate this new set
6. then, add the new second member

Example 3

A replica set has five members, which in the past were healthy. Three of the servers are permanently destroyed. We wish to bring the remaining members online.

As in example 2 we will use the "break the mirror" technique. Unfortunately one of the two members must be re-synced.

1. stop the surviving mongod's
2. consider doing a backup...
3. delete the `local.*` datafiles on server 1
4. delete (ideally just move to a backup location) all the datafiles from server 2
5. restart both mongod's with the new replica set name on the command line for each

6. initiate this new set on server 1
7. then, add the new second member (server 2)

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.

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 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.

1. **Delete all data:** If you stop the failed mongod, delete all data, and restart it, it will automatically resynchronize itself. Of course this may be slow if the database is huge or the network slow.
2. **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. On a slow network, snapshotting all the datafiles from another (inactive) member to a gzipped tarball is a good solution. Also similar strategies work well when using SANs and services such as Amazon Elastic Block Service snapshots.
3. **Find a member with older data:** Each member of the replica set has an oplog. It is possible that a member has a larger oplog than the current primary.

Replica Sets Limits

v1.6

- Authentication mode not supported. [JIRA](#)
- Limits on config changes to sets at first. Especially when a lot of set members are down.
- 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.
- A set can contain
 - A maximum of 12 members
 - A maximum of 7 members that can vote

Replica Set Internals

- [Design Concepts](#)
- [Configuration](#)
 - [Command Line](#)
 - [Node Types](#)
 - [local.system.replset](#)
 - [Set Initiation \(Initial Setup\)](#)
- [Design](#)
 - [Server States](#)
 - [Applying Operations](#)
 - [OpOrdinal](#)
 - [Picking Primary](#)
 - [Heartbeat Monitoring](#)
 - [Assumption of Primary](#)
 - [Failover](#)
 - [Resync \(Connecting to a New Primary\)](#)
 - [Consensus](#)
 - [Increasing Durability](#)

- [Reading from Secondaries and Staleness](#)
- [Example](#)
- [Administration](#)
- [Future Versions](#)

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.

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.

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)

Master Slave

- [Configuration and Setup](#)
- [Command Line Options](#)
 - [Master](#)
 - [Slave](#)
 - [--slavedelay](#)
- [Diagnostics](#)
- [Security](#)
- [Master Slave vs. Replica Sets](#)
- [Administrative Tasks](#)
 - [Failing over to a Slave \(Promotion\)](#)

- [Inverting Master and Slave](#)
- [Creating a slave from an existing master's disk image](#)
- [Creating a slave from an existing slave's disk image](#)
- [Resyncing a slave that is too stale to recover](#)
- [Correcting a slave's source](#)
- [See Also](#)

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 to 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 period.

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

--fastsync is a way to start a slave starting with an existing master disk image/backup. This option declares that the administrator 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.

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.mississippi
```

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.mississippi"}, {$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](#)

Replica Pairs



Replica pairs will be removed for 1.8

Replica pairs should be migrated to their replacement, Replica Sets.

- [Setup of Replica Pairs](#)
- [Consistency](#)
- [Security](#)
- [Replacing a Replica Pair Server](#)
- [Querying the slave](#)
- [What is and when should you use an arbiter?](#)
- [Working with an existing \(non-paired\) database](#)

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, oplogs are quite large - perhaps 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.

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](#)

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 resync 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, deleting 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. 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.* oplog 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 --oplog=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'll be creating ten 2GB datafiles. 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.

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.

Sharding

MongoDB scales horizontally via an auto-sharding architecture.

Sharding offers:

- Automatic balancing for changes in load and data distribution
- Easy addition of new machines
- Scaling out to one thousand nodes
- No single points of failure
- Automatic failover

Documentation

1. What Is Sharding?

Here we provide an introduction to MongoDB's auto-sharding, highlighting its philosophy, use cases, and its core components.

2. How To Set Up and Manage a Cluster

How to set up a sharding cluster and manage it.

- [Configuration](#)
- [Administration](#)
- [Failover](#)

3. Sharding Internals

Auto-Sharding implementation details.

4. Restrictions and Limitations

5. FAQ

Common questions.

Presentations and Further Materials

- [How Sharding Works](#) - O'Reilly Webcast (February 2011)
- [Sharding in Detail](#) - Video from MongoSV (December 2010)
- [Scaling MongoDB](#) - O'Reilly ebook
- [How queries work with sharding, an overview](#)

Sharding Introduction

MongoDB supports an automated sharding 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](#)
 - [Balancing and Failover](#)
 - [Scaling Model](#)
- [Architectural Overview](#)
 - [Shards](#)
 - [Shard Keys](#)
 - [Chunks](#)
 - [Config Servers](#)
 - [Routing Processes](#)
 - [Operation Types](#)
 - [Server Layout](#)
 - [Configuration](#)

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 Virginia	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 sharding mechanism would only kick in if the amount of data you were storing reached the threshold where sharding is advantageous. This probably wouldn't apply to user data but, for now, we can image you'll be dealing with enough data that MongoDB will want to slice and distribute data across servers.

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 continues to look like a single-node database system to our application. But the system's capacity is greatly enhanced. If our users collection receives heavy writes, those writes are now distributed across three shard servers. 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 occurs on a per-collection basis, not on the database as a whole. This makes sense since, as our application grows, certain collections will grow much larger than others. 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. In the context on MongoDB's sharded architecture, non-sharded collections will reside on just one of the sharded nodes.

Balancing and Failover

A sharded architecture needs to handle balancing and failover. 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.

Automated failover is also quite important since proper system functioning requires that each shard node 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, frequently three or more, 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 master. If the master replica fails, the remaining replicas are capable of electing a new master. Thus is automated failover provided for the individual shard.

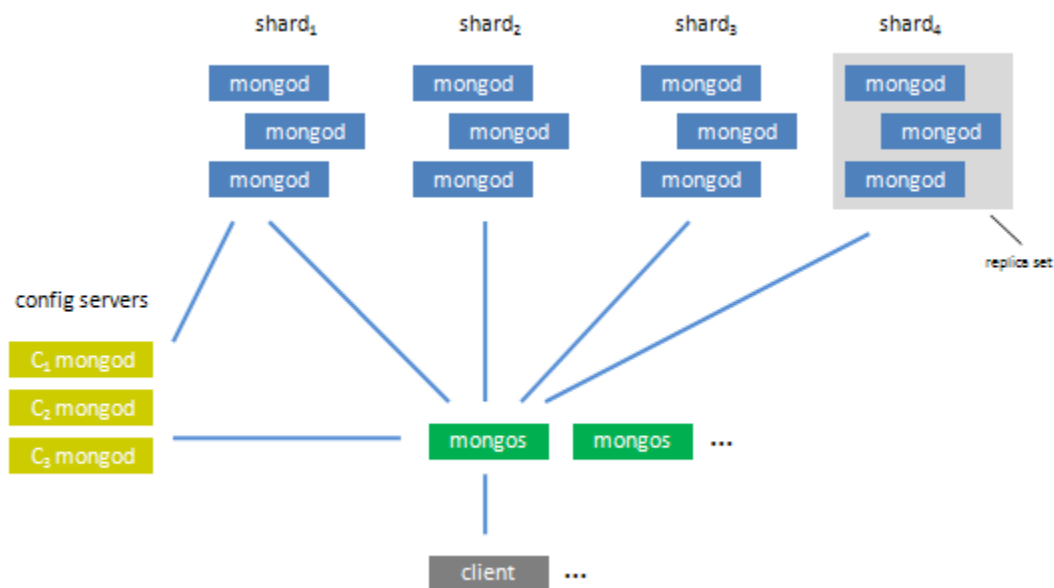
Replica sets were another focus of development in 1.5.x (along with sharding). See the [documentation on replica sets](#) for more details.

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 replicated servers per shard to ensure availability and automated failover. The set of servers/mongod process within the shard comprise a *replica set*.

[Replica sets](#), as discussed earlier, represent an improved version of MongoDB's replication ([SERVER-557](#)).

For testing, you can use sharding with a single mongod instance per shard. If you need redundancy, use one or more slaves for each shard's mongod master. This configuration will require manual failover until replica sets become available.

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 tends 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 $\text{minKey} \leq K < \text{maxKey}$.

Chunks grow to a maximum size, usually 200MB. 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, keep in mind that these values should be *granular* enough to ensure an even distribution of data. For instance, in the above example, where we're sharding on `name`, we have to be careful that we don't have a disproportionate number of users with the same name. In that case, the individual chunk can become too large and find itself unable to split, e.g., where the entire range comprises just a single key.

Thus, if it's possible that a single value within the shard key range might grow exceptionally large, it's best to use a compound shard key instead so that further discrimination of the values will be possible.

Config Servers

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 servers 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

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 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.

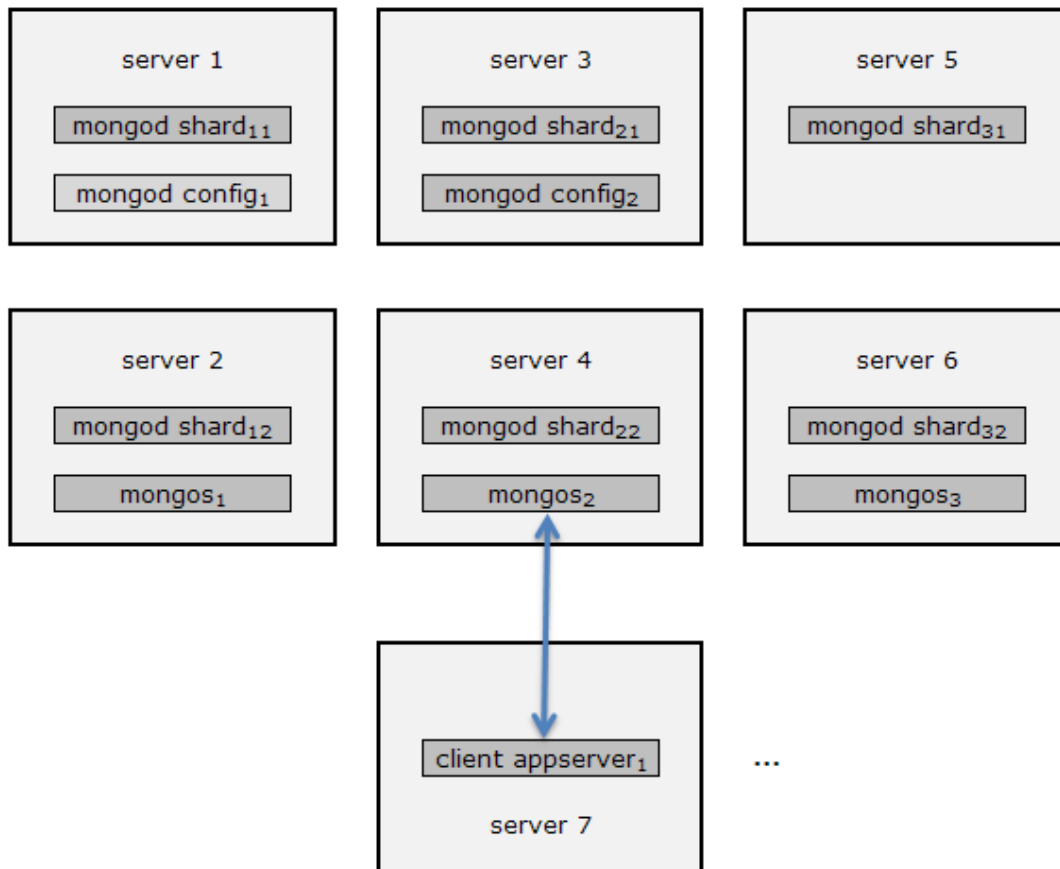
The following table shows various operations and their type. For the examples below, assume a shard key of `{ x : 1 }`.

Operation	Type	Comments
<code>db.foo.find({ x : 300 })</code>	Targeted	Queries a single shard.
<code>db.foo.find({ x : 300, age : 40 })</code>	Targeted	Queries a single shard.
<code>db.foo.find({ age : 40 })</code>	Global	Queries all shards.
<code>db.foo.find()</code>	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>) db.foo.remove({ age : 40 })	Global	
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, then, is an example where some sharing of physical machines is used to lay out a cluster.



Yet more configurations are imaginable, especially when it comes to mongos. For example, it's possible to run mongos processes on all of servers 1-6. Alternatively, as suggested earlier, the mongos processes can exist on each application server (server 7). There is some potential benefit to this configuration, as the communications between app server and mongos then can occur over the localhost interface.

Configuration

Sharding becomes a bit easier to understand once you've set it up. It's even possible to set up a sharded cluster on a single machine. To try it for yourself, see the [configuration docs](#).

Configuring Sharding

Introduction

This document describes the steps involved in setting up a basic sharding cluster. A sharding cluster requires, at minimum, three components:

1. Two or more shards.
2. At least one config server.
3. A mongos routing process.

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, 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; if you're the kind of person who prefers 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.

- [Introduction](#)
- [Sharding Components](#)
 - [Shard Servers](#)
 - [Config Servers](#)
 - [mongos Router](#)
- [Configuring the Shard Cluster](#)
 - [Adding shards](#)
 - [Optional Parameters](#)
 - [Listing shards](#)
 - [Removing a shard](#)
- [Enabling Sharding on a Database](#)
- [Sharding a Collection](#)
- [Relevant Examples and Docs](#)

Sharding Components

First, start the individual shards, config servers, and mongos processes.

Shard Servers

Any mongod can become a shard server. For auto failover support, use a replica set as a shard. See [replica sets](#) for more info.

Note that replica pairs are deprecated will never be supported as shards.

To get started with a simple test, we recommend running a single mongod process per shard, as a test configuration doesn't demand automated failover.

Config Servers

Run mongod on the config server(s) with the `--configsvr` command line parameter. If you're only testing, you can use only one config server. For production, you're expected to run three of them.

Note: config servers take care of replicating data to each other (and 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`.

mongos Router

Run mongos on the servers of your choice. Specify the `--configdb` parameter to indicate location of the config database(s).

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 } );
```

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

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

```
> db.runCommand( { removeshard : "localhost:10000" } );
{ msg : "draining started succesfully" , state: "started" , shard : "localhost:10000" , 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 : "localhost:10000" } );
{ 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. You would need to remove these with the following command:

```
> db.runCommand( { moveprimary : "test", to : "localhost:10001" } );
{ "primary" : "localhost:10001", "ok" : 1 }
```

When the shard is empty, you could issue the 'removeshard' command again and that will clean up all metadata information:

```
> db.runCommand( { removeshard : "localhost:10000" } );
{ msg: "remove shard completed successfully", stage: "completed", host: "localhost:10000", ok : 1 }
```

After the 'removeshard' command reported being done with that shard, you can take that process down.

Enabling Sharding on a Database

Once you've added one or more shards, you can enable sharding on a database. This is an important step; without it, all data in the database will be stored on the same shard.

```
> 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 real partitioning of data, we have to shard an individual collection.

Sharding a Collection

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 create upfront (it speeds up the chunking process) otherwise an index will be automatically created for you.

```
> db.runCommand( { shardcollection : "<namespace>",
                  key : <shardkeypatternobject> } );
```



Important note: running "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 back up 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 }
```

One note: a sharded collection can have only **one unique index, which must exist on the shard key**. No other unique indexes can exist on the collection (except `_id`). You can use the `"unique: true"` option to ensure that the underlying index enforces uniqueness (it only works if the collection is empty until [SERVER-2326](#) is resolved)

```
db.runCommand( { shardcollection : "test.users", key : { email : 1 }, unique : true } );
```

In the end, picking the right shard key for your needs is extremely important for successful sharding. For more information on choosing the key, be sure to check [Choosing a Shard Key](#)

Relevant Examples and Docs

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](#).

Docs

- [Sharding Administration](#)
- [Notes on TCP Port Numbers](#)

A Sample Configuration Session

The following example uses two shards (one server each), 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 200MB 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

Example Document

```
{
  server : "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 fine enough that you won't get large chunks that can't be split.

Using the logging example above, if you chose

```
{server:1}
```

as the shard key, then all data for a single server is 1 chunk. You can easily imagine having more than 200mb (default chunk size) of data for a server. If the shard key was

```
{server:1,time:1}
```

then you can keep splitting up data for a single server down to the millisecond. So as long as you don't have 200mb of data per server per millisecond, no chunk will ever not be splittable.

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 was

```
{server:1,application:1,time:1}
```

then each server,application pair will be written somewhere different. So if there are 100 server,application pairs, and 10 servers, then each server will get 1/10th the writes.

Query isolation

The other consideration is how many shards any query has to hit. Ideally a query would go directly from mongos to the mongod that owns the document requested. So if you know that most common queries start with a condition on *server*, then starting the shard key with *server* will make your setup much more efficient.

All queries will work regardless of shard key, but if mongos cannot determine a single mongod shard that owns the data, it will have to hit all shards sequentially. This will increase the response latency time, and increase the volume of network traffic and load.

Sorting

When a request asks for a `sort()` along a key that is on the left side the shard key, mongos is able to ask each mongod sequentially the pre-sorted result, and just streams it back to the requestor. This results in minimum processing time and resource utilization for mongos.

If instead the `sort()` uses a key that is not on the left of the shard key, mongos will have to request sorted results from all mongod servers in parallel, and merge the results back to the requestor, which will add extra strain on mongos.

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",
}
```

Since the system is write intensive, and you want to spread the writes on all servers, you need to use either the *id* or *user_id* as shard key. The *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 on 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: "2011-01-02_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 server 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 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: "2011-01-02_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.

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 Session](#) to get to know the basic commands involved.



Sharding is a new feature introduced at the 1.6.0 release. This page assumes your non-sharded mongod is on that release.

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-shard0000>
      }
    },
    ...
  ]
}
```

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 200MB in version 1.6.0, 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, is all.

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](#)
- [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.

```
> config = db.getSisterDB("config")
> config.system.namespaces.find()
```

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 the 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. In other words, 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. On the steady state, the size of chunks should be in the range of 100-200MBs of data. This range 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 -- exaggerating 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 between 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, 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.



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](#)

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](#)

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 be between a half and the maximum chunk size in the steady state. The default maximum chunk size has been 200MB (sum of objects in the collection, not counting indices).

Chunk size has been intensely debated -- and much hacked. So let's understand the tradeoff's that that choice of size implies.

When you move data, there's some state resetting in mongos, the query router. Queries that used to hit a given shard for a the migrated chunk, now need to hit a new shard. This state resetting isn't free, so we want to move chunks not too frequently (pressure to move a lot of keys at once). But 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 via the --maxSize parameter of the mongos. 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 Chunks](#)

Backing Up Sharded Cluster

Backing up a sharded cluster is fairly complex because there are so many moving pieces. That is why it is very important to have a very resilient setup when using sharding:

- at least 3 config servers in at least 2 different locations
- each shard should be a 3 server replica set, with at least 2 different locations.

Following is information on how to backup to full sharded environment. Refer to [Backups](#) and [Import Export Tools](#) for more information on backuping.

Small clusters

If you have a small sharded cluster, the easiest thing to do is use mongodump through a mongos.

This is easy, but only works if one machine can store all the data reasonably.

The dump will contain both the sharding config metadata and the data itself.

Note that this procedure cannot be used if you need to modify the hosts information before populating the data (in that case, use the procedure below).

Large clusters

Backup

- Turn off the balancer

```
// connect to mongos
> use config
> db.settings.update( { _id: "balancer" }, { $set : { stopped: true } } , true );
```

- For consistency purpose, if possible, run lock+fsync on each shard, to make sure no more data is getting written to the system.
- On each shard run mongodump, to create a dump independently for each shard.
- Run mongodump from one of the config servers

- Turn balancer back on

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

Restore

- no server should be running (mongod, mongos, config)
- start shard servers (mongod) and use mongorestore to restore each shard independently. This is the same procedure as restoring a replica set.
- start the config servers and use mongorestore to restore them.



Each config server must be restored independently, since they are not using replication themselves.

- if shard IPs/hosts are different, you have to manually update config.shards to use the new servers.
- start mongos and connect to it. Use "printShardingStatus()" or "show collections" to make sure all shards are correctly seen.

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 will be available as of MongoDB v1.6. Read more about [replica set internals](#) or follow the [jira issue](#).

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.

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

Differences from Unsharded Configurations

- Sharding must be ran 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.

[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-unique 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).

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 1.6 we put a cap on the maximum size of the original collection of 25GB (increased to 256GB in 1.8). 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. (*please contact 10gen support directly if you need help in this situation*) 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).
1. **Time:** When sharding an existing collection, please be aware that this process will take some time. If a collection has 200GB of data, the process of re-chunking that data is tantamount to moving 100GB+ of data while maintaining existing read/write rates.

Query limitations

Queries in a sharded environment behave differently than queries in an unsharded environment. MongoDB sharding supports two [styles of operations](#) -- **targeted** and **global**. Before moving to a sharded configuration, it is suggested that you understand these limits.

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 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 – agreed 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 $4 \leq v < 9$, 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"
}
```

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" : 2 }
```

settings

Key/Value table for configurable options (chunkSize)

```
> db.settings.find()
{ "_id" : "chunksize", "value" : 200 }
{ "_id" : "balancer", "who" : "ubuntu:27017", "x" : ObjectId("4bd0cb39503139ae28630ee9") }
```

shards

Stores information about the shards.

```
> db.shards.findOne()
{ "_id" : "shard0", "host" : "localhost:30001" }
```

databases

```
{
  "_id" : "admin",
  "partitioned" : false,
  "primary" : "localhost:20001"
}
```

chunks

```
{
  "_id" : "test.foo-x_MinKey",
  "lastmod" : {
    "t" : 1271946858000,
    "i" : 1
  },
  "ns" : "test.foo",
  "min" : {
    "x" : { $minKey : 1 }
  },
  "max" : {
    "x" : { $maxKey : 1 }
  },
  "shard" : "localhost:30002"
}
```

mongos

Record of all mongos affiliated with this cluster. mongos will ping every 30 seconds so we know who is alive.

```
> 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 thought 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 partitioning (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 partitioning, might help here
 - replicas only work if the dataset is really small as we are using/wasting the same RAM on each replica. thus, partitioning 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 shard's 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 Chunks

Normally, splitting chunks is done automatically for you. Currently, the splits happen as a side effect of inserting (and are transparent). In the future, there may be other cases where a chunk is automatically split.

A recently split chunk may be moved immediately to a new shard if the system finds that future insertions will benefit from that move. (Chunk moves are transparent, too.)

Moreover, MongoDB has a sub-system called Balancer, which constantly monitors shards loads and, as you guessed, moves chunks around if it finds an imbalance. Balancing chunks automatically helps incremental scalability. If you add a new shard to the system, some chunks will eventually be moved to that shard to spread out the load.

That all being said, in certain circumstances one may need to force a split manually.



The Balancer will treat all chunks the same way, regardless if they were generated by a manual or an automatic split.

The following command splits the chunk where the _id 99 would reside in two. The key used as the middle key is computed internally to roughly

divide the chunk in equally sized parts.

```
> use admin
switched to db admin
> db.runCommand( { split : "test.foo" , find : { _id : 99 } } )
...
```

Pre-splitting

There is a second version of the split command that takes the exact key you'd like to split on.

In the example below the command splits the chunk where the `_id` 99 would reside using that key as the split point. 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.

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?
- 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?

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. Typically in production one would use a [replica set](#) for each shard.

Where do unsharded collections go if sharding is enabled for a database?

In alpha 2 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 chunk size is 200mb, so you need at least 200mb 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. If a shard is responding slowly, mongos will wait for it. You won't get partial results.

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. 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 serialize over shards in X-order; if the query sorts by Z, the query will parallelize over shards and perform a merge sort keyed on Z of the documents found.

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 have been 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.

Hosting Center

Cloud-Style

- [MongoHQ](#) provides cloud-style hosted MongoDB instances

- [Mongo Machine](#)
- [MongoLab](#) is currently in private beta
- [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.

Dedicated Servers

- [ServerBeach](#) offers preconfigured, dedicated MongoDB servers [Blog](#)
- [EngineYard](#) supports MongoDB on its private cloud.

VPS

- [\(mt\) Media Temple's \(ve\) server platform](#) is an excellent choice for [easy MongoDB deployment](#).
- [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.

Setup Instructions for Others

- [Amazon EC2](#)
- [Joyent](#)
- [Linode](#)
- [Webfaction](#)



Amazon EC2

- [Instance Types](#)
- [Linux](#)
- [EC2 TCP Port Management](#)
- [EBS Snapshotting](#)
- [EBS vs. Local Drives](#)
- [Distribution Notes](#)
- [Backup, Restore & Verify](#)
- [Presentations](#)

MongoDB runs well on [Amazon EC2](#) . This page includes some notes in this regard.

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.

Linux

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` attributes, for example


```
/dev/mapper/my_vol /var/lib/mongodb xfs noatime,noexec,nodiratime 0 0
```

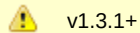
The create the mongodb datafile directory in the desired location and then run the database:

```
mkdir /mnt/db  
./mongod --fork --logpath ~/mongod.log --dbpath /mnt/db/
```

EC2 TCP Port Management

By default the database will now be listening on port 27017. The web administrative UI will be on port 28017.

EBS Snapshotting



If your datafiles are on an EBS volume, you can snapshot them for backups. Use the fsync lock 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 slave databases.

EBS vs. Local Drives

Local drives may be faster than EBS; however, they are impermanent. One strategy is to have a hot server which uses local drives and a slave which uses EBS for storage.

We have seen sequential read rates by MongoDB from ebs (unstriped) of 400Mbps on an extra large instance box. (YMMV)

Distribution Notes

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

Backup, Restore & Verify

[Tips for backing up a MongoDB on EC2 with EBS.](#)

Presentations

- [MongoDB on Amazon EC2 - Webinar \(March 2011\)](#)

Joyent

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/lsinfo -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

```

Monitoring and Diagnostics

- [Query Profiler](#)
- [Http Console](#)
- [mongo Shell Diagnostic Commands](#)
- [Trending/Monitoring Adaptors](#)
- [Hosted Monitoring](#)
- [Database Record/Replay](#)
- [Checking Server Memory Usage](#)
- [Database Profiler](#)
- [Munin configuration examples](#)
- [serverStatus](#)
- [Http Interface](#)
- [mongostat](#)
- [mongosniff](#)
- [Admin UIs](#)

Query Profiler

Use the [Database Profiler](#) to analyze slow queries.

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. (May block other operations too.)
- `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: 1.)ganglia-gmond, 2.)mongodb-ganglia
- cacti
- Mikoomi provides a MongoDB plugin for Zabbix
- Nagios
- mtop - A top like utility for Mongo

Chris Lea from (mt) Media Temple has made an [easy to install Ubuntu package](#) for the munin plugin.

Hosted Monitoring

- [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

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 < 'some_log.bin' | hexdump -c
```

Checking Server Memory Usage

Checking using DB Commands

The serverStatus() command provides memory usage information.

```
> 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). The difference should be relatively small compared to total RAM on the machine. Also watch the delta over time; if it is increasing consistently, that could indicate a leak.

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.

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.

Checking via Unix Commands

mongod uses memory-mapped files; thus the memory stats in top are not that useful. 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, use `vm_stat`.

Memory Mapped Files

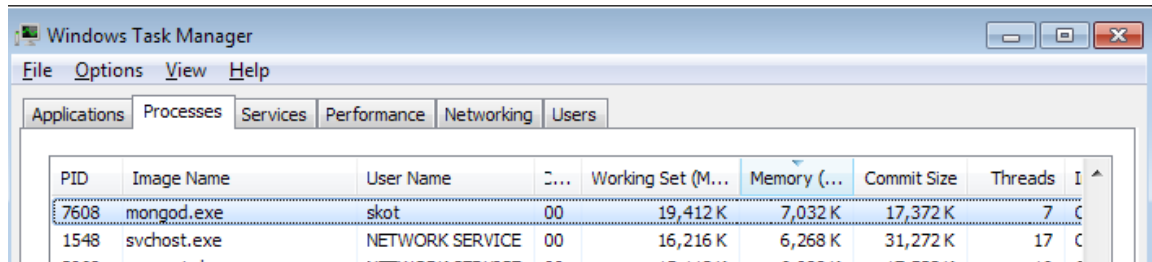
Depending on the platform you may see the mapped files as memory in the process, but this is not strictly correct. Unix top may show way more memory for mongod than is really appropriate. The Operating System (the virtual memory manager specifically, depending on OS) manages the memory where the "Memory Mapped Files" reside. This number is usually shown in a program like "free -lmt".

It is called "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
```

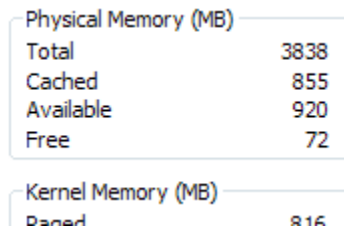
Checking in Windows

By bringing up the Task Manager you can see the process memory allocation for mongod.



PID	Image Name	User Name	...	Working Set (M...	Memory (...)	Commit Size	Threads	I
7608	mongod.exe	skot	00	19,412 K	7,032 K	17,372 K	7	C
1548	svchost.exe	NETWORK SERVICE	00	16,216 K	6,268 K	31,272 K	17	C

In addition in the Performance tab the "cached" memory which represents the memory allocated by the memory mapped (data) files.



Physical Memory (MB)	
Total	3838
Cached	855
Available	920
Free	72

Kernel Memory (MB)	
Committed	816

Historical Memory Leak Bugs (that are fixed)

(12 issues)		
Key	FixVersion	Summary
SERVER-2497	1.7.6	memory leak in ModSet::prepare()
SERVER-2511	1.9.0	memory leak of _MultiCommandJob
SERVER-2522	1.7.6	State::reduceInMemory leaks InMemory on exception
SERVER-2520	1.9 planning	add comments in code about intentional memory leak in ReplSetImpl::initFromConfig()
SERVER-1827		Memory leak when there's multiple query plans with empty result
SERVER-768	1.3.4	Memory leak and high memory usage from snapshots thread
SERVER-774		MessagingPorts are leaking
SERVER-2122	debugging with submitter	memory leak of shard + replication
SERVER-2558	1.8.0-rc0	memory and cursor leak in FindingStartCursor

SERVER-1897		admin page plugins and handlers leak memory
SERVER-2498	2011 - phase 2	small memory leak when closing a database
SERVER-2521	1.9 planning	1300 byte memory leak in PiggyBackData on socket exception

Database Profiler

Mongo includes a profiling tool to analyze the performance of database operations.

See also the [currentOp](#) command.

Enabling Profiling

To enable profiling, from the mongo shell invoke:

```
> db.setProfilingLevel(2);
{"was" : 0 , "slowms" : 100, "ok" : 1}
> db.getProfilingLevel()
2
```

Profiling levels are:

- 0 - off
- 1 - log slow operations (>100ms)
- 2 - log all operations



New in 1.7.x

`getProfilingLevel()` is being replaced with `getProfilingStatus()` which shows both the level and slowms.

In addition to the default levels you can also specify a `slowms` option:

```
> db.setProfilingLevel(1,20)
{"was" : 0, "slowms" : 100, "ok" : 1 }
> db.getProfilingStatus()
{"was" : 1, "slowms" : 20 }
```

Setting via the command-line/config-file

You can also enable profiling on the command line, `--profile=1`, and `--slowms=15`



When profiling is enabling, there is continual writing to the `system.profile` table. This is very fast but does use a write lock which has certain implications for concurrency. An alternative which has no impact on concurrency is to use the `currentOp` command.

Viewing

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}
...
```

To see output without `$cmd` (command) operations, invoke:

```
db.system.profile.find( function() { return this.info.indexOf('$cmd')<0; } )
```

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}
```

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})
```

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, `findOne()` sets **ntoreturn** to 1. `limit()` 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. `save()` 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 `find()` 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 Performance

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.

Configuring "Slow"

Since 1.3.0 there are 2 ways to configure "slow"

- `--slowms` on the command line when starting `mongod` (or file config)
- `db.setProfilingLevel( level , slowms )`

```
db.setProfilingLevel( 1 , 10 );
```

will log all queries over 10ms to `system.profile`

See Also

- [Optimization](#)
- [explain\(\)](#)
- [Viewing and Terminating Current Operation](#)

Munin configuration examples

Overview

[Munin](#) can be used to monitor aspects of your running system. The following is a mini tutorial to help you setup 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 pre-built 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 : determine 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

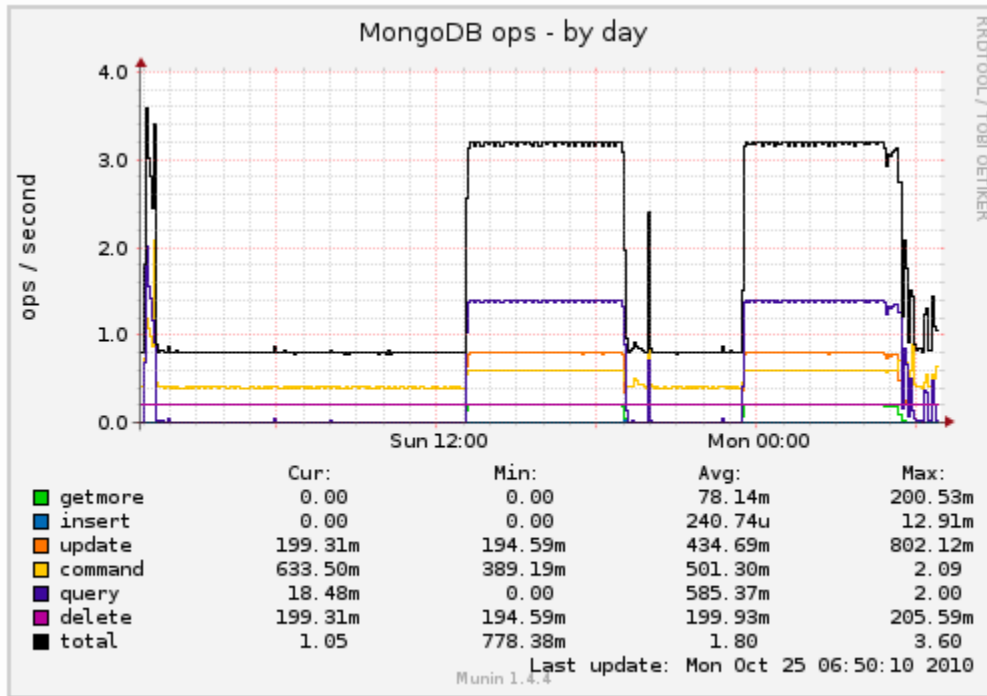
```
shell> wget http://github.com/erh/mongo-munin/tarball/master
shell> tar xvf erh-mongo-munin-*.tar.gz
shell> cp erh-mongo-munin-*/mongo_* /etc/munin/plugins/
```

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 setup 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

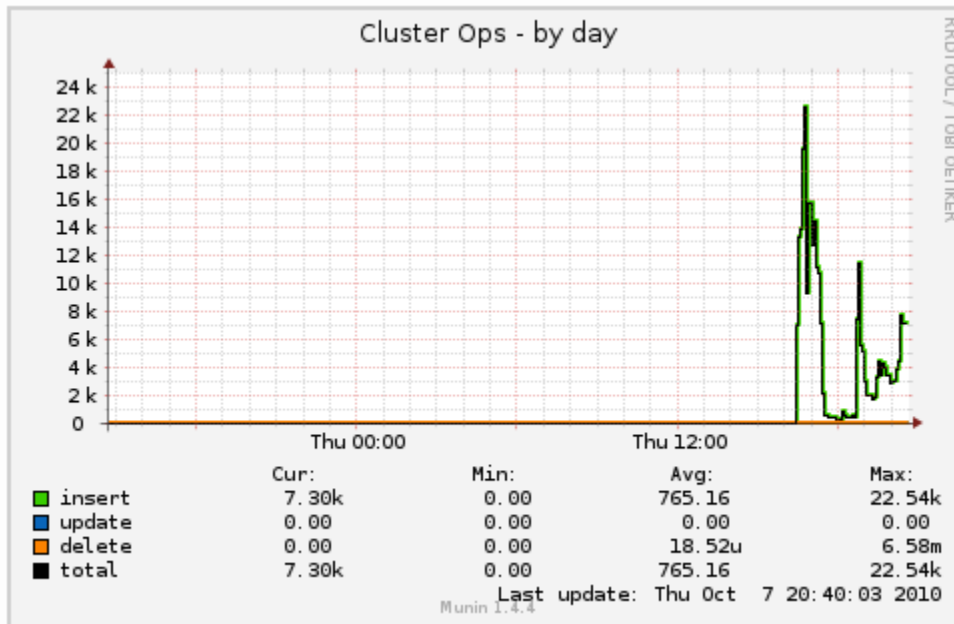
Now lets define a chart to aggregate the inserts, updates and deletefor the cluster

```
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 is looks like



serverStatus

The serverStatus command provides very useful diagnostic information for a mongod instance.

Run

```
> db.serverStatus()
```

in the mongo shell.

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 startup that 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
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	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. 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 access found the document it was looking for
indexCounters.btree.misses	2	The number of times a btree access could not find the document it was looking for
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
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 journal entries waiting to be flushed to disk
dur.commits	0	The number of commits to the journal that have occurred in the last interval
dur.journalMB	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 stats were 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 journaling
dur.timeMs.remapPrivateView	0	The amount of time spent remapping copy-on-write memory mapped views
ok	1	Whether or not serverStatus returned correctly

Http Interface

- [REST Interfaces](#)
 - [Sleepy Mongoose \(Python\)](#)
 - [MongoDB Rest \(Node.js\)](#)
- [HTTP Console](#)
 - [HTTP Console Security](#)
- [Simple REST Interface](#)
 - [JSON in the simple REST 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 REST interface for convenience. For full REST capabilities we recommend using an external tool such as [Sleepy.Mongoose](#).

Note: in v1.3.4+ of MongoDB, 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
```

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 0 1 0 0m 2.36g 4m 0 0 0|0 0|0 62b 1k 1 17:25:35
0 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 0 1 0 0m 2.36g 4m 0 0 0|0 0|0 62b 1k 1 17:25:37
0 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
```

multiple servers:

```
mongostat --host a,b,c
```

find all connected servers (added in 1.7.2):

```
mongostat --discover (--host optional)
```

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.

```
sudo yum install libpcap-devel
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.

Backups

- [Snapshot Backups with Journaling \(--journal\) Enabled](#)
- [Shutdown and Backup](#)
- [Exports](#)
- [Slave Backup](#)
- [Fsync, Write Lock and Backup](#)
- [Community Stuff](#)

Several strategies exist for backing up MongoDB databases. A word of warning: it's not safe to back up 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. See the fsync option below for a way around that.

Snapshot Backups with Journaling (--journal) Enabled

It is safe to take an lvm/ebs snapshot of an entire dbpath (including the journal subdirectory) of a mongod running with --journal. As long as all files are snapshotted at the same point in time, you don't need to fsync-lock the database. It is **not** safe to simply copy the files without locking or snapshotting, even with --journal, since the journal and the datafiles might not be consistent with each other.

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.

Exports

The `mongodump` utility may be used to dump an entire database, even when the database is running and active. The dump can then be restored later if needed.

Slave Backup

Another good technique for backups is replication to a slave database. The slave polls master continuously and thus always has a nearly-up-to-date copy of master.

We then have several options for backing up the slave:

1. Fsync, write lock, and backup the slave.
2. Shut it down, backup, and restart.
3. Export from the slave.

For methods 1 and 2, after the backup the slave will resume replication, applying any changes made to master in the meantime.

Using a slave is advantageous because we then always have backup database machine ready in case master fails (failover). But a slave also gives us the chance to back up the full data set without affecting the performance of the master database.

Fsync, Write Lock and Backup

MongoDB v1.3.1 and higher supports an `fsync` and `lock` command with which we can flush writes, lock the database to prevent writing, and then backup the datafiles.

While in this locked mode, all writes will block. If this is a problem consider one of the other methods below.

Note: 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.

Community Stuff

- <http://github.com/micahwedemeyer/automongobackup>

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

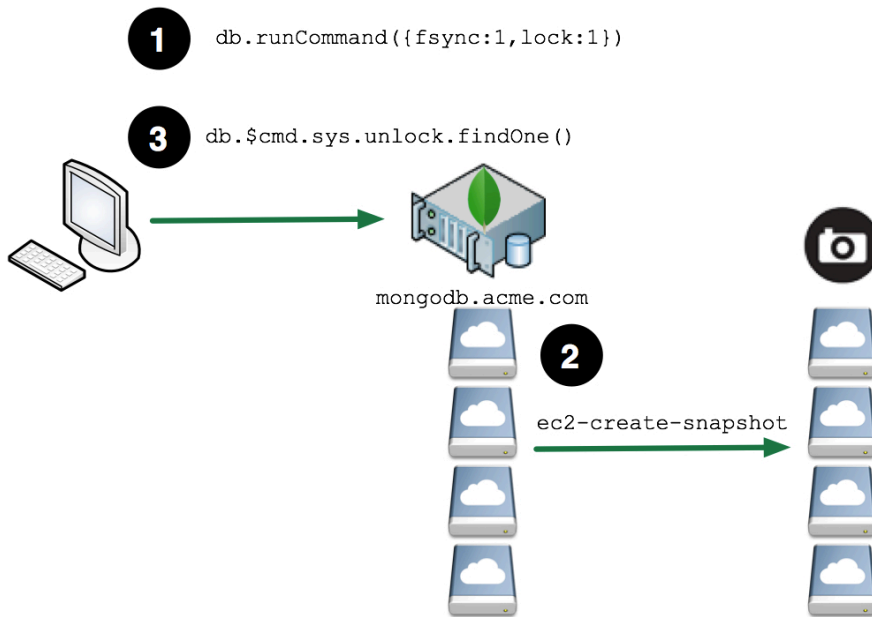
The journal file allows for roll forward recovery. The journal files are located in the `dbpath` directory so will be snapshotted at the same time as the database files.

Proceed to [Backup the database files](#) section.

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                fix0yX-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 device. 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

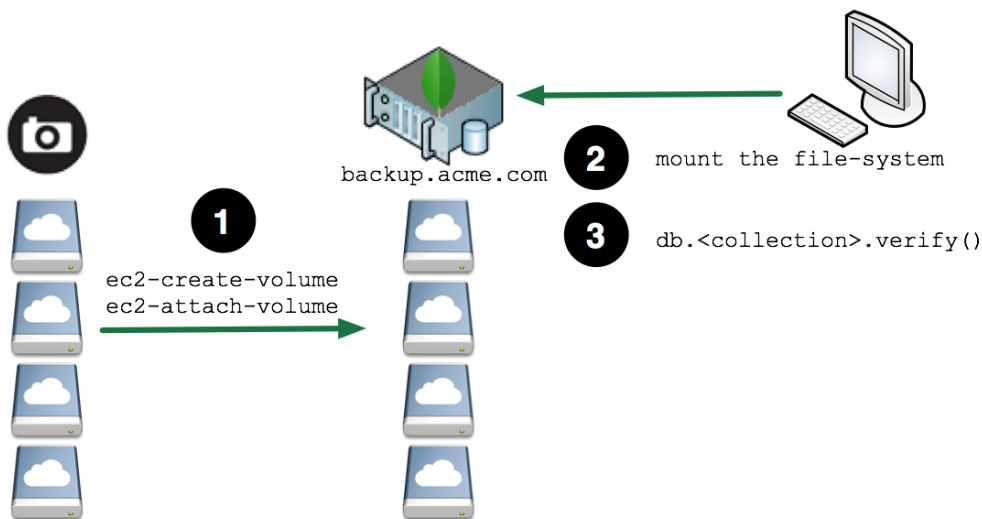
After the snapshots have been created, the database can be unlocked. After this command has been executed the database will be available to process write requests.

```
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 completed

- 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/tokyo0
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.

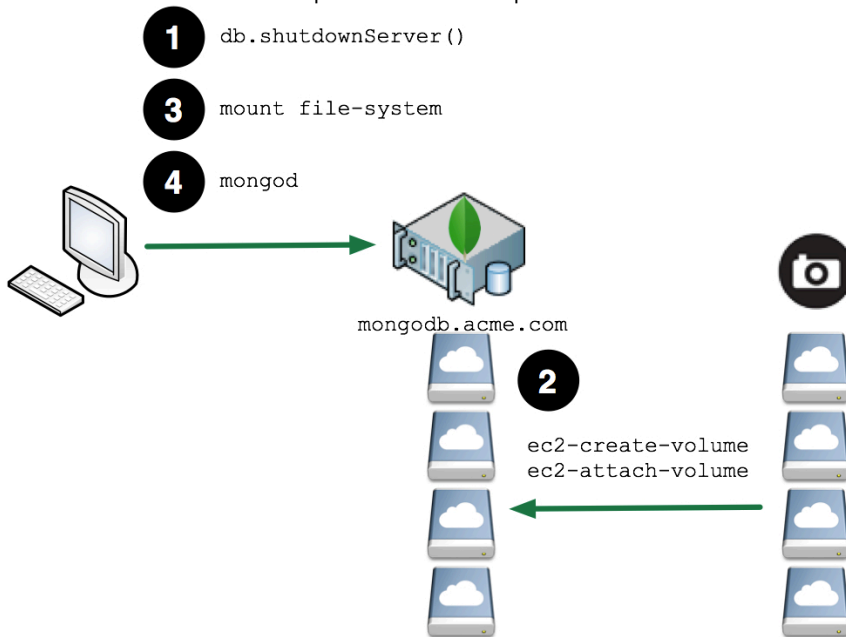
```

mongo shell> db.blogs.verify()

```

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.

Mongo DB 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

- [mongoimport](#)
 - [Example: Importing with upsert](#)
 - [Example: Importing Interesting Types](#)
- [mongoexport](#)
- [mongodump](#)
 - [Example: Dumping Everything](#)
 - [Example: Dumping a Single Collection](#)
 - [Example: Dumping a Single Collection to Stdout](#)
 - [Example: Dumping a Single Collection with a query](#)
- [mongorestore](#)
- [bsondump](#)



If you just want to do [Clone Database](#) from one server to another you don't need these tools.



These tool work with the raw data (the documents in the collection; both user and system); they do not save, or load, the metadata like the (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.

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 separated 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: 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.

Example: Importing Interesting Types

MongoDB supports more types than 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

This utility 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 fidelity please use mongodump.

If you want to output CSV, you have to specify the fields in the order you want them.

Example

```
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 separated list of field names e.g. -f name,age
--csv                 export to csv instead of json
-o [ --out ] arg       output file; if not specified, stdout is used
```

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 try to migrate data this way, this will dump shard configuration information and overwrite configurations upon restore.

```
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)              d/--db)
```

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}
    }'
```

mongorestore

This takes the output from *mongodump* and restores it. Indexes will be created on a restore. *Mongorestore* will do a (fire-and-forget) insert to restore the data; if you want to check for errors during the restore you should look at the server logs.

```
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)
  -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
  --drop                drop each collection before import
  --objcheck            validate object before inserting
  --filter arg          filter to apply before inserting
  --drop               drop each collection before import
  [--indexesLast        wait to add indexes (faster if data isn't inserted in
                        index order)]
  --oplogReplay         Restores the dump and replays the backed up portion of the oplog.
```



If you do not wish to create indexes (the default indexes for `_id` will always be created) you can removed the `system.indexes.bson` file from your database(s) dump directory

bsondump



Added in 1.6

This takes a bson file and converts it to json/debug output.

```
usage: ./bsondump [options] [filename]
options:
  --help                produce help message
  --type arg (=json)    type of output: json,debug
```

Durability and Repair

- [Single Server Durability and Recovery](#)
- [Repair Command](#)
- [Validate Command](#)
- [--syncdelay Command Line Option](#)
- [See Also](#)

Single Server Durability and Recovery

v1.7.5+ of MongoDB supports durability with recovery (via write ahead log – a journal) in the storage engine as an option. See the [Journaling](#) page. When enabled, repairs are not required on a hard crash.



The `--dur` option was used before 1.8; now the option is `--journal`

Repair Command

When not using `--journal`, 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. If a (non `--journal`) slave crashes, another option is just to restart the slave from scratch. When journaling is enabled, it should not 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.

If the databases exited uncleanly and you attempt to restart the database, *mongod* will print:

```
*****
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. After running with `--repair`, *mongod* will start up normally.



When running the repair on a slave server (replica set), you will get an error stating that the server is not master. In order to run the repair, restart the slave without the `--replSet` option so that the server is in single db mode, and run the repair. Then restart one more time with the `--replSet` option on. This may put the slave back in a consistent state, but it is highly recommended to check the data validity by comparing a dump of the master and slave. If there is a suspicion of data being corrupted, it is safer to rebuild the slave from scratch.

Validate Command

Alternatively one could restart and run the *validate* command on select collections. The *validate* command checks 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: 00000000100000000000
  deleted: n: 1 size: 7956
  nIndexes:2
    test.users.$_id_ keys:1
    test.users.$username_1 keys:1 ",
  "ok" : 1,
  "valid" : true,
  "lastExtentSize" : 8192
}

```

--syncdelay Command Line Option

Since 1.1.4, the --syncdelay option controls how often changes are flushed to disk (the default is 60 seconds). If replication is not being used, it may be desirable to reduce this default.

See Also

- [What About Durability? \(MongoDB Blog\)](#)
- [fsync Command](#)
- [MongoDB \(Single-Server\) Data Durability Guide](#)

Security and Authentication

MongoDB includes a basic authentication scheme that works with single server deployments and replica sets. However, the built-in authentication does not work with sharding, so for this and other cases, we recommend running the database in a secure environment.

If you're interested in securing a replica set, see the [replica set authentication](#) docs.

- [Running Without Security \(Trusted Environment\)](#)
- [Mongo Security](#)
- [Configuring Authentication and Security](#)
 - [Changing Passwords](#)
 - [Deleting Users](#)
 - [Ports](#)
 - [IP Address Binding](#)
- [Report a Security Issue](#)

Running Without Security (Trusted Environment)



Trusted environment is the default option and is recommended.

One valid way to run the Mongo database is in a trusted environment, with no 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.

A trusted environment is currently the only way of securing a sharded cluster.

Mongo Security

The current version of Mongo supports only very basic security. One authenticates 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 in question while a read only user only has read access.

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 server process.

Run the database (mongod process) with the `--auth` option to enable security. You **must** either have added a user to the admin db before starting the server with `--auth`, or add the first user from the localhost interface.

Configuring Authentication and Security

Authentication is stored in each database's `system.users` collection. For example, on a database `projectx`, `projectx.system.users` will contain user information.

We should first configure an administrator user for the entire db server process. This user is stored under the special admin database.

If no users are configured in `admin.system.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 a user in `admin.system.users`.

```
> db.auth("theadmin", "anadminpassword")
```

We can view existing users for the database with the command:

```
> db.system.users.find()
```

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)
```

Changing Passwords

The shell `addUser` command may also be used to update a password: if the user already exists, the password simply updates.

Many 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 } )
```

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)

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 127.0.0.0, 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.

Admin UIs

Several administrative user interfaces, or GUIs, are available for MongoDB. [Tim Gourley's blog](#) has a good summary of the tools.

- [Fang of Mongo](#)
- [Futon4Mongo](#)
- [Mongo3](#)
- [MongoHub](#)
- [MongoVUE](#)
- [Mongui](#)
- [Myngo](#)
- [Opricot](#)
- [PHPMoAdmin](#)
- [RockMongo](#)
- [MongoVision](#)
- [MongoExplorer](#)
- The built-in replica set admin UI

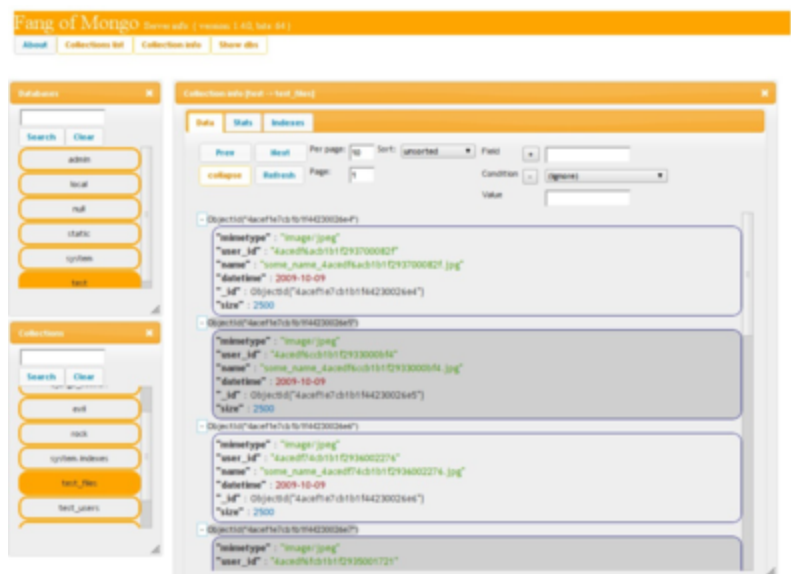
Commercial Offerings

- [DatabaseMaster](#)

Details

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

See it in action at: <http://blueone.pl:8001/fangofmongo/>
 Get it from github: <http://github.com/Fiedzia/Fang-of-Mongo>
 Or track progress on twitter: [@fangofmongo](https://twitter.com/fangofmongo)

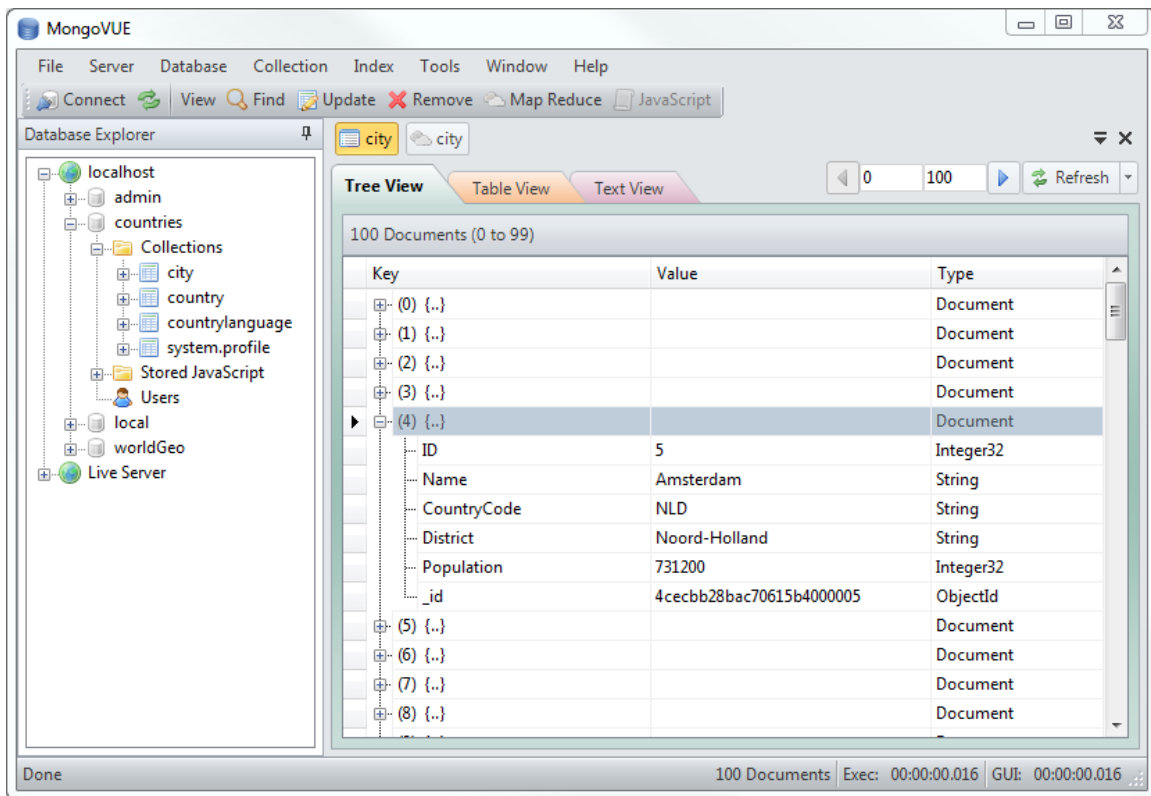
MongoHub

MongoHub is a native OS X GUI.



MongoVUE

MongoVUE is a .NET GUI for MongoDB.



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).



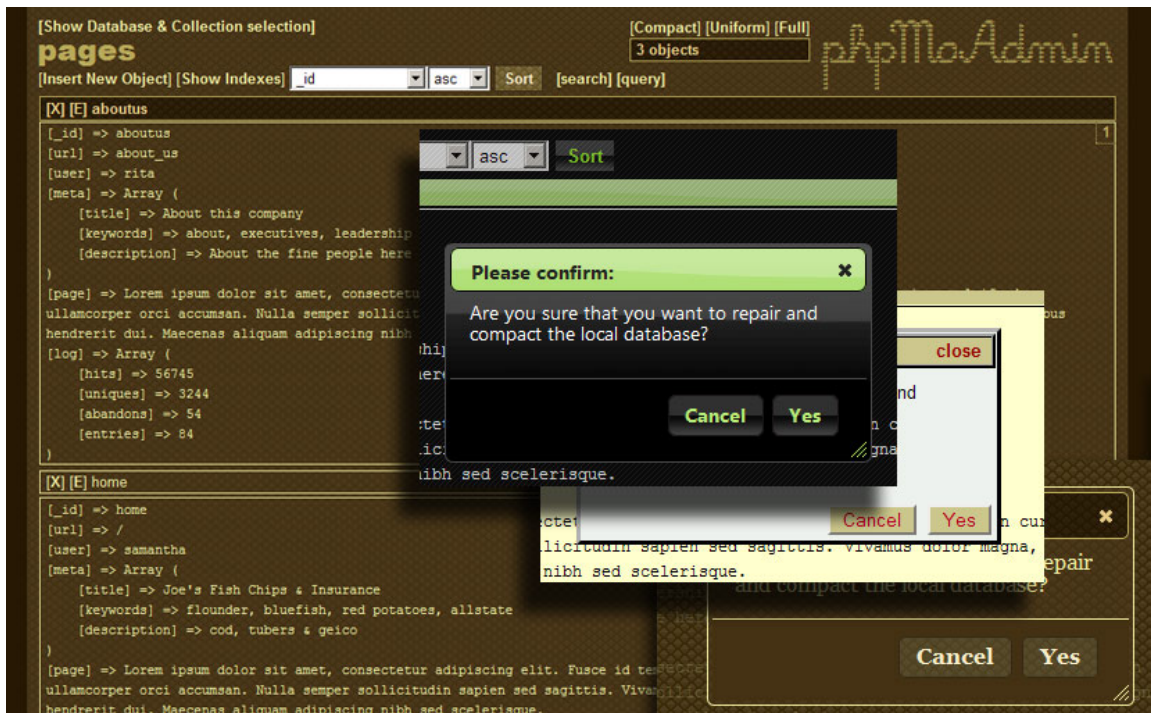
PHPMoAdmin

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](#)

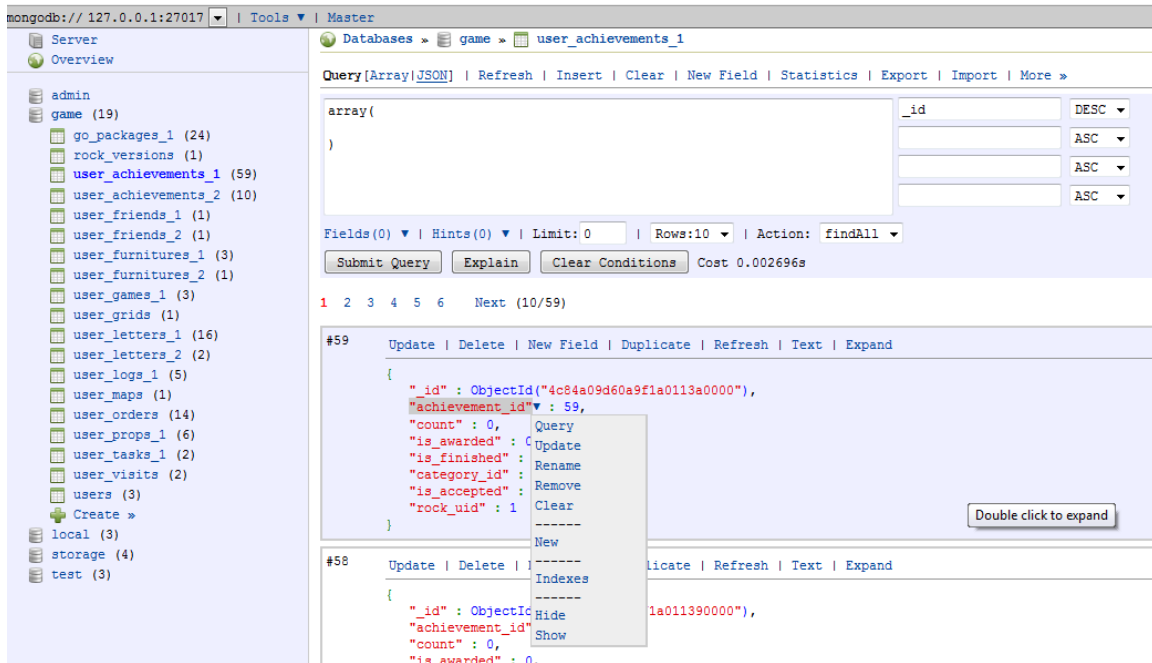


RockMongo

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

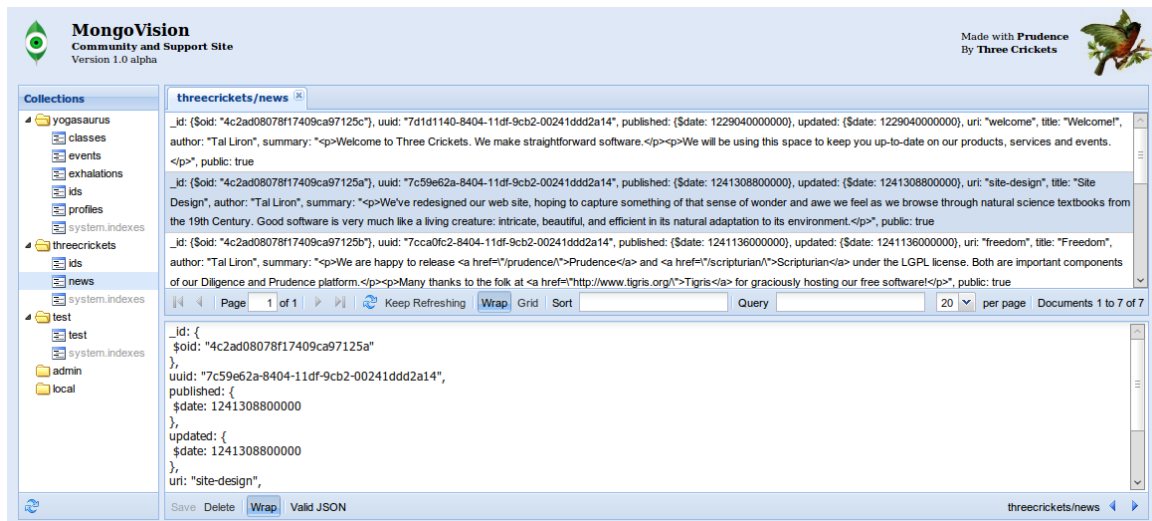


MongoVision

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

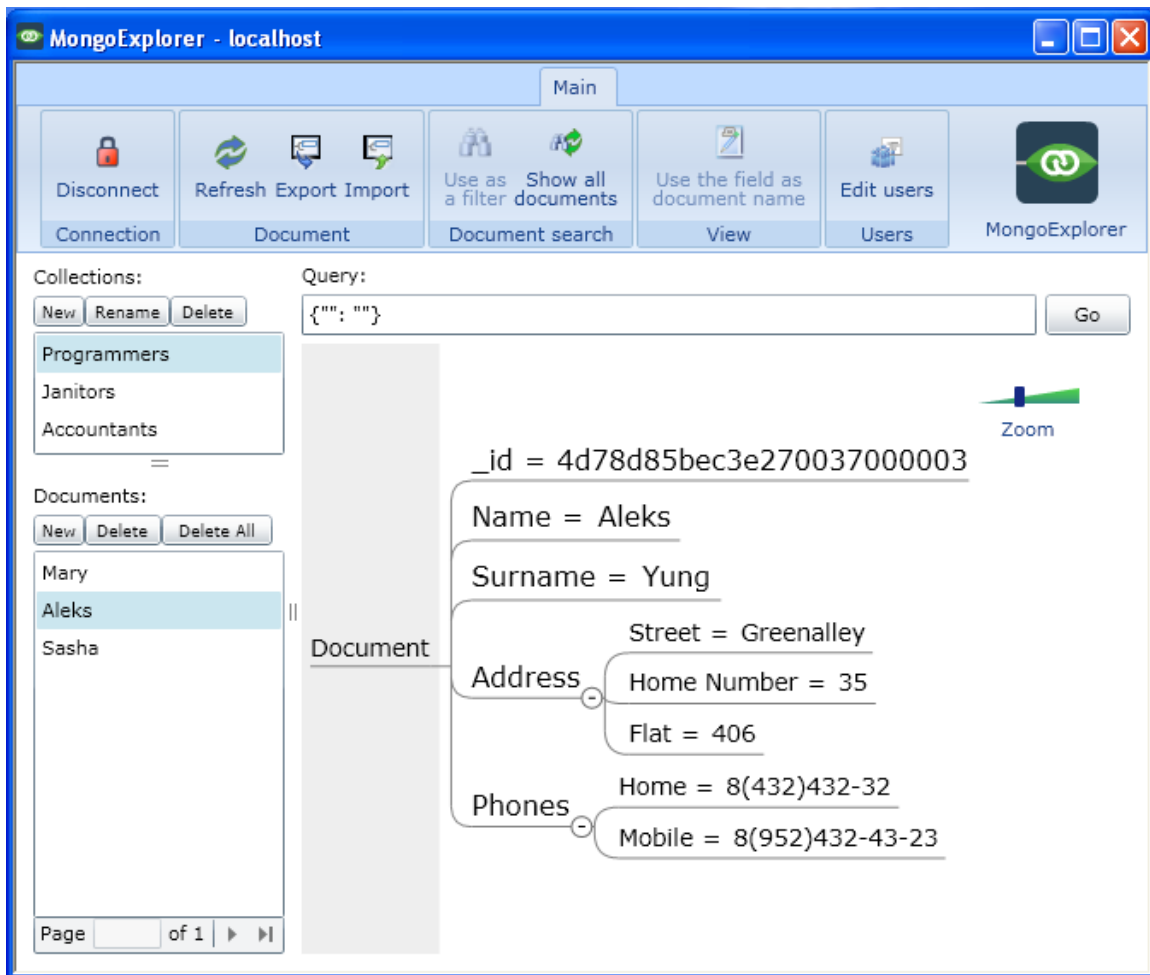


MongoExplorer

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



Commercial Tools

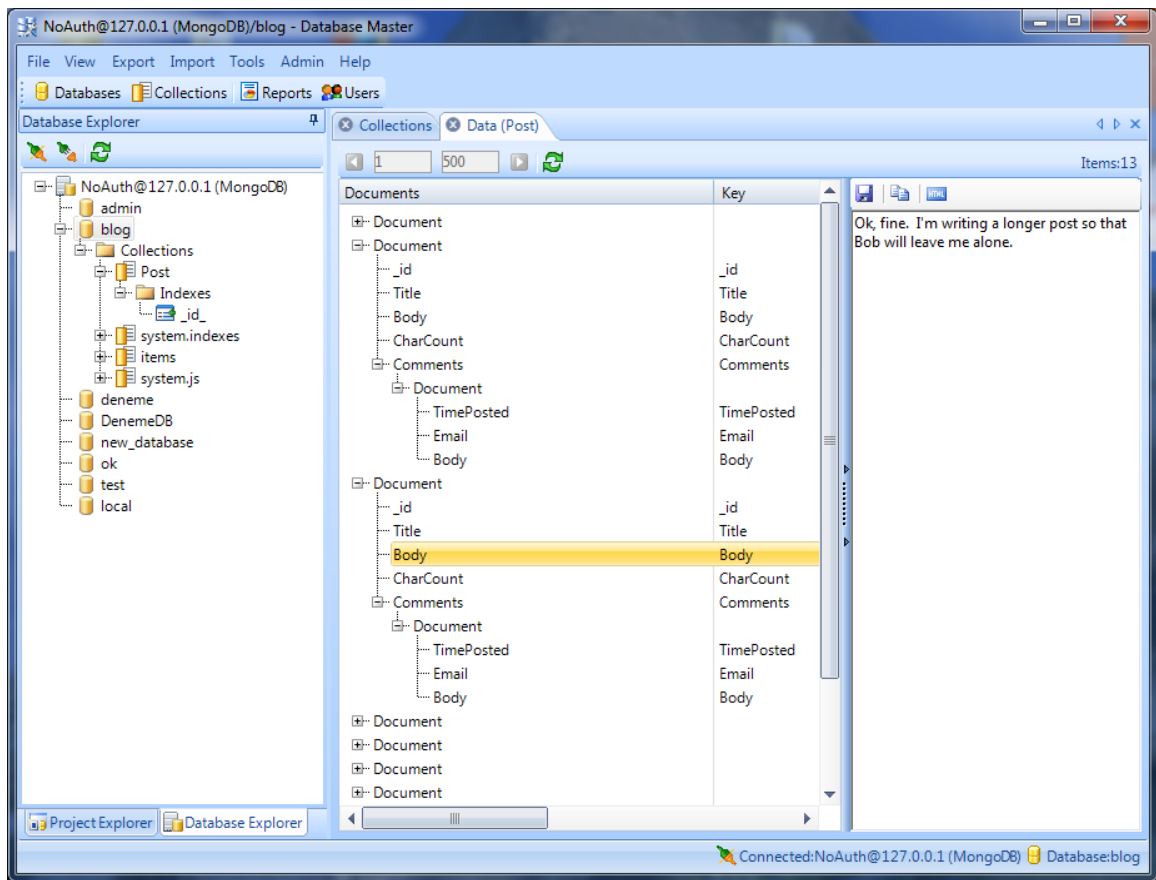
Database Master

[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 RDBMS (MySQL, postgres, ...)



Starting and Stopping Mongo

- Starting Mongo
 - Default Data Directory, Default Port
 - Alternate Data Directory, Default Port
 - Alternate Port
 - Running as a Daemon
- Stopping Mongo
 - Control-C
 - Sending shutdownServer() message from the mongo shell
 - Sending a Unix INT or TERM signal
- Memory Usage

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 Mongo executable is, and the Mongo executable is called mongod.

Starting Mongo

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 Mongo

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 its 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
> 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 }
```

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 as mongod will not be able to cleanly exit. (In such a scenario, run the [repairDatabase](#) command.)

After an unclean shutdown, MongoDB will say it was not shutdown cleanly, and ask you to do a repair. This is absolutely not the same as corruption, this is MongoDB saying it can't 100% verify what's going on, and to be paranoid, run a repair.

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.

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
- `-v` - more verbose output. use more v's (such as `-vvvvvv`) for higher levels of verbosity
- `--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

From the mongo shell

```
> db.runCommand("logRotate");
```

From the unix shell

Note: There is currently no way to rotate logs in Windows from the DOS prompt (or equivalent)

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
```

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.

Basic Options

<code>-h --help</code>	Shows all options
<code>-f --config <file></code>	Specify a configuration file to use
<code>--port <portno></code>	Specifies the port number on which Mongo will listen for client connections. Default is 27017
<code>--dbpath <path></code>	Specifies the directory for datafiles. Default is <code>/data/db</code> or <code>c:\data\db</code>
<code>--fork</code>	Fork the server process
<code>--bind_ip <ip></code>	Specifies a single IP that the database server will listen for
<code>--directoryperdb</code>	Specify use of an alternative directory structure, in which files for each database are kept in a unique directory. (since 1.3.2)
<code>--quiet</code>	Reduces amount of log output
<code>--nohttpinterface</code>	Disable the HTTP interface (localhost:28017)
<code>--rest</code>	Allow extended operations at the Http Interface
<code>--logpath <file></code>	File to write logs to (instead of stdout). You can rotate the logs by sending <code>SIGUSR1</code> to the server.

--logappend	Append to existing log file, instead of overwriting
--repairpath <path>	Root path for temporary files created during database repair. Default is dbpath value.
--cpu	Enables periodic logging of CPU utilization and I/O wait
--noauth	Turns off security. This is currently the default
--auth	Turn on security
-v[v[v[v[v]]]] --verbose	Verbose logging output (-vvvvv is most verbose, -v == --verbose)
--objcheck	Inspect all client data for validity on receipt (useful for developing drivers)
--quota	Enable db quota management. This limits (by default) to 8 files per DB. This can be changed through the --quotaFiles parameter
--quotaFile	Maximum number of files that will be opened per Database. See --quota
--syncdelay arg (=60)	seconds between disk syncs (0=never, but not recommended)
--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
--nohints	Ignore query hints
--noscripting	Turns off server-side scripting. This will result in greatly limited functionality
--notablesan	Turns off table scans. Any query that would do a table scan fails
--noprealloc	Disable data file preallocation
--smallfiles	Use a smaller default file size
--nssize <MB>	Specifies .ns file size for new databases
--sysinfo	Print system info as detected by Mongo and exit
--nunixsocket	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 <= 1.0 to 1.1+)

Master/Slave Replication Options

--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

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
--fastsync	If the node is started with a recent copy of data from another node in the set (including the oplog) it won't need a full resync. You can let it know to skip the resync by running the mongod with --fastsync.

File Based Configuration

In addition to accepting [Command Line Parameters](#), MongoDB can also be configured using a configuration file. A configuration file to use 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.
```



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.

Parameters

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/mongod.log
logappend	Whether the log file will be appended (TRUE) or over-written (FALSE)	logappend=true

Logging

Parameter	Meaning	Example
cpu	Enable periodic logging (TRUE) of CPU utilization and I/O wait	cpu = true
verbose	Verbose logging output	verbose=true

Security

Parameter	Meaning	Example
noauth	Turn authorization on/off. Off is currently the default	noauth = true
auth	Turn authorization on/off. Off is currently the default	auth=false

Administration & Monitoring

Parameter	Meaning	Example
nohttpinterface	Disable the HTTP interface. The default port is 1000 more than the dbport	nohttpinterface = 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
mms-token	Account token for Mongo monitoring server.	mms-token=mytoken
mms-name	Server name for Mongo monitoring server.	mms-name=monitor.example.com
mms-interval	Ping interval for Mongo monitoring server in seconds.	mms-interval=15

quota	Enable quota management	quota = true
quotaFiles	Determines the number of files per Database (default is 8)	quotaFiles=16

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
pairwith	Address of a server to pair with.	pairwith = master.example.com:27017
arbiter	Address of arbiter server	arbiter = aribiter.example.com:27018
autoresync	Automatically resync if slave data is stale	autoresync
opIdMem	Size limit for in-memory storage of op ids in Bytes	opIdMem=1000
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>]
oplogSize	Custom size for replication operation log in MB.	oplogSize=100

Sharding

Parameter	Meaning	Example
shardsvr	Indicates that this mongod will participate in sharding *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

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

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 curen 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()
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
```

Architecture and Components

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.

Database 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 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.

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

Troubleshooting

- [Excessive Disk Space](#)
- [Out of Memory OOM Killer](#)
- [Too Many Open Files](#)
- [mongod process "disappeared"](#)
- [See Also](#)

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
```

See Also

- [Diagnostic Tools](#)

Excessive Disk Space

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 given size. (This is done to prevent file system fragmentation, among other reasons.) The first file for a database is `<dbname>.0`, then `<dbname>.1`, etc. `<dbname>.0` will be 64MB, `<dbname>.1` 128MB, etc., 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 reached.

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; without prefiling in the background this could result in significant delays when a new file must be repopulated.

You can disable preallocation with the `--noprealloc` [command line parameter](#). This flag is nice for tests with small datasets where you drop the db after each test. It shouldn't be used on production servers.

For large databases (hundreds of GB or more) this is of no significant consequence as the unallocated space is small.

Deleted Space

MongoDB maintains deleted lists of space within the datafiles when objects or collections are deleted. This space is reused but never freed to the operating system.

To compact this space, run `db.repairDatabase()` from the mongo shell



this operation will block and is slow.

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.

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: 00000000000001000000
  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)}})}

```

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.

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 `lsdf` (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 nfile X
```

Upstart uses a [different mechanism](#) for setting file descriptor limits - add something like this to your job file:

```
limit nfile 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.

Contributors

- [JS Benchmarking Harness](#)
- [MongoDB kernel code development rules](#)
- [Project Ideas](#)
- [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 } }
]

for ( x = 1; x<=128; x*=2){
  res = benchRun( { parallel : x ,
                    seconds : 5 ,
                    ops : ops
                  } )
  print( "threads: " + x + "\t queries/sec: " + res.query )
}
```


More info:

<http://github.com/mongodb/mongo/commit/3db3cb13dc1c522db8b59745d6c74b0967f1611c>

MongoDB kernel code development rules

Coding conventions for the MongoDB C++ code...

- [Git Commit Rules](#)
- [Kernel class rules](#)
- [Kernel code style](#)
- [Kernel concurrency rules](#)
- [Kernel exception architecture](#)
- [Kernel Logging](#)
- [Kernel string manipulation](#)
- [Memory Management](#)
- [Writing Tests](#)

For anything not mentioned here, default to [google c++ style guide](#)

Git Commit Rules

- 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.

Kernel class rules

Design guidelines

- A class has to have a ***purpose in life*** that's described in a comment in the header of the class. If it takes many lines of heavy comments to state what the class is for, it probably doesn't have a very clear goal. If the comment is just one line but doesn't say anything much clearly than what the name of the class already does, ditto.
- 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. Refrain from obvious, overloaded names (e.g., write, add, shard, clone, ...).
- A ***method's signature must be coherent with its purpose***. If the call is beyond trivial (ie, not an accessor), its side effects in the class private members should be documented. If documenting the call's goal takes many lines of heavy comments, that method likely has very poor coherence and should be refactored.
- The header of a class, or group thereof, is the interface to the class. ***No detail that the caller of the class does not need to know should be in the header***. Conversely, no detail that the caller must know should be omitted.
- Assume that ***any class can throw a DBException***. If a class should never throw (e.g can be called in a destructor), that should be clear. Any other exception should be clearly documented.
- A class, or closely related group thereof, ***is supposed to have a unittest***. That's only one of the reasons why you should design artifacts that are independent from one another. If you can't effectively test your class at that level, consider whether your class is not carrying 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.
- ***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.
- Do not use friend classes. efactor instead.
- Unless necessary make classes ***non-assignable and non-copyable***

Layout guidelines

- Put the public section first, followed by the protected, followed by the private section.
- At each section use the order typedefs, constructors, destructors, methods, members

- 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
- Except when the compiler insists otherwise

Kernel code style

- comments
- case
- inlines
- strings
- brackets
- class members
- functions
- templates
- namespaces
- start of file
- assertions
- return early

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 of 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, what it does, is obvious, you can put a comment on it as to why it exists!

case

Use camelCase for

- most varNames
- --commandLineOptions
- { commandNames : 1, commandOption : "abc" }

In things like serverStatus, include the units in the stat name if there is any chance of ambiguity. For abbreviated units use uppercase. For example:

- writtenMB
- timeMS

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.

strings

See

- utils/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 {  
}
```

class members

```
class Foo {  
    int _bar;  
};
```

functions

```
void foo() {  
}
```

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.

Kernel concurrency rules

All concurrency code must be placed under `utils/concurrency`. You will find several helper libraries there.

Several rules are listed below. Don't break them. If you think there is a real need let's have the group weigh in and get a consensus 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:

- `assert` should be used for internal assertions. However, `massert` is preferred.
- `massert` is an internal assertion with a message.
- `uassert` is used for a user error
- `wassert` warn (log) and continue

Both `massert` and `uassert` take error codes, so that all errors have codes associated with them. These [error codes](#) are assigned randomly, so there aren't segments that have meaning. `scons` checks for duplicates, but if you want the next available code you can run:

```
python buildscripts/errorcodes.py
```

A failed assertion throws an `AssertionException` or a child of that. The inheritance hierarchy is something like:

- `std::exception`
 - `mongo::DBException`
 - `mongo::AssertionException`
 - `mongo::UserAssertionException`
 - `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.

Kernel Logging

- Basic Rules
 - `cout/cerr` should never be used
- Normal Logging
 - debugging with levels of verbosity

```
• log( int x )
```

- informational

```
• log( )
```

- 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.

`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

`str.h`

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.

StringData

See also bson/stringdata.h.

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
- 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.

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 ran by the buildbots only once a night; slowWeekly only once a week. Most other tests are ran every CI cycle (all the time).

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>

GridFS Web Server Modules

There are a couple of modules for different web servers designed to allow serving content directly from GridFS:

Nginx: <http://github.com/mdirolf/nginx-gridfs>
Lighttpd: <http://bitbucket.org/bwmcadams/lighttpd-gridfs>

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 persistence layer!

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
- curoop / 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](#)

(Additionally, community drivers and tools also exist and will be found in other places.)

See Also

- [Building](#)
- [License](#)

Building

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.

Note: see the [Downloads](#) page for prebuilt binaries!

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](#)
- [scons](#)

See Also

- The main [Database Internals](#) page
- [Building with V8](#)

Building Boost

- [Windows](#)

MongoDB uses the [\[www.boost.org\]](http://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\
```

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: There are binaries for many platforms. Most users won't need to compile mongo themselves. See the [Downloads](#) page for these prebuilt binaries!

- [SpiderMonkey, UTF-8, and/or Ubuntu](#)
- [Prerequisites](#)
 - [Fedora 8 or 10](#)
 - [Ubuntu](#)
 - [Ubuntu 8.04](#)
 - [Ubuntu 9.04 and 9.10](#)
 - [Ubuntu 10.04+](#)
- [Building](#)

SpiderMonkey, UTF-8, and/or Ubuntu

Most pre-built Javascript SpiderMonkey binaries do not have UTF-8 compiled in. Additionally, Ubuntu has a weird version of SpiderMonkey that doesn't support everything we use. If you get any warnings during compile time or runtime, we highly recommend building SpiderMonkey from source. See [Building SpiderMonkey](#) for more information.

We currently support SpiderMonkey 1.6 and 1.7, although there is some degradation with 1.6, so we recommend using 1.7. We have not yet tested 1.8, but will once it is officially released.

Prerequisites

Fedora 8 or 10

```
sudo yum -y install git tcsh scons gcc-c++ glibc-devel
sudo yum -y install boost-devel pcre-devel js-devel readline-devel
#for release builds:
sudo yum -y install boost-devel-static readline-static ncurses-static
```

Ubuntu

Note: See SpiderMonkey note above.

Use `cat /etc/lsb-release` to see your version.

Ubuntu 8.04

```
apt-get -y install tcsh git-core scons g++
apt-get -y install libpcre++-dev libboost-dev libreadline-dev xulrunner-1.9-dev
apt-get -y install libboost-program-options-dev libboost-thread-dev libboost-filesystem-dev
libboost-date-time-dev
```

Ubuntu 9.04 and 9.10

```
apt-get -y install tcsh git-core scons g++
apt-get -y install libpcre++-dev libboost-dev libreadline-dev xulrunner-1.9.1-dev
apt-get -y install libboost-program-options-dev libboost-thread-dev libboost-filesystem-dev
libboost-date-time-dev
```

Ubuntu 10.04+

```
apt-get -y install tcsh git-core scons g++
apt-get -y install libpcr++-dev libboost-dev libreadline-dev xulrunner-dev
apt-get -y install libboost-program-options-dev libboost-thread-dev libboost-filesystem-dev
libboost-date-time-dev
```

Building

1. Install Dependencies - see platform specific below
2. get source

```
git clone git://github.com/mongodb/mongo.git
# 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". (Although with
# sharding you will want the latest if the latest is less
# than 1.6.0.) For example:
git checkout r1.4.1
```

3. build

```
scons all
```

4. install

```
scons --prefix=/opt/mongo install
```

Building for OS X

- [Upgrading to Snow Leopard](#)
- [Setup](#)
 - [Sources](#)
 - [Prerequisites](#)
 - [Package Manager Setup](#)
 - [Manual Setup](#)
 - [Install Apple developer tools](#)
 - [Install libraries \(32-bit option\)](#)
 - [Install libraries \(64-bit option\)](#)
- [Compiling](#)
- [Troubleshooting](#)

To set up your OS X computer for MongoDB development:

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](https://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. The easiest way to upgrade gcc is to install the iPhone SDK.

Package Manager Setup

1. Install [MacPorts](#) (snow leopard version). If you have MacPorts installed previously, we've had the most success by running

```
sudo rm -rf /opt/local
```

first.

2. Update/install packages:

```
sudo port install boost pcre++ scon
```

3. Update/install SpiderMonkey with

```
sudo port install spidermonkey
```

. (If this fails, see the note on #2 above.)

Manual Setup

Install Apple developer tools

Install libraries (32-bit option)

1. Download boost `{{boost 1.37.0 http://downloads.sourceforge.net/boost/boost\_1\_37\_0.tar.gz}}` 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)$(<:S)" -L
+   "$(CONFIG_COMMAND)" -dynamiclib -Wl,-single_module -install_name
+   "/usr/local/lib/$(<:B)$(<:S)" -L"$LINKPATH" -o "$(<)" "$(>)" "$(LIBRARIES)" -l$(FINDLIBS-SA)
-   "$(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 <http://www.pcre.org/> (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 {{boost 1.37.0 http://downloads.sourceforge.net/boost/boost_1_37_0.tar.gz}}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 <http://www.pcre.org/> (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

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 a .vcxproj/.sln is 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](#)
 - [Prebuilt from boostpro.com](#)
 - [Building Yourself](#)
- [Additional Notes](#)

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
```

Visual Studio 2010 Project File

A VS2010 vcxproj file is available for building the shell. From the mongo directory open shell/msvc/mongo.vcxproj.

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

The shell uses the [GNU readline library](#) to facilitate command line editing and history. 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

- [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

- [Get the MongoDB Source Code](#)
- [Get Boost Libraries](#)
- [Get SpiderMonkey](#)
- [Building MongoDB from the IDE](#)
- [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, 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.)

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/2.6.4/>.
 - **Note** Make sure to install the 32 bit version of python and not the 64 bit as the scons binaries below are 32 bit.
- 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 located at c:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin (or \amd64) for 32 or 64 bit.

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
scons --64           // build the 64 bit version
scons --dd            // build with debug symbols
```

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.
- 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

- [Building js.lib - Unix](#)
 - [Remove any existing xulrunner](#)
 - [Download](#)
 - [Build](#)
 - [Install](#)
- [Building js.lib - Windows](#)
 - [Prebuilt](#)
 - [Download](#)
 - [Build](#)
 - [Troubleshooting scons](#)
- [See Also](#)

MongoDB uses [SpiderMonkey](#) for server-side Javascript execution. The mongod project requires a file js.lib when linking. This page details how to build js.lib.

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 scones 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](#)

`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
- `--dd` debug build with `_DEBUG` defined (extra asserts etc.)
- `--release`
- `--32` force 32 bit
- `--64` force 64 bit
- `--clean`

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](#)

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 `--dur` 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 --dur --durOptions 1

# recover and terminate
mongod --dur --durOptions 4

# dump journal entries (doing nothing else) and then terminate
mongod --dur --durOptions 7

# extra checks that everything is correct (slow but good for qa)
mongod --dur --durOptions 8
```

Diagrams

- [diagram 1 - process steps](#)
- [diagram 2 - journal file structure](#)

Parsing Stack Traces

addr2line

```
addr2line -e mongod -ifC <offset>
```

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
```

Example

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.

Cursors



Redirection Notice

This page should redirect to [Internals](#).

Error Codes

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	
13440	bad offset accessing a datafile	Run a database --repair

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

- [smoke.py](#)
- [Running a jstest manually](#)
- [Running the C++ unit tests](#)
- [See Also](#)

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
```

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 its 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 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 anthropomorphizing 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	<code>{{ term }}</code>
short code fragment inline	monospace	<code>{{ term }}</code>
file path/name, extension	italic	<i>_term_</i>
programming command, statement or expression	monospace	<code>{{ term }}</code>
variable or "replaceable item"	monospace italic	<i>_term_</i>
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	<i>_term_</i>
tag (in HTML or XML, for example)	monospace	<code>{{ term }}</code>
Extended code sample	code block	<pre>{code} program code {code}</pre>

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



General Community Resources

- **User Forum**

The [user list](#) is for general questions about using, configuring, and running MongoDB and the associated tools and drivers. The list is open to everyone

- **IRC chat**

<irc://irc.freenode.net/#mongodb>

- **Blog**

<http://blog.mongodb.org/>

- **Bugtracker**

File, track, and vote on bugs and feature requests. There is [issue tracking](#) for MongoDB and all supported drivers

- **Announcement Mailing List**

<http://groups.google.com/group/mongodb-announce> - for release announcement and important bug fixes.

- **Events**

Our [events page](#) includes information about MongoDB conferences, webcasts, users groups, local meetups, and open office hours.

- **Store**

Visit our [Cafepress](#) store for Mongo-related swag.

Resources for Driver and Database Developers

- **Developer List**

This [mongodb-dev mailing list](#) is for people developing drivers and tools, or who are contributing to the MongoDB codebase itself.

- **Source**

The source code for the database and drivers is available at the <http://github.com/mongodb>.

- **Project Ideas**

Start or contribute to a MongoDB-related [project](#).

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!

MongoDB Commercial Services Providers

Note: if you provide consultative or support services for MongoDB and wish to be listed here, just let us know.

Production Support

Company	Contact Information

	10gen began the MongoDB project, and offers commercial MongoDB support services. If you are having a production issue and need support immediately, 10gen can usually on-board new clients within an hour. Please contact us or call (866) 237-8815 for more information.

Training

Company	Contact Information
	10gen offers MongoDB training for Developers and Administrators. The 2011 training schedule includes sessions in New York, NY and Redwood Shores, CA.

Hosting

See the [MongoDB Hosting Center](#).

Consulting

Company	Contact Information
	10gen offers consulting services for MongoDB application design, development, and production operation. These services are typically advisory in nature with the goal of building higher in-house expertise on MongoDB for the client. • Lightning Consults - If you need only an hour or two of help, 10gen offers mini consultations for MongoDB projects during normal business hours • MongoDB Health Check - An experienced 10gen / MongoDB expert will work with your IT Team to assess the overall status of your MongoDB deployment • Custom Packages - Contact for quotes on custom packages
	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'.
	MijiX, a software development studio based on Indonesia, provides consulting for MongoDB in Asia-Pacific area.
	Squeejee builds web applications on top of MongoDB with multiple sites already in production.

	TuoJie, based in Beijing, China, provides high quality total solutions and software products for banks, other financial firms, the communications industry, and other multinational corporations. Clients include China Mobile, China Unicom, Microsoft, and Siemens. TuoJie's business scope covers consulting, software development, and technical services.
	ZOPYX Ltd is a German-based consulting and development company in the field of Python, Zope & Plone. Lately we added MongoDB to our consulting and development portofolio. As one of the first projects we were involved in the launch of the BRAINREPUBLIC social platform.

Visit the 10gen Offices

Bay Area

10gen's West Coast office is located on the Peninsula in Redwood Shores.

100 Marine Parkway, Suite 175
Redwood City, CA 94065

[View Larger Map](#) **Directions:**

- **Take 101 to the Ralston Ave Exit.**
 - From the North, turn Left onto Ralston Ave.
 - From the South, turn Right onto Ralston Ave.
- **Continue onto Marine Parkway.**
- **Turn Right onto Lagoon Drive.**
- **Make a Right turn at the "B" marker (see map). Our building, 100 Marine Pkwy, will be on your right.**
- **Park and enter on the North side of the building. Our Suite, 175, will be the first set of double-doors on your right.**
- **Come on in!**

New York City

10gen's East Coast office is located in the Flatiron District of NYC.

134 5th Avenue
3rd Floor
New York, NY 10011

[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

"Maybe we can relax with couchdb but with mongodb we are completely in dreams"
-namlook, #mongodb

"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.

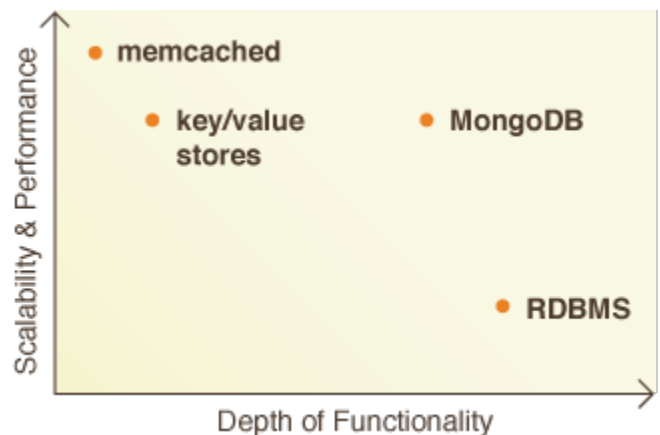
About

- [Philosophy](#)
- [Use Cases](#)
- [Production Deployments](#)
- [Mongo-Based Applications](#)
- [Events](#)
- [Slide Gallery](#)
- [Articles](#)
- [Benchmarks](#)
- [FAQ](#)
- [Product Comparisons](#)
- [Licensing](#)

Philosophy

Design Philosophy

- Databases are specializing - the "one size fits all" approach no longer applies.
- By reducing transactional semantics the db provides, one can still solve an interesting set of problems where performance is very important, and horizontal scaling then becomes easier.
- 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.
- A non-relational approach is the best path to database solutions which scale horizontally to many machines.
- While there is an opportunity to relax certain capabilities for better performance, there is also a need for deeper functionality than that provided by pure key/value stores.
- 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.



Use Cases

See also the [Production Deployments](#) page for a discussion of how companies like Shutterfly, foursquare, bit.ly, Etsy, SourceForge, etc. use MongoDB.

Well Suited

- 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
- Caching. With its potential for high performance, MongoDB works well as a caching tier in an information infrastructure. The persistent backing of Mongo's cache assures that on a system restart the downstream data tier is not overwhelmed with cache population activity.
- 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.
- Storage of program objects and JSON data (and equivalent). 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 require blocking "ALTER TABLE" style operations.
- Document and Content Management Systems - as a document-oriented (JSON) database, MongoDB's flexible schemas are a good fit for this.
- Electronic record keeping - similar to document management.
- [Real-time stats/analytics](#)
- [Archiving and event logging](#)

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 Business Intelligence. Data warehouses are more suited to new, problem-specific BI databases. However note that MongoDB can work very well for several reporting and analytics problems where data is predistilled or aggregated in [runtime](#) -- but classic, nightly batch load business intelligence, while possible, is not necessarily a sweet spot.
- Problems requiring SQL.

Use Case Articles

[Using MongoDB for Real-time Analytics](#)
[Using MongoDB for Logging](#)
[Full Text Search in Mongo](#)
[MongoDB and E-Commerce](#)
[Archiving](#)

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 \(coming soon\)](#)

See other use case guides:

MongoDB for ecommerce and SaaS application
MongoDB for mobile and social applications

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.)

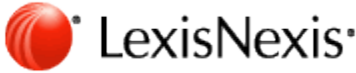
```
> db.createCollection('sessions', { capped: true, size : 1000000 } )
{"ok" : 1}
> p = "";
> for( x = 0; x < 100; x++ ) p += 'x';
> s1 = { info: 'example', _padding : p };
{"info": "example" , "_padding" :
"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"}
}
> db.sessions.save(s1)
> s1
{"info": "example" , "_padding" :
"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
, "_id" : ObjectId( "4aa7b74a5761d147677233b0") }

> // when updating later
> s1 = db.sessions.find( { _id : ObjectId( "4aa7b74a5761d147677233b0") } )
{"_id" : ObjectId( "4aa7b74a5761d147677233b0") , "info" : "example" , "_padding" :
"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"}
}
> delete s._padding;
true
> s.x = 3; // add a new field
3
> db.sessions.save(s);
> s
{"_id" : ObjectId( "4aa7b5a25761d147677233af") , "info" : "example" , "x" : 3}
```

Production Deployments

If you're using MongoDB in production, we'd love to list you here! Please complete this [web form](#) or email meghan@10gen.com and we will add you.

Company	Use Case
	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. • 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
	Foursquare is a location based social network that incorporates gaming elements. • Scaling foursquare with MongoDB - New York MongoDB User Group (December 2010) • MongoDB at foursquare presentation: Slides and Video (May 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)

	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)
	LexisNexis Risk Solutions serves the multi-billion dollar risk information industry, which is comprised of professionals and organizations such as law enforcement, government agencies, financial services firms, collection agencies, insurance and health care providers, and hiring managers. MongoDB is used by the DevOps group to store their CMDB information, static web content, auditing data, and serves as the persistence engine for their asynchronous messaging fabric.
	MongoDB is used for back-end storage on the SourceForge front pages, project pages, and download pages for all projects. • Scaling SourceForge with MongoDB - OSCON Presentation (July 2010) • MongoDB at SourceForge - QCon London Presentation (March 2010) • How Python, TurboGears, and MongoDB are Transforming SourceForge.net - PyCon (February 2010) • SourceForge.net releases Ming - SourceForge blog (December 2009) • TurboGears on Sourceforge - Compound Thinking (July 2009)
	Etsy is a website that allows users to buy and sell handmade items. Read the MongoDB at Etsy blog series: • Part 1 - May 19, 2010 • Part 2 - July 3, 2010
	The New York Times is using MongoDB in a form-building application for photo submissions. Mongo's lack of schema gives producers the ability to define any combination of custom form fields. For more information: • 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)
	CERN uses MongoDB for Large Hadron Collider data. • Holy Large Hadron Collider, Batman! - MongoDB Blog (June 2010)
	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



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:
 - [Humongous Data at Server Density](#) - MongoUK Presentation (June 2010)
 - [MongoDB in Production at Boxed Ice](#) - Webinar (May 2010)



Wordnik stores its entire text corpus in MongoDB - 1.2TB of data in over 5 billion records. The speed to query the corpus was cut to 1/4 the time it took prior to migrating to MongoDB. More about MongoDB at Wordnik:

- [Managing a MongoDB Deployment](#) - Presentation at Large Scale Production Engineering Meetup (February 2011)
- [Wordnik: 10 million API requests a day on MongoDB](#) - High Scalability Blog (February 2011)
- [12 Months with MongoDB](#) - Wordnik Blog (October 2010)
- [B is for Billion](#) - Wordnik Blog (July 2010)
- [MongoDB: Migration from Mysql at Wordnik](#) - Scalable Web Architectures (May 2010)
- [Tony Tam's Presentation at MongoSF](#) (April 2010)
- [What has technology done for words lately?](#) - Wordnik blog (February 2010)



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



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:

- [How This Web Site Uses MongoDB](#) (November 2009 Article)
- [How Business Insider Uses MongoDB](#) (May 2010 Presentation)



GitHub, the social coding site, is using MongoDB for an internal reporting application.



Gilt Groupe is an invitation only luxury shopping site. Gilt uses MongoDB for real time ecommerce analytics.

- Gilt CTO Mike Bryzek's presentation at [MongoSF](#) in April 2010.
- [Hummingbird](#) - a real-time web traffic visualization tool developed by Gilt and powered by MongoDB

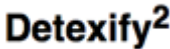
	IGN Entertainment, a unit of News Corporation, is a leading Internet media and services provider focused on the videogame and entertainment enthusiast markets. IGN's properties reached more than 37.3 million unique users worldwide February 2010, according to Internet audience measurement firm comScore 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)
	catch.com is the easiest way to capture what matters. Catch helps you create one place to privately capture & find your notable thoughts, ideas, images, places, lists & anything else you'd rather not forget. • MongoDB and Pylons at Catch.com - PyCon Presentation (March 2011)
	Recently named #8 free app of all-time by Apple, Bump is a quick and easy way to connect two phones, simply by bumping them together! MongoDB is used in Bump's database and queue experience and helps manage and store data from the 25+ million users
	OpenSky is a free online platform that helps people discover amazing products; share them with their friends, family and followers; and earn money. OpenSky uses MongoDB, Symfony 2, Doctrine 2, PHP 5.3, PHPUnit 3.5, jQuery, node.js, Git (with gitflow) and a touch of Java and Python. OpenSky uses MongoDB for just about everything (not just analytics). Along the way they've developed MongoODM (PHP) and MongoDB drivers for Mule and CAS. • MongoDB & Ecommerce: A Perfect Combination - Video from Steve Francia's presentation to the New York MongoDB User Group (October 2010)
	CollegeHumor is a comedy website. MongoDB is used in CollegeHumor for internal analytics and link exchange application.
	Evite uses MongoDB for analytics and quick reporting. • Tracking and visualizing mail logs with MongoDB and gviz_api - Grig Gheorghiu's blog (July 2010)
	Disqus is an innovative blog-commenting system.
	MongoHQ provides a hosting platform for MongoDB and also uses MongoDB as the back-end for its service. Our hosting centers page provides more information about MongoHQ and other MongoDB hosting options.
	Justin.tv is the easy, fun, and fast way to share live video online. MongoDB powers Justin.tv's internal analytics tools for virality, user retention, and general usage stats that out-of-the-box solutions can't provide. Read more about Justin.tv's broadcasting architecture.

	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. • The Secret Weapons Behind Chartbeat - Kushal's coding blog (April 2010) • Kushal Dave's Presentation at MongoNYC (May 2010)
	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, 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)
	Realtime.springer.com is a service that aggregates together downloads of Springer journal articles and book chapters in real time and displays them in a variety of visualizations. The goal of this service is to provide the scientific community with valuable information about how the literature is being used "right now". MongoDB is used to store the details of one million downloads per day from across Springer's sites. Map reduce jobs generate collections for the last 7, 30, and 90 days for around 2,000 journal and 40,000 book titles.
	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)
	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.
	Sugar CRM uses MongoDB to power the backend of its preview feedback mechanism. It captures users' comments and whether they like or dislike portions of the application all from within beta versions of Sugar.

	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)
	WHERE® is a local search and recommendation service that helps people discover places, events and mobile coupons in their area. Using WHERE, people can find everything from the weather, news, and restaurant reviews, to the closest coffee shop, cheapest gas, traffic updates, movie showtimes and offers from local merchants. WHERE is available as a mobile application and as a web service at Where.com. here, Inc. uses MongoDB to store geographic content for the WHERE application and for WHERE Ads™ - a hyper-local ad network.
	Founded by Alibaba Group in May 2003, Taobao facilitates business-to-consumer and consumer-to-consumer retail by providing a platform for businesses and individual entrepreneurs to open online retail stores that cater to consumers in mainland China, Hong Kong, Macau, Taiwan and overseas. Sellers are able to post new and used goods for sale on the Taobao marketplace either through a fixed price or by auction. The overwhelming majority of the products on Taobao are brand new merchandise sold at a fixed price; auctions make up a very small percentage of transactions. MongoDB is used to store the monitoring data.
	PhoneTag is a service that automatically transcribes voicemail to text and delivers it in real-time via e-mail and SMS. PhoneTag stores the metadata and transcriptions for every voicemail it process in MongoDB.
	Harmony is a powerful web-based platform for creating and managing websites. It helps developers with content editors work together with unprecedented flexibility and simplicity. From stylesheets, images and templates, to pages, blogs, and comments, every piece of Harmony data is stored in MongoDB. Switching to MongoDB from MySQL drastically simplified Harmony's data model and increased the speed at which we can deliver features. • Steve Smith's presentation about Harmony at MongoSF (April 2010)
	Hashrocket is an expert web design and development group. Hashrocket built PharmMD, a fully-featured Medication Management application in Ruby on Rails. The system contains functionality for identifying and resolving drug-related problems for millions of patients.
	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.
	The Mozilla open-source Ubiquity Herd project uses MongoDB for back-end storage. Source code is available on bitbucket.
	Sedue is the enterprise search/recommendation engine, provided by Preferred Infrastructure, Inc. Sedue currently uses MongoDB, to store the document in a flexible and highly-available way. • MongoDB as search engine repository - Presentation from Mongo Tokyo (March 2011)
	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.
	Codaset is an open system where you can browse and search through open source projects, and check out what your friends are coding. • The awesomeness that is MongoDB and NoSQL, is taking over Codaset - Codaset Blog (May 2010) • Handling Dates in MongoDB - Codaset Blog (July 2010)
	Shopwiki uses Mongo as a data store for its shopping search engine, where they commit all the data generated, such as custom analytics. Mongo's performance is such that ShopWiki uses it in cases where MySQL would just not be practical. ShopWiki is also using it as a storage engine for all R&D and data-mining efforts where MongoDB's document oriented architecture offers maximum flexibility. • Avery's Talk at MongoNYC - ShopWiki Dev Blog (June 2010)
	Punchbowl.com is a start to finish party planning site that uses MongoDB for tracking user behavior and datamining. • Introducing MongoDB into your organization: Punchbowl Case Study - Ryan Angilly's presentation at Mongo Boston (September 2010) • Ryan Angilly on Replacing MySQL with MongoDB (Zero to Mongo) on The Bitsource • MongoDB for Dummies: How MyPunchbowl went from 0 to production in under 3 days - Presentation at MongoNYC (May 2010)
	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. • Civic Hacking - Video from Luigi Montanez's presentation at MongoNYC (May 2010) • How We Use MongoDB at Sunlight blog post (May 2010)
	photostre.am streams image data from flickr and uses MongoDB as it's only database. • MongoDB in Production at photostre.am - photostre.am blog (June 2010)

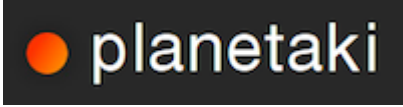
 fotopedia.com The first collaborative photo encyclopedia	Fotopedia uses MongoDB as storage backend for its copy of wikipedia data, storage for users and albums timelines, a feature that is currently under heavy refactoring, and as the "metacache", an index of every tiny html fragment in its varnish cache for proactive invalidation of stale content. • MongoDB: Our Swiss Army Datastore - Presentation at MongoFR in June 2010: Slides and Video
	Grooveshark currently uses Mongo to manage over one million unique user sessions per day.
	Stickybits is a fun and social way to attach digital content to real world objects.
	MongoDB is being used for the game feeds component. It caches game data from different sources which gets served to ea.com, rupture.com and the EA download manager.
	Struq develops technology that personalises the contents and design of online display advertising in real time.
	Pitchfork is using MongoDB for their year-end readers survey and internal analytics.
	Floxee, a web toolkit for creating Twitter directories, leverages MongoDB for back-end storage. The award-winning TweetCongress is powered by Floxee.
	Sailthru is an innovative email service provider that focuses on improving the quality of emails over quantity. Moving to MongoDB from MySQL allowed us extreme flexibility in providing an API to our clients. Passing in arbitrary JSON data is easy – our customers can use objects and arrays inside of their emails. And we've launched Sailthru Alerts, which allows our customers basically whitelabeled Google Alerts: realtime and summary alerts (price, tag match, etc) that are completely up to the customer due to our schema-free data storage. Also, we can gather realtime behavioral data on a client's signed-up users (tag-based interests, geolocale, time of day, etc), and use all that information to power dynamically assembled mass emails that get higher click-through rates than static emails. Plus we just launched an onsite recommendation widget (check it out at refinery29.com), and we're using MongoDB's analytic capabilities to rapidly A/B test different behavioral algorithms. • MongoDB in Production at Sailthru - Presentation to NY MongoDB User Group (January 2011)
	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

	TeachStreet helps people find local and online classes by empowering teachers with robust tools to manage their teaching businesses. MongoDB powers our real-time analytics system which provide teachers with insight into the performance and effectiveness of their listings on TeachStreet. • Slides from Mongo Seattle - TeachStreet blog (July 2010)
	Visibiz is a socially and email infused relationship management solution designed to improve the productivity of business professionals. Within minutes of joining, Visibiz automatically creates fully tagged and organized profiles for everything in the user's network and allows intelligent searching of the information needed to stay productive. Professionals can also track all their business development activities through simplified and streamlined CRM process that is supported by configurable workflows. Visibiz uses MongoDB for all of its data storage needs. • How MongoDB Helps Visibiz Tackle Social CRM - Webinar March 2011
	Defensio is a comment-spam blocker that uses MongoDB for back-end storage.
	TweetSaver is a web service for backing up, searching, and tagging your tweets. TweetSaver uses MongoDB for back-end storage.
	Bloom Digital's AdGear platform is a next-generation ad platform. MongoDB is used for back-end reporting storage for AdGear.
	KLATU Networks designs, develops and markets asset monitoring solutions which helps companies manage risk, reduce operating costs and streamline operations through proactive management of the status, condition, and location of cold storage assets and other mission critical equipment. KLATU uses MongoDB to store temperature, location, and other measurement data for large wireless sensor networks. KLATU chose MongoDB over competitors for scalability and query capabilities.
	songkick lets you track your favorite artists so you never miss a gig again. • Speeding up your Rails app with MongoDB - Presentation at MongoUK (June 2010)
	http://sluggy.com/ is built on MongoDB, mongodb_beaker, and MongoKit. • From MySQL to MongoDB at Sluggy.com - Brendan McAdams' presentation at MongoNYC (May 2010)
	EDITD is using MongoDB to store opinions from social media, blogs, forums and other sources to use in their sentiment analysis system, Zeitgeist.
	@trackmeet helps you take notes with twitter, and is built on MongoDB
	eFlyover leverages the Google Earth Browser Plugin and MongoDB to provide interactive flyover tours of over two thousand golf courses worldwide.

	Shapado is a multi-topic question and answer site in the style of Stack Overflow. Shapado is written in Rails and uses MongoDB for back-end storage.
	Sifino enables students to help each other with their studies. Students can share notes, course summaries, and old exams, and can also ask and respond to questions about particular courses.
	GameChanger provides mobile apps that replace pencil-and-paper scorekeeping and online tools that distribute real-time game updates for amateur sports. • Tornado, MongoDB, and the Realtime Web - Kiril Savino's presentation at MongoNYC (May 2010) • GameChanger and MongoDB: a case study in MySQL conversion - Kiril Savino's blog (September 2010)
	solimap is a map-based ad listings site that uses MongoDB for storage.
	MyBankTracker iPhone App uses MongoDB for the iPhone app's back-end server.
	BillMonitor uses MongoDB to store all user data, including large amounts of billing information. This is used by the live site and also by BillMonitor's internal data analysis tools.
	Tubricator allows you to create easy to remember links to YouTube videos. It's built on MongoDB and Django.
	Mu.ly uses MongoDB for user registration and as a backend server for its iPhone Push notification service. MongoDB is mu.ly's Main backend database and absolute mission critical for mu.ly.
	Avinu is a Content Management System (CMS) built on the Vork enterprise framework and powered by MongoDB.
	edelight is a social shopping portal for product recommendations. • MongoDB: Wieso Edelight statt MySQL auf MongoDB setzt - Exciting Ecommerce blog (September 2010)
	Topsy is a search engine powered by Tweets that uses Mongo for realtime log processing and analysis.
	Cold Hard Code makes featureful, beautiful web applications, including: • Codepeek is using MongoDB and GridFS for storing pastes. • Jarvis uses MongoDB to store asset information.
	Similaria.pl is an online platform, created to connect users with people and products that match them. • One Year with MongoDB - Presentation from PyCon Ukraine (October 2010)

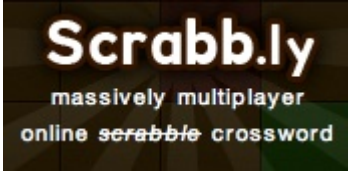
	ToTuTam uses Mongo to store information about events in its portal and also to store and organise information about users preferences. • One Year with MongoDB - Presentation from PyCon Ukraine (October 2010)
	themoviedb.org is a free, user driven movie database that uses MongoDB as its primary database.
	OCW Search is a search engine for OpenCourseWare. It stores all the course materials in MongoDB and uses Sphinx to index these courses. • Full Text Search with Sphinx - Presentation from MongoUK (June 2010)
	Mixero is the new generation Twitter client for people who value their time and are tired of information noise. Mixero uses Mongo to store users' preferences and data.
	Biggo is an advanced site constructor with e-commerce modules. Biggo uses MongoDB for stats collection.
	Kabisa is a web development firm specializing in Ruby on Rails and Java / J2EE. Kabisa uses MongoDB for many of its client projects, including a mobile news application for iPhone and Android.
	DokDok makes it easy and automatic for users to find, work on and share the latest version of any document - right from their inbox. DokDok migrated to a Mongo backend in August 2009. See Bruno Morency's presentation Migrating to MongoDB for more information.
	Enbil is a swedish website for finding, and comparing, rental cars. MongoDB is used for storing and querying data about geographical locations and car rental stations.
	Websko is a content management system designed for individual Web developers and cooperative teams. MongoDB's lack of schema gives unlimited possibilities for defining manageable document oriented architecture and is used for back-end storage for all manageable structure and data content. Websko is written in Rails, uses MongoMapper gem and in-house crafted libraries for dealing with Mongo internals.
	markitfor.me is a bookmarking service that makes your bookmarks available via full-text search so you don't have to remember tags or folders. You can just search for what you're looking for and the complete text of all of your bookmarked pages will be searched. MongoDB is used as the datastore for the marked pages.
	Backpage Pics is a website that displays backpage.com adult classified listings as an image gallery. MongoDB is used to store listing data. <i>Please note that this website is NSFW.</i>

	Joomla Ads uses MongoDB for its back-end reporting services.
	musweet keeps track of what artists and bands publish on the social web.
	Eiwa System Management, Inc. is a software development firm that has been using MongoDB for various projects since January 2010.
	Morango is an internet strategy consultancy based in London, which uses MongoDB in production on several client projects. Building a Content Management System with MongoDB - Presentation from MongoUK (June 2010)
	PeerPong discovers everyone's expertise and connects you to the best person to answer any question. We index users across the entire web, looking at public profiles, real-time streams, and other publicly available information to discover expertise and to find the best person to answer any question.
	ibibo ("I build, I bond") is a social network using MongoDB for its dashboard feeds. Each feed is represented as a single document containing an average of 1000 entries; the site currently stores over two million of these documents in MongoDB.
	MediaMath is the leader in the new and rapidly growing world of digital media trading.
	Zoofs is a new way to discover YouTube videos that people are talking about on Twitter. Zoofs camps in Twitter searching for tweets with YouTube video links, and then ranks them based on popularity.
	Oodle is an online classifieds marketplace that serves up more than 15 million visits a month and is the company behind the popular Facebook Marketplace. Oodle is using Mongo for storing user profile data for our millions of users and has also open sourced its Mongo ORM layer.
	Funadvice relaunched using the MongoDB and MongoMapper. Read the Funadvice CTO's post to MongoDB User Forum from May 2010 for more details.
	Ya Sabe is using MongoDB for the backend storage of business listings. Yasabe.com is the first local search engine built for Hispanics in the US with advanced search functionality. You can find and discover more than 14 million businesses via the web or your mobile phone. All the information is in both Spanish and in English.
	LoteriaFutbol.com is a Fantasy Soccer Portal recently launched for the World Soccer Cup: South Africa 2010. Mongo has been used entirely to store data about users, groups, news, tournaments and picks. It uses the PHP driver with a Mongo module for Kohana v3 (Mango).

	Kehalim switched over to MongoDB 1 year ago after exhausting other cloud and relational options. As a contextual affiliate network, Kehalim stores all of its advertisers, ads and even impressions on top of MongoDB. MongoDB has outed both MySQL and memcached completely and also provides great hadoop-like alternative with its own map-reduce.
	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.
	Givemebeats.net is an e-commerce music site that allows people to buy beats (music instrumentals) produced by some of the best producers in the world. Now we entirely use MongoDB to store users profile, beats information, and transaction statistics.
	Cheméo, a search engine for chemical properties, is built on top of MongoDB. For a fairly extensive explanation of the tools and software used and some MongoDB tips, please go to chemeo.com/doc/technology.
	Planetaki is place were you can read all your favourite websites in one place. MongoDB has replaced MySQL for the storage backend that does all the heavy lifting and caching of each website's news feed. • Planetaki Powered by MongoDB - SamLown.com (June 2010)
	[ChinaVisual.com] is the leading and largest vertical portal and community for creative people in China. ChinaVisual.com moved from mysql to mongoDB in early 2009. Currently MongoDB powers its most major production and service, like file storage, session server, and user tracking.
	RowFeeder is an easy social media monitoring solution that allows people to track tweets and Facebook posts in a spreadsheet. RowFeeder uses MongoDB to keep up with the high volume of status updates across multiple social networks as well as generate basic stats. • MongoDB for Real Time Data Collection and Stats Generation - Presentation at Mongo Seattle (July 2010)
	Mini Medical Record is designed to facilitate medical care for all members of the public. While useful for everyone, it is especially useful for travelers, professional road warriors, homeless, substance dependent, and other members of the public who receive care through multiple medical systems.
	Open Dining Network is a restaurant data and food ordering platform that provides a RESTful API to take web and mobile orders. MongoDB is used to manage all restaurant, customer, and order information.
	URLi.st is a small web application to create and share list of links. The web application is coded in Python (using the pylons framework) and uses MongoDB (with pymongo 1.6) in production to power its data layer.

	Pinterest is a community to share collections of things you love. Pinterest is built in Python and uses MongoDB for its internal analytics tools and huge data sets like contacts imported from gmail and yahoo.
	LearnBoost is a free and amazing gradebook web app that leverages MongoDB for its data storage needs. LearnBoost is the creator of Mongoose, a JavaScript async ORM for MongoDB that is flexible, extensible and simple to use. • Mongoose - LearnBoost blog (May 2010)
	Kidiso is a safe online playground for children up to 13, with advanced parental controls. In the current setup, we are using MongoDB for logging, analysis tasks, and background jobs that aggregate data for performance (ie search results and allowed content).
	Carbon Calculated provides an open platform that aggregates carbon and green house gas emissions for everything in the world, from passenger transport, raw materials, through to consumer goods. Built on top of this platform, Carbon Calculated offers a suite of products that make carbon calculation accessible and intuitive.
	Vowch is a simple platform for telling the world about all the people, places and things that matter most to you. It is a platform for making positive, public endorsements for anyone or anything from a Twitter account. • View a vowch for MongoDB: http://vow.ch/2ij
	HolaDoctor.com is the most comprehensive health and wellness portal available in Spanish for the global online Hispanic community. MongoDB is being used to store all the content for the site, including GridFS to store article images. Session data is also being persisted on our MongoDB cluster using a custom PHP save handler.
	BlueSpark is the first Russian public spending monitoring project. It includes information about 1,400,000 federal government and 210,000 regional government contracts, as well as information about more than 260,000 suppliers and 26,000 customers. MongoDB stores all reports, customer and supplier information, stats and pre-cached queries. The project was initiated by the Institute of Contemporary Development and launched publicly in July 2010 during the Tver economic forum.
	BlueSpark designs and develops iPhone and iPad applications and specializes in Adobe Flash development, we have a passion for creating great user experiences and products that feel simple to use.
	[Aghora] is a time attendance application specially designed for the requirements of the Brazilian governmental requirements. Our whole application is based on PHP and MongoDB. Click here for more information.
	Man of the House is the real man's magazine, a guide for the jack of all trades trying to be better – at work and at home, as a father and as a husband. The entire backend of the site depends on MongoDB.

	PeerIndex is an algorithmic authority ranking web service that uses MongoDB to scale processing of the firehose of social media, as a distributed data store and middle cache for fast site performance.
	sahibinden.com is an online classifieds marketplace that serves more than 14.5 million unique visitors and over 1.5 billion pageviews a month. sahibinden.com is using MongoDB for storing classifieds data and caching.
	Remembersaurus is a flashcard website targeted at language learners which helps the learners focus on the cards that they are having the hardest time remembering. We're using MongoDB for all of our backend storage, but it's been particularly useful for helping log how well each user knows each of the cards they are studying.
	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.
	Ylastic is using MongoDB extensively in production. For example, MongoDB powers Ylastic's monitors capability.
	BRAINREPUBLIC is a social network for anyone who wants to talk face-to-face - or just audio or chat - with like-minded people from anywhere at anytime.
	Friendmaps is a tool that allows users to view all of their social networks on a single map.
	The affiliate marketing platform Jounce has gone live using MongoDB as the main storage solution for its search data. As of August 2010, ~10 million offers are stored in the database.
	Virb Looking for a place to park your portfolio, your band, your website? Build an elegantly simple website with Virb. You provide the content, we'll help with the rest — for only \$10/month.
	Deal Machine is a streamlined CRM that makes sales fun and effective. We use MongoDB as our main storage. It has helped us a lot to make the web app better and more scalable.
	arrivalguides.com is the largest network of free online (and pdf) travel guides. arrivalguides.com recently launched a new site where they rewrote the whole application switching from SQL server to MongoDB using the NoRM Driver for C#. The website is purely driven by MongoDB as the database backend.

	The Hype Machine keeps track of emerging music on the web. We use MongoDB to accelerate storage and retrieval of user preferences, and other core site data. MongoDB's web-native design and high performance in our workloads was what got our attention. It's from the future!
	Scrabbly is a massively multiplayer online ~~scrabble~~ crossword. Uses MongoDB geospatial indexing. • Building a Scrabble MMO in 48 Hours - Startup Monkeys Blog (September 2010)
	ChatPast synchronizes your chat history from multiple chat clients (Live, Skype, etc.), across multiple computers. Search them, slice them, and get just the data you want. Find everything you've ever talked about. Business users can push important IM conversations into Salesforce and 37 Signals products (Highrise, BaseCamp, etc) seamlessly.
	Stockopedia initially began using MongoDB for its internal analytics system - tracking all activity around 20000+ stocks, sectors and investment topics. Stockopedia is now confidently using the same foundation for building real time analytics, recommendation, categorization and discovery features for both publishers and investors conducting and publishing investment research on the Stockopedia platform.
	TravelPost is a community built by travel enthusiasts for travel enthusiasts. Today, the site has millions of reviews, photos and blogs. TravelPost uses MongoDB for backend storage and analytics applications.
	SoulGoal stores or caches all user data and facebook information in MongoDB.
	Top Twitter Trends is an experimental and on-going project built with today's trending and cutting-edge technologies such as node.js, nginx, and MongoDB.
	bongi.mobi is a place for you to build your own mobi free site from your mobile device! Technologies include: fast document orientated database (MongoDB), full handset detection, image/font resizing (based on handset capabilities), mobile ad serving, geolocation, multimedia (images, video, music), analytics and tracking, click-2-call, SMS, email modules, 3rd party API integration.
	CoStore is an online platform for data exchange, collaboration and data entry. CoStore helps you with importing, transforming and collaborating on all sorts of data files. CoStore also provides reporting tools, such as charts, graphs and network visualizations. CoStore runs in the browser, so you can access it wherever you need it. MongoDB is used as the backend; it stores the data and also runs query steps, which are MapReduce operations.
	Vuzz answers questions like "What are everyone else's opinions?" through showing people's ranking charts. At Vuzz, people can vote, talk about, create and share rankings, and communicate with people that share the same interests as you. By adding up people's votes and showing ranking charts, Vuzz wants to be your time machine that shows people's interests and changes in trend from the past to current. Vuzz has been listed on Killerstartups and jp.Techcrunch. Vuzz uses MongoDB as the main application database.

	Bakodo is a barcode search engine with a social component that helps users make informed decisions while they are shopping. Users can scan barcodes using the camera in their mobile phone and get information about the products they are looking at: where to buy it, lower prices, local stores, and most importantly, what their friends think about it. Bakodo uses MongoDB to store their massive index of million of products.
	noclouds.org is a online system, completely open source and still in development, where users can upload and share information about files. MongoDB has been used since the beginning of the project for almost all systems.
	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.
	CafeClimb.com is a travel website for rock climbers and mountaineers. It is a community oriented site which lets people share their travel experiences for friends to learn from. MongoDB is used to save all traveling information that comes from the users of CafeClimb.com. It also executes all searches that come from the user.
	Keekme is free money management web service build on the top of Ruby on Rails and MongoDB. Using Keekme you will easily track all your expenses wherever you are via web, mail and twitter. Keekme uses MongoDB as a primary datastorage for all application data.
	Vitals.com consolidates and standardizes doctor and other health provider data from over 1,000 sources to help users make informed health care choices. Our technology also powers the websites and backends of insurance companies, hospitals, and other health information brokers. In early October, we switched the datasource for our Find A Doctor location-based search functionality from PostgreSQL to a geo-indexed MongoDB collection. Since then, searches are now five to ten times as fast, and the decreased load on our dataservers permits us to serve more clients. Based on this success, we are transitioning other functionality to MongoDB datasources.
	P2P Financial is Canada's first internet-based business financing company. P2P Financial uses Mongo DB for storing business and financial documents.
	Totsy offers moms on-the-go and moms-to-be access to brand-specific sales, up to 70% off retail. Totsy was re-built upon Li3 and MongoDB to correct performance and scaling limitations incurred while on the prior relational-database platform. The transition to MongoDB has resolved all of Totsy's performance and scaling issues. • MongoDB Ecommerce Case Study: Totsy - Mitch Pirtle's presentation at Mongo Boston (September 2010)
	PlumWillow is a Social Shopping Network where girls who like fashion can build outfits by drag-and-drop, selecting from thousands of top-brand items. PlumWillow was built by a "dream team" of core-developers/contributors to PHP, jQuery and MongoDB who utilized the Agile efficiency of MongoDB and the Vork Enterprise PHP Framework to bring PlumWillow from concept-to-launch in just a few months.
	Qwerly is people search for the social web. Qwerly uses MongoDB to store millions of user profiles and links to social networking sites. We offer an API that makes much of our data freely available to the web.

	phpMyEngine is a free, open source CMS licensed under the GPL v.3. For storage, the default database is MongoDB.
	vsChart allows you to compare products to make it easier to make decisions.
	yap.TV is the ultimate companion to the TV watching experience. It is a completely personalized TV show guide fused with a tuned-for-TV Twitter client, and is the best way to interact with your friends and show fans while watching the tube. We store some of the user generated content in MongoDB. We also use MongoDB for analytics.
	BusyConf makes great conferences even better! BusyConf makes life easier for conference organizers by letting them collect and manage proposals, manage speaker info, build and publish the schedule to multiple web platforms. Conference attendees get a gorgeous, fully cached, offline-enabled schedule with all the details preloaded. MongoDB let's us represent rich conference schedule data in a way that's complementary to it's logical structure and the application's access patterns. Thanks to MongoDB, our code is much simpler, and our application is fast out of the gate. • BusyConf presentation at Washington DC MongoDB User Group - December 2010
	Sentimnt is a personal and social search engine. It connects to your daily diet of information and social networks and allows you to search your online world without being distracted by hundreds of "hits" that are not related to you. Sentimnt uses MongoDB to store tens of millions of documents for users. Our MongoDB instances serve around 2000 queries per second and add 200+ documents every second. We switched from MS SQL to MongoDB and haven't looked back since!
	Workbreeze is fast and minimalistic tool for the freelance offers realtime search. MongoDB is used as a global project storage.
	Kompasiana is the biggest citizen journalism In Indonesia. Based on alexa rank, Kompasiana is in top 100 biggest site In Indonesia. MongoDB is used to store all posts data.
	Milaap works with established, grassroots NGOs and Microfinance institutions focused on holistic community development.

	Agent Storm is a complete Real Estate Contact Manager which empowers you and your agents to build and manage your Real Estate business. Import your Real Estate Listings via RETS and easily display IDX listings on your web site, syndicate out to over 100 Real Estate Portals and build custom eye catching eFlyers all at the click of a button. When a API call is received we send that query to MongoDB which performs the search and returns the primary key id's of the properties matching the query. Those id's are then looked up in MySQL using the primary key returned from MongoDB. Once we have the full result set the results are served up and passed back to the client as either XML or JSON. When a Property is updated either via the Web or external the external python importers (not shown) then the id of the property is stored in a Redis/Resque queue and a worker fired to synchronize the MongoDB with the changes. All this means that on average API query results are returned in ~100 milliseconds. • Now with 50-100 millisecond-search-results - Agent Storm Blog (November 2010)
	Mashape is a frustration-free online storefront for developers who want to consume or generate and distribute an API of any kind of service, either an open source project.
	The UK Jobsite is an easy to use job board allowing users to quickly search and apply for job vacancies. We use MongoDB for all aspects of the site - in it's entirety it fully runs the awesome MongoDB from job searches to user information everything is kept in documents. For more information we have written a few of the reasons of why we chose Mongo and continue to use it in a full production site - this can be found at http://www.theukjobsite.co.uk/tech.php.
	Tastebuds music dating lets you meet like-minded single people who share your love for music. Users can connect with their last.fm username – or simply enter a couple of their favourite artists – and are immediately shown single people who share their musical preferences. Tastebuds is also integrated with popular events service Songkick.com allowing users to arrange to meet up at concerts they're attending. MongoDB has dramatically increased the speed at which we can search our database of artists. This allows users to rapidly select their preferred artists using our homepage artist autocomplete.
	Skimlinks enables publishers to easily monetise online content by converting normal product links into equivalent affiliate links on the fly and via SkimWords will create relevant contextual shopping links to products mentioned in the content. Bringing together more than 25 affiliate networks and over 12,000 retailer programs in one place, Skimlinks makes affiliate marketing a viable mainstream business model and ensures no money is left on the table. We use Mongo to log and report on the almost 2 Billion impressions we get every month.
	www.vanilladesk.com is ITIL based servicedesk/helpdesk solution provided as SaaS. MongoDB is used as the main database engine to store all tickets, configuration items, customer and contact details. MongoDB's document oriented approach and high-availability model supported by replica-sets is exactly what enables VanillaDesk to process records fast and keep them safe and available.
	Summify uses MongoDB as our primary database in which we store all of the news articles we crawl (metadata and HTML content), as well as user timelines. We periodically trim the database, i.e. we keep only the most recent 1000-5000 news stories for each user. Besides this, we also use MongoDB as a cache for URL redirects and Twitter user information (usernames, scores, etc.)

	dakwak is the easiest and fastest way to translate your website to any language your audience and visitors want. It is a next-generation website localization tool and service. With dakwak, you are minutes away from getting your website localized and translated to more than 60 languages.
	Kapost's online newsroom technology enables our users to manage a large group of contributors to produce content. We use MongoDB to store all the data that our users create and provide performance analytics for the newsroom content.
	ContactMe is a "light customer relationship management (CRM)" tool designed for small businesses. For many small businesses, enterprise CRM solutions are too cumbersome and technically demanding; ContactMe.com aims to fix this. Besides providing free contact forms and buttons for websites, blogs, and social media sites like Facebook, ContactMe consolidates all of a users' leads and contacts into one place, making for easy and efficient contact management.
	Moontoast Impulse is a free Facebook application that lets fans play, share, and purchase music right from a Facebook fan page. Moontoast is a social commerce company based in Boston and Nashville. Their tools power communities, stores, and private sales clubs that open the doors to a world of new fan-building and revenue-generating opportunities. Moontoast leverages the following technologies in Moontoast Impulse: MongoDB, PHP 5.3, Zend Framework, Doctrine 2, jQuery, Gearman, Ruby, MySQL, and more. Moontoast uses MongoDB to store all product and file system data for Moontoast Impulse.
	Shopperhive is a social price comparison shopping site providing comprehensive search, social product reviews, video reviews and the cheapest prices online. Shopperhive uses MongoDB as a highly scalable data back-end storage.
	tracknose develops location-based applications for mobile phones that detect your GPS position and reward you with discount vouchers when you enter a retail outlet that is part of the scheme. MongoDB was chosen for excellent scalability and capacity to manage geospatial indexes, facilitating the generation of georeferenced statistics.
	TalkAboutHealth matches people with similar health concerns for answers and live chats. Meet others just like you, learn from experienced peers, and share your experiences.
	Wusoup is a free online application for meeting interesting people outside of your social circle. MongoDB handles all user and message data, along with all analytics that power the Wusoup algorithm.
	Fave is a local search application designed to help people find and communicate with their favorite businesses, and to help businesses expand their presence on major search engines. Soon, Fave will be releasing next-generation mobile applications to help its audiences harness the power of geo-aware web services. Infrastructure/architecture containing 14+ million business records; strong emphasis on performance/response time; attempting to leverage flexibility to deliver bulk daily ads/deals and pay per call campaigns
	Doodle is the world's leading consensus-scheduling service with millions of users worldwide. We use MongoDB as an addition to MySQL and plan to slowly but steadily persist more data via MongoDB and less data via MySQL.

	FindTheBest is an objective, socially curated comparison engine that allows you to find a topic, compare your options and decide what's best for you. Ultimately, FindTheBest allows you to make faster and more informed decisions by allowing you to easily compare all the available options, view expert ratings on products and services and read and write reviews on millions of listings. FindTheBest uses MongoDB for traffic monitoring and analysis as well as storage of user reviews.
	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.
	Fuseware provides realtime analysis into social networks, blogs, forums, review sites, multimedia sites and more. This analysis is structured for easy understanding and dissemination by company decision makers. We use Mongo DB clusters to provide scalable data storage for the myriad different web objects that we store. We have hundreds of gigabytes of stored data, and Mongo allows us full horizontal scalability and data access.
	Wherevent is a search engine for geolocated events. It helps people finding local events in their area or in other cities. The entire database is based on a mongoDb sharding. We especially use the geospatial feature.
	Skyline Innovations develops and finances distributed renewable energy projects for small businesses. We deploy custom monitoring solutions to maximize the energy production and efficiency gains. We use MongoDB for data warehousing and fast, flexible multidimensional analyses of our real-world data.
	Athena Capital Research is a quantitative investment manager, specializing in automated trading strategies.
	Memrise combines the art and science of memory to turbocharge learning. We love using MongoDB for all of our learning and web analytics, and any time we want to be able to prototype rapidly without worrying about schema ossification.
	Dather is a content platform that assembles all of your content by events into an interactive timeline. You can invite friends, family, fans, etc... to join your timeline and contribute their own media and events.
	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.

	Fundastic.info is CrunchBase data + visualization. It offers a graphical way to visualize funding by investors and financial organizations implemented using the Crunchbase API. Fundastic provides dedicated pages for each investor and financial organization and provides various charts to help understand the investment pattern of an investor or financial organization. Fundastic.info uses MongoDB to store all its investor and financial organizations funding data and performs aggregations on it. Data is stored in MongoDB as JSON documents and integrates seamlessly with Javascript.
	Foofind Labs is a universal search engine. In Foofind we work creating software that crawls every network and indexes every file. This way, we intend to offer the largest collection of links available. Foofind is a creation by Pablo Soto, P2P technology designer, developed in Madrid, Spain by MP2P Technologies. MongoDB is used to store all the data on the site. This includes several million files, indexed with Sphinx in order to allow text searches.
	Beaconpush is a cloud service for push messaging in the browser. Enables any web site to integrate real-time data with a few lines of JavaScript. Supports native Web Sockets with a wide array of fallback methods for older browsers. Beaconpush uses MongoDB to store all message statistics - in real-time. This data is used for presenting reports, charts and statistics for each user. MongoDB handles this with a very low system impact and has proven very efficient for this kind of data storing.
	Easy Bill, India's first chain of one stop payment collection centres, is yet another initiative in adding value to the life of the common man. Through its vast network of Retail Partners, it offers the consumer never-before convenience in various kinds of transactions. Extensive research revealed that one of the biggest problems faced by the common man is the hassle of dealing with large public and private enterprises for bill payment services. In the last decade, the consumer base of several industries has grown exponentially, though the support services have not been able to keep pace. We use MongoDB for activity logging of our application since our main concern is to write data to db instead of to read.
	DeskMetrics is a real-time analytics service for desktop software that enables developers to understand how their users are using their applications. How many are installing, uninstalling, executing their application, where are the users from and help developers to solve bugs by analyzing the exceptions that happened on the user's machine. We use MongoDB to store all data gathered from the users applications and all generated reports. We have chosen MongoDB most because of its simplicity, specially to shard the collections. Shards are very important for us since we collect and analyze a lot of data.
 SEARCH SOCIAL MOBILE	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!

	Interstate provides businesses and developers alike with a solution for keeping on top of development tasks using awesome roadmaps! All of our client's data is stored using MongoDB (roadmaps, roads, user data, etc).
	Proxlet helps you fight the noise on Twitter, by letting mute users, apps & hash-tags. At it's core, Proxlet is a Twitter API proxy, which allows it to work with a variety of native clients, as well as with Twitter.com by way of browser extensions. Proxlet uses MongoDB to store both code it runs to modify the Twitter stream, per-user setting documents, as well as growing analytics data. Since Proxlet's data set is 100% JSON & designed to scale horizontally, MongoDB has proven to be the perfect fit.
	fiesta.cc makes creating mailing lists dead-simple, and fun. The goal is to bring groups together around a tool that everybody already uses and understands: email. We use MongoDB to store all of the data powering the website and the mail serving infrastructure. We're running a three node replica set.
	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.
	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
	Dayload provides a hassle-free Amazon Web Services usage monitoring. We aggregate customer's AWS resource usage and CloudWatch monitoring logs into MongoDB and deliver daily statistics as a HTML mail. Storing AWS resource usage and CloudWatch monitoring logs. Analyzing data with JavaScript MapReduce function.
	Avivo is creating modern Identities, Brand design concepts and delivers superb service in a form of touch, mobile, web and custom made solution for LOB (line of business) that enhance user digital experience. Through design and branding processes we help our clients to move from static world into digital world of interaction. Ultimately we are delivering engaging software and custom made applications to our clients that build brand loyalty and helps them to innovate. Our passion for design and commitment to innovation is leading us to dive deeply into research and development with new technologies to brings out the very best for our clients. We are multinational team that lives and works in Ljubljana, Slovenia. MongoDB is the primary database for all of our generated links and tracking statistics.

	Abusix is the network and email abuse handling specialist providing network abuse reporting, customized spamfeeds(www.spamfeed.me), consulting and tools against network abuse for ISPs, ESPs and technology providers. We know about the needs network operators and Internet Service Providers have today, while handling thousands of incoming complaints and millions of incoming datasets about abusive behavior in their network manually, but still not solving blacklistings and reputation problems. There is a major need for a solution that handles all these information in a semi or fully automatic way in order to increase customers security and satisfaction, increase infrastructure security and performance, free resources in abuse departments for real anti-abuse work and minimize customer care costs. We use the MongoDB in environments where we need a fast and scalable storage solution which is easy to access. It almost suits all needs and it is fun to work with.
	Freerice is developed by World Food Programme, the largest UN humanitarian organisation. Freerice.com is a social quizz-like game where each good answer to a question make you win 10 rice grains. Since the beginning 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.
	Idea2 is a versatile, customizable data management application. Our platform easily allows anyone to create and implement a hosted business management solution that connects customer relationships, projects, customer care and support, business intelligence, and document management in a browser-based, fully managed environment. Idea2 uses MongoDB as its main storage engine for the application and workflow engine. We moved to MongoDB from a traditional SQL database. In making this transition, we also created a better user experience for our clients due to improved application response.
	FamilyTies is family networking - like social networking but designed to make connecting extended families and friends of families easier. Profiles for households and each household member including children under 13, basic family trees, photo sharing, blogs, comments and birthday reminders with much more to come. Mongo is used to store http sessions, photos, the operational data, logs, audit trails and various cache objects that speed up frequently needed expensive to compute data like family trees and relationship maps.
	Pracanowo.pl is a job portal that helps people create on-line resumes that can be saved into different formats and finds jobs based on experience listed. MongoDB is used to store user account information and in few month to store job postings and board information.
	Republika Online is the biggest Islamic news portal in Indonesia. It's an online version of "Harian Republika", a national newspaper. We use sharded MongoDB as storage of all news and comments.

See also

- [MongoDB Apps](#)
- [Use Cases](#)
- [User Feedback](#)

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

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

phpMyEngine

A free, open source CMS licensed under the GPL v.3. For storage, the default database is MongoDB.

Websko

Websko is a content management system designed for individual Web developers and cooperative teams.

Events

Follow [@mongodb](#), Like MongoDB on [Facebook](#) or join our [LinkedIn](#) group to be the first to hear about new events, updates to MongoDB, and special discounts.

MongoDB Conferences

Submit a [proposal](#) to present at an upcoming MongoDB conference.

- **Mongo Philly: April 26**

One day conference during Philly Tech Week
[Click here for conference agenda and registration.](#)

- **Mongo Moscow: May 16**

Save the date! Details TBA.

- **First Meeting of Stockholm MongoDB User Group: May 19**

Free meetup hosted by Valtech
[Click here for agenda and registration.](#)

MongoDB Training

[MongoDB for Administrators](#)

Redwood Shores, CA
April 26-27

Sold Out

[MongoDB for Administrators and Operations](#)

London, UK
May 5-6

[MongoDB for Administrators](#)

New York, NY
May 24-25

[MongoDB for Developers](#)

Redwood Shores, CA
June 28-29

Click [here](#) for full course content information and training calendar.

- **MongoSF 2011: May 24**

One day multi-track conference.
[Call for Proposals now open.](#)

- **MongoNYC 2011: June 7**

One day multi-track conference.
[Call for Proposals now open.](#)

Webinars

Building your first MongoDB Application

Tuesday, April 5th at 1pm ET / 10am PT
[Register](#)

Attend MongoDB Office Hours and User Group Meetups

New York City

Wednesdays 4 - 6:30pm ET

10gen NYC Office
 134 5th Avenue - 3rd Floor
 Between 18th & 19th Streets

San Francisco

Mondays 5 - 7pm PT

Meet 10gen engineers at the Epicenter Cafe. Look for a laptop with a "Powered by MongoDB" sticker.
 Epicenter Cafe
 764 Harrison St
 Between 4th St & Lapu St

Redwood Shores

Wednesdays 4 - 6pm PT

10gen CA Office
 100 Marine Parkway
 Suite 175
 Redwood City, CA

Mountain View

1st and 3rd Thurs of the Month 4 - 6pm PT

Look for the laptop with the MongoDB sticker!
 Red Rock Coffee
 201 Castro St
 Mountain View, CA

Community Meetups and Conferences

There are MongoDB User Groups in [Atlanta](#), [Chicago](#), [London](#), [New York](#), [San Francisco Bay Area](#), [Stockholm](#), and [Washington, DC](#).

April 2011	Event
April 7-9	TBA talk Ethan Gunderson, Obtiva Scottish Ruby Conference
April 11-14	Building Your First Web Application with MongoDB Roger Bodamer, 10gen MongoDB & MySQL Joined at the Hip Eliot Horowitz, 10gen O'Reilly MySQL Conference & Expo Santa Clara, CA Register using the discount code mys11gen for 20% off
April 12	Philadelphia Ruby User Group MongoDB talk by 10gen engineer Kyle Banker. Philadelphia, PA
April 19	Working with Geo Data in MongoDB Mathias Stearn, 10gen Where 2.0 Santa Clara, CA
April 27-28	Emerging Technologies for the Enterprise Philadelphia, PA
May 2011	Event
May 10	Toronto MongoDB User Group Toronto, ON
May 19	Stockholm-MongoDB Event Hantverkargatan 5 Stockholm
May 25	Mongo, Linq, and the C# driver: Memoirs from a driver test pilot Justin Dearing New York ALT.NET Group

May 26	Glue Conference Bloomfield, CO Get 10% off with discount code: earlysponsor
June 2011	Event
June 6-7	Berlin Buzzwords 2011 10gen Speaker: Mathias Stearn Urania Berlin

[Slide Gallery](#) | [\[More Presentations and Video\]](#)

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](#).

<#comment>

Video & Slides from Recent Events and Presentations

Table of Contents:

[\[MongoDB Conferences \]](#) [\[Ruby/Rails \]](#) [\[Python \]](#) [\[Alt.NET \]](#) [\[User Experiences \]](#) [\[More about MongoDB \]](#)

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.

Slide Gallery

At present, the widgets below will not function in the Chrome browser. We apologize for the inconvenience.

[Click here](#) to visit our full listing of videos & slides from recent events and presentations.

Introduction to MongoDB	User Experience
 Get your SlideShare Playlist	 Get your SlideShare Playlist
Ruby/Rails	Python
 Get your SlideShare Playlist	 Get your SlideShare Playlist
Java	PHP
 Get your SlideShare Playlist	 Get your SlideShare Playlist
MongoDB & Cloud Services	More About MongoDB
 Get your SlideShare Playlist	 Get your SlideShare Playlist

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](#)

Videos

- [Video from MongoDB Conferences](#)
- [MongoDB Blip.tv Channel](#)
- [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. These are links to some benchmarks done by the community, feel free to add your own.

Disclaimer: these benchmarks were created by third parties not affiliated with MongoDB. MongoDB does not guarantee in any way the correctness, thoroughness, or repeatability of these benchmarks.

February 24, 2011 - [MongoDB Performance for Data Bigger than Memory](#) speed tests finding data from disk vs. memory.

March 9, 2010 - [Speed test between django_mongokit and postgresql_psycpg2](#) benchmarks creating, editing, and deleting.

February 15, 2010 - [Benchmarking Tornado's Sessions](#) flatfile, Memcached, MySQL, Redis, and MongoDB compared.

January 23, 2010 - [Inserts](#) and [queries](#) against MySQL, CouchDB, and Memcached.

May 10, 2009 - [MongoDB vs. CouchDB vs. Tokyo Cabinet](#)

July 2, 2009 - [MongoDB vs. MySQL](#)

September 25, 2009 - [MongoDB inserts using Java](#).

August 11, 2009 - [MySQL vs. MongoDB vs. Tokyo Tyrant vs. CouchDB](#) inserts and queries using PHP.

August 23, 2009 - MySQL vs. MongoDB in PHP: [Part 1 \(inserts\)](#), [Part 2 \(queries\)](#), against InnoDB with and without the query log and MyISAM.

November 9, 2009 - MySQL vs. MongoDB in PHP and Ruby inserts ([original Russian](#), [English translation](#))

See Also

- <http://blog.mongodb.org/post/472834501/mongodb-1-4-performance>

FAQ

This FAQ answers basic questions for new evaluators of MongoDB. See also:

- [Developer FAQ](#)
- [Sharding FAQ](#)

MongoDB Intro FAQ

- [MongoDB Intro FAQ](#)
 - [What kind of database is the Mongo database?](#)
 - [What languages can I use to work with the Mongo database?](#)
 - [Does it support SQL?](#)
 - [Is caching handled by the database?](#)
 - [What language is MongoDB written in?](#)
 - [What are the 32-bit limitations?](#)

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.

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. See [the blog post](#) for more information.

Product Comparisons

Interop Demo (Product Comparisons)

Interop 2009 MongoDB Demo

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 (v1.6)	?
Replication	Master-master (with developer supplied conflict resolution)	Master-slave (and " replica sets ")	Master-slave
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 . (As of June 2010, coordinate system is cartesian. Spherical coming soon.)	?
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 require 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 scalability (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.

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 in the MongoDB storage engine via the `--dur` 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.
- 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. For example, [updating real time analytics counters](#) for a web sites (pages views, visits, etc.)

Generally, we find MongoDB to be a very good fit for building web infrastructure.

Licensing



If you are using a vanilla MongoDB server from either source or binary packages you have NO obligations. You can ignore the rest of this page.

- MongoDB Database
 - Free Software Foundation's [GNU AGPL v3.0](#).
 - Commercial licenses are also available from [10gen](#).
- Drivers:
 - [mongodb.org](#) "Supported Drivers": [Apache License v2.0](#).
 - Third parties have created [drivers](#) too; licenses will vary there.
- Documentation: [Creative Commons](#).

From our [blog post](#) on the AGPL:

Our goal with using AGPL is to preserve the concept of copyleft with MongoDB. With traditional GPL, copyleft was associated with the concept of distribution of software. The problem is that nowadays, distribution of software is rare: things tend to run in the cloud. AGPL fixes this "loophole" in GPL by saying that if you use the software over a network, you are bound by the copyleft. Other than that, the license is virtually the same as GPL v3.

*Note however that it is **never** required that applications using mongo be published. The copyleft applies only to the mongod and mongos database programs. This is why Mongo DB drivers are all licensed under an Apache license. Your application, even though it talks to the database, is a separate program and "work".*

If you intend to modify the server and distribute or provide access to your modified version you are required to release the full source code for the modified MongoDB server. To reiterate, you **only** need to provide the source for the MongoDB server and not your application (assuming you use the provided interfaces rather than linking directly against the server).

A few example cases of when you'd be required to provide your changes to MongoDB to external users:

Case	Required
Hosting company providing access MongoDB servers	yes
Public-facing website using MongoDB for content	yes
Internal use website using MongoDB	no
Internal analysis of log files from a web site	no

Regardless of whether you are *required* to release your changes we request that you do. The preferred way to do this is via a [github](#) fork. Then we are likely to include your changes so everyone can benefit.

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 **roger 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 for Web Development
Mitch Pirtle

MongoDB: Sag Ja zu NoSQL
Marc Becker

The Little MongoDB Book
Karl Seguin
Free ebook

Slides and Video

Table of Contents:

- [MongoDB Conferences](#)
- [Webinars](#)
- [Slideshare Galleries, by Topic](#)

MongoDB Conferences

Each MongoDB conference consists a variety of talks, including introductions to MongoDB, features and internals talks, and presentations by production users. Browse through the links below to find slides and video from the 2010 conferences. (Check out the 2011 events [here](#).)

- [MongoSV](#)
- [MongoDC](#)
- [Mongo Chicago](#)
- [Mongo Berlin](#)
- [Mongo Boston](#)
- [Mongo Seattle](#)
- [MongoFR](#)
- [MongoUK](#)
- [MongoNYC](#)
- [MongoSF](#)

Webinars

Webinars are another great way to learn more about MongoDB (or start to if you're a beginner). You can listen to pre-recorded sessions or register to listen to future webinars [here](#).

Slideshare Galleries, by Topic

At present, the widgets below will not function in the Chrome browser. They can be viewed at the [MongoDB Slideshare page](#). We apologize for the inconvenience.

Introduction to MongoDB	User Experience
 Get your SlideShare Playlist	 Get your SlideShare Playlist
Ruby/Rails	Python
 Get your SlideShare Playlist	 Get your SlideShare Playlist

Java	PHP
 Get your SlideShare Playlist	 Get your SlideShare Playlist
MongoDB & Cloud Services	More About MongoDB
 Get your SlideShare Playlist	 Get your SlideShare Playlist

Doc Index

Space Index

0-9 ... 6	A ... 13	B ... 21	C ... 37
F ... 8	G ... 9	H ... 8	I ... 18
L ... 8	M ... 32	N ... 2	O ... 12
R ... 28	S ... 31	T ... 11	U ... 13
X ... 0	Y ... 0	Z ... 0	!@#\$... 0

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. \ 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 ...
-  [1.8 Release Notes](#)
MongoDB 1.8 is a dropin replacement for 1.6. To upgrade, simply shutdown {{mongod}} and the restart with the new binaries. \ Please note that you should upgrade to the latest version of whichever driver you're using The options to the Map/Reduce command have changed in 1.8 ...

A

-  [A Sample Configura](#)
following exampl mongos process below, a python machine ...
-  [About](#)
-  [About the local data](#)
mongod}} reserv that its contents authenticating Si equivalent to aut permissions acrc
-  [Adding a New Set M](#)
Adding a new nc either have an ei set member. Wh
-  [Adding an Arbiter](#)
Arbiters are node have a copy of th readable second (e.g. if a set only
-  [Admin UIs](#)
Several administ Gourley's blog <http://blog.timgoi> has a good sumr <http://github.com> <http://github.com>
-  [Admin Zone](#)
Community Adm deployment <http://blog.boxed> Survey of Admin <http://blog.timgoi> MongoDB Nagio MongoDB Cacti
-  [Advanced Queries](#)
Introduction Mon page lists some i

JSONstyle object database.

 [Aggregation](#)
Mongo includes {{group by}} operator crafted using MapReduce Count {{count()}}

 [Amazon EC2](#)
MongoDB runs v3.2 page includes screenshot of most EC2 types 64 ...

 [Architecture and Components](#)
MongoDB has two types of nodes: the mongod process and the mongos process. The mongod process may be running on a server or on a client.

 [Articles](#)
See also the Use Cases presentations, blog posts, Right Data Mode http://blog.mongodb.org/ nonrelational data

 [Atomic Operations](#)
MongoDB supports atomic operations. It does not support multi-document updates. First, in a single document, and slow.

B

 [Backing Up Sharded Cluster](#)
Backing up a sharded cluster is fairly complex because there are so many moving pieces. That is why it is very important to have a very resilient setup when using sharding: at least 3 config servers in at least 2 different locations each ...

 [Backups](#)
Several strategies exist for backing up MongoDB databases. A word of warning: it's not safe to back up 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.

 [Benchmarks](#)
MongoDB does not publish any official benchmarks. These are links to some benchmarks done by the community, feel free to add your own. Disclaimer: these benchmarks were created by third parties not affiliated with MongoDB. MongoDB does not guarantee in any way ...

 [Books](#)
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> You can download samples at 10gen.com/books <http://www.10gen.com/books>. Kristina Chodorow and Mike Dirolf Peter ...

 [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?pagelId=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 binary-encoded serialization of JSON-like documents. BSON is 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; ... 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 ...

 [bsonspec.org](#)

 [Building](#)
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. Note: see the Downloads

C

 [C Language Center](#)
C Driver {}The M... It's written in pure C, no dependencies, a README <https://github.com/mongodb/c-driver>

 [C Sharp Language Center](#)

 [C Tutorial](#)
this document is an introduction to the C program. First, it shows how to take a look at the C guide for a language.

 [C++ BSON Library](#)
Overview The M... implements the BSON interface. It can be used as a library or as a standalone one is not using

 [C++ Driver Compilation](#)
C\ driver is included in the MongoDB distribution. To compile the "c" driver, you need to have the "c" directory of the driver.

 [C++ Language Center](#)
C\ driver is available for MongoDB. This is the same as the C driver, but it has been compiled for C++.

 [C++ Tutorial](#)
this document is an introduction to the C++ program. First, it shows how to take a look at the C++ guide for a language.

 [Caching](#)
Memory Mapping MongoDB, and its strategy, the open caching.

 [Capped Collections](#)
Capped collections

DOCS:Downloads page for prebuilt binaries! Subsections of this section ...



Building Boost

MongoDB uses the www.boost.org Boost C\ libraries. Windows See also the prebuilt libraries <http://www.mongodb.org/pages/viewpageattachments.action?pagelD=12157032> page. By default c:\boost\ is checked for the boost files. Include files should be under \boost\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: There are binaries for many platforms. Most users won't need to compile mongo themselves. See the Downloads page for these prebuilt binaries! SpiderMonkey, UTF8, and/or Ubuntu Most prebuilt Javascript SpiderMonkey binaries do not have UTF8 compiled in. Additionally, Ubuntu ...



Building for OS X

set up your OS X computer for MongoDB development: Upgrading to Snow Leopard If you have installed Snow Leopard, the builds will be 64bit \ so if moving from a previous OS release, a bit more setup may be required than ...



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

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 <http://www.scons.org/> is the make mechanism, although a .vcxproj ...



Building Spider Monkey

MongoDB uses SpiderMonkey <http://www.mozilla.org/js/spidermonkey/> for serverside Javascript execution. The mongod project requires a file js.lib when linking. This page details how to build js.lib. Note: V8 <http://code.google.com/p/v8/> Javascript support is under ...



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 Visual Studio 2010 Project File A VS2010 vcxproj file is available for building the shell. From the mongo directory open ...



Building with Visual Studio 2008

MongoDB can be compiled for Windows (32 and 64 bit) using Visual C. SCons <http://www.scons.org/> is the make mechanism we use with VS2008. (Although it is possible to build from a sln file with vs2010 DOCS:Building with Visual Studio 2010 ...



Building with Visual Studio 2010

MongoDB can be compiled for Windows (32 and 64 bit) using Visual C. SCons <http://www.scons.org/> 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 ...

autoFIFO ageou
like the "RRD" c
collections auton



CentOS and Fedora

10gen publishes
platforms. The p
production relea
mongo10genuns
releases (odd mi



Checking Server Me

Checking using l
usage informatio
db.serverStatus(
mongod process
values are in me



Choosing a Shard K

Example Docum
chunks. A chunk
key is fine enoug
logging example



Clone Database

MongoDB includ
another. // copy ;
name on another
another on the s



Collections

MongoDB collec
think of them as
MongoDB collec
documents are u
since MongoDB



Command Line Par

MongoDB can be
based configurat
options by runnir
./mongod help In



Commands

Introduction The
Database comm
operations, or to
Database Commr
special ...



Community

tweet.png! <http://>
<http://www.faceb>
<http://www.linker>
Community Resc
<http://groups.goc>



Community Info



Comparing Mongo [

We are getting a
different?"
with schemaless
place \ we are b



Configuring Shardin

Introduction This
sharding cluster.
Two or more she
process. For test



Connecting

C\ driver include:
class DBClientIn
a connection to ;
classes exist for



Connecting Drivers i

Ideally a Mongol
replica set DOC
which to commu

- general steps are
-  [Connecting to Replica Set](#)
Most drivers have a general, this is v pair. Instead of t comma separate
 -  [Connections](#)
MongoDB is a d waits for connect see something lil waiting ...
 -  [Contributing to the C](#)
Qualified volunteer Contact us for m
 -  [Contributing to the F](#)
easiest way to c http://jira.mongodb read on... Finding on Jira http ...
 -  [Contributors](#)
10gen Contributors
 -  [Conventions for Mor](#)
Interface Conventions possible. Of course appropriate. How consistent across
 -  [cookbook.mongodb.](#)
 -  [createCollection Command](#)
Use the createCollection command. Often this is used db.createCollection() to create collection
 -  [Creating and Deleting](#)
 -  [CSharp API Documentation](#)
Under construction the documentation suggestions for are appreciated. In the
 -  [CSharp Community](#)
Community Support http://github.com http://code.google http://github.com Community ...
 -  [CSharp Driver Serialization](#)
Introduction This deserialization) (Serialization is the be saved in MongoDB reconstructing array
 -  [CSharp Driver Tutorial](#)
C# Driver version tutorial introduction consists of two li can be used index
 -  [CSharp Language C](#)
C# Driver The MongoDB. C# Driver Tutorial CSharp http://api.mongodb
 -  [Cursors](#)

D

-  [Data Center Awareness](#)
1.6.0 build of replica sets does not support much in terms of data center awareness. However additional functionality will be added in the future. Below are some suggestions configurations which work today. Primary plus DR site Use one site ...

E

-  [EC2 Backup & Restore](#)
Overview This is running on EC2 instance you backup MongoDB

	Data Processing Manual
	DRAFT TO BE COMPLETED. This guide provides instructions for using MongoDB batch data processing oriented features including map/reduce DOCS:MapReduce. By "data processing", we generally mean operations performed on large sets of data, rather than small ...
	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 To enable profiling, from the {{mongo}} shell invoke: > db.setProfilingLevel(2); > db.getProfilingLevel() 2 Profiling ...
	Database References
	MongoDB is nonrelational (no joins), references ("foreign keys") between documents are generally resolved clientside by additional queries to the server. Two conventions are common for references in MongoDB: first simple manual references, and second, the DBRef standard, which many drivers support ...
	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 ...
	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 ...
	Design Overview
	Developer FAQ
	Also check out Markus Gattol's excellent FAQ on his website http://www.markusgattol.name/ws/mongodb.html . 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 ...
	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
	Generally, with MongoDB we are not building queries from strings, so traditional SQL Injection http://en.wikipedia.org/wiki/SQLInjection attacks are not a problem. More details and some nuances are covered below. MongoDB queries are represented as BSON objects. Typically the programming ...
	Doc Index
	Document-Oriented Datastore
	Documentation
	Dot Notation
	Dot Notation
	Dot Notation (Reaching into Objects)
	MongoDB is designed for store 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' ...
	Downloads
	See also #Packages. Version OS X 32 bit OS X 64 bit Linux 32 bit Linux 64 bit Windows 32 bit Windows 64bit Solaris i86pc Solaris 64 Source Date Change ...
	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\ Perl \ \ \ {{array()}} \ \ {{BasicDBList}} BSONObj or bson::bo \ \ {} {{new stdClass}} {} {{BasicDBObject}} BSONObj ...

	1.8 (or above) or
	Emacs tips for Mong
	You can edit con on http://code.go sample setup to change the defai
	Error Codes
	Error Code \ De grow 11000 dupl 11001 duplicate
	Error Handling in Mc
	an error occurs c object instead of the reserved key
	Events
	Follow @mongo Facebook http:// to be the first to l discounts. Mong
	Excessive Disk Spa
	You may notice t are larger than t reasons for this. DOCS:Replicat

-  [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# CSharp Language Center C\ C Language Center Erlang <http://github.com/TonyGen/mongodberlang> Haskell <http://hackage.haskell.org> ...
-  [Durability and Repair](#)
Single Server Durability and Recovery v1.7.5 of MongoDB supports durability with recovery (via write ahead log a journal) in the storage engine as an option. See the Journaling page. When enabled, repairs are not required on a hard crash. The {{ dur}} option was used ...
-  [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 {{dur}} 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 MongoDB Intro FAQ What kind of database is the Mongo database? MongoDB is an documentoriented DBMS. Think of it as MySQL but JSON (actually ...
-  [Feature Checklist for Mongo Drivers](#)
Functionality Checklist This section lists tasks the driver author might handle. Essential BSON serialization/deserialization Basic operations: {{query}}, {{save}}, {{update}}, {{remove}}, {{ensureIndex}}, {{findOne}}, {{limit}}, {{sort}} Fetch more data from a cursor when necessary ...
-  [File Based Configuration](#)
in addition to accepting command line parameters, MongoDB can also be configured using a configuration file. A configuration file to use can be specified using the {{f}} or {{\config}} command line options. On some packaged installs of MongoDB (for example Ubuntu & Debian ...
-  [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 ...
-  [Forcing a Member to be Primary](#)
Replica sets automatically negotiate which member of the set is primary and which are secondaries. However, if for administrative reasons you want to force a node to be primary at a given point in time, use the replSetFreeze and replSetStepdown commands. If we have members A, B, and C ...
-  [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](#)
fsync Command Version 1.3.1 and higher 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 ...
-  [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 ...

H

-  [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 ...
-  [Home](#)
Getting Started Quickstart \ Downloads \ Tutorial \ Features <http://www.mongodb.org> SQL to Mongo Mapping Chart <http://www.mongodb.org/display/DOCS/SQLtoMongoMappingChart> Development Manual C <http://github.com/mongodb/mongocdriver> \ C# CSharp Language Center \ C\ C ...
-  [Hosting Center](#)
CloudStyle provides cloudstyle hosted MongoDB instances MongoLab <https://mongolab.com/home/> is currently in private beta cloudControl <http://cloudcontrol.com/> offers a fully managed platformasaservice solution with

G

-  [Geospatial Indexing](#)
v1.3.3\ MongoDB with locationbase location." It can & closest N museu
-  [getLastError](#)
Most drivers, and one check the er DOCS:Command getlasterror is pri
-  [Getting Started](#)
-  [Getting the Software](#)
Placeholder \$\$\$
-  [Git Commit Rules](#)
commit message messages should commits should i
-  [GridFS](#)
GridFS is a spec supported driver: Rationale The d: DOCS:BSON ob
-  [GridFS in Ruby](#)
GridFS, which st in MongoDB. It v each of those ch to achieve this: c
-  [GridFS Specification](#)
Introduction Grid works by splitting chunk is stored & about the file, inc
-  [GridFS Tools](#)
File Tools {{mon from the commai ./mongofiles put list connected to

I

-  [Implementing Auth](#)
current version c authenticates a u database. the database in c
-  [Import Export Tools](#)
you just want to i need these tools collection; both t the (capped ...
-  [Index-Related Comr](#)

MongoDB as one of their powerful addons. Read the blog post MongoDB Setup ...



How does concurrency work

mongos For sharded DOCS: Sharding environments, mongos 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 original mongod ...



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 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 ...



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 ...

Create Index {{e creates an index test > db.myColli db.system.index subsequent doc



Indexes

Indexes enhance about the kinds (relevant indexes relatively easy. I



Indexes in Mongo



Indexing Advice and

We get a lot of q of these. There e MongoDB work (techniques for bu



Indexing as a Backg

Slaves and replic releases (includi primary, the slav index builds ther



Inserting

When we insert (Documents are (Ruby hashes, to documentorienta



Installing the PHP D



Internal Commands

Most commands {{ \$cmd.findOne(db.\$cmd.findOne , "ok" : 1 } > adm request ...



Internals

Cursors Tailable side, to support t {{queryOptions ir results when ...



International Docs

Most documenta for volunteers to interested in con roger at 10gen d



International Docum



Internationalized Str

MongoDB supp(Specifically, BS programming lar UTF8 when seri



Interop Demo (Prodi

Interop 2009 Mo <http://github.com>



Introduction

MongoDB wasn't experiences built start from scratch



Introduction - How M



iostat

Linux, use the io database. We ge seconds with ios

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 ...

K



Kernel class rules

Design guideline comment in the I state what the cl

-  [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 ...
-  [Java Language Center](#)
Java Driver Basics Download the Java Driver <http://github.com/mongodb/mongojavadriv...> Tutorial [http://www.mongodb.org/display/DOCS/JavaTutorial API Documentation](http://www.mongodb.org/display/DOCS/JavaTutorial+API+Documentation) <http://api.mongodb.org/java/index.html> Release Notes <https://github.com/mongodb/mongojavadriv...>
-  [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 and higher supports writeahead journaling of operations to facilitate fast crash recovery and durability in the storage engine. Enabling To enable, use the `mongod {{journal}}` command line option. # `mongod journal` Before 1.8 the command line option was dur ...
-  [Journaling Administration Notes](#)
Journal Files 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, a new file is created ...
-  [Joyent](#)
prebuilt DOCS:Downloads 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/lsinfo kv` ...
-  [JS Benchmarking Harness](#)
CODE: `db.foo.drop(); db.foo.insert( { op : "findOne" , ns : "test.foo" , query : } for ( x = 1; x<=128; x=2){ res = benchRun( ) print( "threads: " x "\t queries/sec: " res.query ) }` More info: <http://github.com/mongodb/mongo/commit/3db3cb13dc1c522db8b59745d6c74b0967f1611c>
-  [JVM Languages](#)
moved to Java Language Center

L

-  [Language Support](#)
-  [Last Error Commands](#)
Since MongoDB doesn't wait for a response by default when writing to the database, a couple commands exist for ensuring that these operations have succeeded. These commands can be invoked automatically with many of the drivers when saving and updating in "safe" mode. But what's really happening ...
-  [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](#)
you are using a vanilla MongoDB server from either source or binary packages you have NO obligations. You can ignore the rest of this page. MongoDB Database Free Software Foundation's GNU AGPL v3.0 <http://www.fsf.org/licensing/licenses/agpl3.0> ...
-  [List of Database Commands](#)
`iframe src ="http://api.mongodb.org/internal/current/commands.html" width="100%" height="1000px" frameborder="0"> <a href="http://api.mongodb.org/internal/current/commands.html">List of MongoDB Commands</a> </iframe>` See the Commands page for details on how to invoke a command ...
-  [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 `{{v}}` more verbose output. use more v's (such as `{{vvvvv}}` ...

comment is just i

-  [Kernel code style](#)
comments We fc <http://googlestyle> placement of cor (public or private) has X as a goal i
-  [Kernel concurrency](#)
All concurrency (several helper lit you think there is ...
-  [Kernel exception arc](#)
several different `{{assert}}` should `{{massert}}` is an error `{{wassert}}`
-  [Kernel Logging](#)
Basic Rules cout levels of verbosit replica set node
-  [Kernel string manipu](#)
string manipulati MongoUtils has i not database spe boost headers, b

M

-  [Manual](#)
MongoDB manu: JavaScript for us There is a table l each of the drive
-  [MapReduce](#)
Map/reduce in M operations. It is s coming from a cc where you woul
-  [Master Slave](#)
Configuration an database in a m: database, one in examples explici
-  [Memory Manageme](#)
Overall guideline Prefer scopedptr use autoptr's anc output of new/m:
-  [min and max Query](#)
`min()` and `{{ma: constrain query r keys specified. T in conjunction. T`

-  [mongo - The Interac](#)
Introduction The
MongoDB intera
issue commands
extended Spider
<https://developer>
...
-  [Mongo Administratic](#)
-  [Mongo Concepts an](#)
-  [Mongo Database Ac](#)
-  [Mongo Developers'](#)
-  [Mongo Documentati](#)
page provides in
Confluence. It cc
markup for speci
Notes on Writing
-  [Mongo Driver Requi](#)
highlevel list of fe
group those feat
probably used m
way to learn abo
-  [Mongo Extended JS](#)
Mongo's REST ii
Special represer
JSON mappings
The REST interfi
-  [Mongo Metadata](#)
dbname>.system
system informati
lists all namespa
/ index metadata
...
-  [Mongo Query Lang](#)
Queries in Mong
object as the eq
select from user:
server ...
-  [Mongo Usage Basic](#)
-  [Mongo Wire Protoc](#)
Introduction The
requestresponse
through a regula
configurable and
TCP/IP ...
-  [Mongo-Based Applic](#)
Please list applic
for your applicati
meghan@10gen
using MongoDB
-  [MongoDB - A Devel](#)
-  [MongoDB Commerc](#)
Note: if you prov
be listed here, ju
Information !10g
commercial Mon
-  [MongoDB Data Mod](#)
tutorial discusse
MongoDB.&nbs
goal is to provide
MongoDB.&nbs
...
-  [MongoDB kernel cor](#)
Coding conventi
here, default to c
<http://googlestyle>
-  [MongoDB Language](#)

-  [MongoDB, CouchDB pending... Couch \(JSON http://www http://blog.mongodb string,number,boolean object ...](#)
-  [mongosniff](#)
Unix releases of MongoDB what t situations. The t Usage: mongosr
-  [mongostat](#)
Use the mongos instance. !mongos insert # of inserts second update #
-  [Monitoring](#)
-  [Monitoring and Diag](#)
Admin UIs Query Http Console Th http://localhost:2 Shell Diagnostic
-  [Moving Chunks](#)
any given time, a machinery route: application need balancer http://w
-  [Moving or Replacing](#)
shell you can use Then use rs.add for more informa how ...
-  [Multikeys](#)
MongoDB provid arrays of an object article tagged with db.articles.find()
-  [Munin configuration](#)
Overview Munin of you running sy the MongoDB plugin agent and plugin

N

-  [node.js](#)
Node.js is used to write eventdriven, scalable network programs in serverside JavaScript. It is similar in purpose to Twisted, EventMachine, etc. It runs on Google's V8. Web Frameworks ExpressJS http://expressjs.com Mature web framework with MongoDB session support. 3rd Party ...
-  [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 ...

O

-  [Object IDs](#)
Documents in MongoDB identifies them. The field as its first attribute some capped ...
-  [Old Pages](#)
-  [Older Downloads](#)
-  [Online API Documentation](#)
MongoDB API at Java Driver API | Documentation | Documentation |
-  [Optimization](#)
Additional Article Small Objects Optimization techniques for optimization Suppose our tas
-  [Optimizing MongoDB](#)
-  [Optimizing Object IDs](#)
id field in MongoDB lists some recommendations field. id's can be

-  [Optimizing Storage](#) MongoDB records DOCS:BSON documents (significant, but fields) it would not
-  [OR operations in queries](#) Query objects in 1.5.3 MongoDB there are ways to value in ..." expressions
-  [Out of Memory OOM](#) Linux out of memory event you will see 04:33:23 hostm1
-  [Overview - The MongoDB](#) Starting the Shell distribution. To set type ./bin/mongo {{PATH}} so you
-  [Overview - Writing C](#) section contains protocols of MongoDB Documents of BSON binary documents software ...

P

-  [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 ...
-  [Parsing Stack Traces](#)
addr2line addr2line -e mongod -fC <offset> 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 ...
-  [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 !featuresPerformance.png align=right! Databases are specializing the "one size fits all" approach no longer applies. By reducing transactional semantics the db provides, one can still solve an interesting set of problems where performance is very ...
-  [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. CakePHP MongoDB datasource <https://github.com/ichikaway/cakephpmongodb> for CakePHP. There's also an introductory blog post <http://markstory.com/posts> ...
-  [Product Comparisons](#)
-  [Production Deployments](#)
you're using MongoDB in production, we'd love to list you here! Please complete this web form <https://10gen.wufoo.com/forms/productiondeploymentdetails/> or email meghan@10gen.com and we will add you. <DIV mcestyle="text-align:center;margin:0" style ...
-  [Production Notes](#)
Backups Import Export Tools DOCS:Backups Doc Page Recommended Unix System Settings Turn off atime Set file descriptor limit to 4k (see etc/limits and ulimit) Do not use large VM pages with Linux ...
-  [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](#)

Q

-  [Queries and Cursors](#)
Queries to MongoDB The exact way to queries from the {{mongo}} process
-  [Query Optimizer](#)
MongoDB query client. The supports ad hoc interesting approach
-  [Querying](#)
One of MongoDB Systems that support data; users can find
-  [Quickstart](#)
Quickstart OS X go to <http://try.mongodb.com> Tutorial
-  [Quickstart OS X](#)
Install MongoDB manager or the <http://mxcl.github.io> install mongod
-  [Quickstart Unix](#)
Download If you start, or gives a full Downloads page Ubuntu and Deb
-  [Quickstart Windows](#)
Download The prebuilt binaries. The "Production" and extract the 6

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 good suggestions. One data center If you have just one data center, then the most economical setup is a threenode replica ...
-  [Recommended Production Architectures](#)
-  [Reconfiguring a replica set when members are down](#)
One may modify a set when some members are down as long as a majority is established. In that case, simply send the reconfig command to the current primary. DOCS:Reconfiguring when Members are Up If there is no primary (and this condition is not transient), no majority is available. Reconfiguring ...
-  [Reconfiguring when Members are Up](#)
Version 1.7.1 Use the rs.reconfig() helper in the shell (version 1.7.1). (Run "rs.reconfig" in the shell with no parenthesis to see what it does.) \$ mongo > // shell v1.7.x: > // example : give 1st set member 2 votes > cfg = rs.conf() > cfg.members0.votes = 2 > rs.reconfig ...
-  [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 ...
-  [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. To set it up: # Create a key file that can be copied to each ...
-  [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](#)
How long does failover take? Failover thresholds are configurable. With the defaults, it may take 2030 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 ...
-  [Replica Set Internals](#)
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 setname/seedhostnamelist on the command line. seedhostnamelist is a (partial) list of some members ...
-  [Replica Set Tutorial](#)
tutorial will guide you through the basic configuration of a replica set on a single machine. If you're attempting to deploy replica sets in production, be sure to read the comprehensive replica set documentation Replica Sets. Also, do keep in mind that replica sets ...

S

-  [Schema Design](#)
Introduction With designing a relat you will want one not want ...
-  [scons](#)
Use scons [Scripting the shell](#)
Introduction The scripted using JS also use {{eval}}
-  [Searching and Retri](#)
-  [Security and Authen](#)
MongoDB includ deployments anc with sharding, sc in a secure envir
-  [Server-side Code E](#)
Mongo supports Clauses and Fur specification for string containing
-  [Server-Side Process](#)
-  [serverStatus](#)
serverStatus cor instance. Run > ; Explanation host The version num
-  [Shard Ownership](#)
shard ownership draft/thoughts wi information is in which server owi
-  [Sharding](#)
MongoDB scales Automatic balan new machines S Automatic failove
-  [Sharding Administra](#)
Here we present sharding cluster. configuration DC we're speaking t
-  [Sharding and Failov](#)
properlyconfigu This document d within a shard cli a {{mongos}} rou
-  [Sharding Config Scf](#)
Sharding config version This is a db.getCollection() options (chunkSi shards ...
-  [Sharding Design](#)
concepts config servers and whe replica pair. data or not chunk \ a i



Replica Sets

Replica sets are basically master/slave replication DOCS:Replication, adding automatic failover and automatic recovery of member nodes. The set automatically elects a primary (master) if there is no primary currently available. Drivers can automatically detect when a replica set ...



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

v1.6 Authentication mode not supported. JIRA <http://jira.mongodb.org/browse/SERVER1469> Limits on config changes to sets at first. Especially when a lot of set members are down. Map/reduce writes new collections to the server. Because of this, for now it may only ...



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. With a single active master at any point in time, strong consistency semantics are available ...



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 ...`



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> ...



Sharding FAQ

Should I start out We suggest start data set will not be easy and seamless



Sharding Internals

section includes sharding. Note: some interesting we ...



Sharding Introduction

MongoDB supports scaling across multiple single database automatically managed ...



Sharding Limits

Differences from security mode, with Shard keys are indexed and inserts must



Sharding Use Cases

What specific use techniques) that (e.g., youtube) (for related video:



Slide Gallery

present, the width apologize for the <http://www.mongodb.org> full listing of video to MongoDB Use



Slides and Video

Table of Content a variety of talks, and present find slides and vi



Smoke Tests

smoke.py smoke running tests, it's it down again. You same ...



Sorting and Natural

Natural order" is collection. When objects in forward particularly useful



Source Code

Source for MongoDB hosted at Github <http://github.com> <http://github.com>



Spec, Notes and Su

Assume that the be up to 4MB. The object. We record



Splitting Chunks

Normally, splitting happen as a side may be other cause may be moved in



SQL to MongoDB Map

page not done. For Mongo Program `{{mongo}}` \ MongoDB objects. T



Starting and Stopping

MongoDB is run

Command Line Examples
examples assume
and the MongoDB

-  [Storing Data](#)
-  [Storing Files](#)
-  [Structuring Data for](#)

T

-  [Tailable Cursors](#)
Tailable cursors are only allowed on capped collections and can only return objects in natural order
<http://www.mongodb.org/display/DOCS/SortingandNaturalOrder>. If the field you wish to "tail" is indexed, simply
requerying for `{ field : \ }` is already quite efficient. Tailable ...
-  [Timestamp Data Type](#)
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 32 bits are a time
value (seconds since the UTC epoch). The second ...
-  [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 `lsf` (some variations
shown below): `lsf | grep ...`
-  [Tools and Libraries](#)
Talend Adapters <https://github.com/adrienmogenet>
-  [TreeNavigation](#)
Follow @mongodb <http://www.twitter.com/mongodb>
-  [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` See ...
-  [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](#)
Getting the Database First, run through the Quickstart guide for your platform to get up and running. Getting A
Database Connection Let's now try manipulating the database with the database shell `DOCS:mongo` The
Interactive Shell . (We could perform similar ...
-  [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 ...

U

-  [Ubuntu and Debian](#)
Please read the
`apt-gettable` pack
Debian or Ubuntu
`mongodb10gen`,
-  [UI](#)
Spec/requiremer
clone? collection
`explain()` output :
master ...
-  [Updates](#)
-  [Updating](#)
MongoDB supp
for replacing an
matching criteria
you should use .
-  [Updating Data in Mc](#)
Updating a Docu
previous section
a collection. We
collection. Contr
-  [Upgrading from a N](#)
`{mongod}` proces
that process or d
the Sharding Intr
<http://www.mong>
yourself ...
-  [Upgrading to 1.8.0](#)
First, upgrade yc
can replicate fro
masters. Thus, tl
-  [Upgrading to Replica](#)
Upgrading From
upgrading to rep
replica set with a
we're ...
-  [Use Case - Session](#)
MongoDB is a ge
model is to have
a browser cookie
make updates fa
-  [Use Cases](#)
See also the Pro
a discussion of h
SourceForge, et
site. MongoDB is
-  [User Feedback](#)
I just have to get
`\muckster`, `#mon`
about what is the
`\kunthar@gmail`.
-  [Using a Large Numt](#)
technique one c
collections to sto
this, certain repe
index on that key
-  [Using Multikeys to S](#)
One way to work
use the multikey

	objects. F ... , ... } ; > db.f
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. From the {{mongo}} shell: > db.foo.validate() From a driver one might invoke the driver's equivalent ...  Verifying Propagation of Writes with getLastError v1.5. A client can block until a write operation has been replicated to N servers. Use the getlasterror DOCS:getLastError command with a new parameter {{w}}: db.runCommand() If {{w}} is not set, or equals 1, the command returns immediately, implying the 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 Current Operation(s) in Progress > db.currentOp(); > // same as: db.\$cmd.sys.inprog.findOne() { inprog: { "opid" : 18 , "op" : "query" , "ns" : "mydb.votes" , "query" : " " , "inLock" : 1 } } Fields: opid an incrementing operation number. Use with killOp(). op the operation type ...  Visit the 10gen Offices Bay Area 10gen's West Coast office is located on the Peninsula in Redwood Shores. 100 Marine Parkway, Suite 175 Redwood City, CA 94065 \\ <iframe width="475" height="375" frameborder="0" scrolling="no" marginheight="0" marginwidth ...	W  What is the Compar MongoDB allows differ in type.&nt is utilized as to w arbitrary but well  When to use GridFS page is under co handle large nun User uploaded fi  Why are my datafile:  Why so many "Conr  Windows Windows Quick I See the Quickstz and run the data Service page. Tf  Windows Service windows mongoi service. Service mongod install r option pass the f  Working with Monge  Writing Drivers and ` See Also DOCS: parameter DOCs  Writing Tests We have three g C. These are in t ./test help jstests
X	Y
Z	!@#\$