<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replica Set Versions and Compatibility</td>
<td>369</td>
</tr>
<tr>
<td>Replica Set Authentication</td>
<td>370</td>
</tr>
<tr>
<td>Upgrading to Replica Sets</td>
<td>370</td>
</tr>
<tr>
<td>Replica Set Admin UI</td>
<td>372</td>
</tr>
<tr>
<td>Replica Set Commands</td>
<td>373</td>
</tr>
<tr>
<td>Connecting to Replica Sets from Clients</td>
<td>378</td>
</tr>
<tr>
<td>Replica Sets Troubleshooting</td>
<td>379</td>
</tr>
<tr>
<td>Replica Sets Limits</td>
<td>381</td>
</tr>
<tr>
<td>Replica Set Internals</td>
<td>382</td>
</tr>
<tr>
<td>Master Slave</td>
<td>385</td>
</tr>
<tr>
<td>Halted Replication</td>
<td>389</td>
</tr>
<tr>
<td>Replica Pairs</td>
<td>391</td>
</tr>
<tr>
<td>Sharding</td>
<td>393</td>
</tr>
<tr>
<td>Bulk Inserts</td>
<td>393</td>
</tr>
<tr>
<td>Changing Config Servers</td>
<td>394</td>
</tr>
<tr>
<td>Simple Initial Sharding Architectures</td>
<td>395</td>
</tr>
<tr>
<td>Sharding Introduction</td>
<td>399</td>
</tr>
<tr>
<td>Configuring Sharding</td>
<td>403</td>
</tr>
<tr>
<td>A Sample Configuration Session</td>
<td>406</td>
</tr>
<tr>
<td>Changing a Shard Key</td>
<td>408</td>
</tr>
<tr>
<td>Choosing a Shard Key</td>
<td>408</td>
</tr>
<tr>
<td>Tag Aware Sharding</td>
<td>411</td>
</tr>
<tr>
<td>Sharding Administration</td>
<td>412</td>
</tr>
<tr>
<td>Backing Up Sharded Cluster</td>
<td>416</td>
</tr>
<tr>
<td>Sharding Limits</td>
<td>419</td>
</tr>
<tr>
<td>Sharding Internals</td>
<td>421</td>
</tr>
<tr>
<td>Sharding Config Schema</td>
<td>421</td>
</tr>
<tr>
<td>Sharding Design</td>
<td>424</td>
</tr>
<tr>
<td>Sharding Use Cases</td>
<td>425</td>
</tr>
<tr>
<td>Shard Ownership</td>
<td>425</td>
</tr>
<tr>
<td>Hosting Center</td>
<td>426</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>427</td>
</tr>
<tr>
<td>AWS Marketplace</td>
<td>430</td>
</tr>
<tr>
<td>Automating Deployment with CloudFormation</td>
<td>431</td>
</tr>
<tr>
<td>Amazon EC2 Quickstart</td>
<td>438</td>
</tr>
<tr>
<td>dotCloud</td>
<td>448</td>
</tr>
<tr>
<td>Joyent</td>
<td>449</td>
</tr>
<tr>
<td>Rackspace Cloud</td>
<td>449</td>
</tr>
<tr>
<td>RedHat OpenShift</td>
<td>455</td>
</tr>
<tr>
<td>OpenShift Quickstart</td>
<td>455</td>
</tr>
<tr>
<td>VMware CloudFoundry</td>
<td>460</td>
</tr>
<tr>
<td>Monitoring and Diagnostics</td>
<td>460</td>
</tr>
<tr>
<td>Checking Server Memory Usage</td>
<td>462</td>
</tr>
<tr>
<td>collStats Command</td>
<td>466</td>
</tr>
<tr>
<td>Database Profiler</td>
<td>467</td>
</tr>
<tr>
<td>mongoperf</td>
<td>470</td>
</tr>
<tr>
<td>Munin configuration examples</td>
<td>471</td>
</tr>
<tr>
<td>Http Interface</td>
<td>475</td>
</tr>
<tr>
<td>mongosniff</td>
<td>478</td>
</tr>
<tr>
<td>Wireshark Support for MongoDB Protocol</td>
<td>478</td>
</tr>
<tr>
<td>Backup</td>
<td>480</td>
</tr>
<tr>
<td>EC2 Backup &amp; Restore</td>
<td>482</td>
</tr>
<tr>
<td>How to do Snapshotted Queries in the Mongo Database</td>
<td>487</td>
</tr>
<tr>
<td>Import Export Tools</td>
<td>488</td>
</tr>
<tr>
<td>mongexport</td>
<td>494</td>
</tr>
<tr>
<td>Durability and Repair</td>
<td>495</td>
</tr>
<tr>
<td>Security and Authentication</td>
<td>497</td>
</tr>
<tr>
<td>Admin UIs</td>
<td>501</td>
</tr>
<tr>
<td>Starting and Stopping Mongo</td>
<td>508</td>
</tr>
<tr>
<td>getCmdLineOpts command</td>
<td>510</td>
</tr>
<tr>
<td>Logging</td>
<td>511</td>
</tr>
<tr>
<td>Command Line Parameters</td>
<td>513</td>
</tr>
<tr>
<td>--directory/odb</td>
<td>514</td>
</tr>
<tr>
<td>--quiet</td>
<td>515</td>
</tr>
<tr>
<td>--syncdelay</td>
<td>516</td>
</tr>
<tr>
<td>File Based Configuration</td>
<td>516</td>
</tr>
<tr>
<td>GridFS Tools</td>
<td>518</td>
</tr>
<tr>
<td>DBA Operations from the Shell</td>
<td>519</td>
</tr>
<tr>
<td>Windows</td>
<td>521</td>
</tr>
<tr>
<td>MongoDB on Azure - Overview</td>
<td>522</td>
</tr>
<tr>
<td>Deploying MongoDB Replica Sets to Linux on Azure</td>
<td>523</td>
</tr>
<tr>
<td>MongoDB Installer for Windows Azure</td>
<td>523</td>
</tr>
<tr>
<td>MongoDB on Azure</td>
<td>523</td>
</tr>
<tr>
<td>MongoDB on Azure VM</td>
<td>523</td>
</tr>
<tr>
<td>MongoDB on Azure Worker Roles</td>
<td>538</td>
</tr>
</tbody>
</table>
Troubleshooting .......................................................... 545
Excessive Disk Space ........................................ 546
The Linux Out of Memory OOM Killer ................ 548
Too Many Open Files ....................................... 549
Production Deployments ........................................ 550
Contributors .......................................................... 616
JS Benchmarking Harness ........................................ 616
MongoDB kernel code development rules .............. 617
Kernel class rules ............................................. 618
Kernel code style .............................................. 619
Kernel concurrency rules ...................................... 623
Kernel exception architecture .............................. 623
Kernel logging ................................................ 624
Kernel string manipulation .................................. 625
Memory management .......................................... 625
Writing test tests ............................................. 626
Project Ideas ...................................................... 626
Roadmap ............................................................ 628
UI ................................................................ 628
Source Code ......................................................... 628
Building ............................................................... 628
Building Boost .................................................. 629
Building for FreeBSD ......................................... 630
Building for Linux ............................................. 630
Building for OS X .............................................. 632
Building for Solaris ............................................. 637
Building for Windows ........................................... 637
  Boost 1.41.0 Visual Studio 2010 Binary .............. 637
  Boost and Windows .......................................... 637
  Building the Mongo Shell on Windows ............... 638
  Building with Visual Studio 2008 ....................... 639
  Building with Visual Studio 2010 ....................... 640
Building Spider Monkey ....................................... 642
Building with V8 ................................................ 644
  scons .......................................................... 645
Database Internals ................................................ 645
Caching ............................................................ 646
Durability Internals ............................................. 646
Parsing Stack Traces ........................................... 647
Cursors ............................................................. 648
Error Codes ....................................................... 648
Internal Commands ............................................ 649
Replication Internals .......................................... 649
Smoke Tests ....................................................... 650
Testing Internals ................................................ 652
Contributing to the Documentation ..................... 653
Emacs tips for MongoDB work ......................... 653
Mongo Documentation Style Guide ..................... 653
Community ........................................................ 655
MongoDB Masters .............................................. 657
Technical Support ............................................... 663
MongoDB Commercial Services Providers ........... 664
  Visit the 10gen Offices ................................... 667
User Feedback .................................................... 667
Job Board .......................................................... 669
About ................................................................. 669
Philosophy .......................................................... 669
Use Cases .......................................................... 670
Business Intelligence .......................................... 672
  Hadoop Quick Start ....................................... 673
  Hadoop Scenarios ......................................... 677
How MongoDB is Used in Media and Publishing .. 679
Use Case - Session Objects .................................. 681
MongoDB-Based Applications .......................... 682
Events ............................................................... 683
  Video & Slides from Recent Events and Presentations ... 689
  Your Go-to Resource for Running a MongoDB User Group .... 691
  Presentation Tips for MongoDB User Groups .......... 693
MongoDB Archives and Articles ......................... 695
  Benchmarks ................................................... 696
  FAQ .............................................................. 696
  Misc ............................................................. 696
  Demo App in Python ...................................... 696
  MongoDB, CouchDB, MySQL Compare Grid ............ 697
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing Mongo DB and Couch DB</td>
<td>697</td>
</tr>
<tr>
<td>Licensing</td>
<td>699</td>
</tr>
<tr>
<td>International Docs</td>
<td>699</td>
</tr>
<tr>
<td>Books</td>
<td>699</td>
</tr>
<tr>
<td>Slides and Video</td>
<td>701</td>
</tr>
<tr>
<td>Alerts</td>
<td>701</td>
</tr>
</tbody>
</table>
Slide Gallery

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

.bookmarks

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

The 15 most recent bookmarks in MongoDB
There are no bookmarks to display.

1.1 Development Cycle

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

Creating and Deleting Indexes

Redirection Notice
This page should redirect to Indexes.

C Sharp Language Center

Redirection Notice
This page should redirect to CSharp Language Center.

Diagnostic Tools
Django and MongoDB

Getting Started

International Documentation

Monitoring

Older Downloads

PyMongo and mod_wsgi
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.
- \texttt{db} shell utility
- (Very) basic security
- $\texttt{or}$
- 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
Documentation

Dot Notation

Getting the Software

Language Support

Mongo Administration Guide
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
Building indexes with replica sets

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/administration/indexes/#index-building-replica-sets.

Splitting Shard Chunks

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/administration/sharding/#splitting-chunks.

fsync Command

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/reference/command/fsync/.

Verifying Propagation of Writes with getLastError

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/applications/replication/#replica-set-write-concern.

Replica Set FAQ

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/faq/replica-sets/.

createCollection Command

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/reference/method/db.createCollection/.

renameCollection Command

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/reference/command/renameCollection.
cloneCollection Command

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/reference/command/cloneCollection/.

Design Overview

Redirection Notice
This page should redirect to Developer Zone.

Quickstart OS X

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/.

Mongo Concepts and Terminology

See the Manual page for information on the following:

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

Replication Terms

- Replication. Duplicating data on multiple servers for HA, safety, disaster recovery, and a bit of scaling. Sharding and replication are used together.
- Member. A member (server) in a replica set.
- Primary. The replica set member that is currently “master” and receives all writes. Reads from the primary are “immediately consistent”.
- Secondary. A replica set member that is currently not master and is applying operations it receives from the current primary.
- Arbiter. A replica set member that votes in elections of primary, but that has no data. Arbiters are very lightweight and can be used to break ties in elections. Add an arbiter if a set has an even number of members.
- Oplog. High level log of operations used by replication.

Sharding Terms

- Sharding. The partitioning / distribution of data among machines in a cluster. Each shard has different data. Sharding is the mechanism in MongoDB for building very large clusters. Note: we recommend you begin using MongoDB without sharding. It is easy to transition over to sharding later.
- Shard. Shard Key. See the sharding intro page.
- Chunk. A subset of documents in a collection that fit in a certain shard key range. See the sharding intro page.
- Config server. In a sharded environment, config servers store the metadata of the cluster. Each config server has a mongod process which stores metadata. Typically there are three config servers which have replicas of the same metadata (for data safety and HA). The config server mongod process is fairly lightweight and can be ran on machines performing other work. The config server data is very important be sure to include them in your backups of the cluster.
- mongod, mongos, mongo. MongoDB processes.

Other Concepts
Durability / Journaling. Write-ahead logging for crash safety.
Object IDs. Mongo documents include an _id field in each document.

See Also

- Architecture and Components
- SQL to Mongo Mapping Chart

Database References

**Redirection Notice**
This page should redirect to http://docs.mongodb.org/manual/applications/database-references/.

Indexes in Mongo

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

Optimizing Mongo Performance

**Redirection Notice**
This page should redirect to Optimization.

Components

**Redirection Notice**
This page should redirect to http://docs.mongodb.org/manual/reference/#manual-pages.

Locking in Mongo

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

Server-Side Processing

**Redirection Notice**
This page should redirect to Server-side Code Execution.
Why so many "Connection Accepted" messages logged?

List of Database Commands

Doc Index

movePrimary Command

two-phase commit

Updates

Recommended Configurations
getLog Command

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/reference/commands/#getLog.

Mongo Usage Basics

Redirection Notice
This page should redirect to Tutorial.

Windows Service

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/#mongodb-as-a-windows-service.

slaveOk

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/applications/replication/#read-preference.

What is the Compare Order for BSON Types

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/faq/developers/#what-is-the-compare-order-for-bson-types.

Moving or Replacing a Member

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/administration/replica-sets/#replica-set-admin-procedure-replace-member.

serverStatus Command
Replica Sets - Priority

Compact Command

Upgrading from a Non-Sharded System

Replica Set Configuration

Why are my datafiles so large?

Replica Set Tutorial

Quickstart Windows
SSL

flushRouterConfig command

removeshard command

Storing Data

Searching and Retrieving

Locking

Sharding and Failover
Adding a New Set Member

Reconfiguring when Members are Up

Adding an Arbiter

Replication Oplog Length

Replica Set Design Concepts

Document-Oriented Datastore

Architecture and Components
Overview

The $limit pipeline operator limits the number of JSON documents that passes through it.

Specification

$limit takes a single numeric value as a parameter. Once that many documents have passed through the pipeline operator, no more will.

Here's a simple example:

```javascript
db.article.aggregate(  
    { $limit : 5 }  
);  
```

This pipeline will only pass through the first 5 documents that are seen. $limit has no effect on the content of the documents that are passed through – all fields present will be passed through as they are.

Notes
Overview

$match filters documents. Documents which do not match the specified predicate are filtered out and do not progress further along the aggregation pipeline. Documents which do match are passed along unchanged.

Specification

$match predicate syntax is always exactly the same as that for queries; See Querying and Advanced Queries.

Within a pipeline, the $match operator appears thusly:

```javascript
db.article.aggregate(
   { $match : <match-predicate> }
);
```

Here is an example with a simple field equality test:

```javascript
db.aggregate(
   { $match : { author : "dave" } }
);```

On its own like this, this is the equivalent of db.article.find({ author : "dave" }).

Here is another example, this time with a range test:

```javascript
db.article.aggregate(
   { $match : { score : { $gt : 50, $lte : 90 } } }
);```

Notes

$match should be placed as early in the aggregation pipeline as possible. This minimizes the number of documents after it, thereby minimizing later processing. Placing a $match at the very beginning of a pipeline will enable it to take advantage of indexes in exactly the same way as a regular query (find()/findOne()).

Aggregation Framework - $project

Aggregation Framework - $skip

Aggregation Framework - $sort
Aggregation Framework - $unwind

Aggregation Framework - Expression Reference

SQL to Aggregation Framework Mapping Chart

<table>
<thead>
<tr>
<th>SQL term</th>
<th>MongoDB agg framework term</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>$match</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>$group</td>
</tr>
<tr>
<td>SELECT</td>
<td>$project</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>$sort</td>
</tr>
</tbody>
</table>

Note: you can try the agg framework by testing with the v2.1.0 build. Note that 2.1.x is an unstable branch and not for production. Once ready it will be promoted to v2.2.

MongoDB has a few means of performing aggregation and data processing. In v2.2+ the main means are the aggregation framework and the map/reduce command. In general, if your operation can be expressed in the aggregation framework, use that as it is usually easier than using map/reduce and also performs better.

The table below shows examples of a SQL aggregation statement and the analogous MongoDB agg framework statement. The examples are invocations from the mongo shell, however usage from other programming languages is very similar.

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>Mongo Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT COUNT(*) FROM users</td>
<td>db.users.aggregate({</td>
</tr>
<tr>
<td></td>
<td>{ $group: { _id: null, count: { $sum: 1 } } } })</td>
</tr>
</tbody>
</table>
**SELECT SUM(price) FROM orders**

```javascript
db.orders.aggregate([
  { $group: { _id: null,
    total: { $sum: "$price" } } }
])
```

**SELECT cust_id, SUM(price) FROM orders GROUP BY cust_id**

```javascript
db.orders.aggregate([{
  $group: { _id: "$cust_id",
    total: { $sum: "$price" } } }
])
```

**SELECT cust_id, SUM(price) FROM orders WHERE active=true GROUP BY cust_id**

```javascript
db.orders.aggregate([{
  $match: { active: true }
}, { $group: { _id: "$cust_id",
    total: { $sum: "$price" } } }
])
```

---

**Quickstart Unix**

⚠️ **Redirection Notice**
This page should redirect to [http://docs.mongodb.org/manual/tutorial/install-mongodb-on-linux/](http://docs.mongodb.org/manual/tutorial/install-mongodb-on-linux/).

---

**getLastError_old**

⚠️ **Redirection Notice**
This page should redirect to [getLastError Command](http://docs.mongodb.org/manual/reference/command/getLastError/).

---

**Conference Information**

This is the home page for all information for speakers and sponsors participating in MongoDB Conferences.

- CFP Submission Tips
- Conference Management Tools
- MongoDB Bangalore 2012 Speaker Info
- MongoDB Beijing 2012 Speaker Info
- MongoDB Boston 2012 Speaker Info
- MongoDB Chicago 2012 Speaker Info
- MongoDB Melbourne 2012 Speaker Info
- MongoDB Munich 2012 Speaker Info
- MongoDB Pune 2012 Speaker Info
- MongoDB Seattle 2012 Speaker Info
- MongoDB Seattle 2012 Sponsor Info
CFP Submission Tips

- Where to Submit
- Basic Guidelines
- Popular Topics
- Benefits of Submitting to our CFP
- Examples of Popular 2012 presentations

Where to Submit

10gen Talk Proposal

Basic Guidelines

- The talk should be 30 minutes in length with 10 minutes for Q&A; your talk will be 40 minutes total
- Attendees are mostly developers, so gear your presentation towards a developer heavy audience
- Your talk should contain a 'message' or a 'theme' for attendees to take away
- Have fun with your topic! Each talk should be unique to the speaker giving it
- Have slides ready for review upon submission
- Feel free to email speakerinfo@10gen.com with questions.

Popular Topics

- Analytics
- Scaling
- Unique use cases

FAQ

Q: Can I submit papers in my native language?
A: Yes, you can. We are interesting in having native language presentations for conferences in our non-English speaking cities in order to make talks more accessible to our attendees

Q: My company is interested in sponsoring the conference. Can we sign up as sponsors and have a speaking slot?
A: Yes. We do not include speakings slots with sponsorship, but we accept proposals from everyone and are happy to have speakers, sponsors and attendees attend from one company. You can email sponsor@10gen.com for more information.

Q: I have not heard back yet. How do I know if my talk has been accepted?
A: If you have not heard back, we are still working on the content for the conference you submitted for. As soon as we make a definitive decision, you will hear from us. Feel free to contact speakerinfo@10gen.com if you have questions.

Benefits of Submitting to our CFP

- Every speaker who submits gets a free ticket to the conference
- Speaking at a MongoDB Conference is a great way to promote your product, your company, and yourself!
- MongoDB Conferences are highly valued as recruiting opportunities
- Being active in the MongoDB Community enables you to grow your presence in the open source community

Examples of Popular 2012 presentations

- Get Your Spatial On with MongoDB in the Cloud
- MongoDB Schema Design Insights and Tradeoffs
- Mongodb Versatility Scaling the MapMyFitness Platform
- Mapping Flatland Using MongoDB for an MMO Crossword Game

Conference Management Tools
Tools to Effectively Plan and Execute your Conference

- **Planning**
  - Planning
  - Execution
  - Manage Feedback

**Planning**

There are a few keys to effective planning materials. They should be easy to use, non-exclusive, and they should allow you to clone projects. Below are some tools we use for our conferences.

- **Mavenlink**
  - use Mavenlink as an overall project management system. Syncs with Google Apps.
- **Dropbox**
  - use Dropbox as a collaborative document store for your team.
- **Jira**
  - use Jira as a project management system for contracts or colleagues who only own one item for your conferences, such as design and email campaigns.
  - use google apps to create calendar invites for your speakers and staff, eliminating document loss and confusion.
- **Fiesta.cc**
  - create mailing lists easily for your speakers, sponsors and staff.
- **Wiki Pages**
  - Create wiki pages for your speakers, sponsors and staff to communicate key information in a living document.
- **Eventbrite**
  - use eventbrite for registration and ticket sales. Eventbrite is easy to use with almost any CMS and has standard promotional templates for groups without the need for customizable options.
- **Twitter**
  - use twitter to promote your event and speakers, and on day of to tweet content in real time.
- **Qrious**
  - use qrious to help your attendees network with each other, to autogenerate nametags with unique QR (it syncs with eventbrite), and to provide your sponsors with leads.
  - use guidebook for your mobile agenda and conference information needs.

**Execution**

One drawback to planning conferences in multiple locations is that you often have no control over your environment, and limited knowledge of your resources until the day of the conference. Here are some ways to help keep your conference environment consistent:

- **Wifi Needs:**
  - Order wired lines
  - Presenters can use them and you can use them to connect a custom wireless network.
  - **Meraki**
    - cloud managed wireless solutions put control of conference wifi in your hands.
    - **Tips:**
      - Find out the conferences up bandwidth
      - Kill the conference’s house wifi so you are not fighting with them
      - Get informed before you buy - Meraki has many different solutions for many different conferences.
  - **Walkie Talkies**
    - These help your staff communicate immediately. Simple ones work for most environments. #protip: get on the same channel as your A/V or catering crew, and you will know immediately when there is a problem on your end or the venues.
  - **“Go” Bag:**
    - Fill your "go" bag with these goodies and refresh immediately after a conference, and you will never be scrambling for sharpyes again!
      - scissors, tape, med kit, twine, extra stickie labels, pants, pens, sharpyes, batteries, usb keys, extra charges and extension cords, clickers and mini-display -> VGA adaptors
  - **Film Equipment** (try B&H for your needs!)
    - Record your speakers to add value to your website, or just to provide internal feedback, and avoid the cost of a film crew and the uncertainty that comes from having to use different film crews in different cities!
    - We like: Sony HXR-MC50U Ultra Compact Pro AVCHD Camcorder, Sennheiser EW112-p G3 Camera Mount Wireless Microphone System with ME2 Lavalier Mic (G: 566-608MHz), Sony VCT-60AV Tripod with Remote in Grip
    - #protip: buy a super durable external hard drive to back up your video immediately.
  - **Presentation Equipment:**
    - Buy 3 - 5 laptops to use at conferences, and load them with presentation software. This allows you to pre set hibernate settings, resolution etc and keeps speaker transitions smooth.

**Manage Feedback**
Collection and managing feedback presents several difficulties. You want your feedback to be in depth, but you also want it to be low entry barrier, and you want your attendees to engage in feedback while it is fresh in their minds. Here are some tools we use.

- **Twitter**
  - Set up a twitter handle dedicated to your conferences. Ask people to tweet at you with feedback, and use DM to engage in dialogue. This creates a low entry barrier method of getting feedback with potential for more in depth conversation, and it prevents one negative comment (or even just a silly comment) from turning into spam on your #event twitter feed.

- **Guidebook**
  - Set up low entry barrier questions on guidebook with general multiple choice questions, and a link to your full survey
  - [SurveyMonkey]
  - Use survey monkey to create a conference survey that encompasses every talk, meal and any other aspect of your conference for complete feedback. Use giveaways to encourage participation

- **JIRA**
  - Use JIRA to encourage and promote internal feedback. Your colleagues are probably passionate about your work and have a lot to say, but may feel intimidated by addressing you directly or even feel badly for providing criticism on something you've worked very hard on. Start general and get more specific based on what feedback you get the most!

### MongoDB Bangalore 2012 Speaker Info

- **Dates and Deadlines**
- **General Information**
- **Location**
- **Slide and Presentation Review**
- **Equipment**
- **Presentations**

#### Dates and Deadlines

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Show up at least 10 minutes prior to the start of your talk.

#### General Information

**URL/Dates**

Twitter: 

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Location

<table>
<thead>
<tr>
<th>Conference Location</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link, Address</td>
<td>directions, contact</td>
<td>directions, contact</td>
</tr>
</tbody>
</table>

#### Slide and Presentation Review

- Submitted your slides to speakerinfo@10gen.com.
- An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely
fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.

- We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.

**Equipment**

**Provided**

- Projector, screen, and mic for speaker
- MacBook Mini Display port - VGA adaptor
- Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

**To Bring:**

- A Laptop for your presentation
- A VGA adaptor for anything that is NOT a Mac mini display port
- Conference wifi is always unreliable so we recommend against any live demos that require Internet

**Presentations**

- Each presenter will have a 40 minute session time
- Please factor your Q&A time into this
- Each session has a short break between sessions to allow attendees to transition
- We have staff on hand to make sure you start and end on time.
- the session schedule can be viewed at

**Presentation Tips**

**Filming**

**We Are Filming You!**

- Don’t wear stripes or busy print
- Do repeat questions from the audience before answering them.
- Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers

**MongoDB Beijing 2012 Speaker Info**

- Dates and Deadlines
- General Information
- Location
- Slide and Presentation Review
- Equipment
- Presentations

**Dates and Deadlines**

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Show up at least 10 minutes prior to the start of your talk.

**General Information**

URL/Dates

Twitter: #

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Location

<table>
<thead>
<tr>
<th>Conference Location</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link, Address</td>
<td>directions, contact</td>
<td>directions, contact</td>
</tr>
</tbody>
</table>

Slide and Presentation Review

- Submitted your slides to speakerinfo@10gen.com.
  - An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.
  - We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.

Equipment

Provided

- Projector, screen, and mic for speaker
- MacBook Mini Display port - VGA adaptor
- Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

To Bring:

- A Laptop for your presentation
- A VGA adaptor for anything that is NOT a Mac mini display port
- Conference wifi is always unreliable so we recommend against any live demos that require Internet

Presentations

- Each presenter will have a 40 minute session time
- Please factor your Q&A time into this
- Each session has a short break between sessions to allow attendees to transition
- We have staff on hand to make sure you start and end on time.
- the session schedule can be viewed at

Presentation Tips

Filming

We Are Filming You!

- Don’t wear stripes or busy print
- Do repeat questions from the audience before answering them.
- Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers

MongoDB Boston 2012 Speaker Info

- Dates and Deadlines
- General Information
- Location
- Slide and Presentation Review
- Equipment
Presentations

Dates and Deadlines

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Show up at least 10 minutes prior to the start of your talk.

General Information

URL/Dates
Twitter: 

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Location

<table>
<thead>
<tr>
<th>Conference Location</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link, Address</td>
<td>directions, contact</td>
<td>directions, contact</td>
</tr>
</tbody>
</table>

Slide and Presentation Review

- Submitted your slides to speakerinfo@10gen.com.
  - An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.
  - We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.

Equipment

Provided

- Projector, screen, and mic for speaker
- MacBook Mini Display port - VGA adaptor
- Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

To Bring:

- A Laptop for your presentation
- A VGA adaptor for anything that is NOT a Mac mini display port
- Conference wifi is always unreliable so we recommend against any live demos that require Internet

Presentations

- Each presenter will have a 40 minute session time
- Please factor your Q&A time into this
- Each session has a short break between sessions to allow attendees to transition
• We have staff on hand to make sure you start and end on time.
• the session schedule can be viewed at

Presentation Tips

Filming

We Are Filming You!

• Don’t wear stripes or busy print
• Do repeat questions from the audience before answering them.
• Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers

MongoDB Chicago 2012 Speaker Info

• Dates and Deadlines
• General Information
• Location
• Slide and Presentation Review
• Equipment
• Presentations

Dates and Deadlines

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Show up at least 10 minutes prior to the start of your talk.

General Information

URL/Dates
Twitter: #

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Location

<table>
<thead>
<tr>
<th>Conference Location</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link, Address</td>
<td>directions, contact</td>
<td>directions, contact</td>
</tr>
</tbody>
</table>

Slide and Presentation Review

• Submitted your slides to speakerinfo@10gen.com.
  • An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.
• We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.
Equipment

Provided

- Projector, screen, and mic for speaker
- MacBook Mini Display port - VGA adaptor
- Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

To Bring:

- A Laptop for your presentation
- A VGA adaptor for anything that is NOT a Mac mini display port
- Conference wifi is always unreliable so we recommend against any live demos that require Internet

Presentations

- Each presenter will have a 40 minute session time
- Please factor your Q&A time into this
- Each session has a short break between sessions to allow attendees to transition
- We have staff on hand to make sure you start and end on time.
- the session schedule can be viewed at

PresentationTips

Filming

We Are Filming You!

- Don’t wear stripes or busy print
- Do repeat questions from the audience before answering them.
- Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers

MongoDB Melbourne 2012 Speaker Info

- Dates and Deadlines
- General Information
- Location
- Slide and Presentation Review
- Equipment
- Presentations

Dates and Deadlines

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Show up at least 10 minutes prior to the start of your talk.

General Information

URL/Dates
Twitter: #

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Location

<table>
<thead>
<tr>
<th>Conference Location</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link, Address</td>
<td>directions, contact</td>
<td>directions, contact</td>
</tr>
</tbody>
</table>

Slide and Presentation Review

- Submitted your slides to speakerinfo@10gen.com.
- An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.
- We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.

Equipment

Provided

- Projector, screen, and mic for speaker
- MacBook Mini Display port - VGA adaptor
- Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

To Bring:

- A Laptop for your presentation
- A VGA adaptor for anything that is NOT a Mac mini display port
- Conference wifi is always unreliable so we recommend against any live demos that require Internet

Presentations

- Each presenter will have a 40 minute session time
- Please factor your Q&A time into this
- Each session has a short break between sessions to allow attendees to transition
- We have staff on hand to make sure you start and end on time.
- the session schedule can be viewed at

Filming

We Are Filming You!

- Don’t wear stripes or busy print
- Do repeat questions from the audience before answering them.
- Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers

MongoDB Munich 2012 Speaker Info

- Dates and Deadlines
- General Information
- Location
- Slide and Presentation Review
- Equipment
- Presentations
Dates and Deadlines

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
</table>

- Show up at least 10 minutes prior to the start of your talk.

General Information

URL/Dates
Twitter: #

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
</table>

Location

<table>
<thead>
<tr>
<th>Conference Location</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link, Address</td>
<td>directions, contact</td>
<td>directions, contact</td>
</tr>
</tbody>
</table>

Slide and Presentation Review

- Submitted your slides to speakerinfo@10gen.com.
  - An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.
  - We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.

Equipment

Provided

- Projector, screen, and mic for speaker
- MacBook Mini Display port - VGA adaptor
- Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

To Bring:

- A Laptop for your presentation
- A VGA adaptor for anything that is NOT a Mac mini display port
- Conference wifi is always unreliable so we recommend against any live demos that require Internet

Presentations

- Each presenter will have a 40 minute session time
- Please factor your Q&A time into this
- Each session has a short break between sessions to allow attendees to transition
- We have staff on hand to make sure you start and end on time.
- The session schedule can be viewed at
Presentation Tips

Filming

We Are Filming You!

- Don’t wear stripes or busy print.
- Do repeat questions from the audience before answering them.
- Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers.

MongoDB Pune 2012 Speaker Info

- Dates and Deadlines
- General Information
- Location
- Slide and Presentation Review
- Equipment
- Presentations

Dates and Deadlines

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Show up at least 10 minutes prior to the start of your talk.

General Information

URL/Dates
Twitter: #

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Location

<table>
<thead>
<tr>
<th>Conference Location</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link, Address</td>
<td>directions, contact</td>
<td>directions, contact</td>
</tr>
</tbody>
</table>

Slide and Presentation Review

- Submitted your slides to speakerinfo@10gen.com.
  - An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.
- We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.
Equipment

Provided

- Projector, screen, and mic for speaker
- MacBook Mini Display port - VGA adaptor
- Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

To Bring:

- A Laptop for your presentation
- A VGA adaptor for anything that is NOT a Mac mini display port
- Conference wifi is always unreliable so we recommend against any live demos that require Internet

Presentations

- Each presenter will have a 40 minute session time
- Please factor your Q&A time into this
- Each session has a short break between sessions to allow attendees to transition
- We have staff on hand to make sure you start and end on time.
- the session schedule can be viewed at

PresentationTips

Filming

We Are Filming You!

- Don’t wear stripes or busy print
- Do repeat questions from the audience before answering them.
- Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers

MongoDB Seattle 2012 Speaker Info

- 10gen Contact:
- Dates and Deadlines
- General Information
- Location
- Slide and Presentation Review
- Equipment
- Presentations
- Parking and Transportation

10gen Contact:
Melia Jones p. 773-875-6867 e. melia@10gen.com OR events@10gen.com

Dates and Deadlines

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>08/17/2012</td>
<td>Slides or Outline Due</td>
</tr>
<tr>
<td>09/01/2012</td>
<td>RSVP for Speaker/Sponsor dinner to <a href="mailto:events@10gen.com">events@10gen.com</a></td>
</tr>
<tr>
<td>09/13/2012</td>
<td>MongoDB Seattle Workshop</td>
</tr>
<tr>
<td>09/13/2012</td>
<td>Speaker and Sponsor Thank You Dinner (6 pm - 9 pm )</td>
</tr>
<tr>
<td>09/14/2012</td>
<td>MongoDB Seattle</td>
</tr>
</tbody>
</table>
• Show up at least 10 minutes prior to the start of your talk.

General Information

Event Page
09/13/2012 MongoDB Seattle Workshops
09/14/2012 MongoDB Seattle Conference
Twitter: #MongoDB Seattle
WiFi: Please look for Bell Harbor Conference Center and go. No password needed

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 am - 9 am</td>
<td>breakfast and registration</td>
</tr>
<tr>
<td>9:15 am - 9:45 am</td>
<td>keynote</td>
</tr>
<tr>
<td>10 am - 12:55 pm</td>
<td>sessions</td>
</tr>
<tr>
<td>12 pm - 2 pm</td>
<td>lunch served</td>
</tr>
<tr>
<td>1:45 pm - 5:15 pm</td>
<td>session</td>
</tr>
</tbody>
</table>

Location

<table>
<thead>
<tr>
<th></th>
<th>Conference</th>
<th>Hotel (Sleeping Rooms &amp; Workshops)</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link</td>
<td>Bell Harbor International Conference Center</td>
<td>Seattle Waterfront Marriott</td>
<td>Aqua</td>
<td>Tavern Law</td>
</tr>
<tr>
<td>Address</td>
<td>2211 Alaskan Way, Pier 66 Seattle, WA 98121</td>
<td>2100 Alaskan Way - Seattle, Washington 98121 USA</td>
<td>2801 Alaskan Way, Pier 70 Seattle, WA 98121</td>
<td>1406 12th Avenue, Seattle WA 98122 map</td>
</tr>
<tr>
<td>Phone</td>
<td>206.441.6666</td>
<td>1-206-443-5000</td>
<td>206.956.8171</td>
<td>206.322.9734</td>
</tr>
<tr>
<td>Reservation Times</td>
<td>6 am - 5:30 pm</td>
<td></td>
<td>6 pm</td>
<td>5 pm - 8 pm</td>
</tr>
</tbody>
</table>

Slide and Presentation Review

• Submitted your slides to speakerinfo@10gen.com.
  • An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.
  • We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.

Equipment

Provided

• Projector, screen, and mic for speaker
• MacBook Mini Display port - VGA adaptor
• Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

To Bring:

• A Laptop for your presentation
• A VGA adaptor for anything that is NOT a Mac mini display port
• Conference wifi is always unreliable so we recommend against any live demos that require Internet

Presentations
- Each presenter will have a 40 minute session time
- Please factor your Q&A time into this
- Each session has a short break between sessions to allow attendees to transition
- We have staff on hand to make sure you start and end on time.
- the session schedule can be viewed at

Presentation Tips

Filming

We Are Filming You!

- Don’t wear stripes or busy print
- Do repeat questions from the audience before answering them.
- Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers

Parking and Transportation

complimentary all day Garage Parking is available on site to speakers and sponsor only, please contact events@10gen.com if you would like a parking pass; For all other attendees, all day parking is available for $15

from SeaTac International Airport | driving | public transportation

MongoDB Seattle 2012 Sponsor Info

- 4 Leaf Sponsors Dates and Deadlines
- 3 Leaf Sponsors Dates and Deadlines
- 2 Leaf Sponsors Dates and Deadlines
- 1 Leaf Sponsors Dates and Deadlines
- Media Partner Sponsors Dates and Deadlines
- 10gen Contact:
  - General Information
  - Location
  - Staff Registration
  - Logo Specs
  - Shipping Information
  - Parking and Transportation

4 Leaf Sponsors Dates and Deadlines

Included:

- Linked logo on website, conference emails and collateral
- 150 word message in “Reminder” Email
- 150 word message in “Thank You” Email
- 6 conference passes for your table
- 6 conference passes for your community
- Sponsorship of after party

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Submit logos - see Specs below</td>
</tr>
<tr>
<td>August 27</td>
<td>Submit Reminder and Thank You email copy to <a href="mailto:sponsor@10gen.com">sponsor@10gen.com</a></td>
</tr>
<tr>
<td>August 27</td>
<td>Submit Staffing form to <a href="mailto:events@10gen.com">events@10gen.com</a></td>
</tr>
<tr>
<td>August 27</td>
<td>Hand out your free community passes on <a href="http://mongodbseattle2012.eventbrite.com/">http://mongodbseattle2012.eventbrite.com/</a> with the code “CompanySeattle” (so 10gen would be “10genSeattle”)</td>
</tr>
<tr>
<td>Sept 1</td>
<td>RSVP for speaker and sponsor dinner at ()</td>
</tr>
<tr>
<td>Sept 12</td>
<td>Have your boxes arrive at venue between 9am and 5pm</td>
</tr>
<tr>
<td>Sept 13</td>
<td>MongoDB Workshops</td>
</tr>
</tbody>
</table>
Sept 13  Speaker and sponsor dinner at ( ) 6:30pm
Sept 14  Conference day

3 Leaf Sponsors Dates and Deadlines

Included:

- Linked logo on website, conference emails and collateral
- 150 word message in "Reminder" Email
- 150 word message in "Thank You" Email
- 6 conference passes for your table
- 6 conference passes for your community
- Sponsorship of breakfast, coffee or lunch

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Submit logos - see Specs below</td>
</tr>
<tr>
<td>August 27</td>
<td>Submit Reminder and Thank You email copy to <a href="mailto:sponsor@10gen.com">sponsor@10gen.com</a></td>
</tr>
<tr>
<td>August 27</td>
<td>Submit Staffing form to <a href="mailto:events@10gen.com">events@10gen.com</a></td>
</tr>
<tr>
<td>August 27</td>
<td>Hand out your free community passes on <a href="http://mongodbseattle2012.eventbrite.com/">http://mongodbseattle2012.eventbrite.com/</a> with the code &quot;CompanySeattle&quot; (so 10gen would be &quot;10genSeattle&quot;)</td>
</tr>
<tr>
<td>Sept 1</td>
<td>RSVP for speaker and sponsor dinner at ()</td>
</tr>
<tr>
<td>Sept 12</td>
<td>Have your boxes arrive at venue between 9am and 5pm</td>
</tr>
<tr>
<td>Sept 13</td>
<td>MongoDB Workshops</td>
</tr>
<tr>
<td>Sept 13</td>
<td>Speaker and sponsor dinner at ( ) 6:30pm</td>
</tr>
<tr>
<td>Sept 14</td>
<td>Conference day</td>
</tr>
</tbody>
</table>

2 Leaf Sponsors Dates and Deadlines

Included:

- Linked logo on website, conference emails and collateral
- 50 word message in "Reminder" Email
- 50 word message in "Thank You" Email
- 6 conference passes for your table
- 6 conference passes for your community

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Submit logos - see Specs below</td>
</tr>
<tr>
<td>August 27</td>
<td>Submit Reminder and Thank You email copy to <a href="mailto:sponsor@10gen.com">sponsor@10gen.com</a></td>
</tr>
<tr>
<td>August 27</td>
<td>Submit Staffing form to <a href="mailto:events@10gen.com">events@10gen.com</a></td>
</tr>
<tr>
<td>August 27</td>
<td>Hand out your free community passes on <a href="http://mongodbseattle2012.eventbrite.com/">http://mongodbseattle2012.eventbrite.com/</a> with the code &quot;CompanySeattle&quot; (so 10gen would be &quot;10genSeattle&quot;)</td>
</tr>
<tr>
<td>Sept 1</td>
<td>RSVP for speaker and sponsor dinner at ()</td>
</tr>
<tr>
<td>Sept 12</td>
<td>Have your boxes arrive at venue between 9am and 5pm</td>
</tr>
<tr>
<td>Sept 13</td>
<td>MongoDB Workshops</td>
</tr>
<tr>
<td>Sept 13</td>
<td>Speaker and sponsor dinner at ( ) 6:30pm</td>
</tr>
<tr>
<td>Sept 14</td>
<td>Conference day</td>
</tr>
</tbody>
</table>

1 Leaf Sponsors Dates and Deadlines
Included:

- Linked logo on conference emails, website and collateral
- 50 word message in "Reminder" Email
- 150 word message in "Thank You" Email
- 3 conference passes for your community

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Submit logos - see Specs below</td>
</tr>
<tr>
<td>August 27</td>
<td>Submit Reminder and Thank You email copy to <a href="mailto:sponsor@10gen.com">sponsor@10gen.com</a></td>
</tr>
<tr>
<td>Agust 27</td>
<td>Submit Staffing form to <a href="mailto:events@10gen.com">events@10gen.com</a></td>
</tr>
<tr>
<td>August 27</td>
<td>Hand out your free community passes on <a href="http://mongodbseattle2012.eventbrite.com/">http://mongodbseattle2012.eventbrite.com/</a> with the code &quot;CompanySeattle&quot; (so 10gen would be &quot;10genSeattle&quot;)</td>
</tr>
<tr>
<td>Sept 1</td>
<td>RSVP for speaker and sponsor dinner at ()</td>
</tr>
<tr>
<td>Sept 12</td>
<td>Have your boxes arrive at venue between 9am and 5pm</td>
</tr>
<tr>
<td>Sept 13</td>
<td>MongoDB Workshops</td>
</tr>
<tr>
<td>Sept 13</td>
<td>Speaker and sponsor dinner at () 6:30pm</td>
</tr>
<tr>
<td>Sept 14</td>
<td>Conference day</td>
</tr>
</tbody>
</table>

Media Partner Sponsors Dates and Deadlines

Included:

- Linked logo on conference emails, website, and collateral

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Submit logos - see Specs below</td>
</tr>
<tr>
<td>Agust 27</td>
<td>Submit Staffing form to <a href="mailto:events@10gen.com">events@10gen.com</a></td>
</tr>
<tr>
<td>August 27</td>
<td>Hand out your free community passes on <a href="http://mongodbseattle2012.eventbrite.com/">http://mongodbseattle2012.eventbrite.com/</a> with the code &quot;CompanySeattle&quot; (so 10gen would be &quot;10genSeattle&quot;)</td>
</tr>
<tr>
<td>Sept 1</td>
<td>RSVP for speaker and sponsor dinner at ()</td>
</tr>
<tr>
<td>Sept 12</td>
<td>Have your boxes arrive at venue between 9am and 5pm</td>
</tr>
<tr>
<td>Sept 13</td>
<td>MongoDB Workshops</td>
</tr>
<tr>
<td>Sept 13</td>
<td>Speaker and sponsor dinner at () 6:30pm</td>
</tr>
<tr>
<td>Sept 14</td>
<td>Conference day</td>
</tr>
</tbody>
</table>

10gen Contact:

Melia Jones  p. 773-875-6867 e. melia@10gen.com OR events@10gen.com

General Information

**Public Event URL** Sept 14, 2012
**Official Twitter Hashtag:** #MongoSeattle
WiFi: Please look for Bell Harbor Conference Center and go. No password needed

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:00 am</td>
<td>Sponsor Table Set up (Ends at 8:30)</td>
</tr>
<tr>
<td>8:00 am</td>
<td>Registration start</td>
</tr>
</tbody>
</table>
8:00 am  Breakfast (ends at 9)
9:15 am  Session start
12:00 pm  Lunch (ends at 2)
All Day  Coffee Service

Location

<table>
<thead>
<tr>
<th>Conference</th>
<th>Hotel (Sleeping Rooms &amp; Workshops)</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link</td>
<td>Bell Harbor International Conference Center</td>
<td>Seattle Waterfront Marriott</td>
<td>Aqua</td>
</tr>
<tr>
<td>Address</td>
<td>2211 Alaskan Way, Pier 66 Seattle, WA 98121</td>
<td>2100 Alaskan Way · Seattle, Washington 98121 USA</td>
<td>2801 Alaskan Way, Pier 70 Seattle, WA 98121</td>
</tr>
<tr>
<td>Phone</td>
<td>206.441.6666</td>
<td>1-206-443-5000</td>
<td>206.956.8171</td>
</tr>
<tr>
<td>Reservation Times</td>
<td>6 am - 5:30 pm</td>
<td>6 pm</td>
<td>5 pm - 8 pm</td>
</tr>
</tbody>
</table>

Staff Registration

Please use the attached form to submit your staffing by August 27

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Creator (Last Modifier)</th>
<th>Creation Date</th>
<th>Last Mod Date</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sponsor Staffing Template.xlsx</td>
<td>33 kB</td>
<td>Melia Jones</td>
<td>Aug 08, 2012</td>
<td>Aug 08, 2012</td>
<td>Sponsor Staffing Doc</td>
</tr>
</tbody>
</table>

Logo Specs

Full color vector version (or high res version, if vector version not available)

Knockout vector version, preferably all white

230px wide PNG or JPEG (height dependent on shape of your trademark). Please include white space around trademark in the file.

Shipping Information

Detailed Shipping Information

Bell Harbor Conference Center
2211 Alaskan Way, Pier 66
Seattle, WA 98121
Phone: 206.441.6666

- please clearly label all packages with “10gen, sponsor name” -
  “10gen holds no responsibility for shipped items

Parking and Transportation

complimentary all day Garage Parking is available on site to speakers and sponsor only, please contact events@10gen.com if you would like a parking pass; For all other attendees, all day parking is available for $15

from SeaTac International Airport| driving | public transportation

MongoDB Sydney 2012 Speaker Info
Dates and Deadlines

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Show up at least 10 minutes prior to the start of your talk.

General Information

URL/Dates
Twitter: #

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Location

<table>
<thead>
<tr>
<th>Conference Location</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link, Address</td>
<td>directions, contact</td>
<td>directions, contact</td>
</tr>
</tbody>
</table>

Slide and Presentation Review

- Submitted your slides to speakerinfo@10gen.com.
  - An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.
  - We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.

Equipment

Provided

- Projector, screen, and mic for speaker
- MacBook Mini Display port - VGA adaptor
- Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

To Bring:

- A Laptop for your presentation
- A VGA adaptor for anything that is NOT a Mac mini display port
- Conference wifi is always unreliable so we recommend against any live demos that require Internet
Presentations

- Each presenter will have a 40 minute session time
- Please factor your Q&A time into this
- Each session has a short break between sessions to allow attendees to transition
- We have staff on hand to make sure you start and end on time.
- the session schedule can be viewed at

Presentation Tips

Filming

We Are Filming You!

- Don’t wear stripes or busy print
- Do repeat questions from the audience before answering them.
- Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers

MongoDB Tokyo 2012 Speaker Info

Dates and Deadlines

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Show up at least 10 minutes prior to the start of your talk.

General Information

URL/Dates
Twitter: #

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Location

<table>
<thead>
<tr>
<th>Conference Location</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link, Address</td>
<td>directions, contact</td>
<td>directions, contact</td>
</tr>
</tbody>
</table>
Slide and Presentation Review

- Submitted your slides to speakerinfo@10gen.com.
  - An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.
- We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.

Equipment

Provided

- Projector, screen, and mic for speaker
- MacBook Mini Display port - VGA adaptor
- Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

To Bring:

- A Laptop for your presentation
- A VGA adaptor for anything that is NOT a Mac mini display port
- Conference wifi is always unreliable so we recommend against any live demos that require Internet

Presentations

- Each presenter will have a 40 minute session time
- Please factor your Q&A time into this
- Each session has a short break between sessions to allow attendees to transition
- We have staff on hand to make sure you start and end on time.
- the session schedule can be viewed at

Presentation Tips

Filming

We Are Filming You!

- Don’t wear stripes or busy print
- Do repeat questions from the audience before answering them.
- Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers

MongoDC 2012 Sponsor Info

- 3 Leaf Sponsors Dates and Deadlines
- 2 Leaf Sponsors Dates and Deadlines
- 1 Leaf Sponsors Dates and Deadlines
- Media Partner Sponsors Dates and Deadlines
- General Information
- Location
- Logo Specs
- Shipping Information

3 Leaf Sponsors Dates and Deadlines

Included:

- Linked logo on website, conference emails and collateral
- 50 word message in “Reminder” Email
- 150 word message in “Thank You” Email
- 6 conference passes for your table
- 6 conference passes for your community

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 15th</td>
<td>Submit Reminder and Thank You email copy to <a href="mailto:sponsor@10gen.com">sponsor@10gen.com</a></td>
</tr>
</tbody>
</table>
June 15th | Register your staff with eventbrite (promotional code “DCSponsor”)
June 21st | RSVP for speaker and sponsor dinner at Gordon Biersch
June 21st | Hand out your free community passes on eventbrite with the code “CompanyDC” (so 10gen would be “10genDC”)
June 25th | Have your boxes arrive at venue between 9am and 5pm
June 25th | Speaker and sponsor dinner at Gordon Biersch, 6:30pm
June 26th | Conference day

2 Leaf Sponsors Dates and Deadlines

Included:

- Linked logo on website, conference emails and collateral
- 50 word message in “Reminder” Email
- 150 word message in “Thank You” Email
- 6 conference passes for your table
- 6 conference passes for your community

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 15th</td>
<td>Submit Reminder and Thank You email copy to <a href="mailto:sponsor@10gen.com">sponsor@10gen.com</a></td>
</tr>
<tr>
<td>June 15th</td>
<td>Register with eventbrite (promotional code “DCSponsor”)</td>
</tr>
<tr>
<td>June 21st</td>
<td>RSVP for speaker and sponsor dinner at Gordon Biersch</td>
</tr>
<tr>
<td>June 21st</td>
<td>Hand out your free community passes on eventbrite with the code “CompanyDC” (so 10gen would be “10genDC”)</td>
</tr>
<tr>
<td>June 25th</td>
<td>Have your boxes arrive at venue between 9am and 5pm</td>
</tr>
<tr>
<td>June 25th</td>
<td>Speaker and sponsor dinner at Gordon Biersch, 6:30pm</td>
</tr>
<tr>
<td>June 26th</td>
<td>Conference day</td>
</tr>
</tbody>
</table>

1 Leaf Sponsors Dates and Deadlines

Included:

- Linked logo on conference emails, website and collateral
- 50 word message in “Reminder” Email
- 150 word message in “Thank You” Email
- 3 conference passes for your community

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 21st</td>
<td>Hand out your free community passes on eventbrite with the code “CompanyDC” (so 10gen would be “10genDC”)</td>
</tr>
<tr>
<td>June 21st</td>
<td>RSVP for speaker and sponsor dinner at Gordon Biersch</td>
</tr>
<tr>
<td>June 25th</td>
<td>Have your boxes arrive at venue between 9am and 5pm</td>
</tr>
<tr>
<td>June 25th</td>
<td>Speaker and sponsor dinner at Gordon Biersch, 6:30pm</td>
</tr>
<tr>
<td>June 26th</td>
<td>Conference day</td>
</tr>
</tbody>
</table>

Media Partner Sponsors Dates and Deadlines

Included:

- Linked logo on conference emails, website, and collateral
<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 15th</td>
<td>Register with eventbrite (promotional code ‘DCSponsor’)</td>
</tr>
<tr>
<td>June 21st</td>
<td>RSVP for speaker and sponsor dinner at Gordon Biersch</td>
</tr>
<tr>
<td>June 25th</td>
<td>Have your boxes arrive at venue between 9am and 5pm</td>
</tr>
<tr>
<td>June 25th</td>
<td>Speaker and sponsor dinner at Gordon Biersch, 6:30pm</td>
</tr>
<tr>
<td>June 26th</td>
<td>Conference day</td>
</tr>
</tbody>
</table>

**General Information**

**Public Event URL June 26, 2012**  
Official Twitter Hashtag: #MongoDC

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:00 am</td>
<td>Sponsor Table Set up (Ends at 8:30)</td>
</tr>
<tr>
<td>8:00 am</td>
<td>Registration start</td>
</tr>
<tr>
<td>8:00 am</td>
<td>Breakfast (ends at 10)</td>
</tr>
<tr>
<td>9:15 am</td>
<td>Session start</td>
</tr>
<tr>
<td>1:00 pm</td>
<td>Lunch (ends at 2)</td>
</tr>
<tr>
<td>All Day</td>
<td>Coffee Service</td>
</tr>
</tbody>
</table>

**Location**

<table>
<thead>
<tr>
<th>Conference Location</th>
<th>Speaker Dinner Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Woolly Mammoth Theatre</td>
<td>Gordon Biersch</td>
</tr>
<tr>
<td>641 D Street, NW</td>
<td>900 F Street, NW</td>
</tr>
<tr>
<td>Washington, DC 20004</td>
<td>Washington, DC 20004</td>
</tr>
<tr>
<td>(202) 289-2443</td>
<td>(202) 783-5454</td>
</tr>
</tbody>
</table>

*Get Detailed Map and Directions for Conference*  
*Map for Speaker Dinner*

**Logo Specs**

- Full color vector version (or high res version, if vector version not available)
- Knockout vector version, preferably all white
- 230px wide PNG or JPEG (height dependent on shape of your trademark). Please include white space around trademark in the file.

**Shipping Information**

**Detailed Shipping Information**

Please ship as little as possible to the venue as there is very little holding space there. Arrange for it to arrive on Friday the 22nd or Monday the 25th between 10am and 6pm, as nobody will be there to receive the package outside of those hours.

Attention: Zacory Boatwright/10gen  
Woolly Mammoth Theatre  
641 D Street, NW  
Washington, DC 20004

Please also be aware that we have no palette dolly. All shipments must be able to be broken up into smaller parts that may be moved by one
MongoNYC 2013 Speaker Info

- Dates and Deadlines
- General Information
- Location
- Slide and Presentation Review
- Equipment
- Presentations

Dates and Deadlines

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Show up at least 10 minutes prior to the start of your talk.

General Information

URL/Dates
Twitter: #

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Location

Conference Location | Speaker Dinner Location | After Party Location
Link, Address       | directions, contact    | directions, contact |

equipment

- Submitted your slides to speakerinfo@10gen.com.
  - An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.
  - We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.

Equipment

Provided

- Projector, screen, and mic for speaker
- MacBook Mini Display port - VGA adaptor
- Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

To Bring:
• A Laptop for your presentation
• A VGA adaptor for anything that is NOT a Mac mini display port
• Conference wifi is always unreliable so we recommend against any live demos that require Internet

Presentations

• Each presenter will have a 40 minute session time
• Please factor your Q&A time into this
• Each session has a short break between sessions to allow attendees to transition
• We have staff on hand to make sure you start and end on time.
• the session schedule can be viewed at

PresentationTips

Filming

We Are Filming You!

• Don’t wear stripes or busy print
• Do repeat questions from the audience before answering them.
• Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers

MongoSF 2013 Speaker Info

• Dates and Deadlines
• General Information
• Location
• Slide and Presentation Review
• Equipment
• Presentations

Dates and Deadlines

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Show up at least 10 minutes prior to the start of your talk.

General Information

URL/Dates
Twitter: #

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Location

<table>
<thead>
<tr>
<th>Conference Location</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
</table>
Slide and Presentation Review

- Submitted your slides to speakerinfo@10gen.com.
- An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.
- We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.

Equipment

Provided

- Projector, screen, and mic for speaker
- MacBook Mini Display port - VGA adaptor
- Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

To Bring:

- A Laptop for your presentation
- A VGA adaptor for anything that is NOT a Mac mini display port
- Conference wifi is always unreliable so we recommend against any live demos that require Internet

Presentations

- Each presenter will have a 40 minute session time
- Please factor your Q&A time into this
- Each session has a short break between sessions to allow attendees to transition
- We have staff on hand to make sure you start and end on time.
- the session schedule can be viewed at

Presentation Tips

Filming

We Are Filming You!

- Don’t wear stripes or busy print
- Do repeat questions from the audience before answering them.
- Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers

MongoSV 2012 Speaker Info

- Dates and Deadlines
- General Information
- Location
- Slide and Presentation Review
- Equipment
- Presentations

Dates and Deadlines

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Show up at least 10 minutes prior to the start of your talk.
General Information

URL/Dates
Twitter: #

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Location

<table>
<thead>
<tr>
<th>Conference Location</th>
<th>Speaker Dinner Location</th>
<th>After Party Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link, Address</td>
<td>directions, contact</td>
<td>directions, contact</td>
</tr>
</tbody>
</table>

Slide and Presentation Review

- Submitted your slides to speakerinfo@10gen.com.
  - An outline or partially finished content is totally fine; we just want to make sure we get you the feedback you need in a timely fashion. PowerPoint, KeyNote, Prezi, or PDF are all acceptable formats.
- We are here to help! - we are happy to set up a session (Skype, hangout, call) to go through your presentation.

Equipment

Provided

- Projector, screen, and mic for speaker
- MacBook Mini Display port - VGA adaptor
- Wireless Clickers/Pointers - please do not take the USB adaptor with you on accident!

To Bring:

- A Laptop for your presentation
- A VGA adaptor for anything that is NOT a Mac mini display port
- Conference wifi is always unreliable so we recommend against any live demos that require Internet

Presentations

- Each presenter will have a 40 minute session time
- Please factor your Q&A time into this
- Each session has a short break between sessions to allow attendees to transition
- We have staff on hand to make sure you start and end on time.
- the session schedule can be viewed at

Presentation Tips

Filming

We Are Filming You!

- Don’t wear stripes or busy print
- Do repeat questions from the audience before answering them.
- Do announce results if you poll the audience - such as “raise your hand if you use MongoDB”. Your next comments will much more relevant to viewers
Presentation Tips

Slides

- Be precise in your slides - We will make them available on our website after the conference, so make sure they are correct and to the point.
- Be visual - a clever picture, easy to read chart, or even simple bullet points convey clear information.
- Use Big Font! the rooms are large and it is always hard to see from the back of the room.
- Avoid including full text such as sentences and paragraphs in your slides.
- Know your audience - The audience is very technical, and generally knowledgeable about MongoDB.
- We will briefly introduce all of the sessions at the beginning of the event to give attendees an idea of what sessions their time would best be spent attending. Please familiarize yourself with the sessions as well.
- Turn off Hibernate! You don't want your computer to go to sleep in the middle of your presentation or your audience might as well...

Logistics

- **Arrive 10 minutes early** for your presentation: This gives everyone time to make sure that you and your equipment are ready to go! We will have a room monitor run through a check list to set you up.
- We will have a room monitor in each room, so if you need something, say something.
- Staging: Presenters in SkyTop, PennTop North and South, and the Madison will be presenting from a stage and podium. Presenters in Paris or London will be presenting from a podium.
- Projector and screen: You will be connecting with a Mac mini display adaptor to a VGA connection. If you need a different adaptor for your laptop, please bring it with you.
- Mic's: A room monitor will equip with a lavalier (clip on) when you arrive in your session room.
- Internet connectivity: Please do not rely on wifi for demos as we cannot guarantee it will work properly.
- Power: we will have power running to your podium. Please make sure your machine is fully charged anyway!

Questions

- If there is a microphone, make sure to wait for the person to get it, and use it.
- Repeat or rephrase the question to confirm it is what they are asking and everyone hears it.
- If the question is off-topic or lengthy, table-it or ask the person to find you after the presentation.
- Move towards the person and away from the podium to engage the audience.

Home

Documentation

- “The manual” (docs.mongodb.org)
- Installation Guide
- Tutorial
- SQL to Mongo Mapping Chart
- List of Reference Pages
- Production Notes tips and best practices
- Replication | Sharding
- Security | Backups
- Developer info by language
  - C | C# | C++ | C# & .NET | ColdFusion | Erlang | Haskell | Factor
  - Java | Javascript | PHP | Python | Ruby | Perl | more...

Production

- Hosting
- MongoDB Monitoring Service (MMS) is a free way to monitor your mongodb servers

Support

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

- Use Cases | Philosophy | License | Events

Community

- Conferences and Meetups
- Slides and Videos
- Blog | Articles | Twitter | Forum | MongoDB Masters | Facebook | LinkedIn | Job Board
- Your Go-To Guide to Running a MongoDB User Group

Translations

New Docs (in Progress)

Existing Docs

- Español | Português | | |

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

Introduction

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

– Eliot Horowitz, 10gen CTO and Co-founder

Why MongoDB?

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

Large MongoDB deployment

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

A replica set is one or more servers, each holding copies of the same data. At any given time one is primary and the rest are secondaries. If the primary goes down one of the secondaries takes over automatically as primary. All writes and consistent reads go to the primary, and all eventually consistent reads are distributed amongst all the secondaries.
2. Multiple config servers, each one holds a copy of the meta data indicating which data lives on which shard.

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

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

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

**Small deployment (no partitioning)**

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

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

**Mongo data model**

- A MongoDB 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.
Quickstart

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

Installation Guides

See the install guides from the MongoDB Manual:

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

See Also

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

Downloads

<table>
<thead>
<tr>
<th></th>
<th>OS X 32-bit</th>
<th>OS X 64-bit</th>
<th>Linux 32-bit</th>
<th>Linux 64-bit</th>
<th>Windows 32-bit</th>
<th>Windows 64-bit</th>
<th>Solaris i86pc</th>
<th>Solaris 64</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production Release (Recommended)</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>tgz</td>
</tr>
<tr>
<td>1.8.1</td>
<td>4/6/2011</td>
<td>Changelog</td>
<td>Release Notes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>zip</td>
</tr>
<tr>
<td>Nightly</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>tgz</td>
</tr>
<tr>
<td>Changelog</td>
<td>Nightly</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>download</td>
<td>tgz</td>
</tr>
<tr>
<td>Development Release (Unstable)</td>
<td>new release</td>
<td>new release</td>
<td>new release</td>
<td>new release</td>
<td>new release</td>
<td>new release</td>
<td>new release</td>
<td>new release</td>
<td>tgz</td>
</tr>
<tr>
<td>Archived Releases</td>
<td>list</td>
<td>list</td>
<td>list</td>
<td>list</td>
<td>list</td>
<td>list</td>
<td>list</td>
<td>list</td>
<td>list</td>
</tr>
</tbody>
</table>

2.2 Release Notes

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/release-notes/2.2/.

2.0 Release Notes
Upgrading to 1.8

- Upgrading Replica Sets
- Upgrading Sharded Clusters
- Returning to 1.6
  - Journaling
- See Also
- Download

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

Upgrading Replica Sets

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

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

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

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

1. For each arbiter:
   - Shut down the arbiter
   - Start it back up with the 1.8 binary
2. Change your config (optional)
   It is possible that, when you start shutting down members of the set, a new primary will be elected. If you wish to prevent this, you can give all of the slaves a priority of 0 before upgrading, then change them back afterwards.
   - Record your current config. Run rs.conf() and paste the results into a text file.
   - Update your config so that all secondaries have priority 0. For example:
> config = rs.conf()
> config._id: 'foo',
> "version": 3,
> "members": [ 
>   { 
>     "_id": 0,
>     "host": "ubuntu:27017"
>   },
>   { 
>     "_id": 1,
>     "host": "ubuntu:27018"
>   },
>   { 
>     "_id": 2,
>     "host": "ubuntu:27019",
>     "arbiterOnly": true
>   },
>   { 
>     "_id": 3,
>     "host": "ubuntu:27020"
>   },
>   { 
>     "_id": 4,
>     "host": "ubuntu:27021"
>   }
> ]
> config.version++
> 3
> rs.isMaster()
> { 
>   "setName": "foo",
>   "ismaster": false,
>   "secondary": true,
>   "hosts": [ 
>     "ubuntu:27017",
>     "ubuntu:27018"
>   ],
>   "arbiters": [ 
>     "ubuntu:27019"
>   ],
>   "primary": "ubuntu:27018",
>   "ok": 1
> }
> // for each slave
> config.members[0].priority = 0
> config.members[3].priority = 0
> config.members[4].priority = 0
> rs.reconfig(config)

3. For each slave:
   - Shut down the slave
   - Start it back up with the 1.8 binary

4. If you changed the config, change it back to its original state

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

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

1. Turn off the balancer:

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

2. For each shard:
   - If the shard is a replica set, follow the directions above for replica sets.
   - If the shard is a single mongod process, shut it down and start it back up with the 1.8 binary.

3. For each mongos:
   - Shut down the mongos process
   - Restart with the 1.8 binary

4. For each config server:
   - Shut down the config server process
   - Restart with the 1.8 binary

5. Turn on the balancer

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

Returning to 1.6

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

Journaling

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

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

See Also

- [1.8 Release Notes](#) page for details on changes in v1.8 to map/reduce.

Download

- [Download v1.8](#)

1.6 Release Notes

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

* 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

1.6.x Release Notes

- 1.6.5

1.5.x Release Notes

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

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

Thank you everyone for your support and suggestions!

1.4 Release Notes

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

Release 1.4 includes the following improvements over release 1.2:

Core server enhancements

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

Replication & Sharding

- better handling for restarting slaves offline for a while
- fast new slaves from snapshots (--fastsync)
- configurable slave delay (--slavedelay)
• replication handles clock skew on master
• $inc replication fixes
• sharding alpha 3 - notably 2 phase commit on config servers

**Deployment & production**

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

**Query language improvements**

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

**Geo**

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

**1.2.x Release Notes**

**New Features**

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

**DB Upgrade Required**

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

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

Ask in the forums or IRC for more help.

**Replication Changes**

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

**mongoimport**

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

**field filter changing**

• We’ve changed the semantics of the field filter a little bit. Previously only objects with those fields would be returned. Now the field filter
other notes

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

1.0 Changelist

Wrote MongoDB. See documentation.

Version Numbers

MongoDB uses the odd-numbered versions for development releases.

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

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

For example:

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

What's New by Version

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

- 1.4
  - Geospatial
  - Background indexing
  - --directoryperdb
  - Log rotate
  - $not
  - $ operator for updating arrays
  - $addToSet
  - $unset

- 1.6
  - Sharding
  - Replica Sets
  - getLastError w param
  - $or
  - $slice
  - 64 indexes per collection
  - IPv6

- 1.8
  - Journaling
  - Sparse and covered indexes
  - $rename
  - mongos (i.e., sharded environment) will route SLAVE_OK queries to secondaries in replica sets

Ubuntu and Debian packages

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/tutorial/install-mongodb-on-debian-or-ubuntu-linux/.
Drivers

MongoDB currently has client support for the following programming languages:

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

* **Community Supported**
  - ActionScript3
    - [http://www.mongoas3.com](http://www.mongoas3.com)
  - C
    - [libmongo-client](http://code.google.com/p/jmcnet-full-mongo-flex-driver/)
  - C# and .NET
    - [http://www.mongoas3.com](http://www.mongoas3.com)
  - Clojure
    - See the Java Language Center
  - ColdFusion
    - [cfmongodb](http://code.google.com/p/jmcnet-full-mongo-flex-driver/)
    - [http://github.com/virtix/cfmongodb/tree/0.9](http://github.com/virtix/cfmongodb/tree/0.9)
  - D
    - [Port of the MongoDB C Driver for D](https://bitbucket.org/vadimtsushko/mongo-dart)
  - Dart
    - [https://bitbucket.org/vadimtsushko/mongo-dart](https://bitbucket.org/vadimtsushko/mongo-dart)
  - Delphi
    - [mongo-delphi-driver](https://bitbucket.org/vadimtsushko/mongo-dart) - Full featured Delphi interface to MongoDB built on top of the mongodb.org supported C driver
    - [pebongo](https://bitbucket.org/vadimtsushko/mongo-dart) - Early stage Delphi driver for MongoDB
    - [TMongoWire](https://bitbucket.org/vadimtsushko/mongo-dart) - Maps all the VarTypes of OleVariant to the BSON types, implements IPersistStream for (de)serialization, and uses TTcpClient for networking
  - Entity
    - [entity driver for mongod](http://code.google.com/p/jmcnet-full-mongo-flex-driver/) on Google Code, included within the standard Entity Library
  - Erlang
    - [Erfmongo](http://code.google.com/p/jmcnet-full-mongo-flex-driver/) - an almost complete MongoDB driver implementation in Erlang
  - Factor
  - Fantom
  - F#
    - [http://gist.github.com/218388](http://gist.github.com/218388)
  - Go
    - [gomongo](http://github.com/slavapestov/factor/tree/master/extra/mongodb/)
    - [go-mongo](http://github.com/slavapestov/factor/tree/master/extra/mongodb/)
    - [mgo](http://github.com/slavapestov/factor/tree/master/extra/mongodb/)
    - [mongogo](http://github.com/slavapestov/factor/tree/master/extra/mongodb/)
  - Groovy
    - [gmongo](http://github.com/slavapestov/factor/tree/master/extra/mongodb/)
Get Involved, Write a Driver!

- Writing Drivers and Tools

Hadoop

The MongoDB Hadoop Adapter is a plugin for Hadoop that provides Hadoop the ability to use MongoDB as an input source and/or an output source.

The source code is available on github where you can find a more comprehensive readme

If you have questions please email the mongodb-user Mailing List. For any issues please file a ticket in Jira.

Installation

The Mongo-Hadoop adapter uses the SBT Build Tool tool for compilation. SBT provides superior support for discreet configurations targeting multiple Hadoop versions. The distribution includes self-bootstrapping copy of SBT in the distribution as ‘sbt’. Create a copy of the jar files using the following command:

```
./sbt package
```
The MongoDB Hadoop Adapter supports a number of Hadoop releases. You can change the Hadoop version supported by the build by modifying the value of `hadoopRelease` in the `build.sbt` file. For instance, set this value to:

```scala
hadoopRelease in ThisBuild := "cdh3"
```

Configures a build against Cloudera CDH3u3, while:

```scala
hadoopRelease in ThisBuild := "0.21"
```

Configures a build against Hadoop 0.21 from the mainline Apache distribution.

After building, you will need to place the "core" jar and the "mongo-java-driver" in the `lib` directory of each Hadoop server.

For more complete install instructions please see the install instructions in the readme.

Presentations

- MongoDB, Hadoop and HuMONGOus Data by Steve Francia at MongoSF 2012
- MongoDB + Hadoop by Brendan McAdams at MongoDB Philly 2012
- mongo-hadoop (BigData Analysis with Mongo-Hadoop) by Daichi Morifuji at MongoTokyo 2012

Blog Posts


Scala Language Center

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

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

Community

- Lift-MongoDB - Lift Web Framework supports MongoDB, including object mapping via the Record back-end implementation.
- Rogue: A Type-Safe Scala DSL - Foursquare's DSL for querying MongoDB alongside Lift-MongoDB-Record.
- 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.

Haskell Language Center

The Haskell driver and its API documentation reside on Hackage

C Language Center

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

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

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

Download and build

The C driver is hosted at GitHub. You can download the latest stable version: v0.6:

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

CSharp Language Center

Upcoming webinar - What's New in the .Net Driver on July 19

MongoDB C#/.NET Driver

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

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

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

Downloading the Driver

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

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

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

Visual Studio Versions Supported

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

- Visual Studio 2010
- Visual Studio 2008

Questions and Bug Reports

Questions about the C# driver (or any other MongoDB topic) can be posted at the mongodb-user Google Group:
https://groups.google.com/group/mongodb-user

Bug reports can be created in JIRA at:
https://jira.mongodb.org/browse/CSHARP

See Also
• CSharp Community Projects

Presentations

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

CSharp Community Projects

Community Supported C# Drivers

• See also: the 10gen supported MongoDB C# driver
• mongodb-csharp driver
• simple-mongodb driver
• NoRM

Tools

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

F#

• F# Example

Community Articles

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

Support

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

See Also

• C++ Language Center

CSharp Driver LINQ Tutorial

• Introduction
• Quickstart
• Supported LINQ query operators
  • Any
  • Any (with predicate)
  • Count
  • Count (with predicate)
  • Distinct
  • ElementAt
  • ElementAtOrDefault
  • First
  • First (with predicate)
  • FirstOrDefault
  • FirstOrDefault (with predicate)
  • Last
Introduction

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

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

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

Quickstart

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

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

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

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

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

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

```csharp
var query =
    collection.AsQueryable<Employee>()
    .Where(e => e.FirstName == "John")
    .Select(e => e);

// or
var query =
    var employee in query
    { // process employees named "John"

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

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

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

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

**Supported LINQ query operators**

This section documents the supported LINQ query operators.

**Any**

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

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

// or
var result =
    collection.AsQueryable<C>()
    .Any();

Any (with predicate)

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

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

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

Note that the predicate can be provided either by a where clause or as an argument to Any, so the following are equivalent to the previous query.
```csharp
var result =
(from c in collection.AsQueryable<C>()
where c.X == 1
select c)
.Any();
// or
var result =
collection.AsQueryable<C>()
.Where(c => c.X == 1)
.Any();
```

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

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

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

**Count**

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

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

**Count (with predicate)**

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

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

Note that the predicate can be provided either by a where clause or as an argument to Count, so the following are equivalent to the previous query.
var result =
(from c in collection.AsQueryable<C>()
 where c.X == 1
 select c)
 .Count();
// or
var result =
collection.AsQueryable<C>()
 .Where(c => c.X == 1)
 .Count();

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

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

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

Distinct

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

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

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

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

ElementAt

ElementAt returns a particular document from a result set. Often you will combine this with a sort order.
var result =
(from c in collection.AsQueryable<C>()
 where c.X > 0
 orderby c.X
 select c)
 .ElementAt(index);
// or
var result =
collection.AsQueryable<C>()
 .Where(c => c.X > 0)
 .OrderBy(c => c.X)
 .ElementAt(index);

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

**ElementAtOrDefault**

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

**First**

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

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

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

**First (with predicate)**

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

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

First with a predicate is not supported after a projection (at least not yet). So the following is not valid:
var result =
  collection.AsQueryable<C>()
  .OrderBy(c => c.X)
  .Select(c => c.X)
  .Count(x => x > 0);

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

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

FirstOrDefault

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

FirstOrDefault (with predicate)

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

Last

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

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

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

Last (with predicate)

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

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

Last with a predicate is not supported after a projection (at least not yet). So the following is not valid:
You can usually rewrite such a query by putting an equivalent where clause before the projection.

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

**LastOrDefault**

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

**LastOrDefault (with predicate)**

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

**LongCount**

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

**LongCount (with predicate)**

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

**Max**

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

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

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

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

**Max (with selector)**

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

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

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

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

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

Min (with selector)

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

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

OrderBy

OrderBy is used to specify an ascending sort order for the result set.
var query =
  from c in collection.AsQueryable<C>()
  orderby c.X
  select c;
  // or
var query =
  collection.AsQueryable<C>()
  .OrderBy(c => c.X);

**OrderByDescending**

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

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

**Select**

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

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

**Single**

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

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

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

**Single (with predicate)**

This overload of Single allows you to provide a predicate as an argument to Single. This is an alternative to using a where clause.
```csharp
var result =
(from c in collection.AsQueryable<C>()
orderby c.X
select c)
.Single(c => c.X > 0);
// or
var result =
collection.AsQueryable<C>()
.OrderBy(c => c.X)
.Single(c => c.X > 0);
```

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

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

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

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

**SingleOrDefault**

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

**SingleOrDefault (with predicate)**

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

**Skip**

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

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

**Take**

Use Take to specify how many documents to return from the server. When combining Take with Skip often you will also specify a sort order.
```csharp
var query =
(from c in collection.AsQueryable<C>()
    orderby c.X
    select c);
// or
var query =
    collection.AsQueryable<C>()
    .OrderBy(c => c.X)
    .Skip(100)
    .Take(100);
```

**ThenBy**

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

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

**ThenByDescending**

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

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

**Where**

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

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

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

For example, the following queries are equivalent:
```csharp
var query =
    (from c in collection.AsQueryable<C>()
     where c.X > 0
     where c.Y > 0)
    .First(c.Z > 0);
// or
var query =
    (from c in collection.AsQueryable<C>()
     where c.X > 0 && c.Y > 0 && c.Z > 0)
    .First();
```

### Supported where clauses

This section documents the supported where clauses.

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

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

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

#### And (`&&` operator)

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

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

This is translated to the following MongoDB query:

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

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

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

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

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

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

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

matches any document where A has 1 or more items.

This is translated to the following MongoDB query:

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

Boolean constant

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

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

This is translated to the following MongoDB query:

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

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

Boolean field or property

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

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

This is translated to the following MongoDB query:

No translation is needed in this case.
Contains (Enumerable method)

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

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

This is translated to the following MongoDB query:

```json
{ A : 123 }
```

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

Contains (string method)

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

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

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

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

ContainsAll (LINQ to MongoDB extension method)

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

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

This is translated to the following MongoDB query:
ContainsAny (LINQ to MongoDB extension method)

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

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

This is translated to the following MongoDB query:

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

Count method (array length)

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

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

This is translated to the following MongoDB query:

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

Count property (array length)

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

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

This is translated to the following MongoDB query:
**EndsWith (string method)**

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

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

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

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

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

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

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

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

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

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

This is translated to the following MongoDB query:

```json
{ E : 1 }
```

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

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

**In (LINQ to MongoDB extension method)**

The In method is used to test whether a field or property is equal any of a set of provided values.
```csharp
var query =
    from c in collection.AsQueryable<C>()
    where c.X.In(new [] { 1, 2, 3 })
    select c;
// or
var query =
collection.AsQueryable<C>()
 .Where(c.X.In(new [] { 1, 2, 3 }));
```

This is translated to the following MongoDB query:

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

**Inject**

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

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

This is translated to the following MongoDB query:

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

**IsMatch (regular expression method)**

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

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

This is translated to the following MongoDB query:

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

You can also use the static IsMatch method.
```csharp
var query = from c in collection.AsQueryable<C>()
    where Regex.IsMatch(c.S, "^abc")
    select c;
// or
var query =
    collection.AsQueryable<C>()
        .Where(c => Regex.IsMatch(c.S, "^abc"));
```

This is translated to the following MongoDB query:

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

**Length (array length)**

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

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

This is translated to the following MongoDB query:

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

**Mod (% operator)**

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

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

This is translated to the following MongoDB query:

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

**Not (! operator)**

The ! operator is used to reverse the sense of a test.
```csharp
var query = from c in collection.AsQueryable<C>()
    where !(c.X > 1)
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => !(c.X > 1));
```

This is translated into the following MongoDB query:

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

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

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

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

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

This is translated into the following MongoDB query:

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

**Or (|| operator)**

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

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

This is translated to the following MongoDB query:

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

**StartsWith (string method)**

This method is used to test whether a string field or property of the document starts with a particular substring.
```csharp
var query =
    from c in collection.AsQueryable<C>()
    where c.S.StartsWith("abc")
    select c;
// or
var query =
    collection.AsQueryable<C>()
    .Where(c => c.S.StartsWith("abc"));
```

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

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

**CSharp Driver Quickstart**

- Introduction
- Downloading the C# driver
- Add a reference to the C# driver DLLs
- Add required using statements
- Get a reference to the server object
- Get a reference to a database object
- Decide if you want to work with the BsonDocument object model or with your own domain classes
- Get a reference to a collection object
- Insert a document
- Find an existing document
- Save a document
- Update an existing document
- Remove an existing document
- You do NOT need to call Connect or Disconnect
- Full sample program

**Introduction**

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

**Downloading the C# driver**

You can download the C# driver here:

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

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

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

**Add a reference to the C# driver DLLs**

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

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

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

**Add required using statements**

As a minimum you will need the following using statements:
using MongoDB.Bson;
using MongoDB.Driver;

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

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

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

**Get a reference to the server object**

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

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

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

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

**Get a reference to a database object**

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

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

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

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

There are two ways you can work with collections:

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

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

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

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

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

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

**Get a reference to a collection object**

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

Inserting a document is easy:

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

Find an existing document

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

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

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

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

Save a document

You can save changes to an existing document like this:

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

Update an existing document

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

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

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

Remove an existing document

To remove an existing document from a collection you write:

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

You do NOT need to call Connect or Disconnect

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

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using MongoDB.Bson;
using MongoDB.Driver;
using MongoDB.Driver.Builders;

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

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

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

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

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

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

CSharp Driver Serialization Tutorial

- Introduction
- Creating a class map
- Conventions
  - Field or property level serialization options
    - Opt-In
    - Element name
    - Element order
    - Identifying the Id field or property
    - Selecting an IdGenerator to use for an Id field or property
    - Ignoring a field or property
    - Ignoring null values
    - Default values
    - Ignoring a member based on a ShouldSerializeXyz method
    - Identifying required fields
    - Specifying the serializer
    - Serialization Options
Introduction

This document refers to version 1.6 of the C# Driver.

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

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

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

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

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

Creating a class map

To create a class map in your initialization code write:

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

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

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

```csharp
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.

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

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

Conventions

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

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

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

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

```csharp
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
- SetIgnoreIfDefaultValueConvention
- SetIgnoreIfNullConvention
- SetMemberFinderConvention
- SetSerializationOptionsConvention

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 also 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.

**Opt-In**

A majority of classes will have their properties mapped automatically. There are some circumstances where this does not happen. For instance, if your property is read-only, it will not get included in the automapping of a class by default. In order to include the member, you can use the BsonElementAttribute.
public class MyClass {
    private readonly string _someProperty;

    [BsonElement]
    public string SomeProperty {
        get { return _someProperty; }
    }
}

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

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

**Element name**

To specify an element name using attributes, write:

```csharp
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:

```csharp
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:

```csharp
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:

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

### Selecting an IdGenerator to use for an id field or property

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

- `BsonObjectIdGenerator`
- `CombGuidGenerator`
- `GuidGenerator`
- `NullIdChecker`
- `ObjectIdGenerator`
- `StringObjectIdGenerator`
- `ZeroIdChecker<T>`

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

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

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

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

Or using initialization code instead of attributes:

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

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

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

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

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

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

**Ignoring a field or property**

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

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

Or using initialization code instead of attributes:

```csharp
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:

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

Or using initialization code instead of attributes:

```csharp
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:

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

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

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

Or using initialization code instead of attributes:

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

**Ignoring a member based on a ShouldSerializeXyz method**

Sometimes the decision whether to serialize a member or not is more complicated than just whether the value is null or equal to the default value. You can write a method that determines whether a value should be serialized. Usually the method for member Xyz is named ShouldSerializeXyz. If you follow this naming convention then AutoMap will automatically detect the method and use it. For example:

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

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

Or using initialization code instead of naming conventions:

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

**Identifying required fields**

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

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

```csharp
public class MyClass {
    public ObjectId Id { get; set; }
    [BsonRequired]
    public string X { get; set; }
}
```
Or using initialization code instead attributes:

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

### Specifying the serializer

There are times when a specific serializer needs to be used rather than letting the Bson library choose. This can be done in a couple of ways:

```csharp
public class MyClass {
    [BsonSerializer(typeof(MyCustomStringSerializer))]
    public string X { get; set; }
}
```

Or using initialization code instead attributes:

```csharp
BsonClassMap.RegisterClassMap<MyClass>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.X).SetSerializer(new MyCustomStringSerializer());
});
```

### Serialization Options

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

#### DateTimeSerializationOptions

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

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

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

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

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

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

#### DictionarySerializationOptions

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

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

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

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

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

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

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

You can specify a DictionarySerializationOption as follows:

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

Or using initialization code instead of attributes:

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

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

**RepresentationSerializationOptions**

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

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

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

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

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

Class level serialization options

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

**Ignoring extra elements**

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

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

Or using initialization code instead of attributes:

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

**Supporting extra elements**

You can design your class to be capable of handling any extra elements that might be found in a BSON document during deserialization. To do so, you must have a property of type BsonDocument and you must identify that property as the one that should hold any extra elements that are found (or you can name the property “ExtraElements” so that the default ExtraElementsMemberConvention will find it automatically). For example:
public MyClass {
    // fields and properties
    [BsonExtraElements]
    public BsonDocument CatchAll { get; set; }
}

Or using initialization code instead of attributes:

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

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

**Polymorphic classes and discriminators**

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

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

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

**Setting the discriminator value**

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

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

Or using initialization code instead of attributes:

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

**Specifying known types**

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

```csharp
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:

```json
{  _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:

```csharp
[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:

```csharp
BsonClassMap.RegisterClassMap<Animal>(cm => {
    cm.AutoMap();
    cm.SetIsRootClass(true);
});
BsonClassMap.RegisterClassMap<Cat>();
BsonClassMap.RegisterClassMap<Dog>();
BsonClassMap.RegisterClassMap<Lion>();
BsonClassMap.RegisterClassMap<Tiger>();
```
Now that you have identified Animal as a root class, the discriminator values will look a little bit different:

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

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

```csharp
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. Implement ISupportInitialize
2. Make a class responsible for its own serialization
3. Supplementing the default serializer
4. Write a custom serializer
5. Write a custom attribute
6. Write a custom Id generator
7. Write a custom convention

**Implementing ISupportInitialize**

The driver respects an entity implementing ISupportInitialize which contains 2 methods, BeginInit and EndInit. These method are called before deserialization begins and after it is complete. It is useful for running operations before or after deserialization such as handling schema changes are pre-calculating some expensive operations.

**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:

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

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

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

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

**Supplementing the default serializer provider**

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

```csharp
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.

**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.

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

To implement and register a custom serializer you would:

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

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

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

You can also decorate the target class with a BsonSerializer attribute instead of using the BsonSerializer.RegisterSerializer method:

```csharp
[BsonSerializer(typeOf(MyClassSerializer))]
public class MyClass {
}
```

The IBsonSerializer interface is all that is necessary for serialization. However, there are some extension interfaces that will enable further use in other parts of the api such as saving a class or LINQ.

If your class is used as a root document, you will need to implement the IBsonIdProvider interface in order for "Saving" the document to function. MongoCollection.Save requires a document identity in order to know if it should generate an insert or update statement. Below is the extension to the above MyClassSerializer.

```csharp
public class MyClassSerializer : IBsonSerializer, IBsonIdProvider {
    // ...
    // implement GetDocumentId
    // implement SetDocumentId
}
```

In order to enable LINQ to properly construct type-safe queries using a custom serializer, it needs access to member information or array information. If your custom serializer is for a class, as MyClassSerializer is above, then you should implement IBsonDocumentSerializer.

```csharp
public class MyClassSerializer : IBsonSerializer, IBsonDocumentSerializer {
    // ...
    // implement GetMemberSerializationInfo
}
```

If, however, your class is a collection that should be serialized as an array, it should implement IBsonArraySerializer.
public MyClassSerializer : IBsonSerializer, IBsonArraySerializer {
    // ...
    // implement.GetItemSerializationInfo
}

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

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

**Write a custom attribute**

The auto mapping ability of BSON library utilizes attributes that implement IBsonClassMapModifier or IBsonMemberMapModifier for class level attributes or member level attributes respectively. Each of these interfaces have a single method called Apply that is passed a BsonClassMap or a BsonMemberMap with which it can use any of the public methods and properties to modify the map instance. One example of this would be to create an attribute called BsonEncryptionAttribute that is used to encrypt a string before sending it to the database and decrypt it when reading it back out.

View the existing attributes for examples of how these interfaces function.

**Write a custom Id generator**

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

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

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

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

Or using initialization code instead of attributes:

```csharp
BsonClassMap.RegisterClassMap<Employee >((cm => {  // assumes BsonId is a value type
    cm.AutoMap();
    cm.IdMember.SetIdGenerator(new EmployeeIdGenerator());
    }););
```

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

**Write a custom convention**

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

As an example we will write a custom convention to find the Id member of a class (the default convention looks for a member named "Id"). Our custom convention will instead consider any public property whose name ends in "Id" to be the Id for the class. We can implement this convention as follows:
public class EndsWithIdConvention : IIdMemberConvention {
    public string FindIdMember(Type type) {
        foreach (var property in type.GetProperties()) {
            if (property.Name.EndsWith("Id")) {
                return property.Name;
            }
        }
        return null;
    }
}

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

```csharp
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".

**Handling Schema Changes**

Just because MongoDB is schema-less does not mean that your code can handle a schema-less document. Most likely, if you are using a statically typed language like C# or VB.NET, then your code is not-flexible and needs to be mapped to a known schema.

There are a number of different ways that a schema can change from one version of your application to the next.

1. A new member is added
2. A member is deleted
3. A member is renamed
4. The type of a member is changed
5. The representation of a member is changed

How you handle these is up to you. There primary two different strategies.

1. Write an upgrade script.
2. Incrementally update your documents as they are used

The easiest and most bullet-proof of the strategies is to write an upgrade script. There is effectively no difference to this method between a relational database (SQL Server, Oracle) and MongoDB. Identify the documents that need to be changed and update them.

Alternatively, and not supportable in most relational databases, is the incremental upgrade. The idea is that your documents get updated as they are used. Documents that are never used never get updated. Because of this, there are some definite pitfalls you will need to be aware of.

First, queries against a schema where half the documents are version 1 and half the documents are version 2 could go awry. For instance, if you rename an element, then your query will need to test both the old element name and the new element name to get all the results.

Second, any incremental upgrade code must stay in the code-base until all the documents have been upgraded. For instance, if there have been 3 versions of a document, [1, 2, and 3] and we remove the upgrade code from version 1 to version 2, any documents that still exist as version 1 are un-upgradeable.

So, with that being said, let's talk about handling the schema change variations.

**A member has been added**

When a new member is added to an entity, there is nothing that needs to be done other than restarting the application if you are using the auto mapping features. If not, then you will manually need to map the member in the same way all the other members are getting mapped.

Existing documents will not have this element and it will show up in your class with its default value. You can, of course, specify a default value.

**A member has been removed**

When a member has been removed from an entity, it will continue to exist in the documents. The serializer will throw an exception when this element is seen because it doesn't know what to do with it. The 2 previously discussed items that can be used to combat this are the BsonIgnoreExtraElements class-level attribute and the ExtraElements members.
A member is renamed

When a member has been renamed, it will exist in old documents with the old name and in new documents with the new name. The way to handle incremental upgrades for this rename would be to implement an ExtraElements member in conjunction with ISupportInitialize. For example, let's say that a class used to have a Name property which has now been split into a FirstName and a LastName property.

```csharp
public class MyClass : ISupportInitialize {
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [BsonExtraElements]
    public IDictionary<string, object> ExtraElements { get; set; }

    void ISupportInitialize.BeginInit() {
        // nothing to do at begin
    }

    void ISupportInitialize.EndInit() {
        object nameValue;
        if(!ExtraElements.TryGetValue("Name", out nameValue)) {
            return;
        }
        var name = (string)nameValue;
        // remove the Name element so that it doesn't get persisted back to the database
        ExtraElements.Remove("Name");

        // assuming all names are "First Last"
        var nameParts = name.Split(' ');
        FirstName = nameParts[0];
        LastName = nameParts[1];
    }
}
```

The type of a member is changed

If the .NET type is compatible with the old type (an integer is changed to a double), then everything will continue to work. Otherwise, a custom serializer or a migration script will be required.

The representation of a member is changed

If the representation of a member is changed and the representations are compatible, then everything will continue to work. Otherwise, a custom serializer or a migration script will be required.

CSharp Driver Tutorial

- C# Driver version v1.6.x
- 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
C# Driver version v1.6.x

This tutorial is for v1.6.x of the C# Driver. Api documentation can be found here: http://api.mongodb.org/csharp/current/.

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://msysgit.github.com/.

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

```
$ cd <parentdirectory>
$ git config --global core.autocrlf true
$ git clone git://github.com/mongodb/mongo-csharp-driver.git
$ cd mongo-csharp-driver
$ git config core.autocrlf true
```
You must set the global setting for core.autocrlf to true before cloning the repository. After you clone the repository, we recommend you set the local setting for core.autocrlf to true (as shown above) so that future changes to the global setting for core.autocrlf do not affect this repository. If you then want to change your global setting for core.autocrlf to false run:

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

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

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

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

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

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

Building

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

**Dependencies**

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

**Running unit tests**

There are three projects containing unit tests:

1. BsonUnitTests
2. DriverUnitTests
3. DriverUnitTestsVB

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

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

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

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

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

To run the DriverUnitTests and DriverUnitTestsVB 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 copies the DLLs to your specified installation directory.

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

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

**References and namespaces**

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

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

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

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

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

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

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

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

The BSON Library

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

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

**BsonType**

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

```csharp
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.

**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:

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

**As[Type] Properties**

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

```csharp
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. These can be used as follows:

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

**To[Type] conversion methods**

Unlike the As[Type] methods, the To[Type] methods perform some limited conversion between convertible types, like int and double.

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:

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

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

**Static Create methods**

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

**Implicit conversions**

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

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

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

```csharp
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:

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

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

**ObjectId and BsonObjectId**

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

Here are some common ways of creating ObjectId values:

```csharp
var id1 = new ObjectId(); // same as ObjectId.Empty
var id2 = ObjectId.Empty; // all zeroes
var id3 = ObjectId.GenerateNewId(); // generates new unique Id
var id4 = ObjectId.Parse("4dad90129c294e7a5b6aa8"); // parses a 24 hex digit string
```

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

**BsonElement**

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

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

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

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

**BsonDocument constructor**

BsonDocument has the following constructors:

- `BsonDocument()`
- `BsonDocument(string name, BsonValue value)`
- `BsonDocument(BsonElement element)`
- `BsonDocument(Dictionary<string, object> dictionary)`
- `BsonDocument(IDictionary<string, object> dictionary)`
- `BsonDocument(IDictionary dictionary)`
- `BsonDocument(IDictionary dictionary, IEnumerable<string> keys)`
- `BsonDocument(IDictionary<string, object> dictionary)`
- `BsonDocument(IDictionary<string, object> dictionary, IEnumerable<string> keys)`
- `BsonDocument(IDictionary dictionary, IEnumerable<string> keys)`
- `BsonDocument(IBsonElement[] elements)`
- `BsonDocument(bool allowDuplicateNames)`

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:

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

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

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

```csharp
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:

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

The compiler translates this into calls to the matching Add method:
BsonDocument book = new BsonDocument();
   book.Add("author", "Ernest Hemingway");
   book.Add("title", "For Whom the Bell Tolls");

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

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

is translated by the compiler to:

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

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

Creating nested BSON documents

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

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

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

Add methods

BsonDocument has the following overloaded Add methods:

- Add(BsonElement element)
- Add(Dictionary<string, object> dictionary)
- Add(IDictionary<string, object> dictionary, IEnumerable<string> keys)
- Add(IDictionary dictionary)
- Add(IDictionary dictionary, IEnumerable<string> keys)
- Add(IDictionary<string, object> dictionary)
- Add(IDictionary<string, object> dictionary, IEnumerable<string> keys)
- Add(IDictionary<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;
string author = book["author"].AsString;
DateTime publicationDate = book["publicationDate"].AsDateTime;
int pages = book["pages", -1].AsInt32; // default value is -1

BsonArray

This class is used to represent BSON arrays. While arrays happen to be represented externally as BSON documents (with a special naming convention for the elements), the BsonArray class is unrelated to the BsonDocument class because they are used very differently.

Constructors

BsonArray has the following constructors:

- BsonArray()
- BsonArray(IEnumerable<bool> values)
- BsonArray(IEnumerable<BsonValue> values)
- BsonArray(IEnumerable<DateTime> values)
- BsonArray(IEnumerable<double> values)
- BsonArray(IEnumerable<int> values)
- BsonArray(IEnumerable<long> values)
- BsonArray(IEnumerable<ObjectId> values)
- BsonArray(IEnumerable<string> values)
- BsonArray(IEnumerable values)

All the constructors with a parameter call the matching Add method. The multiple overloads are needed because C# does not provide automatic conversions from IEnumerable<T> to IEnumerable<object>.

Add and AddRange methods

Bson Array has the following Add methods:

- BsonArray Add(BsonValue value)
- BsonArray AddRange(IEnumerable<bool> values)
**BsonArray AddRange(IEnumerable<BsonValue> values)**

**BsonArray AddRange(IEnumerable<DateTime> values)**

**BsonArray AddRange(IEnumerable<double> values)**

**BsonArray AddRange(IEnumerable<int> values)**

**BsonArray AddRange(IEnumerable<long> values)**

**BsonArray AddRange(IEnumerable<ObjectId> values)**

**BsonArray AddRange(IEnumerable<string> values)**

**BsonArray AddRange(IEnumerable values)**

Note that the Add method takes a single parameter. To create and initialize a BsonArray with multiple values use any of the following approaches:

```csharp
// 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:

```csharp
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 MongoClient and all the classes from the BSON Library (except BsonSymbolTable which is thread safe). A class is not thread safe unless specifically documented as being thread safe.

All static properties and methods of all classes are thread safe.

**MongoServer class**

This class serves as the root object for working with a MongoDB server. You will create one instance of this class for each server you connect to. The connections to the server are handled automatically behind the scenes (a connection pool is used to increase efficiency).

When you are connecting to a replica set you will still use only one instance of MongoServer, which represents the replica set as a whole. The driver automatically finds all the members of the replica set and identifies the current primary. MongoServer has several properties you can use to find out more about the current state of the replica set (such as Primary, Secondaries, etc...).

Instances of this class are thread safe.

**Connection strings**

The easiest way to connect to a MongoDB server is to use a connection string. The standard connection string format is:

```
mongodb://[username:password@]hostname[:port]/[database]?[options]
```

The username and password should only be present if you are using authentication on the MongoDB server. These credentials will be the default credentials for all databases. To authenticate against the admin database append "(admin)" to the username. If you are using different credentials with different databases pass the appropriate credentials to the GetDatabase method.
The port number is optional and defaults to 27017.

To connect to multiple servers, specify the seed list by providing multiple hostnames (and port numbers if required) separated by commas. For example:

```
mongodb://server1,server2:27017,server2:27018
```

This connection string specifies a seed list consisting of three servers (two of which are on the same machine but on different port numbers). Because specifying multiple servers is ambiguous as to whether or not it is a replica set or multiple mongos (in a sharded setup), the driver will go through a discovery phase of connecting to the servers to determine their type. This has a little overhead at connection time and can be avoided by specifying a connection mode in the connection string:

```
mongodb://server1,server2:27017,server2:27018/?connect=replicaset
```

The available connection modes are automatic (the default), direct, replica set, and shardrouter. The rules for connection mode are as follows:

1. If a connect mode is specified other than automatic, it is used.
2. If a replica set name is specified on the connection string (replicaset), then replicaset mode is used.
3. If there is only one server listed on the connection string, then direct mode is used.
4. Otherwise, discovery occurs and the first server to respond determines the connection mode.

If you have multiple servers listed, and one is part of a replica set and another is not, then the connection mode is non-deterministic. Be sure that you are not mixing server types on the connection string.

Should the connection mode resolve to a replica set, the driver 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). In addition, other secondaries will also be discovered and added (or removed) into the mix automatically, even after initial connection. This will enable you to add and remove servers from the replica set and the driver will handle the changes automatically.

As alluded to above, the options part of the connection string is used to set various connection options. For example, to turn SafeMode on by default for all operations, you could use:

```
mongodb://localhost/?safe=true
```

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

As another example, suppose you wanted to connect directly to a member of a replica set regardless of whether it was the current primary or not (perhaps to monitor its status or to issue read only queries against it). You could use:

```
mongodb://server2/?connect=direct;readpreference=nearest
```

The full documentation for connection strings can be found at:

http://www.mongodb.org/display/DOCS/Connections

and read preferences at:

http://docs.mongodb.org/manual/applications/replication/#replica-set-read-preference

**SSL Support**

Support for SSL is baked into the driver. You can configure this via the connection string by adding an "ssl=true" option to the options.

```
mongodb://server2/?ssl=true
```

By default, the server certificate will get validated against the local trusted certificate store. This sometimes causes issues in test environments where test servers don't have signed certs. To alleviate this issue, you can also add an "sslverifycertificate=false" as another connection string option to ignore any certificate errors.

**Authentication**
MongoDB supports a simple and straightforward authentication mechanism. You can read about it on the [security and authentication docs page](#).

The C# driver supports authentication in a couple of ways. As noted above in connection strings, you can specify default credentials on the connection string. The default credentials are always used as a fallback if no other credentials are supplied.

Supplying credentials can be done in two ways. First, they can be supplied to certain methods at runtime. These credentials will then be used to execute the desired functionality. The other, and more robust way, is to store credentials in a MongoCredentialsStore. MongoCredentials in the store are keyed by database, so if different databases require different users, then the credentials store is consulted first and, upon a miss, will fallback to the default credentials supplied on the connection string if they exist.

The example below uses the credential store to define admin credentials and credentials for the "foo" database. Access to databases other than "admin" or "foo" will use the connection string supplied default credentials "test".

```csharp
var url = MongoUrl.Create("mongodb://test:user@localhost:27017/?safe=true");
var settings = url.ToServerSettings();
var adminCredentials = new MongoCredentials("admin", "user", true);
settings.CredentialsStore.Add("admin", adminCredentials);
var fooCredentials = new MongoCredentials("foo", "user", false);
settings.CredentialsStore.Add("foo", fooCredentials);
var server = MongoServer.Create(settings);
```

Create method

To obtain an instance of MongoServer use one of the Create methods:

- `MongoServer Create()`
- `MongoServer Create(MongoServerSettings settings)`
- `MongoServer Create(MongoUrl url)`
- `MongoServer Create(string connectionString)`
- `MongoServer Create(Uri uri)`

For example:

```csharp
string connectionString = "mongodb://localhost";
MongoServer server = MongoServer.Create(connectionString);
```

Create maintains a table of MongoServer instances it has returned before, so if you call Create again with the same parameters you get the same instance back again.

The recommended way to call Create is with a connection string in the MongoDB URL format. MongoConnectionStringBuilder is provided for compatibility with how .NET handles SQL Server connection strings, but we recommend you use the URL format instead.

GetDatabase method

You can navigate from an instance of MongoServer to an instance of MongoDatabase (see next section) using one of the following GetDatabase methods or indexers:

- `MongoDatabase GetDatabase(MongoDatabaseSettings settings)`
- `MongoDatabase GetDatabase(string databaseName)`
- `MongoDatabase GetDatabase(string databaseName, MongoCredentials credentials)`
- `MongoDatabase GetDatabase(string databaseName, MongoCredentials credentials, SafeMode safeMode)`
- `MongoDatabase GetDatabase(string databaseName, SafeMode safeMode)`

Sample code:

```csharp
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:
GetDatabase maintains a table of MongoDatabase instances it has returned before, so if you call GetDatabase again with the same parameters you get the same instance back again.

**RequestStart/RequestDone methods**

Sometimes a series of operations needs to be performed on the same connection in order to guarantee correct results. This is rarely the case, and most of the time there is no need to call RequestStart/RequestDone. An example of when this might be necessary is when a series of Inserts are called in rapid succession with SafeMode off, and you want to query that data is in a consistent manner immediately thereafter (with SafeMode off the writes can queue up at the server and might not be immediately visible to other connections). Using RequestStart you can force a query to be on the same connection as the writes, so the query won't execute until the server has caught up with the writes.

A thread can temporarily reserve a connection from the connection pool by using RequestStart and RequestDone. For example:

```csharp
using (server.RequestStart(database)) {
    // a series of operations that must be performed on the same connection
}
```

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.

RequestStart increments a counter (for this thread) which is decremented upon completion. 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 can be nested and the right thing will happen.

> RequestStart returns an IDisposable. If you do no use RequestStart with a using block, it is imperative that RequestDone be called in order to release the connection.

**Other properties and methods**

For a reference of other properties and method, see the api documentation.

**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:

```csharp
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:

```csharp
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

For a reference of other properties and method, see the api documentation.

**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<TDefaultDocument> 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:

```csharp
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:

```csharp
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:

```csharp
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:

```csharp
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:

```csharp
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:

```csharp
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):

```csharp
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:

```csharp
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:

```csharp
MongoCollection<BsonDocument> books;
var query = Query<Book>.EQ(b => b.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 Cradle")
);  
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:

```csharp
MongoCollection<BsonDocument> books;
var query = new QueryDocument {
    { "author", "Kurt Vonnegut" },
    { "title", "Cats Cradle" }
};
var update = new UpdateDocument {
    { "$set", new BsonDocument("title", "Cat's Cradle") }
};
BsonDocument updatedBook = books.Update(query, update);
```

or using Query and Update builders:

```csharp
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:

```csharp
```
```csharp
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.
    var.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.

```csharp
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**

For a reference of other properties and method, see the api documentation.

**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:
You can also use any of the extensions methods defined by LINQ for IEnumerable<T> to enumerate a cursor:

```csharp
var query = Query.EQ("author", "Ernest Hemingway");
var cursor = books.Find(query);
foreach (var book in cursor) {
    // do something with book
}
```

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:

```csharp
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:

```csharp
var query = Query.EQ("status", "pending");
foreach (var task in tasks.Find(query).SetSkip(100).SetLimit(10)) {
    // do something with task
}
```

The fluent interface works well when you are setting only a few values. When setting more than a few you might prefer to use the properties approach.

Once you begin enumerating a cursor it becomes “frozen” and you can no longer change any of its properties. So you must set all the properties before you start enumerating it.

Modifiable properties of a cursor

The following properties of a cursor are modifiable:

- BatchSize (SetBatchSize)
- Fields (SetFields)
- Flags (SetFlags)
- Limit (SetLimit)
- Options (SetOption and SetOptions)
- SerializationOptions (SetSerializationOptions)
- Skip (SetSkip)
- SlaveOk (SetSlaveOk)

The method names in parenthesis are the corresponding fluent interface methods.
The fluent interface also supports additional options that aren't used very frequently and are not exposed as properties:

- SetHint
- SetMax
- SetMaxScan
- SetMin
- SetShowDiskLoc
- SetSnapshot
- SetSortOrder

Other methods

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

- Clone
- Count
- Explain
- Size

SafeMode class

There are various levels of SafeMode, and this class is used to represent those levels. SafeMode applies only to operations that don't already return a value (so it doesn't apply to queries or commands). It applies to the following MongoCollection methods: Insert, Remove, Save and Update.

The gist of SafeMode is that after an Insert, Remove, Save or Update message is sent to the server it is followed by a GetLastErr command so the driver can verify that the operation succeeded. In addition, when using replica sets it is possible to verify that the information has been replicated to some minimum number of secondary servers.

The SafeMode class is no longer immutable. The properties have been made settable to facilitate creation of new instances using object initializer syntax. While it is no longer immutable, SafeMode instances can now be "frozen" to make them immutable at run time. A SafeMode instance is not thread safe until it has been frozen, at which point it becomes thread safe.

CSharp getLastError and SafeMode

In the C# driver SafeMode can be set at different levels.

1. At the server level via the connection string:

   ```csharp
   var connectionString = "mongodb://hostname/?safe=true;w=2;wtimeout=30s";
   var server = MongoServer.Create(connectionString);
   ```

2. At the database level:

   ```csharp
   var safemode = SafeMode.W2; // default timeout
   // or
   var safeMode = SafeMode.Create(2, TimeSpan.FromSeconds(30)); // 30 second timeout
   var database = server.GetDatabase("test", safeMode);
   ```

3. At the collection level:

   ```csharp
   var collection = database.GetCollection("test", safeMode);
   ```

4. At the operation level:

   ```csharp
   var safeModeResult = collection.Insert(document, safeMode);
   ```

   Each level inherits the setting from the level above it unless overridden.

getLastError is called automatically when any SafeMode other than false is used. An exception is thrown if there was an error, otherwise the SafeModeResult has the information returned by GetLastErr.
See Also

- Connections
- getLastError Command
- Replica Set Design Concepts

Erlang Language Center

- Driver Download
  - [https://github.com/mongodb/mongodb-erlang](https://github.com/mongodb/mongodb-erlang)
- API Docs
  - Design of the Erlang Driver post on blog.mongodb.org

Third Party Frameworks and Libs

- Mongrel - A record/document mapper that maps Erlang records to MongoDB documents
  - [API Documentation for Mongrel](https://github.com/mongodb/mongodb-erlang)

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.

<table>
<thead>
<tr>
<th>JavaScript</th>
<th>Python</th>
<th>PHP</th>
<th>Ruby</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>[]</td>
<td>array()</td>
<td>[]</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
<td>new stdClass</td>
<td>{}</td>
</tr>
<tr>
<td>{x: 1}</td>
<td>&quot;x&quot;: 1</td>
<td>array('x' =&gt; 1)</td>
<td><code>{x' =&gt; 1}</code></td>
</tr>
<tr>
<td>connect(&quot;www.example.net&quot;)</td>
<td>Connection(&quot;www.example.net&quot;)</td>
<td>new Mongo(&quot;www.example.net&quot;)</td>
<td>Connection.new(&quot;www.example.net&quot;)</td>
</tr>
<tr>
<td>cursor.next()</td>
<td>cursor.next()</td>
<td>$cursor-&gt;getNext()</td>
<td>cursor.next_document()</td>
</tr>
<tr>
<td>cursor.hasNext()</td>
<td>*</td>
<td>$cursor-&gt;hasNext()</td>
<td>cursor.has_next?</td>
</tr>
<tr>
<td>collection.findOne()</td>
<td>collection.find_one()</td>
<td>$collection-&gt;findOne()</td>
<td>collection.find_one()</td>
</tr>
<tr>
<td>db.eval()</td>
<td>db.eval()</td>
<td>$db-&gt;execute()</td>
<td>db.eval()</td>
</tr>
</tbody>
</table>

* 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](https://github.com/mongodb/mongodb-erlang).
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.

This is an overview of the available tools and suggested practices for using Node.js with MongoDB. Those wishing to skip to more detailed discussion should check out the The Node.js Driver Manual.

- Node.js Driver
  - Installing / Upgrading
  - Object Mappers
  - Presentations
    - 3rd Party Drivers

Node.js Driver

The MongoDB Node.js driver is the 10gen-supported driver for MongoDB. In spring 2012 10gen officially adopted the popular Node MongoDB Native Project and sponsored the maintainer, Christian Kvalheim, to continue its development. It’s written in pure javascript and provides a native asynchronous node.js interface to MongoDB. 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 Mongoose.

- Tutorial
- Node.js Driver README
- Source Code

Installing / Upgrading

The easiest way to install is to use NPM:

```
$ npm install mongodb
```

Object Mappers

Because MongoDB is so easy to use, the basic Node.js driver can be the best solution for many applications.

If you need validations, associations, and other high-level data modeling functions then an Object Document Mapper may be helpful.

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:

- Mongoskin - The future layer for node-mongodb-native.
- Mongoose - Asynchronous JavaScript Driver with optional support for Modeling.
- Mongolia - Lightweight MongoDB ORM/Driver Wrapper.

All the mappers build on top of the native Node.js driver and so some knowledge of that is useful, especially if you work with a custom Mongo configuration.

Presentations

- An introduction to the mongo node.js driver - June 2011
- Using MongoDB with node.js - June 2011
- Node.js & MongoDB - Webinar June 2011
- A beautiful marriage: MongoDB and node.js - MongoNYC June 2011
- Rapid Realtime App Development with Node.JS & MongoDB - MongoSF May 2011

3rd Party Drivers
A few 3rd party drivers exist and are under rapid development. While not officially supported by 10gen, these drivers take a different approach that may be valuable given your needs.

- node-mongodb - Async Node interface to MongoDB (written in C)
- Mongolian DeadBeef - A node.js driver that attempts to closely approximate the MongoDB shell.

**JVM Languages**

---

**Redirection Notice**
This page should redirect to [Java Language Center](#) in about 3 seconds.

---

**Python Language Center**

This is an overview of the available tools for using Python with MongoDB. Those wishing to skip to more detailed discussion should check out the [Python Driver Tutorial](#).

- Python Driver
- Python tools
  - ORM Like Layers
  - Framework Tools
- Alternative drivers
- Presentations
- Tutorials

**Python Driver**

**PyMongo** is the recommended way to work with MongoDB from Python.

- Installation
- Tutorial
- API Documentation
- Changelog
- Source Code

**Python tools**

**ORM Like Layers**

Because MongoDB is so easy to use the basic Python driver is often the best solution for many applications. However, if you need data validation, associations and other high-level data modeling functionality then ORM like layers may be desired.

- ORM like layers

**Framework Tools**

Several tools and adapters for integration with various Python frameworks and libraries also exist.

- Framework Tools

**Alternative drivers**

- Alternative driver list

**Presentations**

- [MongoDB & Python](#) - Workshop materials from PyCon 2012
- [PyCon Poster](#) - PyCon 2012
- [Realtime Analytics using MongoDB, Python, Gevent, and ZeroMQ](#) - Rick Copeland's presentation from Mongo Seattle (December 2011)
- [MongoDB with Python, Pylons, and Pyramid](#) - Niall O'Higgins' presentation from MongoSF (May 2011)
- [Python Development with MongoDB](#) - Bernie Hackett's presentation from MongoSF (May 2011)
- [Building a Social Graph with MongoDB at Eventbrite](#) - Brian Zambrano's presentation from MongoSV (December 2010)
- More Python-related presentations

**Tutorials**
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. 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 Uls
- If you are using Eclipse, you can get Content Assist working by downloading the mongo_version.zip package.

Installing the PHP Driver
PHP Libraries, Frameworks, and Tools

- Libraries and Frameworks
  - CakePHP
  - Codeigniter
  - Doctrine
  - Drupal
  - Fat-Free Framework
  - Kohana Framework
  - Lithium
  - Symfony 2
  - TechMVC
  - Thundergrid
  - YII
  - Zend Framework
- Stand-Alone Tools
  - Autocomplete for IDEs
  - ActiveMongo
  - Comfi
  - MapReduce API
  - Mongofilesystem
  - Mandango
  - MongoDB Pagination
  - MongoDB PHP ODM
  - Mongodloid
  - MongoQueue
  - MongoRecord
  - Morph
  - simplemongophp
- Blogs & HOWTOs
  - How to batch import JSON data output from FFprobe for motion stream analysis

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

**Libraries and Frameworks**

**CakePHP**

- MongoDB datasource for CakePHP. There's also an introductory blog post on using it with Mongo.

**Codeigniter**

- MongoDB-Codeigniter-Driver

**Doctrine**

ODM (Object Document Mapper) is an experimental Doctrine MongoDB object mapper. The Doctrine\ODM\Mongo namespace is an experimental project for a PHP 5.3 MongoDB Object Mapper. It allows you to easily write PHP 5 classes and map them to collections in MongoDB. You just work with your objects like normal and Doctrine will transparently persist them to Mongo.

This project implements the same "style" of the Doctrine 2 ORM project interface so it will look very familiar to you and it has lots of the same features and implementations.

- Documentation - API, Reference, and Cookbook
- Official blog post
- Screencast
- Blog post on using it with Symfony
- Bug tracker

**Drupal**

- MongoDB Integration - Views (query builder) backend, a watchdog implementation (logging), and field storage.
Fat-Free Framework

Fat-Free is a powerful yet lightweight PHP 5.3+ Web development framework designed to help you build dynamic and robust applications - fast!

Kohana Framework

- Mango at github
  An ActiveRecord-like library for PHP, for the Kohana PHP Framework. See also PHP Language Center#MongoDB PHP ODM further down.

Lithium

Lithium supports Mongo out-of-the-box.

- Tutorial on creating a blog backend.

Symfony 2

- Symfony 2 Logger
  A centralized logger for Symfony applications. See the blog post.

- sfMongoSessionStorage - manages session storage via MongoDB with symfony.

- sfStoragePerformancePlugin - This plugin contains some extra storage engines (MongoDB and Memcached) that are currently missing from the Symfony (>= 1.2) core.

TechMVC


Thundergrid

A GridFS Framework for PHP

Thundergrid is a simple framework written in PHP that allows you to rapidly store files in your Mongo database using the GridFS specification.

Using Thundergrid gives you the ability to control exactly how you use GridFS in your scripts, it allows you to list, filter and display objects within GridFS quickly and rapidly.

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
**Autocomplete for IDEs**

**PeclMongoPhpDoc** gives IDEs information about the Pecl extension to help with autocomplete & docs.

**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](http://crodas.org/activemongo.php).

**Comfi**

**Comfi** is a Mongo PHP ORM (part of Comfi.com Homebase Framework)

**MapReduce API**

A MapReduce abstraction layer. See the [blog post](http://http://crodas.org/activemongo.php).

- [MongoDB-MapReduce-PHP](http://http://crodas.org/activemongo.php) at github

**Mongofilesystem**

Filesystem based on MongoDB GridFS. **Mongofilesystem** will help you use MongoDB GridFS like a typical filesystem, using the familiar PHP commands.

**Mandango**

**Mandango** is a simple, powerful and ultrafast Object Document Mapper (ODM) for PHP and MongoDB.

**MongoDB Pagination**

PHP [MongoDB Pagination](http://http://crodas.org/activemongo.php) is the pagination plugin for MongoDB released under MIT License. Simple to install & use. It has been developed under TechMVC 3.0.4, but it's compatible with any 3rd party framework (e.g. Zend (tested)).

**MongoDb PHP ODM**

**MongoDb PHP ODM** is a simple object wrapper for the Mongo PHP driver classes which makes using Mongo in your PHP application more like ORM, but without the suck. It is designed for use with Kohana 3 but will also integrate easily with any PHP application with almost no additional effort.

**Mongodloid**

A nice library on top of the PHP driver that allows you to make more natural queries (`$query->query('a == 13 AND b >= 8 && c % 3 == 4')`), abstracts away annoying $-syntax, and provides getters and setters.

- [Project Page](http://http://crodas.org/activemongo.php)
- [Downloads](http://http://crodas.org/activemongo.php)
- [Documentation](http://http://crodas.org/activemongo.php)

**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](http://http://crodas.org/activemongo.php). 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](http://http://crodas.org/activemongo.php). Oodle’s requirements for a manageable, easy to understand interface for dealing with the super-scalable Mongo datastore was the primary reason for MongoRecord. It was developed to use with PHP applications looking to add Mongo’s scaling capabilities while dealing with a nice abstraction layer.

**Morph**

A high level PHP library for MongoDB. Morph comprises a suite of objects and object primitives that are designed to make working with MongoDB in PHP a breeze.

- [Morph at github](http://http://crodas.org/activemongo.php)
simplemongophp

Very simple layer for using data objects see blog post

  simplemongophp at github

**Blogs & HOWTOs**

*How to batch import JSON data output from FFprobe for motion stream analysis*

FFprobe is a stream analyzer that optionally reports in JSON. This example is a PHP script that reads JSON from STDIN, makes an object using `json_decode`, and inserts the object into a MongoDB database. This script could be used with any program that outputs a JSON stream. A bash script will be used to batch process all files within the current directory. For example, the data may be used for analysis and logging of a day's shoot.

  - Batch import Multimedia Stream Data into MongoDB with FFprobe web site
  - Code and Sample Output at github

**PHP - Storing Files and Big Data**

*Redirection Notice*


**Troubleshooting the PHP Driver**

*Redirection Notice*


**Ruby Language Center**

This is an overview of the available tools and suggested practices for using Ruby with MongoDB. Those wishing to skip to more detailed discussion should check out the Ruby Driver Tutorial, Getting started with Rails or Rails 3, and MongoDB Data Modeling and Rails. There are also a number of good external resources worth checking out.

  - Ruby Driver
    - Installing / Upgrading
    - BSON
  - Object Mappers
  - Notable Projects
  - Presentations

**Ruby Driver**

Install the bson_ext gem for any performance-critical applications.

*The MongoDB Ruby driver* is the 10gen-supported driver for MongoDB. It's written in pure Ruby, with a recommended C extension for speed. The driver is optimized for simplicity. It can be used on its own, but it also serves as the basis of several object mapping libraries, such as MongoMapper.

  - Tutorial
  - Ruby Driver README
  - API Documentation
  - Source Code

**Installing / Upgrading**

The ruby driver is hosted at [Rubygems.org](http://rubygems.org). Before installing the driver, make sure you're using the latest version of rubygems (currently 1.6.0 as of Feb 2012):
$ gem update --system

Then install the gems:

$ gem install mongo

To stay on the bleeding edge, check out the latest source from github:

$ git clone git://github.com/mongodb/mongo-ruby-driver.git
$ cd mongo-ruby-driver/

Then, install the driver from there:

$ rake gem:install

**BSON**

In versions of the Ruby driver prior to 0.20, the code for serializing to BSON existed in the mongo gem. Now, all BSON serialization is handled by the required bson gem.

```
gem install bson
```

For significantly improved performance, install the bson_ext gem. Using compiled C instead of Ruby, this gem speeds up BSON serialization greatly.

```
gem install bson_ext
```

If you're running on Windows, you'll need the Ruby DevKit installed in order to compile the C extensions.

As long it's in Ruby's load path, bson_ext will be loaded automatically when you require bson.

Note that beginning with version 0.20, the mongo_ext gem is no longer used.

To learn more about the Ruby driver, see the [Ruby Tutorial](#).

**Object Mappers**

Because MongoDB is so easy to use, the basic Ruby driver can be the best solution for many applications.

But if you need validations, associations, and other high-level data modeling functions then an Object Document Mapper may be needed.

In the context of a Rails application these provide functionality equivalent to, but distinct from, ActiveRecord. Because Mongo is a document-based database, these mappers are called Object Document Mappers (ODM) as opposed to Object Relational Mappers (ORM).

Several mappers are available:

- [MongoMapper](#) from John Nunemaker
- [Mongoid](#) from Durrant Jordan
- [Mongomatic](#) from Ben Myles
- [MongoODM](#) from Carlos Paramio
- [MongoModel](#) from Sam Pohlenz
- [DriverAPI](#) from Alexey Petrushin

All the mappers build on top of the basic Ruby driver and so some knowledge of that is useful, especially if you work with a custom Mongo configuration.

**Notable Projects**

Tools for working with MongoDB in Ruby are being developed daily. A partial list can be found in the Projects and Libraries section of our external resources page.

If you're working on a project that you'd like to have included, let us know.
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**
  - 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](http://api.mongodb.org/ruby/current/).
A Quick Tour

Using the RubyGem

All of the code here assumes that you have already executed the following Ruby code:

```ruby
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:

```ruby
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

```ruby
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

```ruby
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:

```ruby
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):

```ruby
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:

```ruby
coll = db.collection("testCollection")
```

This is aliased to the `[]` method:

```ruby
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, let's make a little document that in JSON would be represented as

```json
{
  "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.

```ruby
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:

```ruby
doc["name"] = "MongoDB Ruby"
coll.update({"_id" => doc["_id"]}, doc)
```

Or we can use an atomic operator to change a single value:

```ruby
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).

```ruby
my_doc = coll.find_one()
puts my_doc.inspect
```

and you should see:
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:

```json
{
    "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:

```json
{"_id"=>"_id" => #<BSON::ObjectID:0x117de90 ...>, "i" => 71}
```

Getting a Set of Documents With a Query

We can use the query to get a set of documents from our collection. For example, if we wanted to get all documents where "i" > 50, we could write:

```
coll.find("i" => {"$gt" => 50}).each { |row| puts row }
```
which should print the documents where $i > 50. We could also get a range, say $20 < i <= 30:

```
coll.find("i" => {$gt => 20, $lte => 30}).each { |row| puts row }
```

**Selecting a subset of fields for a query**

Use the :fields option. If you just want fields "a" and "b":

```
coll.find("i" => {$gt => 50}, :fields => ['a', 'b']).each { |row| puts row }
```

**Querying with Regular Expressions**

Regular expressions can be used to query MongoDB. To find all names that begin with 'a':

```
coll.find({"name" => /^a/})
```

You can also construct a regular expression dynamically. To match a given search string:

```
search_string = params['search']
# Constructor syntax
coll.find({"name" => Regexp.new(search_string)})
# Literal syntax
coll.find({"name" => /(#{search_string})/})
```

Although MongoDB isn't vulnerable to anything like SQL-injection, it may be worth checking the search string for anything malicious.

**Creating An Index**

MongoDB supports indexes, and they are very easy to add on a collection. To create an index, you specify an index name and an array of field names to be indexed, or a single field name. The following creates an ascending index on the "i" field:

```
# create_index assumes ascending order; see method docs
# for details
coll.create_index("i")
```

To specify complex indexes or a descending index you need to use a slightly more complex syntax - the index specifier must be an Array of [field name, direction] pairs. Directions should be specified as Mongo::ASCENDING or Mongo::DESCENDING:

```
# explicit "ascending"
coll.create_index(["i", Mongo::ASCENDING])
```

**Creating and querying on a geospatial index**

First, create the index on a field containing long-lat values:

```
people.create_index(["loc", Mongo::GEO2D])
```

Then get a list of the twenty locations nearest to the point 50, 50:

```
people.find("loc" => {$near => [50, 50]}, {:limit => 20}).each do |p|
  puts p.inspect
end
```
Getting a List of Indexes on a Collection

You can get a list of the indexes on a collection using `coll.index_information()`.

Database Administration

A database can have one of three profiling levels: off (:off), slow queries only (:slow_only), or all (:all). To see the database level:

```
puts db.profiling_level   # => off (the symbol :off printed as a string)
db.profiling_level = :slow_only
```

Validating a collection will return an interesting hash if all is well or raise an exception if there is a problem.

```
p db.validate_collection('coll_name')
```

See Also

- Ruby Driver Official Docs
- MongoDB Koans A path to MongoDB enlightenment via the Ruby driver.
- MongoDB Manual

Replica Sets in Ruby

⚠️ Redirection Notice
This page should redirect to [http://api.mongodb.org/ruby/current/file.REPLICA_SETS.html](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:

```ruby
# 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 (>= 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:

```ruby
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](#)
- [Replica Set Configuration](#)

### GridFS in Ruby

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
**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:

```ruby
@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:

```ruby
# 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:

```ruby
# 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:

```ruby
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

```ruby
file = File.open("me.jpg")
id2 = @grid.put(file,
    :filename => "my-avatar.jpg",
    :content_type => "application/jpg",
    :_id => 'a-unique-id-to-use-in-lieu-of-a-random-one',
    :chunk_size => 100 * 1024,
    :metadata => {
        'description' => "taken after a game of ultimate"
    })
```

Safe mode

A kind of safe mode is built into the GridFS specification. When you save a file, and MD5 hash is created on the server. If you save the file in safe mode, an MD5 will be created on the client for comparison with the server version. If the two hashes don't match, an exception will be raised.

```ruby
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:

```ruby
@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

```ruby
@db = Mongo::Connection.new.db("social_site")
@fs = GridFileSystem.new(@db)
```

Now suppose we want to save the file 'me.jpg.' This is easily done using a filesystem-like API:

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

We can then retrieve the file by filename:

```ruby
image = @fs.open("me.jpg", "r") { |f| f.read }
```

No problems there. But what if we need to replace the file? That too is straightforward:

```ruby
image = File.open("me-dancing.jpg")
@fs.open("me.jpg", "w") do |f|
    f.write image
end
```
But a couple things need to be kept in mind. First is that the original 'me.jpg' will be available until the new 'me.jpg' saves. From then on, calls to
the #open method will always return the most recently saved version of a file. But, and this the second point, old versions of the file won't be
deleted. So if you're going to be rewriting files often, you could end up with a lot of old versions piling up. One solution to this is to use the
:delete_old options when writing a file:

```ruby
image = File.open("me-dancing.jpg")
@fs.open("me.jpg", "w", :delete_old => true) do |f|
  f.write image
end
```

This will delete all but the latest version of the file.

Deleting files

When you delete a file by name, you delete all versions of that file:

```ruby
@fs.delete("me.jpg")
```

Metadata and safe mode

All of the options for storing metadata and saving in safe mode are available for the GridFileSystem class:

```ruby
image = File.open("me.jpg")
@fs.open('my-avatar.jpg', w,
  :content_type => "application/jpg",
  :metadata     => {'description' => "taken on 3/1/2010 after a game of ultimate"},
  :_id          => 'a-unique-id-to-use-instead-of-the-automatically-generated-one',
  :safe         => true) { |f| f.write image }
```

Advanced Users

Astute code readers will notice that the Grid and GridFileSystem classes are merely thin wrappers around an underlying GridIO class. This means
that it's easy to customize the GridFS implementation presented here; just use GridIO for all the low-level work, and build the API you need in an
external manager class similar to Grid or GridFileSystem.

Rails - Getting Started

Using Rails 3? See Rails 3 - Getting Started

This tutorial describes how to set up a simple Rails application with MongoDB, using MongoMapper as an object mapper. We assume you're
using Rails versions prior to 3.0.

- 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:

```bash
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.

```ruby
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.

```ruby
config.frameworks -= [:active_record]
```

5. Add MongoMapper to the environment. This can be done by opening config/environment.rb and adding the line:

```ruby
config.gem 'mongo_mapper'
```

Once you've done this, you can install the gem in the project by running:

```bash
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.

```ruby
# Drop all columns after each test case.
def teardown
  MongoMapper.database.collections.each do |coll|
    coll.remove
  end
end

# Make sure that each test case has a teardown method to clear the db after each test.
def inherited(base)
  base.define_method teardown do
    super
  end
end
```

This way, all test classes will automatically invoke the teardown method. In the example above, the teardown method clears each collection. We might also choose to drop each collection or drop the database as a whole, but this would be considerably more expensive and is only necessary if our tests manipulate indexes.

Usually, this code is added in test/test_helper.rb. See the aforementioned rails template for specifics.

**Coding**

If you've followed the foregoing steps (or if you've created your Rails with the provided template), then you're ready to start coding. For help on that, you can read about modeling your domain in Rails.
Rails 3 - Getting Started

It’s not difficult to use MongoDB with Rails 3. Most of it comes down to making sure that you're not loading ActiveRecord and understanding how to use Bundler, the new Ruby dependency manager.

- Install the Rails 3
- Configure your application
  - Bundle and Initialize
    - Bundling
    - Initializing
- Running Tests
- Conclusion

Install the Rails 3

If you haven’t done so already, install Rails 3.

```shell
# 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:

```shell
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:

```ruby
require "rails/all"
```

to this:

```ruby
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:

```ruby
# Configure generators values. Many other options are available, be sure to check the documentation.
# config.generators do |g|
#   g.orm :active_record
#   g.template_engine :erb
#   g.test_framework :test_unit, :fixture => true
# end
```

As of this this writing, it's commented out by default, so you probably won't have to change anything here.

Bundle and Initialize

The final step involves bundling any gems you'll need and then creating an initializer for connecting to the database.

Bundling

Edit `Gemfile`, located in the Rails root directory. By default, our `Gemfile` will only load Rails:

```ruby
# Gemfile
```
Normally, using MongoDB will simply mean adding whichever OM framework you want to work with, as these will require the "mongo" gem by default.

```ruby
# 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:

```ruby
# 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:

```bash
bundle install
```

### Initializing

Last item is to create an initializer to connect to MongoDB. Create a Ruby file in `config/initializers` with the following:

```ruby
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:

```ruby
Running Tests
```

A slight modification is required to get rake test working (thanks to John P. Wood). Create a file `lib/tasks/mongo.rake` containing the following:
namespace :db do
  namespace :test do
    task :prepare do
      # Stub out for MongoDB
    end
  end
end

Now the various `rake test` tasks will run properly. See John's post for more details.

**Conclusion**

That should be all. You can now start creating models based on whichever OM you've installed.

**MongoDB Data Modeling and Rails**

This tutorial discusses the development of a web application on Rails and MongoDB. MongoMapper will serve as our object mapper. The goal is to provide some insight into the design choices required for building on MongoDB. To that end, we'll be constructing a simple but non-trivial social news application. The source code for newsmonger is available on github for those wishing to dive right in.

- 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:

```ruby
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:

```shell
db.stories.ensureIndex('voters');
```

Or, using MongoMapper, we can specify the index in `config/initializers/database.rb`:

```ruby
Story.ensure_index(:voters)
```

To find all the stories voted on by a given user:

```ruby
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:

```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:

```ruby
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:

```ruby
@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:

```ruby
@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:

```ruby
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 Diroll'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 mongod 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**

 خطأ توجيه: هذه الصفحة يجب أن توجه إلى 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:

```ruby
# 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:

```ruby
@collection = @db['users']
result = @collection.find({:name => "jones"}).explain
```

The resulting explain plan might look something like this:

```yaml
{ "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":[] }
```

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:

```ruby
@collection = @db['test']
boat_id = @collection.save({:vehicle  => :boat})
car_id  = @collection.save({})

@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`:

   ```ruby
   @collection.find({}, :timeout => false) do |cursor|
     cursor.each do |document|
       # Process documents here
     end
   end
   ```

I periodically see connection failures between the driver and MongoDB. Why can’t the driver retry the operation automatically?

A connection failure can indicate any number of failure scenarios. Has the server crashed? Are we experiencing a temporary network partition? Is there a bug in our ssh tunnel?

Without further investigation, it’s impossible to know exactly what has caused the connection failure. Furthermore, when we do see a connection failure, it’s impossible to know how many operations prior to the failure succeeded. Imagine, for instance, that we’re using safe mode and we send an `$inc` operation to the server. It’s entirely possible that the server has received the `$inc` but failed on the call to `getLastError`. In that case, retrying the operation would result in a double-increment.

Because of the indeterminacy involved, the MongoDB drivers will not retry operations on connection failure. How connection failures should be handled is entirely dependent on the application. Therefore, we leave it to the application developers to make the best decision in this case.

The drivers will reconnect on the subsequent operation.

Java Language Center

Java Driver

- Java Driver
  - Basics
  - Specific Topics and How-To
- Third Party Frameworks and Libs
  - POJO Mappers
  - Code Generation
  - Misc
- Clojure
- Groovy
- JavaScript (Rhino)
- JRuby
- Scala
- Hadoop
- Presentations

Basics

- Download the Java Driver
- Tutorial
- API Documentation
- Release Notes

Specific Topics and How-To

- Concurrency
• Saving Objects
• Data Types
• Aggregation Framework
• Read Preferences and Tagging

Third Party Frameworks and Libs

POJO Mappers

• Morphia - Type-Safe Wrapper with DAO/Datastore abstractions
• Mungbean (w/ clojure support)
• Spring MongoDB – Provides Spring users with a familiar data access features including rich POJO mapping.
• DataNucleus JPA/JDO – JPA/JDO wrapper
• lib-mongomapper JavaBean Mapper (No annotations)
• mongo-jackson-mapper Uses Jackson (annotations) to map to/from POJOs and has a simple wrapper around DBCollection to simply this.
• Kundera - JPA compliant ORM. Works with multiple datastores.
• Jongo - Query in Java as in Mongo shell (using strings), unmarshall results into Java objects (using Jackson)

Code Generation

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

Misc

• log4mongo a log4j appender
• mongo-java-logging a Java logging handler
• (Experimental, Type4) JDBC driver
• Metamodel data exploration and querying library
• Mongodb Java REST server based on Jetty

Clojure

• Congo Mongo
• monger

Groovy

• Groovy Tutorial for MongoDB
• MongoDB made more Groovy
• GMongo, a Groovy wrapper to the mongodb Java driver
  • GMongo 0.5 Release Writeup

JavaScript (Rhino)

• MongoDB-Rhino - A toolset to provide full integration between the Rhino JavaScript engine for the JVM and MongoDB. Uses the MongoDB Java driver.

JRuby

• jmongo A thin ruby wrapper around the mongo-java-driver for vastly better jruby performance.

If there is a project missing here, just add a comment or email the list and we'll add it.

Scala

• Scala Language Center

Hadoop

• Hadoop

Presentations

• Building a Mongo DSL in Scala at Hot Potato - Lincoln Hochberg's Presentation from MongoSF (April 2010)
• Java Development - Brendan McAdams' Presentation from MongoNYC (May 2010)
• Java Development - James Williams' Presentation from MongoSF (April 2010)
• Morphia: Easy Java Persistance for MongoDB - Scott Hernandez' presentation at MongoSF (May 2011)
Java Driver Concurrency

The Java MongoDB driver is thread safe. If you are using in a web serving environment, for example, you should create a single Mongo instance, and you can use it in every request. The Mongo object maintains an internal pool of connections to the database (default pool size of 10). For every request to the DB (find, insert, etc) the java thread will obtain a connection from the pool, execute the operation, and release the connection. This means the connection (socket) used may be different each time.

Additionally in the case of a replica set with slaveOk option turned on, the read operations will be distributed evenly across all slaves. This means that within the same thread, a write followed by a read may be sent to different servers (master then slave). In turn the read operation may not see the data just written since replication is asynchronous. If you want to ensure complete consistency in a "session" (maybe an http request), you would want the driver to use the same socket, which you can achieve by using a "consistent request". Call requestStart() before your operations and requestDone() to release the connection back to the pool:

```java
DB db...;
db.requestStart();
try {
    db.requestEnsureConnection();
    code....
} finally {
    db.requestDone();
}
```

DB and DBCollection are completely thread safe. In fact, they are cached so you get the same instance no matter what.

WriteConcern option for single write operation

Since by default a connection is given back to the pool after each request, you may wonder how calling getLastError() works after a write. You should actually use a write concern like WriteConcern.SAFE instead of calling getLastError() manually. The driver will then call getLastError() before putting the connection back in the pool.

```java
DBCollection coll...;
coll.insert(..., WriteConcern.SAFE);
// is equivalent to
DB db...;
DBCollection coll...;
db.requestStart();
try {
    coll.insert(...);
    DBOBJECT err = db.getLastError();
} finally {
    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:

```java
public class Tweet implements DBOBJECT {
    /* ... */
}
```

Then you can say:
Tweet myTweet = new Tweet();
myTweet.put("user", userId);
myTweet.put("message", msg);
myTweet.put("date", new Date());

collection.insert(myTweet);

When a document is retrieved from the database, it is automatically converted to a DBObject. To convert it to an instance of your class, use DBCollection.setObjectClass():

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

If for some reason you wanted to change the message you can simply take that tweet and save it back after updating the field.

Tweet myTweet = (Tweet)collection.findOne();
myTweet.put("message", newMsg);

collection.save(myTweet);

Java Tutorial

- Introduction
- A Quick Tour
  - Making A Connection
  - Authentication (Optional)
  - Getting A List Of Collections
  - Getting A Collection
  - Inserting a Document
  - Setting write concern
  - 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.mongodbDBObject;
import com.mongodb.DBCursor;
import com.mongodb.ServerAddress;
import java.util.Arrays;

Mongo m = new Mongo();
// or
Mongo m = new Mongo( "localhost" );
// or
Mongo m = new Mongo( "localhost", 27017 );
// or, to connect to a replica set, supply a seed list of members
Mongo m = new Mongo(Arrays.asList(
            new ServerAddress( "localhost", 27017 ),
            new ServerAddress( "localhost", 27018 ),
            new ServerAddress( "localhost", 27019 )));

DB db = m.getDB( "mydb" );

At this point, the db object will be a connection to a MongoDB server for the specified database. With it, you can do further operations.

Note: The Mongo object instance actually represents a pool of connections to the database; you will only need one object of class Mongo even with multiple threads. See the concurrency doc page for more information.

The Mongo class is designed to be thread safe and shared among threads. Typically you create only 1 instance for a given database cluster and use it across your application. If for some reason you decide to create many mongo instances, note that:

- all resource usage limits (max connections, etc) apply per mongo instance
- to dispose of an instance, make sure you call mongo.close() to clean up resources

Authentication (Optional)

MongoDB can be run in a secure mode where access to databases is controlled through name and password authentication. When run in this mode, any client application must provide a name and password before doing any operations. In the Java driver, you simply do the following with the connected mongo object:

    boolean auth = db.authenticate(myUserName, myPassword);

If the name and password are valid for the database, auth will be true. Otherwise, it will be false. You should look at the MongoDB log for further information if available.

Most users run MongoDB without authentication in a trusted environment.

Getting A List Of Collections

Each database has zero or more collections. You can retrieve a list of them from the db (and print out any that are there):

    Set<String> colls = db.getCollectionNames();
    for (String s : colls) {
        System.out.println(s);
    }

and assuming that there are two collections, name and address, in the database, you would see

name
address

as the output.
Getting A Collection

To get a collection to use, just specify the name of the collection to the `getCollection` 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, let's 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);
```

Setting write concern

For typical usage, you should set the default write concern to ensure that your application will be notified if the server is unable to complete an update operation.

```
m.setWriteConcern(WriteConcern.SAFE);
```

There are many options for write concern. Additionally, the default write concern can be overridden on the database, collection, and individual update operations. Please consult the API Documentation for details.

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
Note the _id element has been added automatically by MongoDB to your document. Remember, MongoDB reserves element names that start with "/"/"$" for internal use.

**Adding Multiple Documents**

In order to do more interesting things with queries, let’s add multiple simple documents to the collection. These documents will just be

```json
{
  "i" : value
}
```

and we can do this fairly efficiently in a loop

```java
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.

```java
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:

```java
DBCursor cursor = coll.find();
try {
  while(cursor.hasNext()) {
    System.out.println(cursor.next());
  }
} finally {
  cursor.close();
}
```

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);
    cursor = coll.find(query);
    try {
        while(cursor.hasNext()) {
            System.out.println(cursor.next());
        }
    } finally {
        cursor.close();
    }

and it should just print just one document

    { "_id" : "49903677516250c1008d624e" , "i" : 71 }

You may commonly see examples and documentation in MongoDB which use $ Operators, such as this:

db.things.find({j: {$ne: 3}, k: {$gt: 10} });

These are represented as regular String keys in the Java driver, using embedded DBObjects:

BasicDBObject query = new BasicDBObject();
    query.put("j", new BasicDBObject("$ne", 3));
    query.put("k", new BasicDBObject("$gt", 10));
    cursor = coll.find(query);
    try {
        while(cursor.hasNext()) {
            System.out.println(cursor.next());
        }
    } finally {
        cursor.close();
    }

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
    cursor = coll.find(query);
    try {
        while(cursor.hasNext()) {
            System.out.println(cursor.next());
        }
    } finally {
        cursor.close();
    }

which should print the documents where i > 50. We could also get a range, say 20 < i <= 30:
query = new BasicDBObject();
    query.put("i", new BasicDBObject("$gt", 20).append("$lte", 30)); // i.e. 20 < i <= 30
    cursor = coll.find(query);
    try {
        while (cursor.hasNext()) {
            System.out.println(cursor.next());
        }
    } finally {
        cursor.close();
    }

Creating An Index

MongoDB supports indexes, and they are very easy to add on a collection. To create an index, you just specify the field that should be indexed, and specify if you want the index to be ascending (1) or descending (-1). The following creates an ascending index on the "i" field:

coll.createIndex(new BasicDBObject("i", 1)); // create index on "i", ascending

Getting a List of Indexes on a Collection

You can get a list of the indexes on a collection:

List<DBObject> list = coll.getIndexInfo();
    for (DBObject o : list) {
        System.out.println(o);
    }

and you should see something like

{ "name": "i_1", "ns": "mydb.testCollection", "key": { "i": 1 } }

Quick Tour of the Administrative Functions

Getting A List of Databases

You can get a list of the available databases:

Mongo m = new Mongo();
    for (String s : m.getDatabaseNames()) {
        System.out.println(s);
    }

Dropping A Database

You can drop a database by name using the Mongo object:

m.dropDatabase("my_new_db");

Java Types

- Object Ids
- Regular Expressions
- Dates/Times
Object Ids

`com.mongodb.ObjectId` is used to autogenerate unique ids.

```java
ObjectId id = new ObjectId();
ObjectId copy = new ObjectId(id);
```

Regular Expressions

The Java driver uses `java.util.regex.Pattern` for regular expressions.

```java
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.

```java
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.

```java
DBRef addressRef = new DBRef(db, "foo.bar", address_id);
DBObject address = addressRef.fetch();
DBObject person = BasicDBObjectBuilder.start()  
  .add("name", "Fred")  
  .add("address", addressRef)  
  .get();
collection.save(person);

DBObject fred = collection.findOne();
DBRef addressObj = (DBRef)fred.get("address");
addressObj.fetch()
```

Binary Data

An array of bytes (`byte[]`) as a value will automatically be wrapped as a Binary type. Additionally the Binary class can be used to represent binary objects, which allows to pick a custom type byte.

Timestamp Data

A timestamp is a special object used by Mongo as an ID based on time, represented by a (time in second, incremental id) pair. It is used notably in the replication oplog.

A timestamp is represented by the BSONTimestamp class.

Code Data
A code object is used to represent JavaScript code, for example when saving executable functions into the system.js collection. The Code and CodeWScope classes are used to represent this data. Note that some methods (like map/reduce) take Strings but wrap them into Code objects in the driver.

**Embedded Documents**

Suppose we have a document that, in JavaScript, looks like:

```javascript
{
    "x" : {
        "y" : 3
    }
}
```

The equivalent in Java is:

```java
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:

```javascript
{
    "x" : [
        1,
        2,
        {"foo" : "bar"},
        4
    ]
}
```

you could do:

```java
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);
```

**Read Preferences and Tagging in The Java Driver**

MongoDB’s read preferences and tagging allows application developers to target member nodes within a replica set for read or write operations. Version 2.2 brings several refinements on node tagging that give you greater control over how your data is read or written. Release 2.9.0 of the Java driver has been built in coordination with the release of Mongo 2.2 and provides full access to these newly available preferences and tagging features.

**Read Preferences**

A read preference provides client applications with control over which nodes of a replica set are used for reads. A client application defines its read preference by selecting one of the five behavioral modes:

- **PRIMARY**: The default read mode. Read from primary only. Throw an error if primary is unavailable. Cannot be combined with tags.
- **PRIMARY PREFERRED**: Read from primary if available, otherwise a secondary.
- **SECONDARY**: Read from a secondary node if available, otherwise error.
- **SECONDARY PREFERRED**: Read from a secondary if available, otherwise read from the primary.
- **NEAREST**: Read from any member node from the set of nodes which respond the fastest. The responsiveness of a node is measured with pings. Any node whose ping time is within 15 milliseconds of the node with the lowest ping time is considered near.

**Implementation**
The Java driver implements MongoDB's read preferences with the `ReadPreference` class. Client applications allocate the desired read mode by calling one of `ReadPreference`'s static factory methods. One factory method exists for each mode.

```java
ReadPreference.primary();
ReadPreference.primaryPreferred();
ReadPreference.secondary();
ReadPreference.secondaryPreferred();
ReadPreference.nearest();
```

The factory method returns a private inner subclass of `ReadPreference` that implements the correct behavioral mode. The driver's use of polymorphism in this manner means that your code only needs to interface with the `ReadPreference` class alone.

**Example:**
Suppose we are developing an application and we prefer operate on strongly consistent data, (i.e. read requests always return the most recent updates of a given document). In this case, the primary node will handle all reads and writes. Now suppose our application must be able to perform reads even if the primary becomes unavailable for some reason. Even though we prefer data consistency we can tolerate eventual consistency when the primary is down. We therefore use the primary preferred mode as our read preference.

```java
ReadPreference preference = ReadPreference.primaryPreferred();
DBCursor cur = new DBCursor(collection, query, null, preference);
```

The Java driver maintains knowledge and state of the replica set by periodically pinging the member nodes and can detect the loss of a member. In this example, the driver will automatically detect the loss of the primary and route the read request to a secondary node, just as we have instructed.

**Tags**
As of version 2.0 of MongoDB, each node within your replica set can be marked with additional descriptors called tags. Tags can be used to indicate a node's location, membership in a designated set, or other characteristics. Tags enable your application to read from or write to specific nodes or a set of nodes in a replica set.

As an example, suppose we’re running a replica set of three nodes deployed across three separate data centers. Each of these data centers is in a separate geographic location. We want to ensure that our data will persist in the event of a disaster, so we mark each node with a tag indicating the region where it lives. Our replica set configuration might look similar to this:
Notice the "settings" field in the replication configuration. We've defined a new getLastErrorModes object with the key DRSafe. When our client application uses this error mode in a write concern it is instructing the write operation to replicate to at least two regions before completion. Here's an example in the Java driver:

```java
// create a write concern with the specific getLastErrorMode
WriteConcern concern = new WriteConcern("DRSafe");

// an insert with the custom write concern
coll.insert(new BasicDBObject("name", "simple doc"), concern);
```

By allocating a write concern using the "DRSafe" error mode and passing it in on the insert, we have now ensured that our data has been backed up to two separate regions and will be available should a data center fail.

**Using Tags with Read Preferences**

Continuing with our sample application, we decide that we want to send our read requests to the nearest node to reduce request latency. The Java driver's read preference API gives a couple of ways of doing this, the easiest is to simply use the nearest mode.

```java
DBObject query = new BasicDBObject("name", "simple doc")
DBObject result =
coll.findOne(query, null, ReadPreference.nearest());
```

By using the nearest mode the driver will automatically send the read to one of a set of nodes with the lowest ping time relative to the client application, (the receiving node could be either a primary or secondary). But suppose our client application can determine where its requests originate. We could have explicitly tagged the read preference to use the datacenter nearest to our location.
Look again at the replica set configuration from above. Each node has been tagged by data center. Let’s say that the current read request is coming from southern California. We can configure this read request to be served by the node living in our Los Angeles data center.

```java
// initialize a properly tagged read preference
ReadPreference tagged_pref =
ReadPreference.secondaryPreferred(new BasicDBObject("datacenter", "Los Angeles"));

// include the tagged read preference in this request
DBObject result = coll.findOne(new BasicDBObject("name", "simple doc"), null, tagged_pref);
```

Read preferences can also accept multiple tags. Returning to our example application, suppose we want to send our reads either the “Los Angeles” node, or failing to find a healthy member in Los Angeles a node in the “US_West” region.

```java
// read from either LA or US_West
DBObject tagSetOne = new BasicDBObject("datacenter", "Los Angeles");
DBObject tagSetTwo = new BasicDBObject("region", "US_West");

ReadPreference pref =
ReadPreference.primaryPreferred(tagSetOne, tagSetTwo);
```

In this example, the driver looks first for a member tagged with datacenter “Los Angeles”. If it cannot find an available member, the driver will look for a member tagged with region of “US West”. You may use a tag set to define a set of required tags a member node must have to be used for the read. For example:

```java
// read from either LA or US_West
DBObject tagSetOne = new BasicDBObject("datacenter", "Los Angeles");
tagSetOne.put("rack", "1");
DBObject tagSetTwo = new BasicDBObject("region", "US_West");

ReadPreference pref =
ReadPreference.primaryPreferred(tagSetOne, tagSetTwo);
```

The difference being the driver is looking for a node that is tagged with both data center Los Angeles and on rack 1, or a node that is in region “US_West”.

You can set the Read Preference at the operation, collection, DB, Mongo, MongoOptions, or MongoURI level, and the preference will be inherited similar to the way slaveOk and write concern are. Read preferences work with any server version that supports replica sets (1.6 and up). Tagged read preferences work with any version that supports tagging (2.0 and up). However, tagging will only work on a sharded cluster if you are connecting to a mongos running 2.2 or above.

### Replica Set Semantics

The MongoDB Java driver handles failover in replicated setups with tunable levels of transparency to the user. By default, a Mongo connection object will ignore failures of secondaries, and only reads will throw MongoExceptions when the primary node is unreachable.

The level of exception reporting is tunable however, using a specific WriteConcern; you can set this on the Mongo/DB/Collection/Method level. Several levels are included as static options:

- **WriteConcern.NONE**: No exceptions thrown.
- **WriteConcern.NORMAL**: Exceptions are only thrown when the primary node is unreachable for a read, or the full replica set is unreachable.
- **WriteConcern.SAFE**: Same as the above, but exceptions thrown when there is a server error on writes or reads. Calls getLastError().
- **WriteConcern.REPLICAS_SAFE**: Tries to write to two separate nodes. Same as the above, but will throw an exception if two writes are not possible.
- **WriteConcern.FSYNC_SAFE**: Same as WriteConcern.SAFE, but also waits for write to be written to disk.

Additional errors may appear in the log files, these are for reporting purposes and logged based on the logging settings.

Sample code is provided which illustrates some of these options. To quickly initialize a sample replica set, you can use the mongo shell:

```bash
mongo
```
> var rst = new ReplSetTest({ nodes : 3 })
> rst.startSet() // wait for processes to start
> rst.initiate() // wait for replica set initialization

Java client code demonstrating error handling is available:

- https://github.com/mongodb/mongo-snippets/blob/master/java/Test.java

Using The Aggregation Framework with The Java Driver

Release 2.2.0 of MongoDB introduces the aggregation framework. Designed to be both performant and easy to use, the aggregation framework calculates aggregate values, (such as counts, totals and averages), without the need for complex map-reduce operations. The aggregation framework is both multithreaded and written in C++, thus it executes natively across nodes.

Aggregation tasks are built around the concept of the aggregation pipeline. Just as UNIX-like shells use the pipe operator ‘|’ to connect a series of command-line operations together, the aggregation framework passes documents through a pipeline of operations which transform these objects as they go. Version 2.9.0 of the Java driver provides a new helper method, DBCollection.aggregate() which can be used to create aggregation tasks.

Let’s use a simple example to demonstrate how the aggregation helper works. Suppose I am using MongoDB to store my employee’s travel expenses. I’ve created a collection named “expenses”, which store individual expenses by employee and by department. Here’s a sample document:

```
{"_id" : ObjectId("503d5024ff9038cdbfcc9da4"),
  "employee" : 61,
  "department" : "Sales",
  "amount" : 77,
  "type" : "airfare"
}
```

I am auditing three departments: Sales, Engineering and Human Resources. I want to calculate each department’s average spend on airfare. I’d like to use the Aggregation Framework for the audit, so I think of the operation in terms of a pipeline:

### Pipeline Operations

1. **Match documents where type = airfare into the pipeline**
2. **Pass only the department and amount fields through the pipeline**
3. **Average the expense amount, grouped by department**

I will use the aggregation operators $match, $project and $group to perform each operation. Individual aggregation operations can be expressed as JSON objects, so I can think of my pipeline as:
I use the Java Driver's aggregation helper to build out this pipeline in my application. Let's take a look at the aggregate() method signature.

```java
public AggregationOutput aggregate(DBObject firstOp, DBObject ... additionalOps)
```

The aggregate method uses Java varargs and accepts arbitrary number of DBObjects as parameters. These DBObjects represent aggregation operations, which will be chained together by the helper method to form the aggregation pipeline. Callers of the aggregate method must pass at least one aggregation operation. Here's the Java code we'll use to perform the aggregation task:

```java
// create our pipeline operations, first with the $match
DBObject match = new BasicDBObject("$match", new BasicDBObject("type", "airfare"));

// build the $projection operation
DBObject fields = new BasicDBObject("department", 1);
fields.put("amount", 1);
fields.put("_id", 0);
DBObject project = new BasicDBObject("$project", fields);

// Now the $group operation
DBObject groupFields = new BasicDBObject("_id", "$department");
groupFields.put("average", new BasicDBObject("$avg", "$amount"));
DBObject group = new BasicDBObject("$group", groupFields);

// run aggregation
AggregationOutput output = collection.aggregate( match, project, group );
```

Aggregations are executed as database commands in MongoDB. These commands embed the results of the aggregation task in an object that also contains additional information about how the command was executed. The return value of aggregate() is an instance of the AggregationOutput class, which provides assessors to this information.

```java
public Iterable<DBObject> results() 
public CommandResult getCommandResult 
public DObject getCommand()
```

Let's take a look at the results of my audit:
C++ Language Center

A C++ driver is available for communicating with the MongoDB. As the database is written in C++, the driver actually uses some core MongoDB code -- this is the same driver that the database uses itself for replication.

The driver has been compiled successfully on Linux, OS X, Windows, and Solaris.

- Downloading the Driver
- Compiling and Linking
- MongoDB C++ Client Tutorial
- API Documentation
- Using BSON from C++
- SQL to C++ Mapping Chart
- HOWTO
  - Connecting
  - getLastError
  - Tailable Cursors
  - BSON Arrays in C++
- Mongo Database and C++ Driver Source Code (at github). See the client subdirectory for client driver related files.

Additional Notes

- The Building documentation covers compiling the entire database, but some of the notes there may be helpful for compiling client applications too.

- There is also a pure C driver for MongoDB. For true C++ apps we recommend using the C++ driver.

BSON Arrays in C++
// examples

using namespace mongo;
using namespace bson;

bo an_obj;

/** transform a BSON array into a vector of BSONElements.
    we match array # positions with their vector position, and ignore
    any fields with non-numeric field names.
*/
vector<be> a = an_obj["x"].Array();

be array = an_obj["x"];
assert( array.isABSONObj() );
assert( array.type() == Array );

// Use BSON_ARRAY macro like BSON macro, but without keys
BSONArray arr = BSON_ARRAY( "hello" << 1 << BSON( "foo" << BSON_ARRAY( "bar" << "baz" << "qux" ) ) );

// BSONArrayBuilder can be used to build arrays without the macro
BSONArrayBuilder b;
b.append(1).append(2).arr();

/** add all elements of the object to the specified vector */
bo myarray = an_obj["x"].Obj();
vector<be> v;
myarray elems(v);
list<be> L;
myarray elems(L)

/** add all values of the object to the specified collection. If type mismatches,
    exception.
    template <class T>
    void Vals(vector<T> &) const;
    template <class T>
    void Vals(list<T> &) const;
*/

/** add all values of the object to the specified collection. If type mismatches, skip.
    template <class T>
    void vals(vector<T> &) const;
    template <class T>
    void vals(list<T> &) const;
*/

C++ BSON Library

- Standalone Usage
- API Docs
- Short Class Names

The MongoDB C++ driver library includes a bson package that implements the BSON specification (see [http://www.bsonspec.org/](http://www.bsonspec.org/)). This library can be used standalone for object serialization and deserialization even when one is not using MongoDB at all.

Include bson/bson.h or db/jsobj.h in your application (not both). bson.h is new and may not work in some situations, was is good for light header-only usage of BSON (see the bsondemo.cpp example).

Key classes:

- `mongo::BSONObj (aka bson::bo)` a BSON object
- `mongo::BSONElement (bson::be)` a single element in a bson object. This is a key and a value.
- `mongo::BSONObjBuilder (bson::bob)` used to make BSON objects
- `mongo::BSONObjIterator ( bson::bo::iterator )` to enumerate BSON objects

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

```cpp
BSONObjBuilder b;
b.append("name", "Joe");
b.append("age", 33);
BSONObj p = b.obj();
```

Or more concisely:

```cpp
BSONObj p = BSONObjBuilder().append("name", "Joe").append("age", 33).obj();
```

We can also create objects with a stream-oriented syntax:

```cpp
BSONObjBuilder b;
b << "name" << "Joe" << "age" << 33;
BSONObj p = b.obj();
```

The macro BSON lets us be even more compact:

```cpp
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.

```cpp
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:

```cpp
BSONObj p =
BSONObjBuilder().genOID().append("name","Joe").append("age",33).obj();
```

**Standalone Usage**

You can use the C++ BSON library without MongoDB. Most BSON methods under the `bson/` directory are header-only. They require boost, but headers only.

See the `bsondemo.cpp` example at github.com.

**API Docs**

- [http://api.mongodb.org/cpp/plusplus/](http://api.mongodb.org/cpp/plusplus/)

**Short Class Names**

Add

```cpp
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.:

```bash
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**

```bash
$ 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:

```bash
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:

```bash
scons mongoclient
```

Alternatively, include client/mongo_client_lib.cpp in your project.

For Windows, see also:

- bson/bsondemo/bsondemo.vcxproj
- client/examples/simple_client_demo.vcxproj
- Prebuilt Boost Libraries
  - Prebuild Boost Libraries only necessary for versions < 2.1.1
  - Boost 1.49 source is now included directly in version 2.1.1+
Building with Visual Studio 2010

Other notes for Windows:

- Compile with /MT
- You may wish to define _CRT_SECURE_NO_WARNINGS to avoid warnings on use of strncpy and such by the MongoDB client code.
- Include the WinSock library in your application: Linker/Input/Additional Dependencies - add ws2_32.lib.

Windows Troubleshooting

- error LNK2005: ___ already defined in msvcprt.lib(MSVC100.dll) libboost_thread-vc100-mt-1_42.lib(thread.obj)
  - The boost library being linked against is expecting a /MT build. If this is a release build, try using /MT instead of /MD for compilation (under Properties.C++.Code Generation).

C++ Driver Download

Driver tarballs

The C++ client library can be found at http://dl.mongodb.org/dl/cxx-driver/.

Note: despite the word 'linux' in the filenames, these files are mostly source code and thus should be applicable to all operating systems.

From the server source code

If you have the full MongoDB source code, the driver code is part of it. This is available on github and also on the MongoDB Downloads page.

The full server source is quite large, so the tarballs above may be easier. Also if using the full server source tree, use "scons mongoclient" to build just the client library.

Next: Compiling and Linking

C++ getLastError

- string mongo::DBClientWithCommands::getLastError();
  - Get error result from the last operation on this connection. Empty string if no error.

- BSONObj DBClientWithCommands::getLastErrorDetailed();
  - Get the full last error object. See the getLastError Command page for details.

See client/simple_client_demo.cpp for an example.

See Also

- getLastError Command

C++ Tutorial

- 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
  - Example
- 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):
```cpp
#include <iostream>
#include "client/dbclient.h"

using namespace mongo;

void run() {
    DBClientConnection c;
    c.connect("localhost");
}

int main() {
    try {
        run();
        cout << "connected ok" << endl;
    } catch (DBException &e) {
        cout << "caught " << e.what() << endl;
    }
    return 0;
}
```

If you are using gcc on Linux or OS X, you would compile with something like this, depending on location of your include files and libraries:

```
$ g++ tutorial.cpp -lmongoclient -lboost_thread -lboost_filesystem -lboost_program_options -o tutorial
$ ./tutorial
connected ok
```

- Depending on your boost version you might need to link against the `boost_system` library as well: `-lboost_system`.
- You may need to append `"-mt"` to `boost_filesystem` and `boost_program_options`. If using a recent boost, `"-mt"` is not needed anymore.
- Of course, you may need to use `-I` and `-L` to specify the locations of your mongo and boost headers and libraries.

**BSON**

The Mongo database stores data in **BSON** format. BSON is a binary object format that is JSON-like in terms of the data which can be stored (some extensions exist, for example, a Date datatype).

To save data in the database we must create objects of class `BSONObj`. The components of a `BSONObj` are represented as `BSONElement` objects. We use `BSONObjBuilder` to make BSON objects, and `BSONObjIterator` to enumerate BSON objects.

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

```cpp
BSONobjBuilder b;
b.append("name", "Joe");
b.append("age", 33);
BSONObj p = b.obj();
```

Or more concisely:

```cpp
BSONObj p = BSONObjBuilder().append("name", "Joe").append("age", 33).obj();
```

We can also create objects with a stream-oriented syntax:

```cpp
BSONObjBuilder b;
b << "name" << "Joe" << "age" << 33;
BSONObj p = b.obj();
```

The macro BSON lets us be even more compact:
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.

```cpp
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:

```cpp
BSONObj p = BSONObjBuilder().genOID().append("name","Joe").append("age",33).obj();
```

Other helpers are listed here, see bsonmisc.h for a full list.

**Inserting**

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

```cpp
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().

```cpp
cout << "count:" << c.count("tutorial.persons") << endl;
auto_ptr<DBClientCursor> cursor =
c.query("tutorial.persons", BSONObj());
while( cursor->more() ) {
  BSONObj p = cursor->next();
  cout << p.getStringField("name") << endl;
}
```

BSONObj() is an empty BSON object -- it represents {} which indicates an empty query pattern (an empty query is a query for all objects).

We use BSONObj::toString() above to print out information about each object retrieved. BSONObj::toString is a diagnostic function which prints an abbreviated JSON string representation of the object. For full JSON output, use BSONObj::jsonString.

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

```cpp
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:

```cpp
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
Queries are BSON objects of a particular format -- in fact, we could have used the BSON() macro above instead of QUERY(). See class Query in dbclient.h for more information on Query objects, and the Sorting section below.

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

```
use tutorial;
db.persons.find( { age : 33 } );
```

**Indexing**

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

```
c.ensureIndex("tutorial.persons", fromjson("{age:1}");
```

The ensureIndex method checks if the index exists; if it does not, it is created. ensureIndex is intelligent and does not repeat transmissions to the server; thus it is safe to call it many times in your code, for example, adjacent to every insert operation.

In the above example we use a new function, fromjson. fromjson converts a JSON string to a BSONObj. This is sometimes a convenient way to specify BSON. Alternatively we could have written:

```
c.ensureIndex("tutorial.persons", BSON("age" << 1));
```

**Sorting**

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

```
auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons", QUERY("age" << age ));
```

to

```
to auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons", QUERY("age" << age ).sort("name") );
```

Here we have used Query::sort() to add a modifier to our query expression for sorting.

**Updating**

Use the update() method to perform a database update. For example the following update in the mongo shell:

```
> use tutorial
> db.persons.update( { name : 'Joe', age : 33 },
...                  { $inc : { visits : 1 } } )
```

is equivalent to the following C++ code:

```
db.update("tutorial.persons",
    BSON( "name" << "Joe" << "age" << 33 ),
    BSON( "$inc" << BSON( "visits" << 1 ) ) );
```

**Example**

A simple example illustrating usage of BSON arrays and the "$nin" operator is available here.

**Further Reading**
This overview just touches on the basics of using Mongo from C++. There are many more capabilities. For further exploration:

- See the language-independent Developer's Tour;
- Experiment with the mongo shell;
- Review the doxygen API docs;
- See connecting pooling information in the API docs;
- See GridFS file storage information in the API docs;
- See the HOWTO pages under the C++ Language Center;
- Consider getting involved to make the product (either C++ driver, tools, or the database itself) better!

List of helper functions

This is a non-exhaustive list of helper functions for use in the C++ stream syntax. An exhaustive list is here: bsonmisc.h

Typical example of stream syntax:

```cpp
BSONObj p = BSON( "name" << "Joe" << "age" << 33 );
```

GENOID -the server will add an _id automatically if it is not included explicitly.

```cpp
BSONObj p = BSON( GENOID << "name" << "Joe" << "age" << 33 );
// result is: { _id : ..., name : "Joe", age : 33 }
```

LT,GT,LTE,GTE, NE -less than, greater than, etc

```cpp
BSONObj p = BSON( "age" << GT << 21 );
// result is: { age : { $gt : 21 } }
```

DATENOW -translates to current date

```cpp
BSONObj p = BSON( "created" << DATENOW );
// result is: { created : "2009-10-09 11:41:42" }
```

MINKEY,MAXKEY

```cpp
BSONObj p = BSON( "a" << MINKEY);
// result is: { "a" : { "$minKey" : 1 } }
```

OR

```cpp
OR(BSON("x" << GT << 7), BSON("y" << LT << 6))
// result is: { $or: [{x: {$gt: 7}}, {y: {$lt: 6}}] }
```

BSONNULL -translates to null value (will appear in MongoDB 2.1)

```cpp
BSONObj p = BSON( "name" << "Methuselah" << "age" << BSONNULL );
// result is: { name : "Methuselah", age : null }
```

Connecting

The C++ driver includes several classes for managing collections under the parent class DBClientInterface.

DBClientConnection is our normal connection class for a connection to a single MongoDB database server (or shard manager). Other classes exist for connecting to a replica set.

See http://api.mongodb.org/cplusplus/ for details on each of the above classes.
Perl Language Center

- Installing
  - CPAN
  - Manual (Non-CPAN) Installation
  - Big-Endian Systems
- Next Steps
- MongoDB Perl Tools
  - BSON
  - Entities::Backend::MongoDB
  - MojoX::Session::Store::MongoDB
  - MongoDB::Admin
  - Mongoose
  - MongoDBI
  - MongoDBx-Class
  - MongoX
  - OOP Perl CMS

### 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

```bash
$ 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:

```bash
$ perl Makefile.PL
$ make
$ make test # make sure mongod is running, first
$ sudo make install
```

Please note that the tests will not pass without a `mongod` process running.

Note that the Perl driver requires some libraries available in CPAN. As of April, 2010, these are Any::Moose, Class::Method::Modifiers, Data::Types, DateTime, File::Slurp, Test::Exception, Try::Tiny, boolean, and Module::Install. (Additionally, Tie::IxHash is required for testing.) On Debian or Ubuntu systems, these prerequisites can be easily installed with the following command:

```bash
```

### Big-Endian Systems

The driver will work on big-endian machines, but the database will not. The tests assume that `mongod` will be running on localhost unless `%ENV{MONGOD}` is set. So, to run the tests, start the database on a little-endian machine (at, say, "example.com") and then run the tests with:
A few tests that require a database server on "localhost" will be skipped.

Next Steps

There is a tutorial and API documentation on CPAN.

If you're interested in contributing to the Perl driver, check out Contributing to the Perl Driver.

**MongoDB Perl Tools**

**BSON**

**BSON** is a pure-Perl BSON implementation.

**Entities::Backend::MongoDB**

**Entities::Backend::MongoDB** is a backend for the Entities user management and authorization system stores all entities and relations between them in a MongoDB database, using the MongoDB module. This is a powerful, fast backend that gives you all the features of MongoDB.

**MojoX::Session::Store::MongoDB**

**MojoX::Session::Store::MongoDB** is a store for **MojoX::Session** that stores a session in a MongoDB database. Created by Ask Bjørn Hansen.

**MongoDB::Admin**

**MongoDB::Admin** is a collection of MongoDB administrative functions. Created by David Burley.

**Mongoose**

**Mongoose** is an attempt to bring together the full power of Moose with MongoDB. Created by Rodrigo de Oliveira Gonzalez.

**MongoDBI**

**MongoDBI** is an Object-Document-Mapper (ODM) for MongoDB. It allows you to create Moose-based classes to interact with MongoDB databases.

At-a-glance, most will enjoy MongoDBI for its ability to easily model classes while leveraging the power of MongoDB's schemaless and expeditious document-based design, dynamic queries, and atomic modifier operations.

Also noteworthy is MongoDBI's ease-of-use, chainable search facilities (filters), automated indexing, moose-integration (inheritance support, etc), lean document updates via dirty field tracking, and ability for each class to be configured to use a different database and connection, etc.

**MongoDBx-Class**

**MongoDBx-Class** is an ORM for MongoDB databases. MongoDBx::Class takes advantage of the fact that Perl's MongoDB driver is Moose-based to extend and tweak the driver's behavior, instead of wrapping it. This means MongoDBx::Class does not define its own syntax, so you simply use it exactly as you would the MongoDB driver directly. That said, MongoDBx::Class adds some sugar that enhances and simplifies the syntax unobtrusively (either use it or don't). Thus, it is relatively easy to convert your current MongoDB applications to MongoDBx::Class. A collection in MongoDBx::Class isa('MongoDB::Collection'), a database in MongoDBx::Class isa('MongoDB::Database'), etc. Created by Ido Perlmuter.

**MongoX**

**MongoX** - DSL sugar for MongoDB

**OOP Perl CMS**

**OOP Perl CMS** is based on Khurt Williams' Object Oriented Perl methodology and can be used as a basic CMS framework or as a basis for your own CMS system. It uses Apache & mod_perl with MongoDB backend. Created by Waitman Gobble.

**Contributing to the Perl Driver**

The easiest way to contribute is to file bugs and feature requests on Jira.
If you would like to help code the driver, read on...

**Finding Something to Help With**

**Fixing Bugs**

You can choose a bug on [Jira](http://example.com/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 Guidelines**

The best way to make changes is to create an account on [Github](https://github.com), 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](http://search.cpan.org/dist/MongoDB/lib/MongoDB/Tutorial.pod).

**Online API Documentation**
Writing Drivers and Tools

See Also

- MongoDB 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:

<table>
<thead>
<tr>
<th>Description</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification for the basic socket communications protocol used between Mongo and clients.</td>
<td></td>
</tr>
<tr>
<td>Description of what functionality is expected from a Mongo Driver</td>
<td></td>
</tr>
<tr>
<td>Specification of GridFS - a convention for storing large objects in Mongo</td>
<td></td>
</tr>
<tr>
<td>Description of the extended JSON protocol for the REST-ful interface (ongoing development)</td>
<td></td>
</tr>
</tbody>
</table>

Additionally we recommend driver authors take a look at existing driver source code as an example.

- Back to driver development home page
- Drivers Home Page

Mongo Driver Requirements

This is a high-level list of features that a driver for MongoDB might provide. We attempt to group those features by priority. This list should be taken with a grain of salt, and probably used more for inspiration than as law that must be adhered to. A great way to learn about implementing a driver is by reading the source code of any of the existing drivers, especially the ones listed as "mongodb.org supported".

High priority

- BSON serialization/deserialization
- full cursor support (e.g. support OP_GET_MORE operation)
- close exhausted cursors via OP_KILL_CURSORS
- support for running database commands
- handle query errors
- convert all strings to UTF-8 (part of proper support for BSON)
- hint, explain, count, $where
- database profiling: set/get profiling level, get profiling info
- advanced connection management (replica sets, slave okay)
- automatic reconnection

Medium priority

- validate a collection in a database
- buffer pooling
- Tailable cursor support

A driver should be able to connect to a single server. By default this must be localhost:27017, and must also allow the server to be specified by hostname and port.
Mongo m = new Mongo();  // go to localhost, default port

Mongo m = new Mongo(String host, int port);

How the driver does this is up to the driver - make it idiomatic. However, a driver should make it explicit and clear what is going on.

Replica Sets

A driver must be able to support "Replica-Set" configurations, where more than one mongod servers are specified, and configured for hot-failover.

The driver should determine which of the nodes is the current master, and send all operations to that server. In the event of an error, either socket error or a "not a master" error, the driver must restart the determination process.

1. **Cluster Mode** Connect to master in master-slave cluster

   ```
   ServerCluster sc = new ServerCluster(INETAddr...); // again, give one and discover?
   Mongo m = new Mongo(sc);
   ```

   ## Connect to slave in read-only mode in master-slave cluster

   ```
   ServerCluster sc = new ServerCluster(INETAddr...); // again, give one and discover?
   sc.setTarget(...)
   Mongo m = new Mongo(sc);
   or maybe make it like *Default/Simple* w/ a flag?
   ```

Other than that, we need a way to get a DB object:

```
Mongo m = new Mongo();
DB db = m.getDB(name);
```

And a list of db names (useful for tools...):

```
List<String> getDBNamelist();
```

Database Object

Simple operations on a database object:
get name of database

String dbName = db.getName();

Get a list of all the collection names in this database

List<String> cols = db.getCollectionNames();

get a collection object. Can optionally create it if it
doesn't exist, or just be strict. (XJDM has strictness as an option)

Collection coll = db.getCollection(string);

Create a collection w/ optional options. Can fault
* if the collection exists, or can just return it if it already does

Collection coll = db.createCollection( string);
Collection coll = db.createCollection( string, options);

Drop a collection by its name or by collection object.
* Driver could invalidate any outstanding Collection objects
* for that collection, or just hope for the best.

boolean b = db.dropCollection(name);
boolean b = db.dropCollection(Collection);

Execute a command on the database, returning the
* BSON doc with the results

Document d = db.executeCommand(command);

Close the [logical] database

void db.close();

Erase / drop an entire database

bool dropDatabase(dbname)

Database Administration

These methods have to do with database metadata: profiling levels and collection validation. Each admin object is associated with a database. These methods could either be built into the Database class or provided in a separate Admin class whose instances are only available from a database instance.
/* get an admin object from a database object. */
Admin admin = db.getAdmin();

/**
 * Get profiling level. Returns one of the strings "off", "slowOnly", or
 * "all". Note that the database returns an integer. This method could
 * return an int or an enum instead --- in Ruby, for example, we return
 * symbols.
 */
String profilingLevel = admin.getProfilingLevel();

/**
 * Set profiling level. Takes whatever getProfilingLevel() returns.
 */
admin.setProfilingLevel("off");

/**
 * Retrieves the database's profiling info.
 */
Document profilingInfo = admin.getProfilingInfo();

/**
 * Returns true if collection is valid; raises an exception if not.
 */
boolean admin.validateCollection(collectionName);

Collection Basic Ops

/**
 * full query capabilities - limit, skip, returned fields, sort, etc
 */
Cursor find(...);
void insert(...) // insert one or more objects into the collection, local variants on args
void remove(query) // remove objects that match the query
void update(selector, modifier) // modify all objects that match selector w/ modifier object
void updateFirst(selector, object) // replace first object that match selector w/ specified object
void upsert(selector, object) // replace first object that matches, or insert
long getCount();
long getCount(query);

Index Operations

void createIndex( index_info)
void dropIndex(name)
void dropIndexes()
List<info> getIndexInformation()

Misc Operations

document explain(query)
options getOptions();
string getName();
void close();

Cursor Object

document getNextDocument()
iterator getIterator() // again, local to language
bool hasMore()
void close()
Spec, Notes and Suggestions for Mongo Drivers

Assume that the BSON objects returned from the database may be up to 16MB. This size may change over time.

See Also

- Driver Requirements
- BSON
- The main Database Internals page

Feature Checklist for Mongo Drivers

Functionality Checklist

This section lists tasks the driver author might handle.

Essential

- BSON serialization/deserialization
- Basic operations: query, insert, update, remove, ensureIndex, findOne, limit, sort
- Fetch more data from a cursor when necessary (dbGetMore)
- Sending of KillCursors operation when use of a cursor has completed (ideally for efficiently these are sent in batches)
- Convert all strings to utf8
- Authentication

Recommended

- automatic _id generation
- Database $cmd support and helpers
- Detect { $err: ... } response from a db query and handle appropriately --see Error Handling in Mongo Drivers
- [Automatically connect to proper server, and failover], when connecting to a Replica Set
- ensureIndex commands should be cached to prevent excessive communication with the database. (Or, the driver user should be informed that ensureIndex is not a lightweight operation for the particular driver.)
- Support detecting max BSON size on connection (e.g., using buildinfo or isMaster commands) and allowing users to insert docs up to that size.

More Recommended

- lasterror helper functions
- count() helper function
- $where clause
- eval()
- File chunking (GridFS)
- hint fields
- explain helper

More Optional

- addUser, logout helpers
- Allow client user to specify Option SlaveOk for a query
- Tailable cursor support
- In/out buffer pooling (if implementing in a garbage collected languages)

More Optional

- [connection pooling]
- Automatic reconnect on connection failure
- DBRef Support:
  - Ability to generate easily
  - Automatic traversal

See Also

- The Driver and Integration Center for information about the latest drivers
- The [top page] for this section
- The main Database Internals page
- The starting point for all Home
Conventions for Mongo Drivers

**Interface Conventions**

It is desirable to keep driver interfaces consistent when possible. Of course, idioms vary by language, and when they do adaptation is appropriate. However, when the idiom is the same, keeping the interfaces consistent across drivers is desirable.

**Terminology**

In general, use these terms when naming identifiers. Adapt the names to the normal "punctuation" style of your language -- foo_bar in C might be fooBar in Java.

- database - what does this mean?
- collection
- index

**Driver Testing Tools**

**Object IDs**

- driverOIDTest for testing toString

```
> db.runCommand( { "driverOIDTest" : new ObjectId() } )
{ "oid" : ObjectId("4b8991f221752a6e61a88267"), "str" : "4b8991f221752a6e61a88267", "ok" : 1 }
```

**Mongo Wire Protocol**

- Introduction
- 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

**Table Of Contents**

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:

```c
struct MsgHeader {
    int32 messageLength; // total message size, including this
    int32 requestId; // identifier for this message
    int32 responseTo; // requestID from the original request
    // (used in responses 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. `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

**Table Of Contents**

The following are the currently supported opcodes:

<table>
<thead>
<tr>
<th>Opcode Name</th>
<th>OpCode Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP_REPLY</td>
<td>1</td>
<td>Reply to a client request. responseTo is set</td>
</tr>
<tr>
<td>OP_MSG</td>
<td>1000</td>
<td>generic msg command followed by a string</td>
</tr>
<tr>
<td>OP_UPDATE</td>
<td>2001</td>
<td>update document</td>
</tr>
<tr>
<td>OP_INSERT</td>
<td>2002</td>
<td>insert new document</td>
</tr>
<tr>
<td>RESERVED</td>
<td>2003</td>
<td>formerly used for OP_GET_BY_OID</td>
</tr>
<tr>
<td>OP_QUERY</td>
<td>2004</td>
<td>query a collection</td>
</tr>
<tr>
<td>OP_GETMORE</td>
<td>2005</td>
<td>Get more data from a query. See Cursors</td>
</tr>
<tr>
<td>OP_DELETE</td>
<td>2006</td>
<td>Delete documents</td>
</tr>
</tbody>
</table>
**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

```c
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**:

<table>
<thead>
<tr>
<th>bit num</th>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Upsert</td>
<td>If set, the database will insert the supplied object into the collection if no matching document is found.</td>
</tr>
<tr>
<td>1</td>
<td>MultiUpdate</td>
<td>If set, the database will update all matching objects in the collection. Otherwise only updates first matching doc.</td>
</tr>
<tr>
<td>2-31</td>
<td>Reserved</td>
<td>Must be set to 0.</td>
</tr>
</tbody>
</table>

**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

```c
struct {
    MsgHeader header;       // standard message header
    int32     flags;         // bit vector - see below
    cstring   fullCollectionName; // *dbname.collectionname*
    document* documents;     // one or more documents to insert into the collection
}
```

**fullCollectionName**: The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

**documents**: One or more documents to insert into the collection. If there are more than one, they are written to the socket in sequence, one after another.

**flags**:

<table>
<thead>
<tr>
<th>bit num</th>
<th>name</th>
<th>description</th>
</tr>
</thead>
</table>
ContinueOnError: If set, the database will not stop processing a bulk insert if one fails (e.g. due to duplicate IDs). This makes bulk insert behave similarly to a series of single inserts, except lastError will be set if any insert fails, not just the last one. If multiple errors occur, only the most recent will be reported by getLastError. (new in 1.9.1)

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:

```c
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:**

<table>
<thead>
<tr>
<th>bit num</th>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
<td>Must be set to 0.</td>
</tr>
<tr>
<td>1</td>
<td>TailableCursor</td>
<td>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 &quot;latent cursor&quot;, the cursor may become invalid at some point (CursorNotFound) – for example if the final object it references were deleted.</td>
</tr>
<tr>
<td>2</td>
<td>SlaveOk</td>
<td>Allow query of replica slave. Normally these return an error except for namespace &quot;local&quot;.</td>
</tr>
<tr>
<td>3</td>
<td>OplogReplay</td>
<td>Internal replication use only - driver should not set</td>
</tr>
<tr>
<td>4</td>
<td>NoCursorTimeout</td>
<td>The server normally times out idle cursors after an inactivity period (10 minutes) to prevent excess memory use. Set this option to prevent that.</td>
</tr>
<tr>
<td>5</td>
<td>AwaitData</td>
<td>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.</td>
</tr>
<tr>
<td>6</td>
<td>Exhaust</td>
<td>Stream the data down full blast in multiple &quot;more&quot; 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.</td>
</tr>
<tr>
<td>7</td>
<td>Partial</td>
<td>Get partial results from a mongos if some shards are down (instead of throwing an error)</td>
</tr>
<tr>
<td>8-31</td>
<td>Reserved</td>
<td>Must be set to 0.</td>
</tr>
</tbody>
</table>

**fullCollectionName**: The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

**numberToSkip**: Sets the number of documents to omit - starting from the first document in the resulting dataset - when returning the result of the query.

**numberToReturn**: Limits the number of documents in the first CONTRIB.OP_REPLY message to the query. However, the database will still establish a cursor and return the cursorID to the client if there are more results than numberToReturn. If the client driver offers 'limit' functionality (like the SQL LIMIT keyword), then it is up to the client driver to ensure that no more than the specified number of document are returned to the calling application. If numberToReturn is 0, the db will use the default return size. If the number is negative, then the database will return that number and close the cursor. No further 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 a 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 a CONTRIB:OP_REPLY message.

### OP_DELETE

The OP_DELETE message is used to remove one or more documents from a collection. The format of the OP_DELETE message is:

```
struct {
    MsgHeader header;     // standard message header
    int32 ZERO;            // 0 - reserved for future use
    cstring fullCollectionName; // "dbname.collectionname"
    int32 flags;           // bit vector - see below for details.
    document selector;     // query object. See below for details.
}
```

- **fullCollectionName**: The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

- **flags**:

```
<table>
<thead>
<tr>
<th>bit num</th>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SingleRemove</td>
<td>If set, the database will remove only the first matching document in the collection. Otherwise all matching documents will be removed.</td>
</tr>
<tr>
<td>1-31</td>
<td>Reserved</td>
<td>Must be set to 0.</td>
</tr>
</tbody>
</table>
```

- **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 

```c
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 OP_QUERY or OP_GET_MORE message. The format of an OP_REPLY message is:

```c
struct {
    MsgHeader header;       // standard message header
    int32     responseFlags; // bit vector - see details below
    int64     cursorID;      // cursor id
    int32     startingFrom;  // where in the cursor this reply is starting
    int32     numberReturned; // number of documents in the reply
    document* documents;     // documents
}  
```

**responseFlags**: 

<table>
<thead>
<tr>
<th>bit num</th>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CursorNotFound</td>
<td>Set when getMore is called but the cursor id is not valid at the server. Returned with zero results.</td>
</tr>
<tr>
<td>1</td>
<td>QueryFailure</td>
<td>Set when query failed. Results consist of one document containing an &quot;$err&quot; field describing the failure.</td>
</tr>
<tr>
<td>2</td>
<td>ShardConfigStale</td>
<td>Drivers should ignore this. Only mongos will ever see this set, in which case, it needs to update config from the server.</td>
</tr>
<tr>
<td>3</td>
<td>AwaitCapable</td>
<td>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.</td>
</tr>
<tr>
<td>4-31</td>
<td>Reserved</td>
<td>Ignore</td>
</tr>
</tbody>
</table>

**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 OP_GET_MORE messages used to get more data, and also must be closed by the client when no longer needed via a OP_KILL_CURSORS message.

**BSON**
BSON Document Types

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 and BSON and Data Interchange

Language-Specific Examples

We often map from a language's "dictionary" type – which may be its native objects – to BSON. The mapping is particularly natural in dynamically typed languages:

JavaScript: {
  "foo": "bar"
}
Perl: {
  "foo" => "bar"
}
PHP: array("foo" => "bar")
Python: {
  "foo": "bar"
}
Ruby: {
  "foo" => "bar"
}
Java: DBObjce 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 );


C++

BSNObj 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:

```php
$bson = bson_encode(null);
$bson = bson_encode(true);
$bson = bson_encode(4);
$bson = bson_encode("hello, world");
$bson = bson_encode(array("foo" => "bar"));
$bson = bson_encode(new MongoDate());
```

Mongo-specific objects (MongoId, MongoDate, MongoRegex, MongoCode) will be encoded in their respective BSON formats. For other objects, it will create a BSON representation with the key/value pairs you would get by running `for ($object as $key => $value)`.

`bson_decode` takes a string representing a BSON object and parses it into an associative array.

Python
```python
>>> from bson import BSON
>>> bson_string = BSON.encode({"hello": "world"})
>>> bson_string
b'\x16\x00\x00\x00\x02hello\x00\x06\x00\x00\x00world\x00\x00'
>>> bson_string.decode()
{u'hello': u'world'}
```

PyMongo also supports "ordered dictionaries" through the `bson.son` module. The `BSON` class can handle `SON` instances using the same methods you would use for regular dictionaries. Python2.7's collections.OrderedDict is also supported.

Ruby
There are now two gems that handle BSON-encoding: bson and bson_ext. These gems can be used to work with BSON independently of the MongoDB Ruby driver.

```ruby
irb
>> require 'rubygems'
=> true
>> require 'bson'
=> true
>> doc = {'hello' => "world"}
>> bson = BSON.serialize(doc).to_s
=> "\x02\00\000\00\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](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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Strict</th>
<th>JS</th>
<th>TenGen</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_binary</td>
<td></td>
<td></td>
<td></td>
<td><code>&lt;bindata&gt;</code> is the base64 representation of a binary string. <code>&lt;t&gt;</code> is the hexadecimal representation of a single byte indicating the data type.</td>
</tr>
<tr>
<td>data_date</td>
<td></td>
<td>Date( <code>&lt;date&gt;</code> )</td>
<td>Date( <code>&lt;date&gt;</code> )</td>
<td><code>&lt;date&gt;</code> is the JSON representation of a 64 bit signed integer for milliseconds since epoch (unsigned before version 1.9.1).</td>
</tr>
<tr>
<td>data_timestamp</td>
<td></td>
<td>Timestamp( <code>&lt;t&gt;</code>, <code>&lt;i&gt;</code></td>
<td>Timestamp( <code>&lt;t&gt;</code>, <code>&lt;i&gt;</code></td>
<td><code>&lt;i&gt;</code> is the JSON representation of a 32 bit unsigned integer for seconds since epoch. <code>&lt;i&gt;</code> is the JSON representation of a 32 bit unsigned integer for increment.</td>
</tr>
</tbody>
</table>
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 the 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:

```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:

```json
{
    "_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:
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:

```json
{
    "_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:

```
cursor = db.fs.chunks.findOne({files_id: myFileID}).sort({n:1});
```

### Implementing Authentication in a Driver

The current version of Mongo supports only very basic authentication. One authenticates a username and password in the context of a particular database. Once authenticated, the user has full read and write access to the database in question.

The `admin` database is special. In addition to several commands that are administrative being possible only on `admin`, authentication on `admin` gives one read and write access to all databases on the server. Effectively, `admin` access means root access to the `db`.

Note on a single socket we may authenticate for any number of databases, and as different users. This authentication persists for the life of the database connection (barring a `logout` command).

**The Authentication Process**

Authentication is a two step process. First the driver runs a `getnonce` command to get a nonce for use in the subsequent authentication. We can view a sample `getnonce` invocation from `dbshell`:

```
> db.$cmd.findOne({getnonce:1})
{ "nonce": "7268c5046fcf36e1", "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:
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:

```javascript
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);
}
```

```javascript
DB.prototype.auth = function(username, pass){
    n = this.runCommand( { getnonce: 1 } );
    var a = this.runCommand({
        authenticate : 1 ,
        user : username ,
        nonce : n.nonce ,
        key : hex_md5( n.nonce + username + hex_md5(username +":mongo:" + pass ) )
    });
    return a.ok;
}
```

Logout

Drivers may optionally implement the logout command which deauthorizes usage for the specified database for this connection. Note other databases may still be authorized.

Alternatively, close the socket to deauthorize.

```
> db.$cmd.findOne({logout:1})
{
    "ok" : 1.0
}
```

Replica Sets and Authentication

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

When switching from one server in a replica set to another, for example in a failover situation, you must reauthenticate. Clients will likely want to cache authentication from the user so that the client can reauthenticate with the new server when appropriate.
Be careful also with operations such as Logout. If you log out from only some members of a replica set, that could be an issue.

Authenticating with a server in slave mode is allowed.

**See Also**

- Security and Authentication

### Notes on Pooling for Mongo Drivers

Note that with the db write operations can be sent asynchronously or synchronously (the latter indicating a getlasterror request after the write).

When asynchronous, one must be careful to continue using the same connection (socket). This ensures that the next operation will not begin until after the write completes.

**Pooling and Authentication**

An individual socket connection to the database has associated authentication state. Thus, if you pool connections, you probably want a separate pool for each authentication case (db + username).

**Pseudo-code**

The following pseudo-code illustrates our recommended approach to implementing connection pooling in a driver's connection class. This handles authentication, grouping operations from a single "request" onto the same socket, and a couple of other gotchas:

```python
class Connection:
    def 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]
        thissockets = Socket[pool_size]
        this.socket_auth = String[pool_size][]
        this.auth = {}  
        this.find_master()

    def find_master():
        for address in this.addresses:
            if address.is_master():
                this.master = address

    def 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

    def get_socket():
        sock_number = this.thread_map[current_thread]
        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

del 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.
Connecting Drivers to Replica Sets

Ideally a MongoDB driver can connect to a cluster of servers which represent a replica set, and automatically find the right set member with which to communicate. Failover should be automatic too. The general steps are:

1. The user, when opening the connection, specifies host[:port] for one or more members of the set. Not all members need be specified -- in fact the exact members of the set might change over time. This list for the connect call is the seed list.
2. The driver then connects to all servers on the seed list, perhaps in parallel to minimize connect time. Send an ismaster command to each server.
3. When the server is in replSet mode, it will return a hosts field with all members of the set that are potentially eligible to serve data. The client should cache this information. Ideally this refreshes too, as the set's config could change over time.
4. Choose a server with which to communicate.
   a. If ismaster == true, that server is primary for the set. This server can be used for writes and immediately consistent reads.
   b. If secondary == true, that server is not primary, but is available for eventually consistent reads. In this case, you can use the primary field to see which server the master should be. (If primary is not set, you may want to poll other nodes at random; it is conceivable that the member to which we are talking is partitioned from the other members, and thus it cannot determine who is primary on its own. This is unlikely but possible.)
5. If an error occurs with the current connection, find the new primary and resume use there.

For example, if we run the ismaster command on a non-primary server, we might get something like:

```plaintext
> 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:

```plaintext
{ $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.

```c
/* 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
  - Updacting
  - 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.

**Tutorial**

- Running MongoDB
- Getting A Database Connection
- Dynamic Schema ("Schema Free")
- Inserting Data into A Collection
- Accessing Data From a Query
- Specifying What the Query Returns
  - `findOne()` - Syntactic Sugar
  - Limiting the Result Set via `limit()`
- More Help
- What Next

**Running MongoDB**

First, run through the Quickstart guide for your platform to get Mongo installed.

**Getting A Database Connection**

Let's now try manipulating the database with the database shell. (We could perform similar operations from any programming language using an appropriate driver. The shell is convenient for interactive and administrative use.)

Start the MongoDB JavaScript shell with:

```bash
# '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:

```javascript
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:
Switching to a database with the `use` command won't immediately create the database - the database is created lazily the first time data is inserted. This means that if you `use` a database for the first time it won't show up in the list provided by `show dbs` until data is inserted.

To see a list of handy commands, type `help`.

Tip for Developers with Experience in Other Databases

You may notice, in the examples below, that we never create a database or collection. MongoDB does not require that you do so. As soon as you insert something, MongoDB creates the underlying collection and database. If you query a collection that does not exist, MongoDB treats it as an empty collection.

Dynamic Schema ("Schema Free")

MongoDB has databases, collections, and indexes much like a traditional RDBMS. In some cases (databases and collections) these objects can be implicitly created, however once created they exist in a system catalog (db.systems.collections, db.system.indexes).

Collections contain (BSON) documents. Within these documents are fields. In MongoDB there is no predefined fields (what would be columns in an RDBMS). There is no schema for fields within documents – the fields and their value datatypes can vary. Thus there is no notion of an "alter table" operation which adds a "column". In practice, it is highly common for a collection to have a homogenous structure across documents; however this is not a requirement. This flexibility means that schema migration and augmentation are very easy in practice - rarely will you need to write scripts which perform "alter table" type operations. In addition to making schema migration flexible, this facility makes iterative software development atop the database easier.

Inserting Data into A Collection

Let's create a test collection and insert some data into it. We will create two objects, 
j and t, and then save them in the collection `things`.

In the following examples, `>` indicates commands typed at the shell prompt.

```plaintext
> 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("4c2209f9f3924d31102bd84b"), x: 3 }
```

A few things to note:

- We did not predefine the collection. The database creates it automatically on the first insert.
- The documents we store can have different fields - in fact in this example, the documents have no common data elements at all. In practice, one usually stores documents of the same structure within collections.
- Upon being inserted into the database, objects are assigned an object ID (if they do not already have one) in the field `_id`.
- When you run the above example, your ObjectID values will be different.

Let's add some more records to this collection:
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 \texttt{it} shortcut. Continuing from the code above:

Technically, \texttt{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 \texttt{it} command.

But we can also work with the cursor directly; just how that's done is discussed in the next section.

\textbf{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 \texttt{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 \texttt{mongo shell}, we need to explicitly use the cursor returned from the \texttt{find()} operation.

Let's repeat the same query, but this time use the cursor that \texttt{find()} returns, and iterate over it in a while loop:
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:

```javascript
> var cursor = db.things.find();
> while (cursor.hasNext()) printjson(cursor.next());
( "_id" : ObjectId("4c220a42f3924d31102bd85f"), "name" : "mongo" )
( "_id" : ObjectId("4c220a42f3924d31102bd862"), "x" : 4, "y" : 1 )
( "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "y" : 2 )
( "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "y" : 3 )
( "_id" : ObjectId("4c220a42f3924d31102bd861"), "x" : 4, "y" : 4 )
( "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "y" : 5 )
( "_id" : ObjectId("4c220a42f3924d31102bd865"), "x" : 4, "y" : 6 )
( "_id" : ObjectId("4c220a42f3924d31102bd863"), "x" : 4, "y" : 7 )
( "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "y" : 8 )
( "_id" : ObjectId("4c220a42f3924d31102bd864"), "x" : 4, "y" : 9 )
( "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "y" : 10 )
```

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:

```javascript
> db.things.find().forEach(printjson);
( "_id" : ObjectId("4c220a42f3924d31102bd85f"), "name" : "mongo" )
( "_id" : ObjectId("4c220a42f3924d31102bd862"), "x" : 4, "y" : 1 )
( "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "y" : 2 )
( "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "y" : 3 )
( "_id" : ObjectId("4c220a42f3924d31102bd861"), "x" : 4, "y" : 4 )
( "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "y" : 5 )
( "_id" : ObjectId("4c220a42f3924d31102bd865"), "x" : 4, "y" : 6 )
( "_id" : ObjectId("4c220a42f3924d31102bd863"), "x" : 4, "y" : 7 )
( "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "y" : 8 )
( "_id" : ObjectId("4c220a42f3924d31102bd864"), "x" : 4, "y" : 9 )
( "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "y" : 10 )
```

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:

```javascript
> var arr = db.things.find().toArray();
> arr[5];
{ "_id" : ObjectId("4c220a42f392431102bd859"), "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, let's 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.

```sql
SELECT * FROM things WHERE name="mongo"
```

```javascript
> db.things.find({name: "mongo"]).forEach(printjson);
{ "_id" : ObjectId("4c220a42f392431102bd84a"), "name" : "mongo" }
```

```sql
SELECT * FROM things WHERE x=4
```

```javascript
> db.things.find({x:4}).forEach(printjson);
{ "_id" : ObjectId("4c220a42f392431102bd856"), "x" : 4, "j" : 4 } ...
```

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, let's 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.thing.find({x:4}, {j: true}).forEach(printjson);

true

{
  "_id": ObjectId("4c220a42f3924d31102bd856"),
  "j": 1
}

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.thing.findOne({name: "mongo"}));

{
  "_id": ObjectId("4c220a42f3924d31102bd84a"),
  "name": "mongo"
}

This is more efficient because the client requests a single object from the database, so less work is done by the database and the network. This is the equivalent of `find({name:"mongo"}).limit(1)`.

Another example of finding a single document by _id:

> var doc = db.thing.findOne({_id:ObjectId("4c2209f9f3924d31102bd84a")});
> doc

{
  "_id": ObjectId("4c2209f9f3924d31102bd84a"),
  "name": "mongo"
}

Limiting the Result Set via `limit()`

You may limit the size of a query’s result set by 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.thing.find().limit(3);

{
  "_id": ObjectId("4c2209f9f3924d31102bd84a"),
  "name": "mongo"
}

More Help
In addition to the general "help" command, you can call help on db and db.collection (where "collection" is the name of a collection) to see a summary of methods available.

If you are curious about what a function is doing, you can type it without the ()s and the shell will print the source, for example:

```javascript
> printjson
function (x) {
    print(tojson(x));
}
```

mongo is a full JavaScript shell, so any JavaScript function, syntax, or class can be used in the shell. In addition, MongoDB defines some of its own classes and globals (e.g., db). You can see the full API at http://api.mongodb.org/js/.

What Next

- After completing this tutorial the next step to learning MongoDB is to dive into the rest of the documentation for more details, you can see the new Manual for additional documentation.
- See also SQL to Mongo Mapping Chart
- The MongoDB Manual
- Tutorials in the MongoDB Manual

Manual

For the manual see http://docs.mongodb.org/.

More docs on specific topics:

- Connections
- Databases
  - Commands
    - Copy Database Commands
    - getLastError Command
    - Viewing and Terminating Current Operation
  - Mongo Metadata
- Collections
  - Capped Collections
  - Using a Large Number of Collections
- Documents
- Data Types and Conventions
  - Dates
  - Timestamp data type
  - Internationalized Strings
  - Object IDs
    - ObjectId
- GridFS
  - When to use GridFS
- Indexes
  - Index Versions
  - Geospatial Indexing
    - Geospatial Haystack Indexing
  - Indexing as a Background Operation
  - Multikeys
    - Using Multikeys to Simulate a Large Number of Indexes
    - Indexing Advice and FAQ
- Inserting
  - Legal Key Names
  - Schema Design
    - Tweaking performance by document bundling during schema design
  - Trees in MongoDB
- Optimization
  - Explain
  - Optimizing Object IDs
  - Optimizing Storage of Small Objects
  - Query Optimizer
- Querying
  - Mongo Query Language
  - Querying and nulls
  - Retrieving a Subset of Fields
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
Thu 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 parameter introducing the options. Options are name=value pairs and the pairs are separated either by “&” or “;”.

As many hosts as necessary may be specified (for connecting to replica pairs/sets).

The options are:

Replica set:

```
• `replicaSet-name`
  • The driver verifies that the name of the replica set it connects to matches this name. Implies that the hosts given are a seed list, and the driver will attempt to find all members of the set.
```
Single server:
- slaveOk=true|false

Any configuration:
- safe=true|false
  - true: the driver sends a getLastError command after every update to ensure that the update succeeded (see also w and wtimeoutMS).
  - false: the driver does not send a getLastError command after every update.
- w=n
  - The driver adds \{ w : n \} to the getLastError command. Implies safe=true.
- wtimeoutMS=ms
  - The driver adds \{ wtimeout : ms \} to the getLastError command. Implies safe=true.
- fsync=true|false
  - true: the driver adds \{ fsync : true \} to the getLastError command. Implies safe=true.
  - false: the driver does not not add fsync to the getLastError command.
- journal=true|false
  - true: Sync to journal. Implies safe=true.
- connectTimeoutMS=ms
  - How long a connection can take to be opened before timing out.
- socketTimeoutMS=ms
  - How long a send or receive on a socket can take before timing out.

These options are not case sensitive.

**Examples**

Connect to a database server running locally on the default port:

```
mongodb://localhost
```

Connect and login to the admin database as user "fred" with password "foobar":

```
mongodb://fred:foobar@localhost
```

Connect and login to the "baz" database as user "fred" with password "foobar":

```
mongodb://fred:foobar@localhost/baz
```

Connect to a replica pair, with one server on example1.com and another server on example2.com:

```
mongodb://example1.com:27017,example2.com:27017
```

Connect to a replica set with three servers running on localhost (on ports 27017, 27018, and 27019):

```
mongodb://localhost,localhost:27018,localhost:27019
```

Connect to a replica set with three servers, sending all writes to the primary and distributing reads to the secondaries:

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

Connect to localhost with safe mode on:

```
mongodb://localhost/?safe=true
```

Connect to a replica set with safe mode on, waiting for replication to succeed on at least two machines, with a two second timeout:

```
mongodb://host1,host2,host3/?safe=true;w=2;wtimeoutMS=2000
```
Connection Pooling

The server will use one thread per TCP connection, therefore it is highly recomended that your application use some sort of connection pooling. Luckily, most drivers handle this for you behind the scenes. One notable exception is setups where your app spawns a new process for each request, such as CGI and some configurations of PHP.

Databases

Each MongoDB server can support multiple databases. Each database is independent, and the data for each database is stored separately, for security and ease of management.

A database consists of one or more collections, the documents (objects) in those collections, and an optional set of security credentials for controlling access.

See Also

- Mongo Metadata

Commands

Introduction

The Mongo database has a concept of a database command. Database commands are ways to ask the database to perform special operations, or to request information about its current operational status.

- Introduction
- Privileged Commands
- Getting Help Info for a Command
- The MongoDB Manual has complete documentation of all MongoDB Database Commands.
- Commands Quick Reference Card

A command is sent to the database as a query to a special collection namespace called $cmd. The database will return a single document with the command results - use findOne() for that if your driver has it.

The general command syntax is:

```
db.$cmd.findOne( { <commandname>: <value> [, options] } );
```

The shell provides a helper function for this:

```
db.runCommand( { <commandname>: <value> [, options] } );
```

For example, to check our database's current profile level setting, we can invoke:

```
> db.runCommand({profile:-1});
{
   "was" : 0.0,
   "ok" : 1.0
}
```

For many db commands, some drivers implement wrapper methods are implemented to make usage easier. For example, the mongo shell offers

```
> db.getProfilingLevel()
0.0
```

Let's look at what this method is doing:
> print( db.getProfilingLevel )
function () {
    var res = this._dbCommand({profile:-1});
    return res ? res.was : null;
}

> print( db._dbCommand )
function (cmdObj) {
    return this.$cmd.findOne(cmdObj);
}

Many commands have helper functions - see your driver's documentation for more information.

Privileged Commands

Certain operations are for the database administrator only. These privileged operations may only be performed on the special database named admin.

> use admin;
> db.runCommand("shutdown"); // shut down the database

If the db variable is not set to 'admin', you can use adminCommand (_adminCommand in versions earlier than 1.8) to switch to the right database automatically (and just for that operation):

> db.adminCommand("shutdown");

(For this particular command there is also a shell helper function, db.shutdownServer.)

Getting Help Info for a Command

Use commandHelp in shell to get help info for a command:

> db.commandHelp("datasize")
help for: datasize example: { datasize:"blog.posts", keyPattern:{x:1}, min:{x:10}, max:{x:55} } 
NOTE: This command may take awhile to run

(Help is not yet available for some commands.)

Copy Database Commands

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

These options are faster than the alternative of doing a mongodump and mongorestore since they skip the step of creating a temporary on-disk BSON representation.

copydb Command

Copies an entire database from one name on one server to another name on another server. Omit <from_hostname> to copy from one name to another on the same server. Command must be sent/run on the "destination" server.
cloneDatabase

Similar to copydb but with a simplified syntax for the case where we simply want to copy a database to this server, and leave its name as-is.

> db.cloneDatabase(<from_hostname>);

The copy database command does not work when copying to or from a sharded collection.

Other Notes

- copyDatabase may be run against a slave/secondary (i.e., the source server can be a slave/secondary).
- copyDatabase does not snapshot in any sense: if the source database is changing during the copy, the destination will receive documents representing different points in time in the source during the copying interval.
- The command must be run on the destination server.
- The command does not lock either the source or the destination server for the duration of the operation. Each side will yield periodically to allow other reads/writes through.

getErrorCommand

MongoDB does not wait for a response by default when writing to the database. Use the getLastError command to ensure that operations have succeeded.

Many of the drivers can invoke the getLastError command automatically on a write operation. Enabling this driver option is called “safe mode” (“write concern” in some drivers). When enabled, the driver piggybacks a getLastError message with each write message. It then awaits the result of the getLastError command before returning.

A few other alternative modes of operation exist. First, an application could not call getLastError at all. This might be appropriate if, for example, one is writing data to a log, and would not report the error to the user anyway. Another option would be to only call getLastError after a series of operations.

Running in the Shell

When to Use

Options

- j
- w
  - Tagging
  - majority
- fsync
- Combining

Return Value

Using getLastError from Drivers

Mongo Shell REPL Behavior

getLastError Command

getPrevError Command

See Also
The `getlasterror` command checks for an error on the last database operation for this connection. Since it's a command, there are a few ways to invoke it:

```
> db.$cmd.findOne({getlasterror:1})
```

Or in the shell:

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

Or you can use the shell helper:

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

In the mongo shell, `db.getLastError()` returns the last error – null if no error. Use `db.getLastErrorObj()` to see the full error result. `db.getLastErrorObj().err` should be null if there is no error condition.

For more about commands, see the command documentation.

**When to Use**

`getlasterror` is primarily useful for write operations (although it is set after a command or query too). Write operations by default do not have a return code: this saves the client from waiting for client/server turnarounds during write operations. One can always call `getLastError` if one wants a return code.

If you’re writing data to MongoDB on multiple connections, then it can sometimes be important to call `getlasterror` on one connection to be certain that the data has been committed to the database. For instance, if you’re writing to connection #1 and want those writes to be reflected in reads from connection #2, you can assure this by calling `getlasterror` after writing to connection #1.

For maximum speed, skip the `getLastError` (or "safe mode") for noncritical writes. Use it when you need to confirm to the user that the write succeeded.

**Options**

`j`

_v2.0+:_ When `j:true` is specified, the `getlasterror` call awaits the journal commit before returning. If the server is running without journaling, it returns immediately, and successfully.

```
> db.runCommand({getlasterror:1,j:true})
```

`w`

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

`wtimeout` may be used in conjunction with `w`. The default is no timeout (wait forever).

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

Note the above options can be combined: waiting for journal acknowledgement and acknowledgement that the write has reached a majority of a replica set can make sense:

```
> db.runCommand({getlasterror:1,j:true,w:2,wtimeout:5000})
```

**Tagging**

_v2.0+:_ You can specify acknowledgement rules based on tags in the configuration.

majority

You can also specify a string of "majority":

```
Once a majority of set members have applied the write, it is **truly committed across the cluster**.

**fsync**

(Generally not recommended. Use `j` instead.)

When running `mongod` without journaling (`--nojournal`), the `fsync` option forces the database to fsync all files before returning.

When running with **journaling**, the `fsync` option awaits the next group commit before returning.

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

**Use `j`, not `fsync`**.

**Combining**

A good combination for highly critical writes is `j:true` and `w:"majority"`.

```javascript
> db.runCommand({getlasterror:1,j:true,w:'majority',wtimeout:10000})
true
```

**Return Value**

The return value from the command is an object with various fields. The common fields are listed below; there may also be other fields.

- **ok** - true indicates the `getLastError` command completed successfully. This does NOT indicate there wasn't a last error.
- **err** - if non-null, indicates an error occurred. Value is a textual description of the error.
- **code** - if set, indicates the error code which occurred.
- **connectionId** - the id of the connection
- **lastOp** - the op-id from the last operation

For updates:

- **n** - if an update was done, this is the number of documents updated.

With `w:<n>/<tag>`

- **wnote** - if set indicates that something unusual happened that is related to using `w`:
- **wtimeout** - if timed out, set to the value true
- **waited** - if timed out, how long waited in milliseconds
- **wtime** - the time spent waiting for the operation to complete

**Using getLastError from Drivers**

The drivers support `getLastError` in the command form and many also offer a "safe" mode for operations. For more on "safe" mode, see each driver's documentation.

* **C#**

- `C++`

  Python. If you're using Python, you automatically call `getlasterror` on insert as follows:

```python
collection.save({"name": "MongoDB"}, safe=True)
```

If the save doesn't succeed, an exception will be raised.

* **Java**

  Java supports various `getLastError()` semantics using `WriteConcern Objects`.

**Mongo Shell REPL Behavior**

The database shell performs a `resetError()` before each read/eval/print loop command evaluation - and automatically prints the error, if one occurred, after each evaluation. Thus, after an error, at the shell prompt `db.getLastError()` will return null. However, if called before returning
getPrevError Command

When performing bulk write operations, resetError() and getPrevError() can be an efficient way to check for success of the operation. For example if we are inserting 1,000 objects in a collection, checking the return code 1,000 times over the network is slow. Instead one might do something like this:

```javascript
db.resetError();
for ( loop 1000 times... )
    db.foo.save(something...);
if( db.getPrevError().err )
    print("didn't work!");
```

- getPrevError only holds 1 previous error, so whole batch may have to be retried in case of error.
- A better alternative is to use the "batch" insert method provided by many drivers, along with safe writes. This way getLastError() is only called after each batch, but still writes will stop at the 1st error.
- getPrevError may be deprecated in the future.

Viewing and Terminating Current Operation

- View Operation(s) in Progress
- Killing an In Progress Operation
- Sharded Databases
- Finding blocking ops
  - Finding Waiting Write
  - Finding Active Write
  - Finding Active Reads
- See Also

View Operation(s) in Progress

```javascript
> db.currentOp();
{ inprog: [ { "opid" : 18 , "op" : "query" , "ns" : "mydb.votes" ,
    "query" : "{ score : 1.0 }" , "inLock" : 1 } ] }
```

Fields:

- opid - an incrementing operation number. Use with killOp().
- active - if the operation is active, false if the operation is queued
- waitingForLock - if true, lock has been requested but not yet granted
- op - the operation type (query, update, etc.)
- ns - namespace for the operation (database + collection name) (Note: if the ns begins with a question mark, the operation has yielded.)
- query - the query spec, if operation is a query
- lockType - the type of lock the operation requires, either read or write or none. See concurrency page.
- client - address of the client who requested the operation
- desc - the type of connection. conn indicates a normal client connections. Other values indicate internal threads in the server.
• `threadId` - id of the thread
• `numYields` - the number of the times the operation has yielded (to some other operation)

From a driver one runs `currentOp` by executing the equivalent of:

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

### Killing an In Progress Operation

**v1.4+**

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

Be careful about terminating internal operations, for example the a replication sync thread. Typically you only kill operations from **external clients**.

### Sharded Databases

In a sharded environment, operations named "writebacklistener" will appear. These are long-lived socket connections between mongos and mongod. These can be ignored. See [Sharding FAQ](#).

### Finding blocking ops

In general you will find that either a (active/waiting) write will block (future) readers. If there are active readers then writes will need to wait for the readers to finish (or temporarily yield) so the write can become active.

#### Finding Waiting Write

```javascript
db.currentOp().inprog.forEach(
    function(d){
        if(d.waitingForLock && d.lockType != "read")
            printjson(d)
    })
```

#### Finding Active Write

```javascript
db.currentOp().inprog.forEach(
    function(d){
        if(d.active && d.lockType == "write")
            printjson(d)
    })
```

#### Finding Active Reads

```javascript
db.currentOp().inprog.forEach(
    function(d){
        if(d.active && d.lockType == "read")
            printjson(d)
    })
```

### See Also

- [How does concurrency work](#)

### Mongo Metadata
The `<dbname>.system.*` namespaces in MongoDB are special and contain database system information. System collections include:

- `<dbname>.system.namespaces` lists all namespaces.
- `<dbname>.system.indexes` lists all indexes. Additional namespace / index metadata exists in the database.ns files, and is opaque.
- `<dbname>.system.profile` stores database profiling information.
- `<dbname>.system.users` lists users who may access the database.

Additionally, in the local database only there is replication information in system collections, e.g., `local.system.replset` contains the replica set configuration.

Information on the structure of a stored document is stored within the document itself. See BSON.

There are several restrictions on manipulation of objects in the system collections. Inserting into `system.indexes` adds an index, but otherwise that table is immutable (the special drop index command updates it for you). `system.users` is modifiable. `system.profile` is droppable.

Note: $ is a reserved character. Do not use it in namespace names or within field names. Internal collections for indexes use the $ character in their names. These collection store b-tree bucket data and are not in BSON format (thus direct querying is not possible).

**Collections**

MongoDB collections are essentially named groupings of documents. You can think of them as roughly equivalent to relational database tables.

**Overview**

A MongoDB collection is a collection of BSON documents. These documents usually have the same structure, but this is not a requirement since MongoDB is a schema-free (or more accurately, “dynamic schema”) database. You may store a heterogeneous set of documents within a collection, as you do not need to define the collection’s “columns” or fields.

A collection is created when the first document is inserted.

Collection names should begin with letters or an underscore and may include numbers; $ is reserved. Collections can be organized in namespaces; these are named groups of collections defined using a dot notation. For example, you could define collections blog.posts and blog.authors, both reside under “blog”. Note that this is simply an organizational mechanism for the user -- the collection namespace is flat from the database’s perspective.

The maximum size of a collection name is 128 characters (including the name of the db and indexes). It is probably best to keep it under 80/90 chars.

**Shell**

Programmatically, we access these collections using the dot notation. For example, using the mongo shell:

```bash
if ( db.blog.posts.findOne() )
    print("blog.posts exists and is not empty.");
```

Alternative ways to access collections are:

```bash
> db."mycol".find()
> db.getCollection("mycol").find()
```

Note that though the underscore character is allowed, it has a special function in the shell if it is the 1st character: the shell considers the property to be an actual javascript value, not a collection name. Consequently it is not accessible using the dot notation, but it works fine with `getCollection()`.

```bash
> db._mycol.find() --> error
> db.getCollection("_mycol").find() --> success
```

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

Capped collections are not shard-able.

Creating

Unlike a standard collection, you must explicitly create a capped collection, specifying a collection size in bytes. The collection's data space is then preallocated. Note that the size specified includes database headers.

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

Behavior

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

Usage and Restrictions

- You may insert new documents in the capped collection.
- You may update the existing documents in the collection. However, the documents must not grow in size. If they do, the update will fail. Note if you are performing updates, you will likely want to declare an appropriate index (given there is no _id index for capped collections by default).
- The database does not allow deleting documents from a capped collection. Use the `drop()` method to remove all rows from the collection. (After the drop you must explicitly recreate the collection.)
- Capped collection are not shard-able.

Warning

By default, capped collections do not have a unique index on _id. However, replication refers to document by their _ids, so if you are lacking an index on _id, replication can fall behind or stall. Thus, if you are using a capped collections and replication, you should create a unique index on _id. Having duplicate _ids in your capped collection can also confuse replication, so it is important to insure uniqueness by using the default client-generated mongo _ids, or the `autoIndexId` field (see `autoIndexId` section below).

Applications

- **Logging.** Capped collections provide a high-performance means for storing logging documents in the database. Inserting documents in an unindexed capped collection will be close to the speed of logging to a filesystem. Additionally, with the built-in FIFO mechanism, you are not at risk of using excessive disk space for the logging.
- **Automatic Maintaining of Insertion Order.** Capped collections keep documents in their insertion order automatically, with no index being required for this property. The logging example above is a good example of a case where keeping items in order is important.
- **Caching.** If you wish to cache a small number of documents in the database, perhaps cached computations of information, the capped tables provide a convenient mechanism for this. Note that for this application you will typically want to use an index on the capped table as there will be more reads than writes.
- **Automatic Age Out.** If you know you want data to automatically "roll out" over time as it ages, a capped collection can be an easier way
to support than writing manual removal via cron scripts. Ejection from the capped collection is also inexpensive compared to explicit remove operations.

**Recommendations**

- When appropriate, do not create indexes on a capped collection. If the collection will be written to much more than it is read from, it is better to have no indexes. Note that you may create indexes on a capped collection; however, you are then moving from "log speed" inserts to "database speed" inserts -- that is, it will still be quite fast by database standards.
- Use natural ordering to retrieve the most recently inserted elements from the collection efficiently. This is (somewhat) analogous to tail on a log file.

**Options**

**size.**

The size, in bytes, of the capped collection. This must be specified.

**max**

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

**Note:** When specifying a cap on the number of documents, you must also cap on size. Be sure to leave enough room for your chosen number of documents or items will roll out faster than expected. You can use the validate() utility method to see how much space an existing collection uses, and from that estimate your size needs.

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

**autoIndexId**

The autoIndexId field may be set to true or false to explicitly enable or disable automatic creation of a unique key index on the _id field.

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

If you will be using the _id field, you should create an index on _id.

Given these are used sometimes without an _id index, it can be useful to insert documents without an _id field. Most drivers and the mongo shell add an _id client-side. See each driver's documentation for how to suppress this (behavior might vary by driver). In the mongo shell one could invoke:

```javascript
> db.mycollection._mongo.insert(db.mycollection._fullName, myDocWithoutAnId)
```

**Checking if a collection is capped**

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

Here is the definition.

```javascript
> db.c.isCapped
function () { 
  var e = this.exists();
  return e && e.options && e.options.capped ? true : false; }
```

**Converting a collection to capped**

You can convert a (non-capped) collection to a capped collection with the convertToCapped command:

```javascript
> db.runCommand("convertToCapped": "mycoll", size: 100000});
{ "ok": 1 }
```
Note that the size is in bytes.

---

**Indexes not copied**
No indexes will be created when the new capped collection is. If you want the old indexes you will need to recreate them after it has been converted.

---

**See Also**

- The Sorting and Natural Order section of this Guide
- Creating collections manually `createCollection Command`

---

**Using a Large Number of Collections**

A technique one can use with MongoDB in certain situations is to have several collections to store information instead of a single collection. By doing this, certain repeating data no longer needs to be stored in every object, and an index on that key may be eliminated. More importantly for performance (depending on the problem), the data is then clustered by the grouping specified.

For example, suppose we are logging objects/documents to the database, and want to have M logs: perhaps a dev log, a debug log, an ops log, etc. We could store them all in one collection 'logs' containing objects like:

```
{ log : 'dev', ts : ..., info : ... }
```

However, if the number of logs is not too high, it might be better to have a collection per log. We could have a 'logs.dev' collection, a 'logs.debug' collection, 'logs.ops', etc.:

```
// logs.dev:
( ts : ..., info : ... )
```

Of course, this only makes sense if we do not need to query for items from multiple logs at the same time.

Generally, having a large number of collections has no significant performance penalty, and results in very good performance.

---

**Limits**

By default MongoDB has a limit of approximately 24,000 namespaces per database. Each namespace is 628 bytes, the `.ns` file is 16MB by default.

Each collection counts as a namespace, as does each index. Thus if every collection had one index, we can create up to 12,000 collections. The `--nssize` parameter allows you to increase this limit (see below).

Be aware that there is a certain minimum overhead per collection -- a few KB. Further, any index will require at least 8KB of data space as the b-tree page size is 8KB. Certain operations can get slow if there are a lot of collections and the meta data gets paged out.

`--nssize`

If more collections are required, run mongod with the `--nssize` parameter specified. This will make the <database>.ns file larger and support more collections. Note that `--nssize` sets the size used for newly created `.ns` files -- if you have an existing database and wish to resize, after running the `db` with `--nssize`, run the `db.repairDatabase()` command from the shell to adjust the size.

Maximum `.ns` file size is 2GB.

---

**Documents**

- **Maximum Document Size**

MongoDB can be thought of as a document-oriented database. By ‘document’, we mean structured documents, not freeform text documents. These documents can be thought of as objects but only the data of an object, not the code, methods or class hierarchy. Additionally, there is much less linking between documents in MongoDB data models than there is between objects in a program written in an object-oriented programming language.

In MongoDB the documents are conceptually JSON. More specifically the documents are represented in a format called **BSON** (standing for Binary JSON).
Documents are stored in Collections.

**Maximum Document Size**

MongoDB limits the data size of individual BSON objects/documents. At the time of this writing the limit is 16MB.

This limit is designed as a sanity-check; it is not a technical limit on document sizes. The thinking is that if documents are larger than this size, it is likely the schema is not ideal. Further it allows drivers to make some assumptions on the max size of documents.

The concept is that the maximum document size is a limit that ensures each document does not require an excessive amount of RAM from the machine, or require too much network bandwidth to fetch. For example, fetching a full 100MB document would take over 1 second to fetch over a gigabit ethernet connection. In this situation one would be limited to 1 request per second.

Over time, as computers grow in capacity, the limit will be adjusted upward.

For cases where larger sizes are required, use GridFS.

**Data Types and Conventions**

**MongoDB (BSON) Data Types**

Mongo uses special data types in addition to the basic JSON types of string, integer, boolean, double, null, array, and object. These types include date, object id, binary data, regular expression, and code. Each driver implements these types in language-specific ways, see your driver's documentation for details.

See [BSON](#) for a full list of database types.

**Internationalization**

- See [Internationalized strings](#)

**Database References**

- See [Database References](#) and [Schema Design](#)

**Checking Types from the Shell**

Floats and ints are treating as standard javascript numbers, and are thus hard to tell apart in the shell.
 Dates

The BSON Date/Time data type is referred to as "UTC DateTime" in the BSON spec.

Note – There is a Timestamp data type but that is a special internal type for MongoDB that typically should not be used.

A BSON Date value stores the number of milliseconds since the Unix epoch (Jan 1, 1970) as a 64-bit integer. **v2.0+** : this number is signed so dates before 1970 are stored as a negative numbers.

Before MongoDB v2.0 dates were incorrectly interpreted as an unsigned integer, adversely affected sorting, range queries, and indexes on Date fields. Indexes are not recreated when upgrading. Thus if you created on index on Date values with pre v2.0 versions, and dates before 1970 are relevant to your application, please reindex.

In the shell

```
> x = new Date()
ISODate("2011-10-12T14:54:02.069Z")
> x.toString()
Wed Oct 12 2011 10:54:02 GMT-0400 (Eastern Daylight Time)
> d = ISODate()       // like Date() but behaves more intuitively when used
> d = ISODate('YYYY-MM-DD hh:mm:ss')  // without an explicit "new" prefix on construction,
> d.getMonth()
9
```

See Also


Timestamp data type
This is not the normal Date datatype. This is a special type for internal MongoDB use.

**BSON** includes a timestamp data type with special semantics in MongoDB.

Timestamps are stored as 64 bit values which, on a single mongod, are guaranteed unique. The first 32 bits are a time_t value (seconds since the UTC epoch). The second 32 bits are an incrementing ordinal for operations within a given second.

MongoDB uses the Timestamp datatype as “OpTimes” in the replication oplog’s ts field.

Timestamps have special semantics when null. If null, and the timestamp is one of the first two fields of the object, the timestamp will automatically be converted to a unique value. (It must be one of the first two top level fields of the document for performance reasons; the entire document is not scanned for timestamps.)

An example from the mongo shell follows (example works with shells v1.7.5 and higher).

```bash
> // 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("4d1d4cfddba04eeb294c099"), "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("4d1d4cfddba04eeb294c09a"), "y" : { "t" : 1293765911000, "i" : 1 }, "x" : 1 }
{ "_id" : ObjectId("4d1d4cfddba04eeb294c09b"), "y" : { "t" : 1293765911000, "i" : 2 }, "x" : 2 }
{ "_id" : ObjectId("4d1d4cfddba04eeb294c09c"), "y" : { "t" : 1293765911000, "i" : 3 }, "x" : 3 }
{ "_id" : ObjectId("4d1d4cfddba04eeb294c09d"), "y" : { "t" : 1293765911000, "i" : 4 }, "x" : 4 }
{ "_id" : ObjectId("4d1d4cfddba04eeb294c09e"), "y" : { "t" : 1293765911000, "i" : 5 }, "x" : 5 }
{ "_id" : ObjectId("4d1d4cfddba04eeb294c09f"), "y" : { "t" : 1293765911000, "i" : 6 }, "x" : 6 }
{ "_id" : ObjectId("4d1d4cfddba04eeb294c0a0"), "y" : { "t" : 1293765911000, "i" : 7 }, "x" : 7 }
{ "_id" : ObjectId("4d1d4cfddba04eeb294c0a1"), "y" : { "t" : 1293765911000, "i" : 8 }, "x" : 8 }
{ "_id" : ObjectId("4d1d4cfddba04eeb294c0a2"), "y" : { "t" : 1293765911000, "i" : 9 }, "x" : 9 }
{ "_id" : ObjectId("4d1d4cfddba04eeb294c0a3"), "y" : { "t" : 1293765911000, "i" : 10 }, "x" : 9 }
```

From MongoDB 2.1.0+, timestamps are displayed in the shell with a wrapper, as Timestamp(1293765911000, 1).

**Internationalized Strings**

MongoDB supports UTF-8 for strings in stored objects and queries. (Specifically, BSON strings are UTF-8.)

Generally, drivers for each programming language convert from the language’s string format of choice to UTF-8 when serializing and deserializing BSON. For example, the Java driver converts Java Unicode strings to UTF-8 on serialization.

In most cases this means you can effectively store most international characters in MongoDB strings. A few notes:

- MongoDB regex queries support UTF-8 in the regex string.
- Currently, sort() on a string uses strcmp: sort order will be reasonable but not fully international correct. Future versions of MongoDB may support full UTF-8 sort ordering.

**Object IDs**

Documents in MongoDB required a key, _id, which uniquely identifies them.
**The _id Field**

Almost every MongoDB document has an _id field as its first attribute (there are a few exceptions for system collections and capped collections). The _id value can be of any type with type ObjectID being the most common. _id must be unique for each document in a collection. In most cases collections automatically have an _id index which includes a unique key constraint that enforces this.

If a user tries to insert a document without providing an _id field, the database will automatically generate an object id and store it in the _id field. The _id field is immutable. If you try to change its value on an update you will get a "Mod on _id not allowed" error from getLastError.

The _id value may be of any type, other than arrays, so long as it is unique. If your document has a natural primary key that is immutable we recommend you use that in _id instead of the automatically generated ids. Arrays are not allowed _ids because they are Multikeys.

**The BSON ObjectID Datatype**

Although _id values can be of any type, a special BSON datatype is provided for object ids. This type is a 12-byte binary value designed to have a reasonably high probability of being unique when allocated. All of the officially-supported MongoDB drivers use this type by default for _id values. Also, the MongoDB 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.

```javascript
> 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: ObjectId( "47cc67093475061e3d95369d" ) } )
{ _id : ObjectId( "47cc67093475061e3d95369d" ) , name : "joe" }
```

**BSON ObjectID Specification**

A BSON ObjectID is a 12-byte value consisting of a 4-byte timestamp (seconds since epoch), a 3-byte machine id, a 2-byte process id, and a 3-byte counter. Note that the timestamp and counter fields must be stored big endian unlike the rest of BSON. This is because they are compared byte-by-byte and we want to ensure a mostly increasing order. The format:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>machine/pid</td>
<td>inc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **TimeStamp.** This is a unix style timestamp. It is a signed int representing the number of seconds before or after January 1st 1970 (UTC).
- **Machine.** This is the first three bytes of the (md5) hash of the machine host name, or of the mac/network address, or the virtual machine id.
- **Pid.** This is 2 bytes of the process id (or thread id) of the process generating the Objectid.
- **Increment.** This is an ever incrementing value starting with a random number.

BSON ObjectIds can be any 12 byte binary string that is unique; however, the server itself and almost all drivers use the format above.

**Sequence Numbers**

Traditional databases often use increasing sequence numbers for primary keys. In MongoDB, the preferred approach is to use Object IDs instead. The concept is that in a very large cluster of machines, it is easier to create an object ID than have global, uniformly increasing sequence numbers.

However, sometimes you may want a sequence number. You can do this by creating "counter" documents and using the findAndModify Command.
See also "Insert if Not Present" section of the Atomic Operations page for another method.

**UUIDs**

The `_id` field can be of any type; however, it must be unique. Thus you can use UUIDs in the `_id` field instead of BSON ObjectIds (BSON ObjectIds are slightly smaller; they need not be worldwide unique, just unique for a single db cluster). When using UUIDs, your application must generate the UUID itself. Ideally the UUID is then stored in the `[DOCS:BSON]` type for efficiency—however you can also insert it as a hex string if you know space and speed will not be an issue for the use case.

**See Also**

- Optimizing Object IDs

**ObjectId**

**Redirection Notice**

This page should redirect to Object IDs.

**GridFS**

GridFS is a specification for storing large files in MongoDB. All of the mongodb.org supported drivers implement the GridFS spec.

- Rationale
- Implementation
- Language Support
- Command Line Tools
- See also

**Rationale**

The database supports native storage of binary data within BSON objects. However, BSON objects in MongoDB are limited in size (4MB older versions, 16MB in v1.7/1.8, higher limits in the future). The GridFS spec provides a mechanism for transparently dividing a large file among multiple documents. This allows us to efficiently store large objects, and in the case of especially large files, such as videos, permits range operations (e.g., fetching only the first N bytes of a file).

**Implementation**

To facilitate this, a standard is specified for the chunking of files. Each file has a metadata object in a files collection, and one or more chunk objects in a chunks collection. Details of how this is stored can be found in the GridFS Specification; however, you do not really need to read that, instead, just look at the GridFS API in each language's client driver or mongofiles tool.

**Language Support**

Most drivers include GridFS implementations; for languages not listed below, check the driver's API documentation. (If a language does not include support, see the GridFS specification -- implementing a handler is usually quite easy.)

**Command Line Tools**

Command line tools are available to write and read GridFS files from and to the local filesystem.

**See also**
When to use GridFS

When to use GridFS

- Lots of files. GridFS tends to handle large numbers (many thousands) of files better than many file systems.
- User uploaded files. When users upload files you tend to have a lot of files, and want them replicated and backed up. GridFS is a perfect place to store these as then you can manage them the same way you manage your data. You can also query by user, upload date, etc... directly in the file store, without a layer of indirection
- Files that often change. If you have certain files that change a lot - it makes sense to store them in GridFS so you can modify them in one place and all clients will get the updates. Also can be better than storing in source tree so you don't have to deploy app to update files.

When not to use GridFS

- Few small static files. If you just have a few small files for a website (js,css,images) its probably easier just to use the file system.
- Note that if you need to update a binary object atomically, and the object is under the document size limit for your version of MongoDB (16MB for 1.8), then you might consider storing the object manually within a single document. This can be accomplished using the BSON bindata type. Check your driver's docs for details on using this type.

Indexes

Indexes enhance query performance, often dramatically. It's important to think about the kinds of queries your application will need so that you can define relevant indexes. Once that's done, actually creating the indexes in MongoDB is relatively easy.

Indexes in MongoDB are conceptually similar to those in RDBMSes like MySQL. You will want an index in MongoDB in the same sort of situations where you would have wanted an index in MySQL.

Basics

An index is a data structure that collects information about the values of the specified fields in the documents of a collection. This data structure is used by Mongo's query optimizer to quickly sort through and order the documents in a collection. Formally speaking, these indexes are implemented as "B-Tree" indexes.

In the shell, you can create an index by calling the ensureIndex() function, and providing a document that specifies one or more keys to index.
Referring back to our examples database from Mongo Usage Basics, we can index on the 'j' field as follows:

```javascript
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:

```javascript
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

```javascript
db.things.getIndexes()
```

in the shell to see the existing indexes on the collection. Run

```javascript
db.system.indexes.find()
```

to see all indexes for the database.

`ensureIndex()` creates the index if it does not exist. A standard index build will block all other database operations. If your collection is large, the build may take many minutes or hours to complete - if you must build an index on a live MongoDB instance, we suggest that you build it in the background using the `background : true` option. This will ensure that your database remains responsive even while the index is being built. Note, however, that background indexing may still affect performance, particularly if your collection is large.

> If you use replication, background index builds will block operations on the secondaries. To build new indices on a live replica set, it is recommended you follow the steps described here.

In many cases, not having an index at all can impact performance almost as much as the index build itself. If this is the case, we recommend the application code check for the index at startup using the chosen mongoDB driver's `getIndexes()` function and terminate if the index cannot be found. A separate indexing script can then be explicitly invoked when safe to do so.

**The `_id` Index**

For all collections except capped collections, an index is automatically created for the `_id` field. This index is special and cannot be deleted. The `_id` index enforces uniqueness for its keys (except for some situations with sharding). `_id` values are invariant.

**Indexing on Embedded Fields ("Dot Notation")**

With MongoDB you can even index on a key inside of an embedded document. Reaching into sub-documents is referred to as Dot Notation. For example:

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

**Compound Keys**

In addition to single-key basic indexes, MongoDB also supports multi-key "compound" indexes. Just like basic indexes, you use the `ensureIndex()` function in the shell to create the index, but instead of specifying only a single key, you can specify several:

```javascript
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
you can use it query on

\[ a \]

\[ a, b \]

\[ a, b, c \]

**v1.6+**

Now you can also use the compound index to service any combination of equality and range queries from the constitute fields. If the first key of the index is present in the query, that index may be selected by the query optimizer. If the first key is not present in the query, the index will only be used if hinted explicitly. While indexes can be used in many cases where an arbitrary subset of indexed fields are present in the query, as a general rule the optimal indexes for a given query are those in which queried fields precede any non queried fields.

### Indexing Array Elements

When a document's stored value for a index key field is an array, MongoDB indexes each element of the array. See the [Multikeys](#) page for more information.

### Creation Options

The second argument for `ensureIndex` is a document/object representing the options. These options are explained below.

<table>
<thead>
<tr>
<th>option</th>
<th>values</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>background</td>
<td>true/false</td>
<td>false. see doc page for caveats</td>
</tr>
<tr>
<td>dropDups</td>
<td>true/false</td>
<td>false</td>
</tr>
<tr>
<td>sparse</td>
<td>true/false</td>
<td>false</td>
</tr>
<tr>
<td>unique</td>
<td>true/false</td>
<td>false</td>
</tr>
<tr>
<td>v</td>
<td>index version. 0 = pre-v2.0, 1 = smaller/faster (current)</td>
<td>1 in v2.0. Default is used except in unusual situations.</td>
</tr>
</tbody>
</table>

`name` is also an option but need not be specified and will be deprecated in the future. The name of an index is generated by concatenating the names of the indexed fields and their direction (i.e., 1 or -1 for ascending or descending). Index names (including their namespace/database), are limited to 128 characters.

**sparse: true**

**v1.8+**

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

A "sparse index" is an index that only includes documents with the indexed field. Any document that is missing the sparsely indexed field will not be stored in the index; the index will therefor be `sparse` because of the missing documents when values are missing.

Sparse indexes, by definition, are not complete (for the collection) and behave differently than complete indexes. When using a "sparse index" for sorting (or in some cases just filtering) some documents in the collection may not be returned. This is because only documents in the index will be returned.
You can combine sparse with unique to produce a unique constraint that ignores documents with missing fields.

Note that MongoDB's sparse indexes are not block-level indexes. MongoDB sparse indexes can be thought of as dense indexes with a specific filter.

```
unique:true
```

MongoDB indexes may optionally impose a unique key constraint, which guarantees that no documents are inserted whose values for the indexed keys match those of an existing document.

Note that if a document is missing the indexed field, the value is treated as null for that document. If the index is unique, two documents cannot both have the null value. If you want to allow missing fields or nulls for your unique indexed field, be sure to also use the sparse option.

Here are some examples using unique index constraints:

```
// everyone's username must be unique:
db.things.ensureIndex({email:1},{unique:true});

// in this variation, it's ok to not have an email address, // but if you have one, it must be unique:
db.things.ensureIndex({email:1},{unique:true,sparse:true});

// a compound index example. firstname+lastname combination must be unique:
db.things.ensureIndex({firstname: 1, lastname: 1}, {unique: true});
```

For non-sparse indexes (see the sparse option documentation), if a field is not present its value is treated as null for purposes of indexing. This is important to keep in mind when writing an application. For example:

```
db.customers.ensureIndex({firstname:1,lastname:1},{unique:true});
assert ( db.customers.count() == 0 );
db.customers.insert({firstname:'jane',lastname:'doe'}); // ok
db.customers.insert({firstname:'jane',lastname:'doe'}); // dup key error
db.customers.insert({firstname:'jane',lastname:'smith'}); // ok
db.customers.insert({firstname:'jane'}); // ok, treated as {firstname:'jane',lastname:undefined}
db.customers.insert({lastname:'smith'}); // ok
db.customers.insert({firstname:'john',lastname:'smith'}); // ok
db.customers.insert({firstname:'jane'}); // dup key error
db.customers.insert({email:'sally@abc.com',age:33}); // ok, treated as {firstname:undefined,lastname:undefined}
db.customers.insert({email:'pete@abc.com',age:39}); // dup key error
```

```
dropDups:true
```

A unique index cannot be created on a key that has pre-existing duplicate values. If you would like to create the index anyway, keeping the first document the database indexes and deleting all subsequent documents that have duplicate values, add the dropDups option.
db.things.ensureIndex({firstname : 1}, {unique : true, dropDups : true})

dropDups deletes data. A “fat finger” with drop dups could delete almost all data from a collection. Backup before using. Note also that if the field is missing in multiple records, that evaluates to null, and those would then be consider duplicates – in that case using sparse, or not using dropDups, would be very important.

**background:true**

By default, building an index blocks other database operations. v1.4+ has a background index build option – however this option has significant limitations in a replicated cluster (see doc page).

**Dropping Indexes**

To delete all indexes on the specified collection:

```javascript
db.collection.dropIndexes();
```

To delete a single index:

```javascript
db.collection.dropIndex({x: 1, y: -1})
```

Running directly as a command without helper:

```javascript
// 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.

```javascript
db.myCollection.reIndex()
```

See here for more documentation: [reIndex Command](#)

**Performance Notes**

**Updates**

When you update an object, if the object fits in its previous allocation area, only those indexes whose keys have changed are updated. This improves performance. Note that if the object has grown and must move, all index keys must then update, which is slower.

**How many indexes?**

Indexes make retrieval by a key, including ordered sequential retrieval, very fast. Updates by key are faster too as MongoDB can find the document to update very quickly.

However, keep in mind that each index created adds a certain amount of overhead for inserts and deletes. In addition to writing data to the base collection, keys must then be added to the B-Tree indexes. Thus, indexes are best for collections where the number of reads is much greater than the number of writes. For collections which are write-intensive, indexes, in some cases, may be counterproductive. Most collections are read-intensive, so indexes are a good thing in most situations.

**Using sort() without an Index**

You may use `sort()` to return data in order without an index if the data set to be returned is small (less than 32 megabytes in version 2.0, or less than four megabytes in version 1.8 and earlier). For these cases it is best to use `limit()` and `sort()` together.
Additional Notes

Behaviors

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

Using Documents as Keys

Indexed fields may be of any type, including (embedded) documents:

```javascript
db.factories.insert( { name: "xyz", metro: { city: "New York", state: "NY" } } );
db.factories.ensureIndex( { metro : 1 } );
// this query can use the above index:
db.factories.find( { metro: { city: "New York", state: "NY" } } );
// this one too, as {city:"New York"} < {city:"New York",state: "NY"}
db.factories.find( { metro: { $gte : { city: "New York" } } } );

// this query does not match the document because the order of fields is significant
db.factories.find( { metro: { state: "NY", city: "New York" } } );
```

An alternative to documents as keys is to create a compound index:

Document indexes like these cannot be used to search for queries using their (embedded) parts, like "state" in this example. When you search you must use a prefix, or the whole document, of the embedded document as if they were stored as opaque blobs; that is essentially how they are stored in the index entry. This is even more understandable when you think about how each embedded document could include any number of fields and/or types of data.

```javascript
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().sort( { "metro.city": 1, "metro.state": 1 } );
```

There are pros and cons to the two approaches. When using the entire (sub-)document as a key, compare order is predefined and is ascending key order in the order the keys occur in the BSON document. With compound indexes reaching in, you can mix ascending and descending keys, and the query optimizer will then be able to use the index for queries on solely the first key(s) in the index too.

Keys Too Large To Index

Index entries have a limitation on their maximum size (the sum of the values), currently 1024 bytes (prior to v2.0 the maximum size was 819 bytes). Documents whose fields have values (key size in index terminology) greater than this size cannot be indexed. You will see log messages similar to:

```
...Btree::insert: key too large to index, skipping...
```

Queries against this index will not return the unindexed documents. You can force a query to use another index, or really no index, using this special index hint:

```javascript
db.myCollection.find({<key>: <value too large to index>}).hint({$natural: 1})
```

This will cause the document to be used for comparison of that field (or fields), rather than the index.

This limitation will eventually be removed (see SERVER-3372).

Presentations

Indexing and Query Optimization - Presentation from MongoSV (December 2011)

More Presentations

- Video introduction to indexing and the query optimizer
- More advanced video and accompanying slides, with many examples and diagrams
Intermediate level webinar and accompanying slides
Another set of intermediate level slides

Index Versions

- Converting an Existing Index to `{v:1}` format
- Rolling Back to MongoDB < v2.0
- Building a `{v:0}` Index

MongoDB v2.0 uses a new data structure for its indexes. The new `{v:1}` indexes are, on average, 25% smaller than the old `{v:0}` indexes. This usually results in significantly increased performance. In addition, signed (pre 1970) dates are supported in `{v:1}` indexes.

Backward-compatibility with `{v:0}` indexes is maintained. Thus, you can upgrade to MongoDB v2.0 seamlessly. However, to get the benefits of the new index format, you'll need to reindex.

Converting an Existing Index to `{v:1}` format

All operations that create a new index will result in a `{v:1}` index by default.

- **Reindexing** will result in a replacement of any `{v:0}` index of with a `{v:1}` type.
  - Due to SERVER-3866, reindexing on a secondary does not work in <=v2.0.0. Do not reindex on a secondary as the drop phase of reindex is done and then the rest of the work aborts. See jira for workaround.
  - A `compact command` invocation will convert all indexes for the given collection to `{v:1}` type.

In v2.0.1, the following operations will convert `{v:0}` indexes to `{v:1}` type:

- The `repair database command`
- Creation of a new `replica set` secondary from scratch (initial sync).

Rolling Back to MongoDB < v2.0

While MongoDB v2.0 supports the old index format, old versions will not support the new format. If you need to roll back to an old version, the server will run, but queries and other operations involving the newer indexes will log and return an error. Thus, you'll need to recreate any new index you'd like to use on an old server.

Versions >= 1.8.3 are aware of the index version field, but version <= 1.8.2 are not. So if you rollback a `{v:1}` index to 1.8.2 and re-index it its version will still be marked `{v:1}` although it actual version is `{v:0}`. Then if you upgrade again to 2.0 this index won't work even though they are marked as `{v:1}` in `db.system.indexes`. So if you have to rollback to a version <= 1.8.2, you must delete the index then create it again (instead of simply re-indexing).

Building a `{v:0}` Index

You can still create a `{v:0}` index with MongoDB v2.0. To do so, add the option `{v:0}` in the index creation command. For example in the `mongo shell`:

```
> // defaults to a v:1 index
> db.foo.ensureIndex({name:1})
> db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "mydb.foo", "name" : "_id_" }
{ "v" : 1, "key" : { "name" : 1 }, "ns" : "mydb.foo", "name" : "name_1" }
>
> // create a v:0 index
> db.foo.ensureIndex({name:1},{v:0})
> db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "mydb.foo", "name" : "_id_" }
{ "v" : 0, "key" : { "name" : 1 }, "ns" : "mydb.foo", "name" : "name_1" }
```

Geospatial Indexing

- Creating the Index
- Querying
- Compound Indexes
- `geoNear` Command
- Bounds Queries
- The Earth is Round but Maps are Flat
  - New Spherical Model
MongoDB supports two-dimensional geospatial indexes. It is designed with location-based queries in mind, such as "find me the closest N items to my location." It can also efficiently filter on additional criteria, such as "find me the closest N museums to my location."

In order to use the index, you need to have a field in your object that an array where the first 2 elements are x,y coordinates (or y,x - just be consistent; it might be advisable to use order-preserving dictionaries/hashes in your client code, to ensure consistency).

To make sure ordering is preserved from all languages use a 2 element array

```
[ x, y ]
```

Some examples:

```
{ loc : [ 50 , 30 ] } //SUGGESTED OPTION
{ loc : { x : 50 , y : 30 } }
{ loc : { foo : 50 , y : 30 } }
{ loc : { lon : 40.739037, lat: 73.992964 } }
```

Creating the Index

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

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

The index space bounds are inclusive of the lower bound and exclusive of the upper bound.

If you are indexing something else, you can specify some options:

```
db.places.ensureIndex( { loc : "2d" }, { min : -500 , max : 500 } )
```

that will scale the index to store values between -500 and 500. Bounded geospatial searches are currently limited to rectangular and circular areas with no "wrapping" at the outer boundaries. You cannot insert values outside the boundary interval [min, max). For example, using the code above, the point (-500, 500) could not be inserted and would raise an error (the point (-500, 499), however, would be fine).

Pre-1.9 releases of mongo do not allow the insertion of points exactly at the boundaries.

```
db.places.ensureIndex( { loc : "2d" }, { bits : 26 } )
```

The bits parameter sets the precision of the 2D geo-hash values, the smallest "buckets" in which locations are stored. By default, precision is set to 26 bits which is equivalent to approximately 2 feet given (longitude, latitude) location values and default (-180, 180) bounds. To index other spaces which may have very large bounds, it may be useful to increase the number of bits up to the maximum of 32.

You may only have 1 geospatial index per collection, for now. While MongoDB may allow to create multiple indexes, this behavior is unsupported. Because MongoDB can only use one index to support a single query, in most cases, having multiple geo indexes will produce undesirable behavior.

Implicit array expansion syntax is only supported in v1.9+, where "foo.bar" : "2d" may reference a nested field like so:
This restriction holds even if there are not multiple locations per document and the array size is 1. In older versions you will need to embed the nested location without a parent array:

```javascript
{ foo : { bar : [ ... ] } }
```

### Querying

The index can be used for exact matches:

```javascript
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:

```javascript
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):

```javascript
db.places.find( { loc : { $near : [50,50] } } ).limit(20)
```

You can also use $near with a maximum distance

```javascript
db.places.find( { loc : { $near : [50,50] , $maxDistance : 5 } } ).limit(20)
```

Prior to v1.9.1, geospatial indexes can be used for exact lookups only when there is no other criteria specified in the query and exact locations are specified in arrays. For any type of $near, $within, or geoNear query, this restriction does not apply and any additional search criteria can be used.

All distances in geospatial queries are specified in the same units as the document coordinate system (aside from spherical queries, discussed below). For example, if your indexed region is of size [300, 300], representing a 300 x 300 meter field, and you have documents at locations (10, 20) and (10, 30), representing objects at points in meters (x, y), you could query for points $near : [10, 20], $maxDistance : 10. The distance unit is the same as in your coordinate system, and so this query looks for points up to 10 meters away.

When using longitude and latitude, which are angular measures, distance is effectively specified in approximate units of "degrees," which vary by position on the globe but can very roughly be converted to distance using 69 miles per degree latitude or longitude. The maximum error in northern or southernmost populated regions is -2x longitudinally - for many purposes this is acceptable. Spherical queries (below) take the curvature of the earth into account.

### Compound Indexes

MongoDB geospatial indexes optionally support specification of secondary key values. If you are commonly going to be querying on both a location and other attributes at the same time, add the other attributes to the index. The other attributes are annotated within the index to make filtering faster. For example:

```javascript
db.places.ensureIndex( { location : "2d", category : 1 } );
db.places.find( { location : { $near : [50,50] }, category : 'coffee' } );
```

Limits in geospatial queries are always applied to the geospatial component first - this can cause unexpected results when also re-sorting results by additional criteria, pending resolution of SERVER-4247.
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".

```javascript
> db.runCommand( { geoNear : "places" , near : [50,50], num : 10 } )
{ 
  "ns" : "test.places",
  "near" : "11001100000011111110000001111110000001111110000001111",
  "results" : [
    { 
      "dis" : 69.29646421910687,
      "obj" : { 
        "_id" : ObjectId("4b8bd6b93b83c574d8760280"),
        "y" : [ 1,
            1,
          ],
        "category" : "Coffee"
      },
    },
    { 
      "dis" : 69.29646421910687,
      "obj" : { 
        "_id" : ObjectId("4b8bd6b03b83c574d876027f"),
        "y" : [ 1,
            1,
          ]
      } 
    }
  ],
  "stats" : { 
    "time" : 0,
    "btreelocs" : 1,
    "btreeindex" : 1,
    "nscanned" : 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:

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

query can be any regular mongo query.

Bounds Queries

$within can be used instead of $near to find items within a shape. Results are not sorted by distance, which may result in faster queries when this sorting is not required. Shapes of type $box (rectangles), $center (circles), and $polygon (concave and convex polygons) are supported. All bounds queries implicitly include the border of the shape as part of the boundary, though due to floating-point inaccuracy this can’t strictly be relied upon.

To query for all points within a rectangle, you must specify the lower-left and upper-right corners:
A circle is specified by a center point and radius:

```javascript
> center = [50, 50]
> radius = 10
> db.places.find({ "loc": { "$within": { "$center": center, "$radius": radius } } })
```

A polygon is specified by an array or object of points, where each point may be specified by either an array or an object. The last point in the polygon is implicitly connected to the first point in the polygon.

```javascript
> polygonB = { a : { x : 10, y : 20 } , b : { x : 15, y : 25 } , c : { x : 20, y : 20 } } 
> db.places.find({ "loc": { "$within": { "$polygon": polygonA } } }) 
> db.places.find({ "loc": { "$within": { "$polygon": polygonB } } })
```

Polygon searches are strictly limited to looking for points inside polygons, polygon shapes in documents can't currently be indexed in MongoDB.

!!! IMPORTANT

Polygon searches are supported in versions >= 1.9

---

**The Earth is Round but Maps are Flat**

The current implementation assumes an idealized model of a flat earth, meaning that an arc degree 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 arc degree of longitude is about 52 miles or 83 km (latitude is unchanged). This means that something 1 mile to the north would seem closer than something 1 mile to the east.

**New Spherical Model**

v1.8+

Spherical distances can be used by adding "Sphere" to the name of the query. For example, use `$nearSphere` or `$centerSphere` ( `$boxSphere` and `$polygonSphere` don't make as much sense and so aren't supported). If you use the geoNear command to get distance along with the results, you just need to add `spherical: true` to the list of options.

There are a few caveats that you must be aware of when using spherical distances. The biggest is:

1. The code assumes that you are using decimal degrees in \((\text{longitude}, \text{latitude})\) order. This is the same order used for the GeoJSON spec. Using (latitude, longitude) will result in very incorrect results, but is often the ordering used elsewhere, so it is good to double-check.

   The names you assign to a location object (if using an object and not an array) are completely ignored, only the ordering is detected. A few examples:

   ```javascript
   /* assuming longitude is 13, latitude is -50 */
   [13, -50] // ok
   [ x : 13, y : -50 ] // ok
   [ lon : 13, lat : -50 ] // ok
   /* wrong, will make lat = longitude and lon = latitude */
   [ lat : -50, lon : 13 ]
   ```

   As above, the use of order-preserving dictionaries is required for consistent results.

   Also:

   1. All distances use radians. This allows you to easily multiply by the radius of the earth (about 6371 km or 3959 miles) to get the distance in your choice of units. Conversely, divide by the radius of the earth when doing queries.
   2. We don't currently handle wrapping at the poles or at the transition from -180° to +180° longitude, however we detect when a search would wrap and raise an error.
   3. While the default Earth-like bounds are \([-180, 180]\), valid values for latitude are between -90° and 90°.

**Spherical Example**
Below is a simple example of a spherical distance query, demonstrating how to convert a specified range in kilometers to a maxDistance in radians as well as converting the returned distance results from radians back to kilometers. The same conversion of kilometer to radian distance bounds is required when performing bounded $nearSphere and $centerSphere queries.

```javascript
> db.points.insert({ pos : { lon : 30, lat : 30 } })
> db.points.insert({ pos : { lon : -10, lat : -20 } })
> db.points.ensureIndex({ pos : "2d" });
> var earthRadius = 6378 // km
> var range = 3000 // km
> distances = db.runCommand({ geoNear : "points", near : [0, 0], spherical : true, maxDistance : range / earthRadius /* to radians */ }).results
[ { "dis" : 0.3886630122897946,
  "obj" : { 
    "_id" : ObjectId("4d9123026ccc7e2cf22925c4"),
    "pos" : { 
      "lon" : -10,
      "lat" : -20
    }
  }
}
]
> pointDistance = distances[0].dis * earthRadius // back to km
2478.89269238431
```

**Multi-location Documents**

**v.1.9+**

MongoDB now also supports indexing documents by multiple locations. These locations can be specified in arrays of sub-objects, for example:

```javascript
> 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:

```javascript
> db.places.insert({ lastSeenAt : [ { x : 45.3, y : 32.2 }, [54.2, 32.3], { lon : 44.2, lat : 38.2 } ] })
> db.places.ensureIndex({ "lastSeenAt" : "2d" })
```

By default, when performing geoNear or $near-type queries on collections containing multi-location documents, the same document may be returned multiple times, since $near queries return ordered results by distance. Queries using the $within operator by default do not return duplicate documents.

**v2.0**

In v2.0, this default can be overridden by the use of a $uniqueDocs parameter for geoNear and $within queries, like so:

```javascript
> db.runCommand( { geoNear : "places", near : [50,50], num : 10, uniqueDocs : false } )
> db.places.find( { loc : { $within : { $center : [[0.5, 0.5], 20], $uniqueDocs : true } } } )
```
Currently it is not possible to specify $uniqueDocs for $near queries

Whether or not uniqueDocs is true, when using a limit the limit is applied (as is normally the case) to the number of results returned (and not to the docs or locations). If running a geoNear query with uniqueDocs : true, the closest location in a document to the center of the search region will always be returned - this is not true for $within queries.

In addition, when using geoNear queries and multi-location documents, often it is useful to return not only distances, but also the location in the document which was used to generate the distance. In v2.0, to return the location alongside the distance in the geoNear results (in the field loc), specify includeLocs : true in the geoNear query. The location returned will be a copy of the location in the document used.

If the location was an array, the location returned will be an object with "0" and "1" fields in v2.0.0 and v2.0.1.

```javascript
> db.runCommand({ geoNear : "places", near : [ 0, 0 ], maxDistance : 20, includeLocs : true })

"results" : [
    {
        "dis" : 5.830951894845301,
        "loc" : {
            "x" : 3,
            "y" : 5
        },
        "obj" : {
            "_id" : ObjectId("4e52672c15f59224bdb2544d"),
            "name" : "Final Place",
            "loc" : {
                "x" : 3,
                "y" : 5
            }
        }
    },
    {
        "dis" : 14.142135623730951,
        "loc" : {
            "0" : 10,
            "1" : 10
        },
        "obj" : {
            "_id" : ObjectId("4e5266a915f59224bdb2544b"),
            "name" : "Some Place",
            "loc" : [
                [ 10, 10 ],
                [ 50, 50 ]
            ]
        }
    },
    {
        "dis" : 14.142135623730951,
        "loc" : {
            "0" : -10,
            "1" : -10
        },
        "obj" : {
            "_id" : ObjectId("4e5266ba15f59224bdb2544c"),
            "name" : "Another Place",
            "loc" : {
                [ -10, -10 ],
                [ -50, -50 ]
            };
```
Sharded Collections

v1.8+. Creating a geospatial index for a sharded collection is supported with some caveats: see [http://jira.mongodb.org/browse/SHARDING-83](http://jira.mongodb.org/browse/SHARDING-83). There are no caveats for using geospatial indexes with unsharded collections in a sharded cluster.

Implementation

The current implementation encodes geographic hash codes atop standard MongoDB B-trees. Results of $near queries are exact. One limitation with this encoding, while fast, is that prefix lookups don't give exact results, especially around bit flip areas. MongoDB solves this by doing a grid-neighbor search after the initial prefix scan to pick up any straggler points. This generally ensures that performance remains very high while providing correct results.

Presentations

- Geospatial Indexing with MongoDB - MongoSF (May 2011)
- Storing and Querying location data with MongoDB - MongoSF (May 2011)
- Community blog posts

Geospatial Haystack Indexing

In addition to ordinary 2d geospatial indices, mongodb supports the use of bucket-based geospatial indexes. Called "Haystack indexing", these indices can accelerate small-region type longitude / latitude queries when additional criteria is also required. For example, "find all restaurants within 25 miles with name 'foo'". Haystack indices allow you to tune your bucket size to the distribution of your data, so that in general you search only very small regions of 2d space for a particular kind of document. They are not suited for finding the closest documents to a particular location, when the closest documents are far away compared to bucket size.

For now, only a single coordinate field and optional single additional field can be used in a haystack index.

To use haystack indexing, documents with a (longitude, latitude) position stored as a sub-document or array are required, with an optional additional field to be indexed. For example:

```javascript
> db.foo.insert({ pos : { long : 34.2, lat : 33.3 }, type : "restaurant" })
> db.foo.insert({ pos : { long : 34.2, lat : 37.3 }, type : "restaurant" })
> db.foo.insert({ pos : { long : 59.1, lat : 87.2 }, type : "office" })
...
> db.foo.ensureIndex({ pos : "geoHaystack", type : 1 }, { bucketSize : 1 })
```

The bucketSize parameter is required, and determines the granularity of the bucket index - our value of 1 above creates an index where keys within 1 unit of longitude or latitude are stored in the same bucket.

The haystack index can only be used by a database command, it is not at present chosen by the query optimizer. As an example of finding all restaurants in a particular area with a given maximum distance of 6 degrees longitude / latitude, with a maximum of 30 results returned (by default, there is a 50 document result limit):
Spherical queries are not currently supported by haystack indices.

Indexing as a Background Operation

Slaves and replica secondaries build indexes in the foreground even if background:true is specified. The slave/secondary will block queries while the index builds on it. Indexing on the secondaries begins after the index is completely built on the primary.

v1.4+

By default the ensureIndex() operation is blocking, and stops other operations on the database from proceeding until completed.

To build an index in the background, add background:true to your index options. Examples:

```javascript
> db.things.ensureIndex({x:1}, {background: true});
true
> db.things.ensureIndex({name:1}, {background: true, unique: true, ... dropDups: true});
```

The index build operation is not a background build when it replicates to secondaries. This could be a significant issue if you are reading from your secondaries – in this case back up each replica set member one by one – see the doc page for this.

With background mode enabled, other non-indexing operations, including writes, will not be obstructed during index creation. The index is not used for queries until the build is complete. Further calls to ensureIndex() while the background indexing is progressing will fail with a message indicating a background operation is in progress.

Although the operation is 'background' in the sense that other operations may run concurrently, the command will not return to the shell prompt until completely finished. To do other operations at the same time, open a separate mongo shell instance.

Please note that background mode building uses an incremental approach to building the index which is slower than the default foreground mode: time to build the index will be greater. If the resulting index is larger than RAM, background indexing can be much slower.

While the index build progresses, it is possible to see that the operation is still in progress with the db.currentOp() command (will be shown as an insert to system.indexes). You may also use db.killOp() to terminate the index build on the primary.
While the build progresses, the index is visible in system.indexes, but it is not used for queries until building completes.

**Notes**

- Only one index build at a time, whether foreground or background, is permitted per collection.
- Some administrative operations, such as `repairDatabase`, are disallowed while a background indexing job is in progress.

**Multikeys**

MongoDB provides an interesting “multikey” feature that can automatically index arrays of an object’s values. A good example is tagging. Suppose you have an article tagged with some category names:

```
$ dbshell
> db.articles.save( { name: "Warm Weather", author: "Steve",
>                    tags: ['weather', 'hot', 'record', 'april'] } )

> db.articles.find()
{"name" : "Warm Weather", "author" : "Steve",
  "tags" : "weather","hot","record","april"}

> db.articles.find( { tags: 'april' } )
{"name" : "Warm Weather", "author" : "Steve",
  "tags" : "weather","hot","record","april"}
```

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"}
```

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"}
```

```
> db.articles.find( { tags: 'april' } ).explain()
{"cursor" : "BtreeCursor tags_1", "startKey" : "tags" : "april"},
  "endKey" : "tags" : "april"}, "nscanned" : 1, "n" : 1, "millis" : 0 }
```

**Incrementally adding and removing keys**

You can use `$addToSet` to add a new key to the array, and `$pull` to remove one.

```
> db.articles.update({name: "Warm Weather"},{$addToSet:{tags: "northeast"}});
...
> db.articles.update({name: "Warm Weather"},{$pull:{tags: "northeast"}});
```

**Embedded object fields in an array**

The same technique can be used to find fields within objects embedded in arrays:
// find posts where julie commented
> db.posts.find( { "comments.author" : "julie" } )
{ "title" : "How the west was won",
 "comments" : [{ "text" : "great!", "author" : "sam" },
{ "text" : "ok", "author" : "julie" }],
"_id" : "497ce79f1ca9ca6d3efca325"}

Querying on all values in a given set

By using the $all query option, a set of values may be supplied each of which must be present in a matching object field. For example:

> db.articles.find( { tags: [ $all: [ 'april', 'record' ] ] } )
{ "name" : "Warm Weather", "author" : "Steve",
 "tags" : ["weather","hot","record","april"],
"_id" : "497ce4051ca9ca6d3efca323"}

> db.articles.find( { tags: [ $all: [ 'april', 'june' ] ] } )
> // no matches

Exact Array Matching with an Index

Since indexing an array, which creates the multikey index, only indexes each element in the array it is not possible to do an exact array match using an index on the array field. The index will be used to lookup a subset of the values (currently the first one) and then the document will be inspected for the exact match. This is also true for range queries, but the rules for types are bit more complicated since it follows the matching rules for single values.
Caveats with Parallel Arrays

When using a compound index, at most one of indexed values in any document can be an array. So if we have an index on {a: 1, b: 1}, the following documents are both fine:

(a: [1, 2], b: 1)
(a: 1, b: [1, 2])

This document, however, will fail to be inserted, with an error message "cannot index parallel arrays".
The problem with indexing parallel arrays is that each value in the cartesian product of the compound keys would have to be indexed, which can get out of hand very quickly.

See Also

- The Multikeys section of the Full Text Search in Mongo document for information about this feature.

Using Multikeys to Simulate a Large Number of Indexes

One way to work with data that has a high degree of options for queryability is to use the multikey indexing feature where the keys are objects. For example:

```javascript
> x = {
...   _id : "abc",
...   cost : 33,
...   attribs : [
...       { color : 'red' },
...       { shape : 'rect' },
...       { color : 'blue' },
...       { avail : true }
...   ];
> db.foo.insert(x);
> db.foo.ensureIndex({attribs:1});
> db.foo.find( { attribs : {color:'blue'} } ); // uses index
> db.foo.find( { attribs : {avail: false} } ); // uses index
```

In addition to being able to have an unlimited number of attributes types, we can also add new types dynamically.

This is mainly useful for simply attribute lookups; the above pattern is not necessary helpful for sorting or certain other query types.

See Also

Discussion thread MongoDB for a chemical property search engine for a more complex real world example.

Indexing Advice and FAQ

We get a lot of questions about indexing. Here we provide answers to a number of these. There are a couple of points to keep in mind, though. First, indexes in MongoDB work quite similarly to indexes in MySQL, and thus many of the techniques for building efficient indexes in MySQL apply to MongoDB.

Second, and even more importantly, know that advice on indexing can only take you so far. The best indexes for your application should always be based on a number of important factors, including the kinds of queries you expect, the ratio of reads to writes, and even the amount of free memory on your system. This means that the best strategy for designing indexes will always be to profile a variety of index configurations with data sets similar to the ones you'll be running in production, and see which perform best. There's no substitute for good empirical analyses.

Note: if you're brand new to indexing, you may want to read this introductory article first.

- **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?
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:

```javascript
db.posts.find({ slug: 'state-of-mongodb-2010' })
```

In this case, a unique index on a single key is best:

```javascript
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:

```javascript
db.comments.find({ tags: 'mongodb' }).sort({ created_at: -1 });
```

And here’s the proper index:

```javascript
db.comments.ensureIndex({ tags: 1, created_at: -1 });
```

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. The only exception to this is

```javascript
$or
```

which can run multiple query plans and then de-dup the results.

Make sure your indexes can fit in RAM.

The shell provides a command for returning the total index size on a given collection:

```javascript
db.comments.totalIndexSize();
```

If your queries seem sluggish, you should verify that your indexes are small enough to fit in RAM. For instance, if you’re running on 4GB RAM and you have 3GB of indexes, then your indexes probably aren’t fitting in RAM. You may need to add RAM and/or verify that all the indexes you’ve created are actually being used.

Be careful about single-key indexes with low selectivity.

Suppose you have a field called ‘status’ where the possible values are ‘new’ and ‘processeed’. If you add an index on ‘status’ then you’ve created a low-selectivity index, meaning that the index isn’t going to be very helpful in locating records and might just be be taking up space.

A better strategy, depending on your queries, of course, would be to create a compound index that includes the low-selectivity field. For instance, you could have a compound-key index on ‘status’ and ‘created_at.’

Another option, again depending on your use case, might be to use separate collections, one for each status. As with all the advice here, experimentation and benchmarks will help you choose the best approach.

Use `explain`.

MongoDB includes an `explain` command for determining how your queries are being processed and, in particular, whether they’re using an index. `explain` can be used from of the drivers and also from the shell:
db.comments.find({ tags : 'mongodb'}).sort({ created_at : -1 }).explain();

This will return lots of useful information, including the number of items scanned, the time the query takes to process in milliseconds, which indexes the query optimizer tried, and the index ultimately used.

If you've never used `explain`, now's the time to start.

Understanding `explain`'s output.

There are three main fields to look for when examining the `explain` command's output:

- **cursor**: the value for cursor can be either `BasicCursor` or `BtreeCursor`. The second of these indicates that the given query is using an index.
- **nscanned**: the number of index entries 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:

```javascript
db.foo.ensureIndex({a: 1, b: 1, c: 1})
```

Here's some advice on using an index like this:

- **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)`

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.
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
- **Bulk inserts**

**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 MongoDB are JSON-like. JSON is a good way to store object-style data from programs in a manner that is language-independent and standards based.

To be efficient, MongoDB uses a format called **BSON** which is a binary representation of this data. BSON is faster to scan for specific fields than JSON. Also BSON adds some additional types such as a data data type and a byte-array (bindata) datatype. BSON maps readily to and from JSON and also to various data structures in many programming languages.

Client drivers serialize data to BSON, then transmit the data over the wire to the db. Data is stored on disk in BSON format. Thus, on a retrieval, the database does very little translation to send an object out, allowing high efficiency. The client driver unserialized a received BSON object to its native language format.
For example the following "document" can be stored in Mongo DB:

```json
{ 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:

```bash
> 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

```bash
> 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 work pretty well, it doesn't harness the real power of Mongo. Mongo is designed for and works best with a rich object model.

**Store Example**

If you're building a simple online store that sells products with a relation database, you might have a schema like:

```json
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

```json
item : {
  "title" : <title> ,
  "price" : <price> ,
  "sku" : <sku> ,
  "features" : {
    "optical zoom" : <value> ,
    ...
  }
}
```

This does a few nice things
• you can load an entire item with one query
• all the data for an item is on the same place on disk, thus only one seek is required to load

Now, at first glance there might seem to be some issues, but we’ve got them covered.

• you might want to insert or update a single feature. mongo lets you operate on embedded files like:

```
db.items.update( { sku : 123 } , { "$set" : { "features.zoom" : 5 } } )
```

• Does adding a feature require moving the entire object on disk? No. mongo has a padding heuristic that adapts to your data so it will leave some empty space for the object to grow. This will prevent indexes from being changed, etc.

**Bulk inserts**

It is possible to insert many documents in a single db call. Consult your driver’s documentation for how to do bulk inserts in that language. There is currently no support for bulk inserts from the mongo shell, see SERVER-2395.

MongoDB is quite fast at a series of singleton inserts. Thus one often does not need to use this specialized version of insert.

In v2.0+ one can set the ContinueOnError flag for bulk inserts to signal inserts should continue even if one or more from the batch fails. In that case, getLastError will be set if any insert fails, not just the last one. If multiple errors occur, only the most recent will be reported by getLastError.

For a sharded collection, ContinueOnError is implied and cannot be disabled.

**Legal Key Names**

Key names in inserted documents are limited as follows:

• The ‘$’ character must not be the first character in the key name.
• The ‘.’ character must not appear anywhere in the key name.

**Schema Design**

• Embedding and Linking
• Collections
• Atomic Operations
• Indexes
• Sharding
• Example
• Summary of Best Practices
• More Details
  • Choosing Indexes
  • How Many Collections?
• See Also
  • Books
  • Blog posts
  • Related Doc Pages
  • Videos

Schema design in MongoDB is very different from schema design in a relational DBMS. However it is still very important and the first step towards building an application.

In relational data models, conceptually there is a "correct" design for a given entity relationship model independent of the use case. This is typically a third normal form normalization. One typically only diverges from this for performance reasons. In MongoDB, the schema design is not only a function of the data to be modeled but also of the use case. The schema design is optimized for our most common use case. This has pros and cons – that use case is then typically highly performant; however there is a bias in the schema which may make certain ad hoc queries a little less elegant than in the relational schema.

As we design the schema, the questions we must answer are:

1. When do we embed data versus linking (see below)? Our decisions here imply the answer to question #2:
2. How many collections do we have, and what are they?
3. When do we need atomic operations? These operations can be performed within the scope of a BSON document, but not across documents.
4. What indexes will we create to make query and updates fast?
5. How will we shard? What is the shard key?
Embedding and Linking

A key question when designing a MongoDB schema is when to embed and when to link. Embedding is the nesting of objects and arrays inside a BSON document. Links are references between documents.

There are no joins in MongoDB – distributed joins would be difficult on a 1,000 server cluster. Embedding is a bit like “prejoined” data. Operations within a document are easy for the server to handle; these operations can be fairly rich. Links in contrast must be processed client-side by the application; the application does this by issuing a follow-up query.

Generally, for "contains" relationships between entities, embedding should be chosen. Use linking when not using linking would result in duplication of data.

Collections

Collections in MongoDB are analogous to tables in a relational database. Each collection contains documents. As mentioned above these documents can be fairly rich.

There is no explicit declaration of the fields within the documents of a collection. However there is a conceptual design from the schema designer of what those fields will be and how the documents in the collection will be structured. MongoDB does not require documents in a collection to have the same structure. However, in practice, most collections are highly homogenous. We can move away from this when we need to though; for example when adding a new field. In a situation such as that, an “alter table” style operation is not necessary.

Atomic Operations

Some problems require the ability to perform atomic operations. For example, simply incrementing a counter is often a case where one wants atomicity. MongoDB can also perform more complex operations such as that shown in the pseudocode below:

```plaintext
atomically {
    if( doc.credits > 5 ) {
        doc.credits -= 5;
        doc.debits += 5;
    }
}
```

Another example would be a user registration scenario. We would never want two users to register the same username simultaneously:

```plaintext
atomically {
    if( exists a document with username='jane' ) {
        print "username already in use please choose another";
    } else {
        insert a document with username='jane' in the users collection;
        print("thanks you have registered as user jane.");
    }
}
```

The key aspect here in terms of schema design is that our scope of atomicity / ACID properties is the document. Thus we need to assure that all fields relevant to the atomic operation are in the same document.

Indexes

MongoDB supports the declaration of indexes. Indexes in MongoDB are highly analogous to indexes in a relational database : they are needed for efficient query processing, and must be explicitly declared. Thus we need to think about what indexes we will define as part of the schema design process. Just like in a relational database, indexes can be added later – if we decide later to have a new one, we can do that.

Sharding

Another consideration for schema design is sharding. A BSON document (which may have significant amounts of embedding) resides on one and only one shard.

A collection may be sharded. When sharded, the collection has a shard key, which determines how the collection is partitioned among shards. Typically (but not always) queries on a sharded collection involve the shard key as part of the query expression.

The key here is that changing shard keys is difficult. You will want to choose the right key from the start.

Example

Let’s consider an example, which is a content management system. The examples below use mongo shell syntax but could also be done in any programming language – just use the appropriate driver for that language.
Our content management system will have posts. Posts have authors. We'd like to support commenting and voting on posts. We'd also like posts to be taggable for searching.

One good schema design for this scenario would be to have two MongoDB collections, one called posts and one called users. This is what we will use for the example.

Our users have a few properties - a user id they registered with, their real name, and their karma. For example we could invoke:

```javascript
> db.users.insert( { _id: "alex", name: { first:"Alex", last:"Benisson" }, karma : 1.0 } )
```

The _id field is always present in MongoDB, and is automically indexed with a unique key constraint. That is perfect for our usernames so we store them in the _id field. We don't have to though; we could instead make a username field and let the system automically generate a unique id.

Let's now consider posts. We'll assume some posts are already populated. Let's query one:

```javascript
> db.posts.findOne()
{ 
    _id : ObjectId("4e77bb3b8a3e0000000004f7a"),
    when : Date("2011-09-19T02:11:3Z"),
    author : "alex",
    title : "No Free Lunch",
    text : "This is the text of the post. It could be very long."
    tags : [ "business", "ramblings" ],
    votes : 5,
    voters : [ "jane", "joe", "spencer", "phyllis", "li" ],
    comments : [ 
        { who : "jane", when : Date("2011-09-19T04:10.112Z"),
          comment : "I agree." },
        { who : "meghan", when : Date("2011-09-20T14:36:06.958Z"),
          comment : "You must be joking. etc etc ..." } ]
}
```

It's interesting to contrast this with how we might design the same schema in a relation database. We would likely have a users collection and a posts collection. But in addition one would typically have a tags collection, a voters collection, and a comments collection. Grabbing all the information on a single post would then be a little involved. Here to grab the whole post we might execute:

```javascript
> db.posts.findOne( { _id : ObjectId("4e77bb3b8a3e0000000004f7a") } );
```

To get all posts written by alex:

```javascript
> db.posts.find( { author : "alex" } )
```

If the above is a common query we would create an index on the author field:

```javascript
> db.posts.ensureIndex( { author : 1 } )
```

The post documents can be fairly large. To get just the titles of the posts by alex :

```javascript
> db.posts.find( { author : "alex" }, { title : 1 } )
```

We may want to search for posts by tag:

```javascript
> // make and index of all tags so that the query is fast:
> db.posts.ensureIndex( { tags : 1 } )
> db.posts.find( { tags : "business" } )
```

What if we want to find all posts commented on by meghan?
Let's index that to make it fast:

```javascript
> db.posts.ensureIndex( { "comments.who" : 1 } )
```

We track voters above so that no one can vote more than once. Suppose Calvin wants to vote for the example post above. The following update operation will record Calvin's vote. Because of the `$nin` sub-expression, if Calvin has already voted, the update will have no effect.

```javascript
> db.posts.update( { _id : ObjectId("4e77bb3b8a3e000000004f7a"),
                  voters : { $nin : ["calvin"] },
                  votes : { $inc : 1 },
                  voters : { $push : "calvin" } } );
```

Note the above operation is atomic: if multiple users vote simultaneously, no votes would be lost.

Suppose we want to display the title of the latest post in the system as well as the full user name for the author. This is a case where we must use client-side linking:

```javascript
> var post = db.posts.find().sort( { when : -1 } ).limit(1);
> var user = db.users.find( { _id : post.author });
> print( post.title + " " + user.name.first + " " + user.name.last );
```

A final question we might ask about our example is how we would shard. If the users collection is small, we would not need to shard it at all. If posts is huge, we would shard it. We would need to choose a shard key. The key should be chosen based on the queries that will be common. We want those queries to involve the shard key.

- Sharding by `_id` is one option here.
- If finding the most recent posts is a very frequent query, we would then shard on the `when` field. (There is also an optimization trick which might work here.)

### Summary of Best Practices

- "First class" objects, that are at top level, typically have their own collection.
- Line item detail objects typically are embedded.
- Objects which follow an object modelling "contains" relationship should generally be embedded.
- Many to many relationships are generally done by linking.
- Collections with only a few objects may safely exist as separate collections, as the whole collection is quickly cached in application server memory.
- Embedded objects are a bit harder to link to than "top level" objects in collections.
- It is more difficult to get a system-level view for embedded objects. When needed an operation of this sort is performed by using MongoDB's `map/reduce` facility.
- If the amount of data to embed is huge (many megabytes), you may reach the limit on size of a single object. See also GridFS.
- If performance is an issue, embed.

### More Details

#### Choosing Indexes

A second aspect of schema design is index selection. As a general rule, *where you want an index in a relational database, you want an index in Mongo.*

- The `_id` field is automatically indexed.
- Fields upon which keys are looked up should be indexed.
- Sort fields generally should be indexed.

The MongoDB `profiling` facility provides useful information for where an index should be added that is missing.

Note that adding an index slows writes to a collection, but not reads. Use lots of indexes for collections with a high read : write ratio (assuming one does not mind the storage overhead). For collections with more writes than reads, indexes are expensive as keys must be added to each index for each insert.

#### How Many Collections?

As MongoDB collections are polymorphic, one could have a collection `objects` and put everything in it! This approach is taken by some object databases. This is not recommended in MongoDB for several reasons, mainly performance. Data within a single collection is roughly contiguous on disk. Thus, "table scans" of a collection are possible, and efficient. Just like in relational dbs, independent collections are very important for high throughput batch processing.
Tweaking performance by document bundling during schema design

Note: this page discusses performance tuning aspects – if you are just getting started skip this for later. If you have a giant collection of small documents that will require significant tuning, read on.

During schema design one consideration is when to embed entities in a larger document versus storing them as separate small documents. Tiny documents work fine and should be used when that is the natural way to go with the schema. However, in some circumstances, it can be better to group data into larger documents to improve performance.

Consider for example a collection which contains some documents that are fairly small. Documents are indicated in the figures below as squares. Related documents – perhaps all associated with some larger entity in our program, or else that correlate in their access, are indicated in figure 1 with the same color.

MongoDB caches data in pages, where the page size is that of the operating system's virtual memory manager (almost always 4KB). Page units are indicated by the black lines – for this example 8 boxes fit per page.

Let's suppose we wish to fetch all of the dark blue documents – indicates with stripes in figure 2. If this data is in RAM, we can (assuming we have an index) fetch them very efficiently. However note that the eight entities span eight pages, even though they could in theory fit on a single page.

With an alternate schema design we could "roll up" some of these entities into a larger document which includes an array of subdocuments. By doing that the items will be clustered together – a single BSON document in MongoDB is always stored contiguously. Figure 3 shows an example where the eight entities roll up into two documents (perhaps they could have rolled up to just one document; the point here is that it isn't essential that it be one, we are simply doing some bundling). In this example the two new documents are stored within three pages. While this isn't a huge reduction – eight to three – in many situations the documents are much smaller than a page – sometimes 100 documents fit within a single page. (The diagram example is not very granular to make reading easy.)

The benefits of this rolled-up schema design are

- Better RAM cache utilization. If we need to cache the dark blue items (but not the others), we can now cache three pages instead of eight. Note this is really only important if the data is too large to fit entirely in RAM – if it all fits, there is no gain here.
- Fewer disk seeks. If nothing was cached in RAM, less random i/o's are necessary to fetch the objects.
- Smaller index sizes. The common key the eight items contain can be stored in less copies, with less associated key entries in its corresponding index.
Caveats:

- Do not optimize prematurely; if grouping the entities would be awkward, don't do it. The goal of Mongo is to make development easier, not harder.
- Note we simply want to get to a document size that is on the order of the page cache unit size — about 4KB. If your documents are already of roughly that size, there is less benefit to the above (some still regarding random disk i/o, but no benefit for ram cache efficiency).
- If you often only need a subset of the items you would group, this approach could be inefficient compared to alternatives.

Trees in MongoDB

- Patterns
  - Full Tree in Single Document
  - Parent Links
  - Child Links
  - Array of Ancestors
  - Materialized Paths (Full Path in Each Node)
  - acts_as_nested_set
- Use Cases
  - Find Nodes by a Partial Path
- 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. Further there is a limit on the size of documents in MongoDB – 16MB in v1.8 (limit may rise in future versions).

Parent Links
Storing all nodes in a single collection, with each node having the id of its parent, is a simple solution. The biggest problem with this approach is getting an entire subtree requires several query turnarounds to the database (or use of db.eval).

```
> t = db.tree1;
> t.find()
{ "_id" : 1 }
{ "_id" : 2, "parent" : 1 }
{ "_id" : 3, "parent" : 1 }
{ "_id" : 4, "parent" : 2 }
{ "_id" : 5, "parent" : 4 }
{ "_id" : 6, "parent" : 4 }

> // find children of node 4
> t.ensureIndex({parent:1})
> t.find( {parent : 4 } )
{ "_id" : 5, "parent" : 4 }
{ "_id" : 6, "parent" : 4 }
```

Child Links
Another option is storing the ids of all of a node's children within each node's document. This approach is fairly limiting, although ok if no operations on entire subtrees are necessary. It may also be good for storing graphs where a node has multiple parents.

```
> t = db.tree2
> t.find()
{ "_id" : 1, "children" : [ 2, 3 ] }
{ "_id" : 2 }
{ "_id" : 3, "children" : [ 4 ] }
{ "_id" : 4 }

> // find immediate children of node 3
> t.findOne({_id:3}).children
[ 4 ]

> // find immediate parent of node 3
> t.ensureIndex({children:1})
> t.find({children:3})
{ "_id" : 1, "children" : [ 2, 3 ] }
```

Array of Ancestors
Here we store all the ancestors of a node in an array. This makes a query like "get all descendents of x" fast and easy.
> t = db.mytree;

> t.find()
{ "_id": "a" }
{ "_id": "b", "ancestors": [ "a" ], "parent": "a" }
{ "_id": "c", "ancestors": [ "a", "b" ], "parent": "b" }
{ "_id": "d", "ancestors": [ "a", "b" ], "parent": "b" }
{ "_id": "e", "ancestors": [ "a" ], "parent": "a" }
{ "_id": "f", "ancestors": [ "a", "e" ], "parent": "e" }
{ "_id": "g", "ancestors": [ "a", "b", "d" ], "parent": "d" }

> t.ensureIndex( { ancestors : 1 } )

> 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" }

> // 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:
Ruby example: http://github.com/banker/newsmonger/blob/master/app/models/comment.rb

acts_as_nested_set

See http://ar.rubyonrails.org/classes/ActiveRecord/Acts/NestedSet/ClassMethods.html

This pattern is best for datasets that rarely change as modifications can require changes to many documents.

Use Cases

Find Nodes by a Partial Path

Suppose we want to find a given node in a tree, given some path through a portion of the tree to that node, and then get back that node, and perhaps also everything below it.

With a materialized paths approach we can do the above. The main thing that needs tweaking is to make the operation fast if there is a path "a..b..c..d..e" to a document and we want to find documents with a path "..b..c..d..". If we are starting from the very top it is easy (and described above in the materialized paths section). However here we aren't starting at the top. One approach is to use a combination of materialized path plus an array of the node's ancestors, something like:

```javascript
{ path : "a,b,c,d,e,"
  ancestor : ['a','b','c','d','e'] }
```

We could index on ancestors which will create a multikey index. Then we would do a query like the following to find nodes on path "...b,c,d..." with some efficiency:

```
find({ path : /b,c,d/, ancestor : 'd', <more_query_expressions_optionally> })
```

In the above the index on ancestor would be used and only docs from 'd' down need be inspected. The following could be tried which might be even better depending on how smart the query optimizer is:
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:

```javascript
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:

```javascript
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.

```javascript
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:

```javascript
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:

```javascript
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.

```javascript
db.posts.ensureIndex({author:1});
db.posts.find({author:"george"}).count();
```

Increment Operations

MongoDB supports simple object field increment operations; basically, this is an operation indicating "increment this field in this document at the server". This can be much faster than fetching the document, updating the field, and then saving it back to the server and are particularly useful for implementing real time counters. See the Updates section of the Mongo Developers' Guide for more information.

Circular Fixed Size Collections

MongoDB provides a special circular collection type that is pre-allocated at a specific size. These collections keep the items within well-ordered even without an index, and provide very high-speed writes and reads to the collection. Originally designed for keeping log files - log events are stored in the database in a circular fixed size collection - there are many uses for this feature. See the Capped Collections section of the Mongo Developers' Guide for more information.

Server Side Code Execution

Occasionally, for maximal performance, you may wish to perform an operation in process on the database server to eliminate client/server network turnarounds. These operations are covered in the Server-Side Processing section of the Mongo Developers' Guide.

Explain

A great way to get more information on the performance of your database queries is to use the explain plan feature. See the Explain doc page.

Hint

While the mongo query optimizer often performs very well, explicit "hints" can be used to force mongo to use a specified index, potentially improving performance in some situations. When you have a collection indexed and are querying on multiple fields (and some of those fields are indexed), pass the index as a hint to the query.

To set the hint for a particular query, call the `hint()` function on the cursor before accessing any data, and specify a document with the key to be used in the query:

```javascript
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":

```javascript
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:

```javascript
> db.collection.find().hint({$natural:1})
```

See Also

- Query Optimizer
- `currentOp()`
- Sorting and Natural Order

Explain

- Basics
- Output Fields
  - `cursor`
  - `nsampled`
  - `nsampledObjects`
  - `nYields`
  - `n`
  - `millis`
  - `scanAndOrder`
  - `indexOnly`
  - `isMultiKey`
- With Sharding
  - `clusteredType`
  - `shards`
  - `nChunkSkips`
  - `millisTotal`
  - `millisAvg`
  - `numQueries`
  - `numShards`
- See Also

Basics

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

When using the `mongo shell`, invoke the `explain()` method on a cursor. The result will be a document that contains the explain output. Note that `explain` runs the actual query to determine the result. If the query is slow, the explain will be slow too.

```javascript
> db.collection.find(query).explain()
```

Some of the options below may not appear depending on the version of mongod you are running.

```javascript
> db.collection.find(query).explain()
```

provides information such as the following:
The type of cursor used for scanning.

- BasicCursor - indicates a table scan style operation
- BtreeCursor - an index was used. When an index is used, indexBounds will be set to indicate the key bounds for scanning in the index.

Number of items (documents or index entries) examined. Items might be objects or index keys. If a "covered index" is involved, nscanned may be higher than nsccessedObjects.

Number of documents scanned.

Number of times this query yielded the read lock to let waiting writes execute.

Number of documents matched (on all criteria specified).

Time for the query to run, in milliseconds.

True if the index could not be used for sorting.

True if the results of the query can be returned using only the index. See SERVER-5759 for more information on the behavior of this value.

If true, a multikey index was used.

In a sharded deployment, explain outputs some additional fields. An example:
clusteredType: "ParallelSort",
shards: {
  "shard1:27017": {
    "cursor": "BtreeCursor a_1",
    "nscanned": 1,
    "nscannedObjects": 1,
    "n": 1,
    "millis": 0,
    "nYields": 0,
    "nChunkSkips": 0,
    "isMultiKey": false,
    "indexOnly": false,
    "indexBounds": {
      "a": [
        1,
        1
      ]
    }
  }
}
}

clusteredType

- ParallelSort means all shards are accessed in parallel
- SerialServer means shards are queried one by one in order

shards

List of all the shards accessed during the query and the explain output for each shard.

nChunkSkips

The number of documents skipped because of active chunk migrations in a sharded system. Typically this will be zero. A number greater than zero is ok, but indicates a little bit of inefficiency.

millisTotal

Total time for the query to run, in milliseconds.

millisAvg

Average time for the query to run on each shard, in milliseconds.

numQueries

Total number of queries executed.

numShards

Total number of shards queried.

See Also

- Optimization
- Hint
- Database Profiler
- Viewing and Terminating Current Operation
Optimizing Object IDs

- Use the collections ‘natural primary key’ in the _id field.
- When possible, use _id values that are roughly in ascending order.
- Store Binary GUIDs as BinData, rather than as hex encoded strings.
- Extract insertion times from _id rather than having a separate timestamp field.
- Sort by _id to sort by insertion time.
- See Also

The _id field in a MongoDB document is very important and is always indexed for normal collections. This page lists some recommendations. Note that it is common to use the BSON ObjectID datatype for _id's, but the values of an _id field can be of any type.

Use the collections ‘natural primary key’ in the _id field.

_id's can be any type, so if your objects have a natural unique identifier, consider using that in _id to both save space and avoid an additional index.

When possible, use _id values that are roughly in ascending order.

If the _id's are in a somewhat well defined order, on inserts the entire b-tree for the _id index need not be loaded. BSON ObjectIds have this property.

Store Binary GUIDs as BinData, rather than as hex encoded strings.

BSON includes a binary data datatype for storing byte arrays. Using this will make the id values, and their respective keys in the _id index, twice as small.

Note that unlike the BSON Object ID type (see above), most UUIDs do not have a rough ascending order, which creates additional caching needs for their index.

Extract insertion times from _id rather than having a separate timestamp field.

The BSON ObjectIds format provides documents with a creation timestamp (one second granularity) for free. Almost all drivers implement methods for extracting these timestamps; see the relevant api docs for details. In the shell:

Sort by _id to sort by insertion time

BSON ObjectIds begin with a timestamp. Thus sorting by _id, when using the ObjectID type, results in sorting by time. Note: granularity of the timestamp portion of the ObjectID is to one second only.

See Also

- Object IDs
Optimizing Storage of Small Objects

MongoDB records have a certain amount of overhead per object (BSON document) in a collection. This overhead is normally insignificant, but if your objects are tiny (just a few bytes, maybe one or two fields) it would not be. Below are some suggestions on how to optimize storage efficiently in such situations.

Using the _id Field Explicitly

Mongo automatically adds an object ID to each document and sets it to a unique value. Additionally this field 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 and then executes the plan to return results. Thus, MongoDB supports ad hoc queries much like say, Oracle or MySQL.

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

First, the optimizer might consistently pick a bad query plan. For example, there might be correlations in the data of which the optimizer is unaware. In a situation like this, the developer might use a query hint.

Also with the traditional approach, query plans can change in production with negative results. No one thinks rolling out new code without testing is a good idea. Yet often in a production system a query plan can change as the statistics in the database change on the underlying data. The query plan in effect may be a plan that never was invoked in QA. If it is slower than it should be, the application could experience an outage.

The MongoDB query optimizer is different. It is not cost based -- it does not model the cost of various queries. Instead, the optimizer simply tries different query plans and learn which ones work well. Of course, when the system tries a really bad plan, it may take an extremely long time to run. To solve this, when testing new plans, MongoDB executes multiple query plans in parallel. As soon as one finishes, it terminates the other executions, and the system has learned which plan is good. This works particularly well given the system is non-relational, which makes the space of possible query plans much smaller (as there are no joins).

Sometimes a plan which was working well can work poorly -- for example if the data in the database has changed, or if the parameter values to the query are different. In this case, if the query seems to be taking longer than usual, the database will once again run the query in parallel to try different plans.

This approach adds a little overhead, but has the advantage of being much better at worst-case performance.

Testing of queries repeats after 1,000 operations and also after certain manipulations of a collection occur (such as adding an index).

Query plan selection is based on a "query pattern". For example the query
is treated as the same pattern as \{ x : <anothervalue> \}.

**See Also**

- MongoDB hint() and explain() operators

### Querying

One of MongoDB's best capabilities is its support for dynamic (ad hoc) queries. Systems that support dynamic queries don't require any special indexing to find data; users can find data using any criteria. For relational databases, dynamic queries are the norm. If you're moving to MongoDB from a relational databases, you'll find that many SQL queries translate easily to MongoDB's document-based query language.

**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:

```javascript
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:

```javascript
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:

```javascript
// retrieve ssn field for documents where last_name == 'Smith':
db.users.find({last_name: 'Smith'}, {'ssn': 1});
```

```javascript
// 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:

```javascript
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:

```javascript
db.users.find().skip(20).limit(10);
```

**slaveOk (Querying Secondaries)**

When querying a replica set, drivers route their requests to the primary mongod by default; to perform a query against an (arbitrarily-selected) secondary, the query can be run with the slaveOk option. See the slaveOk page for more information.

### 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:

```javascript
var cur = db.example.find();
cur.forEach(function(x) { print(tojson(x))});
```

---

**Quick Reference Card**

Download the Query and Update Modifier Quick Reference Card

**More Info**

This was just an introduction to querying in Mongo. More information:

- **Mongo Query Language**
- Querying and nulls
- Retrieving a Subset of Fields
- Advanced Queries
- Dot Notation (Reaching into Objects)
- Full Text Search in Mongo
- min and max Query Specifiers
- OR operations in query expressions
- Queries and Cursors
- Server-side Code Execution
- Sorting and Natural Order
- Aggregation

**See Also**

- Advanced Queries
- Indexes
- Queries and Cursors
- Query Optimizer

---

**Mongo Query Language**

Queries in MongoDB are expressed as JSON (BSON). Usually we think of query object as the equivalent of a SQL "WHERE" clause:
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:

```javascript
> db.users.find( { x : 3, y : "abc" } ).sort({x:1});
// select * from users where x=3 and y='abc'
order by x asc;
```

The possible specifies in the query object are:
- `$query` - the evaluation or "where" expression
- `$orderby` - sort order desired
- `$hint` - hint to query optimizer
- `$explain` - if true, return explain plan results instead of query results
- `$snapshot` - if true, "snapshot mode"

**Querying and nulls**

The `null` value in javascript carries different meanings. When a query involves `null`, this can have multiple interpretations.

Take the following examples:

```javascript
> db.foo.insert({ x : 1, y : 1 })
> db.foo.insert({ x : 2, y : "string" })
> db.foo.insert({ x : 3, y : null })
> db.foo.insert({ x : 4 })

// Query #1
> db.foo.find({ y : null })
{ "_id" : ObjectId("4dc1975312c677fc83b5629f"), "x" : 3, "y" : null }
{ "_id" : ObjectId("4dc1975a12c677fc83b562a0"), "x" : 4 }

// Query #2
> db.foo.find({ y : { $type : 10 } })
{ "_id" : ObjectId("4dc1975312c677fc83b5629f"), "x" : 3, "y" : null }

// Query #3
> db.foo.find({ y : { $exists : false } })
{ "_id" : ObjectId("4dc1975a12c677fc83b562a0"), "x" : 4 }
```

To summarize the three queries:
1. documents where `y` has the value `null` or where `y` does not exist
2. documents where `y` has the value `null`
3. documents where `y` does not exist

**Retrieving a Subset of Fields**

- **Field Negation**
  - The `_id` Field
- **Mixing Includes/Excludes**
- **Covered Indexes**
- **Dot Notation**
- **Retrieving a Subrange of Array Elements**
- **See Also**

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)).
Field Negation

We can say “all fields except x” – for example to remove specific fields that you know will be large:

```javascript
// get all posts about 'tennis' but without the comments field
db.posts.find( { tags : 'tennis' }, { comments : 0 } );
```

The `_id` Field

The `_id` field will be included by default.

If you do not want it, you must exclude it specifically. (Typically you will want to do that if using the covered index feature described below.)

```javascript
// get all posts about 'tennis' but without the _id field
db.posts.find( { tags : 'tennis' }, { _id : 0 } );
```

Mixing Includes/Excludes

You cannot mix them, with the exception of the `_id` field. Note also that the `$slice` operator does not conflict with exclusions on other fields.

Covered Indexes

v1.8+

MongoDB can return data from the index only when the query only involves keys which are present in the index. Not inspecting the actual documents can speed up responses considerably since the index is compact in size and usually fits in RAM, or is sequentially located on disk.

MongoDB will automatically use covered index when it can. But be sure that:

- you provide list of fields to return, so that it can determine that it can be covered by index
- you must explicitly exclude the `_id` field by using `{ _id: 0 }` (unless the index includes that)
- as soon as you insert one array value for one of the index keys, the index will immediately become a multikey index and this disables covered index functionality
- use `Explain` to determine if the covered index is used: the `indexOnly` field should be true

```javascript
// do a login with a covered index, returning the users roles/groups
> db.users.ensureIndex( { username : 1, password : 1, roles : 1 } );
> db.users.save({username: "joe", password: "pass", roles: 2})
> db.users.save({username: "liz", password: "pass2", roles: 4})
> db.users.find({username: "joe"}, { _id: 0, roles: 1})
{ "roles" : 2 }
> db.users.find({username: "joe"}, { _id: 0, roles: 1}).explain()
{ "cursor" : "BtreeCursor username_1_password_1_roles_1",
...
"indexOnly" : true,
...
}
```

Dot Notation

You can retrieve partial sub-objects via Dot Notation.

```javascript
> 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.

```javascript
In MongoDB, just like in an RDBMS, creating appropriate indexes for queries is quite important for performance. See the Indexes page for more info.
Queries in MongoDB are represented as JSON-style objects, very much like the documents we actually store in the database. For example:

```javascript
// i.e., select * from things where x=3 and y="foo"
db.thing.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:

```javascript
db.thing.find({j: {$ne: 3}, k: {$gt: 10} });
```

Unless otherwise noted, the operations below can be used on array elements in the same way that they can be used on "normal" fields. For example, suppose we have some documents such as:

```javascript
> db.thing.insert({colors : ["blue", "black"]})
> db.thing.insert({colors : ["yellow", "orange", "red"]})
```

Then we can find documents that aren’t "red" using:

```javascript
> db.thing.find({colors : {$ne : "red"}})
{"_id": ObjectId("4dc9acea045bbf04348f9691"), "colors": ["blue","black"]}
```

Retrieving a Subset of Fields

See Retrieving a Subset of Fields

Conditional Operators

<, <=, >, >=

Use these special forms for greater than and less than comparisons in queries, since they have to be represented in the query document:

```javascript
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:

```javascript
db.thing.find({j : {$lt: 3}});
db.thing.find({j : {$gte: 4}});
```

You can also combine these operators to specify ranges:

```javascript
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

```javascript
( a: [ 1, 2, 3 ] )
```

would be matched by

```javascript
db.thing.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.

```javascript
db.things.find( { a: { $exists: true } } ); // return object if a is present
db.things.find( { a: { $exists: false } } ); // return if a is missing
```

Before v2.0, $exists is not able to use an index. Indexes on other fields are still used. $exists is not very efficient even with an index, and esp. with {$exists:true} since it will effectively have to scan all indexed values.

$mod

The $mod operator allows you to do fast modulo queries to replace a common case for where clauses. For example, the following $where query:

```javascript
db.things.find("this.a % 10 == 1")
```

can be replaced by:

```javascript
db.things.find( { a : { $mod : [ 10 , 1 ] } } )
```

$ne

Use $ne for “not equals”.

```javascript
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.

```javascript
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:

```javascript
db.things.find({j:{$in: [2,4,6]}});
```

Suppose the collection updates is a list of social network style news items; we want to see the 10 most recent updates from our friends. We could invoke:

```javascript
db.updates.ensureIndex( { ts : 1 } ); // ts == timestamp
var myFriends = myUserObject.friends; // let's assume this gives us an array of DBRef's of my friends
var latestUpdatesForMe = db.updates.find( { user : { $in : myFriends } } ).sort( { ts : -1 } ).limit(10);
```

The target field's value can also be an array; if so then the document matches if any of the elements of the array's value matches any of the $in field's values (see the Multikeys page for more information.

$nin

The $nin operator is similar to $in except that it selects objects for which the specified field does not have any value in the specified array. For example
db.things.find({j:{$nin: [2,4,6]}});

would match \{j:1,b:2\} but not \{j:2,c:9\}.

**$nor**

The $nor operator lets you use a boolean or expression to do queries. You give $nor a list of expressions, none of which can satisfy the query.

**$or**

v1.6+

The $or operator lets you use boolean or in a query. You give $or an array of expressions, any of which can satisfy the query.

Simple:

```javascript
db.foo.find( { $or : [ { a : 1 } , { b : 2 } ] } )
```

With another field

```javascript
db.foo.find( { name : "bob" , $or : [ { a : 1 } , { b : 2 } ] } )
```

The $or operator retrieves matches for each or clause individually and eliminates duplicates when returning results.

⚠️ $or can be nested as of v2.0, however nested $or clauses are not handled as efficiently by the query optimizer as top level $or clauses.

**$and**

v2.0+

The $and operator lets you use boolean and in a query. You give $and an array of expressions, all of which must match to satisfy the query.

```javascript
db.foo.insert( { a: [ 1, 10 ] } )
db.foo.find( { $and: [ { a: 1 }, { a: { $gt: 5 } } ] } )
```

In the above example documents with an element of \(a\) having a value of \(a\) equal to 1 and a value of \(a\) greater than 5 will be returned. Thus the inserted document will be returned given the multikey semantics of MongoDB.

**$size**

The $size operator matches any array with the specified number of elements. The following example would match the object \{a:["foo"]\}, since that array has just one element:

```javascript
db.things.find( { a : { $size: 1 } } );
```

You cannot use $size to find a range of sizes (for example: arrays with more than 1 element). If you need to query for a range, create an extra size field that you increment when you add elements. Indexes cannot be used for the $size portion of a query, although if other query expressions are included indexes may be used to search for matches on that portion of the query expression.

**$type**

The $type operator matches values based on their BSON type.

```javascript
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:
<table>
<thead>
<tr>
<th>Type Name</th>
<th>Type Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>1</td>
</tr>
<tr>
<td>String</td>
<td>2</td>
</tr>
<tr>
<td>Object</td>
<td>3</td>
</tr>
<tr>
<td>Array</td>
<td>4</td>
</tr>
<tr>
<td>Binary data</td>
<td>5</td>
</tr>
<tr>
<td>Object id</td>
<td>7</td>
</tr>
<tr>
<td>Boolean</td>
<td>8</td>
</tr>
<tr>
<td>Date</td>
<td>9</td>
</tr>
<tr>
<td>Null</td>
<td>10</td>
</tr>
<tr>
<td>Regular expression</td>
<td>11</td>
</tr>
<tr>
<td>JavaScript code</td>
<td>13</td>
</tr>
<tr>
<td>Symbol</td>
<td>14</td>
</tr>
<tr>
<td>JavaScript code with scope</td>
<td>15</td>
</tr>
<tr>
<td>32-bit integer</td>
<td>16</td>
</tr>
<tr>
<td>Timestamp</td>
<td>17</td>
</tr>
<tr>
<td>64-bit integer</td>
<td>18</td>
</tr>
<tr>
<td>Min key</td>
<td>255*</td>
</tr>
<tr>
<td>Max key</td>
<td>127</td>
</tr>
</tbody>
</table>

Although the Min key is 255 per the BSON spec, for internal reasons, mainly to do with how it is required to evaluate, the Min key is actually of $type -1. Hence to find the minimum value for a shard key you would do something like this:

```javascript
db.chunks.find( { "min.shardKey" : { $type : -1 } } ); // matches for the minimum shard key value
```

For more information on types and BSON in general, see [http://www.bsonspec.org](http://www.bsonspec.org).

**Regular Expressions**

You may use regexes in database query expressions:

```javascript
db.customers.find( { name : /acme.*corp/i } );
db.customers.find( { name : { $regex : 'acme.*corp', $options : 'i' } } );
```

If you wish to specify both a regex and another operator for the same field, you need to use the $regex clause. For example, to find the customers where name matches the above regex but does not include 'acmeblahcorp', you would do the following:

```javascript
db.customers.find( { name : { $regex : /acme.*corp/i, $nin : ['acmeblahcorp'] } } );
```

Note that $regex queries are escaped slightly differently than JavaScript native regular expressions. For example:

```javascript
> db.x.insert({someId : "123[456]*"})
> db.x.find({someId : /123\[456\]/i}) // use "\" to escape
> db.x.find({someId : {$regex : "123\[456\]\*"}}) // use "\" to escape
```

An index on the field queried by regexp can increase performance significantly, as follows:
Simple prefix queries (also called rooted regexps) like `/^prefix/` will make efficient use of the index (much like most SQL databases that use indexes for a LIKE `prefix%` expression). This only works if the expression is left-rooted and the `i` (case-insensitivity) flag is not used.

All other queries will not make an efficient use of the index: all values in the index will be scanned and tested against the regular expression.

While `/^a/, /^a.*/, and /^a.*$/` are equivalent, they will have different performance characteristics. The latter two will be slower as they have to scan the whole string. The first format can stop scanning after the prefix is matched.

MongoDB uses PCRE for regular expressions. Valid flags are:

- `i` - Case insensitive. Letters in the pattern match both upper and lower case letters.
- `m` - Multiline. By default, Mongo treats the subject string as consisting of a single line of characters (even if it actually contains newlines). The "start of line" metacharacter (^) matches only at the start of the string, while the "end of line" metacharacter ($) matches only at the end of the string, or before a terminating newline. When `m` is set, the "start of line" and "end of line" constructs match immediately following or immediately before internal newlines in the subject string, respectively, as well as at the very start and end. If there are no newlines in a subject string, or no occurrences of `^` or `$` in a pattern, setting `m` has no effect.
- `x` - Extended. If set, whitespace data characters in the pattern are totally ignored except when escaped or inside a character class. Whitespace does not include the VT character (code 11). In addition, characters between an unescaped # outside a character class and the next newline, inclusive, are also ignored. This option makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence `(? which introduces a conditional subpattern.
- `s` - Dot all. v2.0+. Allows the dot (.) to match all characters including new lines. By default, `/a.*b/` will not match the string "apple\nbanana", but `/a.*b/s` will.

Note that javascript regex objects only support the `i` and `m` options (and `g` which is ignored by mongod). Therefore if you want to use other options, you will need to use the `$regex, $options query syntax.

Value in an Array

To look for the value "red" in an array field `colors`:

```javascript
db.things.find( { colors : "red" } );
```

That is, when "colors" is inspected, if it is an array, each value in the array is checked. This technique may be mixed with the embedded object technique below.

$elemMatch

v1.4+

Use $elemMatch to check if an element in an array matches the specified match expression.

```javascript
> t.find( { x : { $elemMatch : { a : 1, b : { $gt : 1 } } } } )
{ "id" : ObjectId("4b5783300334000000000aa9"),
  "x" : [{ "a" : 1, "b" : 3 }, 7, { "b" : 99 }, { "a" : 11 }] }
```

Note that a single array element must match all the criteria specified; thus, the following query is semantically different in that each criteria can match a different element in the x array:

```javascript
> t.find( { "x.a" : 1, "x.b" : { $gt : 1 } } )
```

See the dot notation page for more.

You only need to use this when more than one field must be matched in the array element.

A frequently asked question is how to query for a range.
As a range query, the above correctly matches single values, but probably doesn't do what you expect for arrays – this is why `$elemMatch` is needed for range queries on an array.

```javascript
> t.find({ x: { $gt: 2, $lt: 5 } })
{ "_id": ObjectId("4ffb67009a563373b772e33c"), "x": 3 }
{ "_id": ObjectId("4ffb67009a563373b772e33d"), "x": [ 4 ] }
{ "_id": ObjectId("4ffb67009a563373b772e33e"), "x": [ 1, 6 ] }
```

Value in an Embedded Object

For example, to look in a postings collection with embedded author objects:

```javascript
db.postings.find( { "author.name": "joe" } );
```

See the dot notation page for more.

**Meta operator: $not**

The `$not` meta operator can be used to negate the check performed by a standard operator. For example:

```javascript
db.customers.find( { name: { $not: /acme.*corp/i } } );

db.things.find( { a: { $not: { $mod: [ 10, 1 ] } } } );
```

The `$not` meta operator can only affect other operators. The following do not work. For such a syntax use the `$ne` operator.

```javascript
db.things.find( { a: { $not: true } } ); // syntax error
```

$not is not supported for regular expressions specified using the `{regex: ...}` syntax. When using $not, all regular expressions should be passed using the native BSON type (e.g. `{"$not": re.compile("acme.*corp")}` in PyMongo).

**Javascript Expressions and $where**

In addition to the structured query syntax shown so far, you may specify query expressions as Javascript. To do so, pass a string containing a Javascript expression to `find()`, or assign such a string to the query object member `$where`. The database will evaluate this expression for each object scanned. When the result is true, the object is returned in the query results.

For example, the following mongo shell statements all do the same thing:

```javascript
> db.myCollection.find( { a: { $gt: 3 } } );
> db.myCollection.find( { $where: "this.a > 3" } );
> db.myCollection.find("this.a > 3");
> f = function() { return this.a > 3; } db.myCollection.find(f);
```

You may mix mongo query expressions and a `$where` clause. In that case you must use the `$where` syntax, e.g.:

```javascript
> db.myCollection.find({registered:true, $where:"this.a>3"})
```

Javascript executes more slowly than the native operators listed on this page, but is very flexible. See the server-side processing page for more information.

**Cursor Methods**
**count()**

The `count()` method returns the number of objects matching the query specified. It is specially optimized to perform the count in the MongoDB server, rather than on the client side for speed and efficiency:

```javascript
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:

```javascript
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.

```javascript
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.

```javascript
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:

```javascript
function printStudents(pageNumber, nPerPage) {
    print("Page: " + pageNumber);
    db.students.find().skip((pageNumber-1)*nPerPage).limit(nPerPage).forEach( function(student) {
        print(student.name + ":<p>");
    });
}
```

**Paging Costs**

Unfortunately `skip` can be (very) costly and requires the server to walk from the beginning of the collection, or index, to get to the offset/skip position before it can start returning the page of data (limit). As the page number increases `skip` will become slower and more cpu intensive, and possibly IO bound, with larger collections.

Range based paging provides better use of indexes but does not allow you to easily jump to a specific page.

**snapshot()**

Indicates use of snapshot mode for the query. Snapshot mode assures no duplicates are returned, or objects missed, which were present at both the start and end of the query’s execution (even if the object were updated). If an object is new during the query, or deleted during the query, it may or may not be returned, even with snapshot mode.

Note that short query responses (less than 1MB) are effectively snappedotted.

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

For more information, see [How to do Snapshotted Queries in the Mongo Database](#).

**sort()**
sort() is analogous to the ORDER BY statement in SQL - it requests that items be returned in a particular order. We pass `sort()` a key pattern which indicates the desired order for the result.

```
db.myCollection.find().sort( { ts : -1 } ); // sort by ts, descending order
```

`sort()` may be combined with the `limit()` function. In fact, if you do not have a relevant index for the specified key pattern, `limit()` is recommended as there is a limit on the size of sorted results when an index is not used. Without a `limit()`, or an index, a full in-memory sort must be done. However, using a `limit()` reduces the required memory footprint and increases the speed of the operation by using an optimized sorting algorithm.

**batchSize()**

`batchSize()` determines the number of documents MongoDB returns in each batch to the client. MongoDB considers `batchSize()`, `limit()`, and the size in bytes of each document when deciding how many documents to send in each batch.

```
db.myCollection.find().batchSize(10);
```

The shell and most drivers present results to your application code as if they came in a single batch, so you normally do not need to think about `batchSize`.

**Meta query operators**

**$returnKey**

Only return the index key:

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

**$maxScan**

Limit the number of items to scan:

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

**$orderby**

Sort results:

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

**$explain**

Explain the query instead of actually returning the results:

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

**$snapshot**

Snapshot query:

```
db.foo.find().__addSpecial("$snapshot", true )
// same as
db.foo.find().snapshot()
```
**$min and $max**

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

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

**$showDiskLoc**

Show disk location of results:

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

**$hint**

Force query to use the given index:

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

**$comment**

You can put a $comment field on a query to make looking in the profiler logs simpler.

```javascript
db.foo.find()._addSpecial("$comment", "some comment to help find a query")
```

**group()**

The group() method is analogous to GROUP BY in SQL. group() is more flexible, actually, allowing the specification of arbitrary reduction operations. See the Aggregation section of the Mongo Developers' Guide for more information.

**See Also**

- Indexes
- Optimizing Queries (including explain() and hint())

**Dot Notation (Reaching into Objects)**

- Dot Notation vs. Subobjects
- Array Element by Position
- Matching with $elemMatch
- See Also

MongoDB is designed for storing JSON-style objects. The database understands the structure of these objects and can reach into them to evaluate query expressions.

Let's suppose we have some objects of the form:

```javascript
> 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:

```javascript
> 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: {name : "Jane", id : 1}},
comments : [{ by : "Abe", text : "First" },
{ by : "Ada", text : "Good post" }]
```

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" })
```
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:

```javascript
// 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:

```javascript
db.foo.find({
  "foo" : {
    "$elemMatch" : {
      "shape" : "square",
      "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

```javascript
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:

```javascript
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

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:

```javascript
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

```javascript
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:

```javascript
> print(db.articles.findOne( { tags: } ).name);
Apollo
```

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:

```javascript
{ 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 website uses MongoDB for its blog search function in production.

Mark Watson's opinions on Java, Ruby, Lisp, AI, and the Semantic Web - A recipe example in Ruby.

Full text search with MongoDB at Flowdock

**min and max Query Specifiers**

The `min()` and `max()` functions may be used in conjunction with an index to constrain query matches to those having index keys between the `min()` and `max()` keys specified. The `min()` and `max()` functions may be used individually or in conjunction. The index to be used may be specified with a `hint()` or one may be inferred from pattern of the keys passed to `min()` and/or `max()`.

```javascript
db.f.find().min({name: "barry"}).max({name: "larry"}).hint({name: 1});
```

If you're using the standard query syntax, you must distinguish between the `$min` and `$max` keys and the query selector itself. See here:

```javascript
db.f.find({$min: {name: "barry"}, $max: {name: "larry"}, $query: {}});
```

The `min()` value is included in the range and the `max()` value is excluded.

Normally, it is much preferred to use `$gte` and `$lt` rather than to use `min` and `max`, as `min` and `max` require a corresponding index. `Min` and `max` are primarily useful for compound keys: it is difficult to express the `last_name/first_name` example above without this feature (it can be done using `$where`).

`min` and `max` exist primarily to support the mongos (sharding) process.

**OR operations in query expressions**

Query objects in Mongo by default AND expressions together.

**$in**

The `$in` operator indicates a "where value in ..." expression. For expressions of the form `x == a OR x == b`, this can be represented as

```javascript
( x : { $in : [ a, b ] } )
```

**$or**

v1.5.3+

The `$or` operator lets you use a boolean or expression to do queries. You give `$or` a list of expressions, any of which can satisfy the query.

Simple:

```javascript
db.foo.find( { $or : [ { a : 1 } , { b : 2 } ] } )
```

With another field

```javascript
db.foo.find( { name : "bob" , $or : [ { a : 1 } , { b : 2 } ] } )
```

The `$or` operator retrieves matches for each or clause individually and eliminates duplicates when returning results.

**$where**
We can provide arbitrary Javascript 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:

```javascript
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:

```javascript
db.mycollection.find( function() { return this.a == 3 || this.b == 4; } );
```

See Also

- Advanced Queries

Queries and Cursors

Queries to MongoDB return a cursor, which can be iterated to retrieve results. The exact way to query will vary with language driver. Details below focus on queries from the MongoDB shell (i.e. the mongo process).

The shell `find()` method returns a cursor object which we can then iterate to retrieve specific documents from the result. We use `hasNext()` and `next()` methods for this purpose.

```javascript
for( var c = db.parts.find(); c.hasNext(); ) {
  print( c.next());
}
```

Additionally in the shell, `forEach()` may be used with a cursor:

```javascript
db.users.find().forEach( function(u) { print("user: " + u.name); } );
```

Topics:

- Array Mode in the Shell
- Getting a Single Item
- Querying Embedded Objects
- Greater Than / Less Than
- Latent Cursors and Snapshotting
- Execution of queries in batches
- Performance implications
- Auditing allocated cursors
- Closing and Timeouts
- See Also

**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:

```javascript
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:

```javascript
db.order.find( { "shipping.carrier" : "usps" } );
```

See the [dot notation] docs for more information.

### Greater Than / Less Than

```javascript
db.myCollection.find( { a : { $gt : 3 } } );
db.myCollection.find( { a : { $gte :3 } } );
db.myCollection.find( { a : { $lt :3 } } );
db.myCollection.find( { a : { $lte :3 } } ); // a <= 3
```

### Latent Cursors and Snapshotting

A latent cursor has (in addition to an initial access) a latent access that occurs after an intervening write operation on the database collection (i.e., an insert, update, or delete). Under most circumstances, the database supports these operations.

Conceptually, a cursor has a current position. If you delete the item at the current position, the cursor automatically skips its current position forward to the next item.

MongoDB cursors do not provide a snapshot: if other write operations occur during the life of your cursor, it is unspecified if your application will see the results of those operations or not. See the [snapshot] docs for more information.

### Execution of queries in batches

The MongoDB server returns query results to the client in batches. You can modify this behavior in two ways: You can specify `batchSize()`, which tells the server how many documents to return in each batch, or `limit()`, which determines the total number of documents to return for this query. (See Advanced Queries for setting limit and `batchSize`.)

If `limit` and `batchSize` are not specified, the first batch contains 101 documents, or enough documents to exceed 1 MB, whichever comes first. Otherwise, the server returns enough documents to satisfy the lesser of the `batchSize` or the `limit`. If the query matches more than that quantity of results and you would like them all to be returned, you need to either specify a larger limit or `batchSize`, or iterate through the result set to retrieve all results. Iterating a cursor will return enough documents in each batch to satisfy `batchSize` or to exceed 4 MB, whichever comes first.

A special case is if you sort a set of documents without an index. In that case, since MongoDB must load all the documents in order to sort them in memory, so it returns them all in the first batch.

Regardless of `limit` and `batchSize`, no batch will contain more than enough documents to exceed 4 MB.

Examples:
Performance implications

If, for example, you do

```javascript
cursor = db.foo.find( { x : 1 } )
for ( i=0; i<100; i++ ) {
    printjson( cursor.next() );
}
```

The server will only find the first the first 100 results. If the result set is large, finding the first 100 results may be much faster than finding all results and printing the first 100.

Note that counts are performed against the entire result set, so for example

```javascript
db.foo.find( { x : 1 } ).count()
```

Could be much slower than finding the first 100 results above.

Auditing allocated cursors

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

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

Closing and Timeouts

By default a cursor will timeout after 10 minutes of inactivity. The server will close the cursor if it isn't accessed in that time or it has been exhausted.

You can specify `NoTimeout` optionally for a query. If you do this please be careful to close the cursor manually – otherwise they will consume memory on the server.

See Also

- Advanced Queries
- Multikeys in the HowTo
Tailable Cursors

A tailable cursor "tails" the end of a capped collection, much like the Unix "tail -f" command. They key idea is that if we "catch up" and have reached the end of the collection, our position is remembered rather than the cursor being closed. Thus, after new objects are inserted, we can resume retrieving from where we left off -- which is then very inexpensive. If you reach the end of the collection and request more data, the cursor will block waiting for new documents to be inserted.

If the field you wish to "tail" is indexed, do not use tailable cursors; instead simply (re)query for \{ field: { $gt: value } \} where value is where you last left off. This is normally quite efficient. Tailable cursors are for cases where having an index would be prohibitive (extremely high write collections). If performance is not problematic, use a normal query and cursor, tailable adds some complexity.

As no index will be used for the query, the initial scanning to find the first object to return will likely be quite costly. However once found, retrieving additional data from new inserts then becomes very inexpensive.

The cursor may become invalid if, for example, the last object returned is at the end of the collection and is deleted. Thus, you should be prepared to requery if the cursor is "dead". You can determine if a cursor is dead by checking its id. An id of zero indicates a dead cursor (use isDead in the c++ driver). In addition, the cursor will be in "dead" state after a query which returns no matches.

MongoDB replication uses tailable cursors to follow the end of the primary's replication op log collection. Writes to the oplog would be slower with an index. The tailable feature eliminates the need to create an index for replication's use case.

C++ example:

```cpp
#include "client/dbclient.h"

using namespace mongo;

/* "tail" the namespace, outputting elements as they are added. Cursor blocks
 * waiting for data if no documents currently exist. For this to work something
 * field -- _id in this case -- should be increasing when items are added.
 */
void tail(DBClientBase& conn, const char *ns) {
    // minKey is smaller than any other possible value
    BSONElement lastId = minKey.firstElement();

    Query query = Query().sort("$natural"); // { $natural : 1 } means in forward
    // capped collection insertion order
    while(1) {
        auto_ptr<DBClientCursor> c = conn.query(ns, query, 0, 0, 0,
                                               QueryOption_CursorTailable | QueryOption_AwaitData);
        while(1) {
            if( !c->more() ) {
                if( c->isDead() ) {
                    // we need to requery
                    break;
                }
            // No need to wait here, cursor will block for several sec with _AwaitData
                continue; // we will try more() again
            }
            BSONObj o = c->next();
            lastId = o["_id"];
            cout << o.toString() << endl;
        }
    // prepare to requery from where we left off
    query = QUERY( "_id" << GT << lastId ).sort("$natural");
}
}
```

Javascript example:
var coll = db.some.capped.collection;
var lastVal = coll.find().sort({ '$natural' : 1 })
    .limit(1).next()['increasing'];

while(!)
    cursor = coll.find({ 'increasing' : { '$gte' : lastVal } });

    // tailable
    cursor.addOption(2);
    // await data
    cursor.addOption(32);

    // Waits several sec for more data
    while( cursor.hasNext() ){
        var doc = cursor.next();
        lastVal = doc['increasing'];
        printjson(doc);
    }

See Also

- A detailed blog post on tailable cursors
  http://github.com/mongodb/mongo-snippets/blob/master/cpp-examples/tailable_cursor.cpp
- http://api.mongodb.org/cplusplus/2.0.3/dbclient_8h_source.html

Server-side Code Execution

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

Mongo supports the execution of code inside the database process.

Map/Reduce

MongoDB supports Javascript-based map/reduce operations on the server. See the map/reduce documentation for more information.

Using db.eval()

⚠️ Locking
By default db.eval() requires a write lock so it will block all read/write operations while it runs.

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 `db.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 `db.eval()` to do equivalent of the Mongo `count()` function:**

```
function mycount(collection) {
    return db.eval( function(){
        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(name, howMuch){
        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;
    }, name, //first argument to function above
    howMuch //second argument to function above
    );
}
```

```
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:

```javascript
db.eval("rs.add('ip-address:27017')");
```

As we just mentioned, eval will take a write lock on the current node. Therefore, this won't work because you can't add a new replica set member if any of the existing nodes is write-locked.

The proper approach is to run the commands to add a node manually. rs.add simply queries the local.system.replSet collection, updates the config object, and run the replSetReconfig command. You can do this from the driver, which, in addition to not taking out the eval write lock, manages to more directly perform the operation.

In 1.7.2, a noLock option was added to eval. To use noLock you have to use the command interface directly:

```javascript
db.runCommand({$eval: function() {return 42;}, noLock: true})
```

or with args

```javascript
db.runCommand({$eval: function(x,y) {return x*y;}, args: [6,7], noLock: true})
```

Sharding

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

Storing functions server-side

Note: we recommend not using server-side stored functions when possible. As these are code it is likely best to store them with the rest of your code in a version control system.

There is a special system collection called system.js that can store JavaScript functions to be reused. To store a function, you would do:

```javascript
db.system.js.save( { _id : "foo", value : function( x , y ){ return x + y; } } );
```

_id is the name of the function, and is unique per database.

Once you do that, you can use foo from any JavaScript context (db.eval, $where, map/reduce)

Here is an example from the shell:

```shell
> db.system.js.save({ "_id" : "echo", "value" : function(x){return x;} })
> db.eval("echo('test')")
test
```

See [http://github.com/mongodb/mongo/tree/master/jstests/storefunc.js](http://github.com/mongodb/mongo/tree/master/jstests/storefunc.js) for a full example.

In MongoDB 2.1 you will also be able to load all the scripts saved in db.system.js into the shell using db.loadServerScripts()

```shell
> db.loadServerScripts()
> echo(3)
3
```

$where Clauses and Functions in Queries

In addition to the regular document-style query specification for find() operations, you can also express the query either as a string containing a
SQL-style WHERE predicate clause, or a full JavaScript function. Note: if a normal data-driven BSON query expression is possible, use that construction. Use $where only when you must it is significantly slower.

When using this mode of query, the database will call your function, or evaluate your predicate clause, for each object in the collection.

In the case of the string, you must represent the object as "this" (see example below). In the case of a full JavaScript function, you use the normal JavaScript function syntax.

The following four statements in mongo - The Interactive Shell are equivalent:

```js
    db.myCollection.find( { a : { $gt: 3 } } );
    db.myCollection.find( { $where: "this.a > 3" });
    db.myCollection.find( { $where: function() { return this.a > 3; } });
```

The first statement is the preferred form. It will be faster to execute because the query optimizer can easily interpret that query and choose an index to use.

You may mix data-style find conditions and a function. This can be advantageous for performance because the data-style expression will be evaluated first, and if not matched, no further evaluation is required. Additionally, the database can then consider using an index for that condition's field. To mix forms, pass your evaluation function as the $where field of the query object. For example, both of the following would work:

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

Do not write to the database from a $where expression.

Notes on Concurrency

If you don't use the "nolock" option, db.eval() requires a write lock which will block all reads and writes until it is done. Thus, its operations are atomic but prevent other operations from processing. Even with the nolock option javascript will still be limited to a single executing block at a time – multiple javascript threads cannot run concurrently.

When more concurrency is needed consider using an external client/program.

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:

```js
> 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:

```js
> c=db.collection.find().sort({name : 1, age : -1})
```

See Also
MongoDB includes utility functions which provide server-side `count`, `distinct`, and `group by` operations. More advanced aggregate functions can be crafted using the Aggregation Framework or MapReduce.

### Aggregation Framework

The Aggregation Framework provides a means to calculate aggregated values without having to use MapReduce. While MapReduce is powerful, it is often more difficult than necessary for many simple aggregation tasks such as totaling or averaging field values.

#### 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.

```javascript
db.collection.count(selector);
```

For example:

```javascript
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

```javascript
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:

```javascript
{ distinct : <collection_name>, key : <key>[, query : <query>] }
```

although many drivers have a helper function for distinct.
distinct may also reference a nested key:

```javascript
> 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 }
```

You can add an optional query parameter to distinct as well

```javascript
> 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 (i.e. greater than the maximum document size of 4 or 16MB supported by your MongoDB version), use MapReduce 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

Use of Group in Sharded Environments
Currently one must use map/reduce instead of group() in sharded MongoDB configurations. It is also not to be confused with the `$group` operator in the Aggregation Framework which is supported in sharded environments.

group returns an array of grouped items. The command is similar to SQL's group by. The SQL statement

```sql
select a,b,sum(c) csum from coll where active=1 group by a,b
```

corresponds to the following in MongoDB:

```javascript
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 MapReduce.

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 \texttt{obj} and \texttt{prev}.

- \texttt{initial:} initial value of the aggregation counter object.
- \texttt{key:} An optional \textbf{function} returning a "key object" to be used as the grouping key. Use this instead of \texttt{key} to specify a key that is not a single/multiple existing fields. Could be used to group by day of the week, for example. Set in lieu of \texttt{key}.
- \texttt{cond:} An optional condition that must be true for a row to be considered. This is essentially a \texttt{find()} query expression object. If null, the reduce function will run against all rows in the collection.
- \texttt{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 \texttt{jstests/group3.js} for examples.

To order the grouped data, simply sort it client-side upon return. The following example is an implementation of \texttt{count()} using \texttt{group()}.

```javascript
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;
}
```

\textbf{Examples}

The examples assume data like this:

```javascript
{
    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 \texttt{http_action} in November 2009:

```javascript
db.test.group({
    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 }
});
```

Show me stats for each domain for each day in November 2009:

```javascript
db.test.group({
    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 }
});
```
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:

```javascript
> db.test.group(
  { cond: { "$gte": "2009-11", "$lt": "2009-12"} },
  { key: { domain: true, invoked_at: 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
  } }
);
```

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:

```javascript
> db.foo.find()
{ "_id" : ObjectId("4a92af2db3d09cb83d985f6f"), "x" : 1}
{ "_id" : ObjectId("4a92af2fb3d09cb83d985f70"), "x" : 3}
{ "_id" : ObjectId("4a92afdb3d09cb83d985f71"), "x" : 3}
```

```javascript
> db.$cmd.findOne({group : {
  ... ns : "foo",
  ... cond : {},
  ... key : {x : 1},
  ... initial : {count : 0},
  ... $reduce : function(obj,prev){prev.count++;}}})
{ "retval" : [1,3], "count" : 3, "avg_time" : 0.05 }
```

If you use the database command with keyf (instead of key) it must be prefixed with a $. For example:

```javascript
db.$cmd.findOne({group : {
  ... ns : "foo",
  ... $keyf : function(doc) {
    return {"x" : doc.x}; },
  ... initial : {count : 0},
  ... $reduce : function(obj,prev) { prev.count++; }}})
```

MapReduce

MongoDB provides a MapReduce facility for more advanced aggregation needs. CouchDB users: please note that basic queries in MongoDB do not use MapReduce.

See Also

- Aggregation Framework Examples
- 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:

```javascript
db.things.remove({});  // removes all
db.things.remove({n:1});  // removes all where n == 1
```

The most efficient method to delete a specific document is to use item's document _id value as a criteria:

```javascript
> // myobject is some document that is in our db.things collection
> db.things.remove({ _id: myobject._id });
```

Note that passing an entire document to remove() would work, but is inefficient as in that case we are basically specifying every single field of the document as part of the query expression.

```javascript
> db.things.remove(myobject);  // don't do this
```

Isolation

If a simultaneous update (on the same collection) grows an object which matched the remove criteria, the updated object may not be removed (as the operations are happening at approximately the same time, this may not even be surprising). In situations where this is undesirable, pass {$atomic : true} in your filter expression:

```javascript
db.videos.remove( { rating : { $lt : 3.0 }, $atomic : true } )
```

The remove operation is then isolated – however, it will also block other operations while executing. The collection must be unsharded to use this option.

**Updating**

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

- `update()` replaces the document matching criteria entirely with objNew. If you only want to modify some fields, you should use the $ modifiers below.

Here's the MongoDB shell syntax for `update()`:
db.collection.update( criteria, objNew, upsert, multi )

Arguments:

- criteria - query which selects the record to update;
- objNew - updated object or $ operators (e.g., $inc) which manipulate the object
- upsert - if this should be an "upsert" operation; that is, if the record(s) do not exist, insert one. **Upsert only inserts a single document.**
- multi - indicates if all documents matching criteria should be updated rather than just one. Can be useful with the $ operators below.

If you are coming from SQL be aware that by default, update() only modifies the first matched object. If you want to modify all matched objects, you need to use the multi flag.

**Upserts**

To perform an upsert, pass true as the third argument to update.

Upsert means "update the document if present; insert (a single document) if missing". MongoDB determines that the document is missing solely via examining the criteria document.

It is important to remember that upsert can only insert a single document.

**save() in the mongo shell**

The save() helper method in the mongo shell provides a shorthand syntax to perform an update of a single document with upsert semantics:

```shell
> // x is some JSON style object
> db.mycollection.save(x); // updates if exists; inserts if new
> // equivalent to:
> db.mycollection.update({ _id: x._id }, x, /*upsert*/ true);
```

**Modifier Operations**

Modifier operations are highly-efficient and useful when updating existing values; for instance, they're great for incrementing a number.

So, while a conventional implementation does work:

```javascript
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 as well as resulting in very little network data transfer. The modifier update is also also **atomic** (per document that is; on a multi-document update, there is effectively an auto-commit after each individual document update).

You specify any of the special update operators (which always start with a '$' character) with a relevant update document:

```javascript
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:

```json
{ $set : { x : 1 , y : 2 } }
```

Also, multiple operators are valid too:

```json
{ $set : { x : 1 }, $inc : { y : 1 } }
```

$inc

```json
( $inc : { field : value } )
```

increments `field` by the number `value` if `field` is present in the object, otherwise sets `field` to the number `value`. This can also be used to decrement by using a negative `value`.

$set

```json
( $set : { field : value } )
```

sets `field` to `value`. All datatypes are supported with `$set`.

$unset

```json
( $unset : { field : 1 })
```

Deletes a given field. v1.3+

$push

```json
( $push : { field : value } )
```

appends `value` to `field`, if `field` is an existing array, otherwise sets `field` to the array `[value]` if `field` is not present. If `field` is present but is not an array, an error condition is raised.

Multiple arrays may be updated in one operation by comma separating the `field`: `value` pairs:

```json
( $push : { field : value, field2 : value2 } )
```

$pushAll

```json
( $pushAll : { field : value_array } )
```

appends each value in `value_array` to `field`, if `field` is an existing array, otherwise sets `field` to the array `value_array` if `field` is not present. If `field` is present but is not an array, an error condition is raised.

$addToSet and $each

```json
( $addToSet : { field : value } )
```

Adds value to the array only if it's not in the array already, if `field` is an existing array, otherwise sets `field` to the array `value` if `field` is not present. If `field` is present but is not an array, an error condition is raised.
To add a list of several values to the set use the `$each` qualifier:

```
{ $addToSet : { a : { $each : [ 3 , 5 , 6 ] } } }
```

$pop

```
{ $pop : { field : 1 } }
```

removes the last element in an array (ADDED in 1.1)

```
{ $pop : { field : -1 } }
```

removes the first element in an array (ADDED in 1.1)

$pull

```
{ $pull : { field : _value } }
```

removes all occurrences of `value` from `field`, if `field` is an array. If `field` is present but is not an array, an error condition is raised.

In addition to matching an exact value you can also use expressions ($pull is special in this way):

```
{ $pull : { field : {field2: value} } } removes array elements with field2 matching value
```

```
{ $pull : { field : {$gt: 3} } } removes array elements greater than 3
```

```
{ $pull : { field : {<match-criteria>} } } removes array elements meeting match criteria
```

Because of this feature, to use the embedded doc as a match criteria, you cannot do exact matches on array elements.

$pullAll

```
{ $pullAll : { field : value_array } }
```

removes all occurrences of each value in `value_array` from `field`, if `field` is an array. If `field` is present but is not an array, an error condition is raised.

$rename

Version 1.7.2+ only.

```
{ $rename : { old_field_name : new_field_name } }
```

Renames the field with name 'old_field_name' to 'new_field_name'. Does not expand arrays to find a match for 'old_field_name'.

$bit

v1.8+. 
Does a bitwise update of `field`. Can only be used with integers.

The `$ positional operator

v1.4+

The `$` operator (by itself) means “position of the matched array item in the query”. Use this to find an array member and then manipulate it. For example:

```javascript
> t.find()
{ 
  "_id": ObjectId("4b97e62b1d8c7152c9ccbf74"),
  "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("4b97e62b1d8c7152c9ccbf74"),
  "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:

```javascript
> t.find();
{ 
  
  "_id": ObjectId("4b9e4a1f583fa1c76198319"),
  "x": [ 1, 2, 3, 2 ]
}
> t.update({x: 2}, { $inc: { "x.$": 1 }}, false, true);
> t.find();
{ 
  
  "_id": ObjectId("4b9e4a1f583fa1c76198319"),
  "x": [ 1, 3, 3, 2 ]
}
```

The positional operator cannot be combined with an `upsert` since it requires a matching array element. If your update results in an insert then the `$` will literally be used as the field name.

**Using "$unset" with an expression "array.$" will result in the array item becoming null, not being removed. You can issue an update with "{$pull:{x:null}}" to remove all nulls. $pull can now do much of this so this example is now mostly historical.**

```javascript
> t.insert({x: [1,2,3,4,3,2,3,4]})
> t.find()
{ 
  
  "_id": ObjectId("4bde2ad3755d00000000710e"),
  "x": [ 1, 2, 3, 4, 3, 2, 3, 4 ]
}
> t.update({x:3}, { $unset: { x.$:1} })
> t.find()
{ 
  
  "_id": ObjectId("4bde2ad3755d00000000710e"),
  "x": [ 1, 2, null, 4, 3, 2, 3, 4 ]
}
```

**Upserts with Modifiers**

You may use `upsert` with a modifier operation. In such a case, the modifiers will be applied to the update criteria member and the resulting object will be inserted. The following upsert example may insert the object `{name:"Joe",x:1,y:1}.`  

```javascript
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:

```javascript
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

```javascript
update( {_id:'joe'},{ "$addToSet": { tags : "baseball" } } );
```

For older versions, add $ne : <value> to your query expression:

```javascript
update( {_id:'joe', tags: { }}, { "$ne" : "baseball" }, { "$push": { tags : "baseball" } } );
```

**Checking the Outcome of an Update Request**

As described above, a non-upsert update may or may not modify an existing object. An upsert will either modify an existing object or insert a new object. The client may determine if its most recent message on a connection updated an existing object by subsequently issuing a `getlasterror` command (db.runCommand( "getlasterror" )). If the result of the `getlasterror` command contains an `updatedExisting` field, the last message on the connection was an update request. If the `updatedExisting` field's value is true, that update request caused an existing object to be updated; if `updatedExisting` is false, no existing object was updated. An "upserted" field will contain the new `_id` value if an insert is performed (new as of 1.5.4).

**Notes**

**Object Padding**

When you update an object in MongoDB, the update occurs in-place if the object has not grown in size. This is good for insert performance if the collection has many indexes.

Mongo adaptively learns if objects in a collection tend to grow, and if they do, it adds some padding to prevent excessive movements. This statistic is tracked separately for each collection. [More info here.](#)

**Blocking**

Starting in 1.5.2, multi updates yield occasionally so you can safely update large amounts of data. If you want a multi update to be truly isolated (so no other writes happen while processing the affected documents), you can use the `$atomic` flag in the query expression. For example:

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

- **sharding**
  - `$atomic` is not supported with Sharding. If used it results in no updates and the `getLastError` Command returns error information.

- **atomic**
  - `$atomic` only means that the update will not yield for other operations while running. It does **not** mean that the operation is "all or nothing". See [here.](#)

**Field (re)order**

During an update the field order may be changed. There is no guarantee that the field order will be consistent, or the same, after an update. At the moment, if the update can be applied in place then the order will be the same (with additions applied at the end), but if a move is required for the document (if the currently allocated space is not sufficient for the update) then the fields will be reordered (alphanumerically).

**See Also**

- findandmodify Command
- Atomic Operations

**Atomic Operations**
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. MongoDB'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 MongoDB to work well for real-time 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 MongoDB 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 MongoDB update command supports several modifiers, all of which atomically update an element in a document. They include:

- `$set` - set a particular value
- `$unset` - delete a particular field (v1.3+)
- `$inc` - increment a particular value by a certain amount
- `$push` - append a value to an array
- `$pushAll` - append several values to an array
- `$pull` - remove a value(s) from an existing array
- `$pullAll` - remove several value(s) from an existing array
- `$bit` - bitwise operations

These modifiers are convenient ways to perform certain operations atomically.

"Update if Current"

Another strategy for atomic updates is "Update if Current". This is what an OS person would call Compare and Swap. For this we

1. Fetch the object.
2. Modify the object locally.
3. Send an update request that says "update the object to this new value if it still matches its old value".

Should the operation fail, we might then want to try again from step 1.

For example, suppose we wish to fetch one object from inventory. We want to see that an object is available, and if it is, deduct it from the inventory. The following code demonstrates this using mongo shell syntax (similar functions may be done in any language):

```javascript
> t=db.inventory
> s = t.findOne({sku:'abc'})
"_id": "a9df43e96432c73ea865a", "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.

```javascript
> t.update({sku:"abc", qty:{$gt:0} }, { $inc : { qty : -1 } } ) ; db.$cmd.findOne({getlasterror:1})
{"err": , "updatedExisting": true , "n": 1 , "ok": 1} // it worked
```

The ABA Nuance

In the first of the examples above, we basically did "update object if qty is unchanged". However, what if since our read, sku had been modified? We would then overwrite that change and lose it!

There are several ways to avoid this problem; it's mainly just a matter of being aware of the nuance.
1. Use the entire object in the update's query expression, instead of just the _id and qty field.
2. Use $set to set the field we care about. If other fields have changed, they won't be affected.
3. Put a version variable in the object, and increment it on each update.
4. When possible, use a $ operator instead of an update-if-current sequence of operations.

"Insert if Not Present"

Another optimistic concurrency scenario involves inserting a value when not already there. When we have a unique index constraint for the criteria, we can do this. See the How to Make an Auto Incrementing Field page for an example.

Find and Modify (or Remove)

See the findandmodify Command documentation for more information.

Applying to Multiple Objects At Once

You can use multi-update to apply the same modifier to every relevant object. By default a multi-update will allow some other operations (which could be writes) to interleave. Thus, this will only be pseudo-atomic (pseudo-isolated). To make it 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 than you are the only one writing when the update is done; this means each update is done without any interference from any other.

Atomic operation examples

A key goal of MongoDB is to handle a good breadth of use cases, and to handle in a way that is easy for the developer. We found that a good number of use cases require atomic operations / strong consistency; thus that is a feature of the product. Below are some examples (in mongo shell syntax).
// register a new user (atomically)
db.users.insert( { _id : 'joe123' } )
if( db.getLastErrorObj().err )
    print("name is use try another")
else
    print("you are registered")

// decrement y if y > 0 (atomically)
if
    db.stats.update(
        { _id : 'myid', y : { $gt : 0 } },
        { $inc : { y : -1 } }
    )

// assure everyone’s email address is unique
db.users.ensureIndex( {email:1} , {unique:true} )

// if joe hasn’t already voted, let him vote (without races)
if
    db.posts.update(
        { _id : 'some_post_of_interest' , voters : { $ne : 'joe' } },
        { votes : { $inc : 1 } }
    )

How to Make an Auto Incrementing Field

Generally in MongoDB, one does not use an auto-increment pattern for _id’s (or other fields), as this does not scale up well on large database clusters. Instead one typically uses Object IDs.

Side counter method

One can keep a counter of the current _id in a side document, in a collection dedicated to counters. Then use FindAndModify to atomically obtain an id and increment the counter.

> db.counters.insert({_id: "userId", c: 0});

> var o = db.counters.findAndModify(
...    {query: {_id: "userId"}, update: {$inc: {c: 1}}});
> db.mycollection.insert({_id:o.c, stuff:"abc"});

> o = db.counters.findAndModify(
...    {query: {_id: "userId"}, update: {$inc: {c: 1}}});
> db.mycollection.insert({_id:o.c, stuff:another one});

Once you obtain the next id in the client, you can use it and be sure no other client has it.

Optimistic loop method

One can do it with an optimistic concurrency “insert if not present” loop. The following example, in Mongo shell Javascript syntax, demonstrates.
function insertObject(o) {
    x = db.myCollection;
    while( 1 ) {
        // determine next _id value to try
        var c = x.find({},{\_id:1}).sort(\{\_id:-1\}).limit(1);
        var i = c.hasNext() ? c.next()._id + 1 : 1;
        o._id = i;
        x.insert(o);
        var err = db.getLastErrorObj();
        if( err && err.code ) {
            if( err.code == 11000 /* dup key */ )
                continue;
            else
                print("unexpected error inserting data: " + toJSON(err));
        }
        break;
    }
}

The above should work well unless there is an extremely high concurrent insert rate on the collection. In that case, there would be a lot of looping potentially.

See Also

- Atomic Operations

findAndModify Command

Find and Modify (or Remove)

MongoDB 1.3+ supports a "find, modify, and return" command. This command can be used to atomically modify a document 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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>a filter for the query</td>
<td>{}</td>
</tr>
<tr>
<td>sort</td>
<td>if multiple docs match, choose the first one in the specified sort order as the object to manipulate</td>
<td>{}</td>
</tr>
<tr>
<td>remove</td>
<td>set to a true to remove the object before returning</td>
<td>N/A</td>
</tr>
<tr>
<td>update</td>
<td>a modifier object</td>
<td>N/A</td>
</tr>
<tr>
<td>new</td>
<td>set to true if you want to return the modified object rather than the original. Ignored for remove.</td>
<td>false</td>
</tr>
<tr>
<td>fields</td>
<td>see Retrieving a Subset of Fields (1.5.0+)</td>
<td>All fields</td>
</tr>
<tr>
<td>upsert</td>
<td>create object if it doesn't exist; a query must be supplied! examples (1.5.4+)</td>
<td>false</td>
</tr>
</tbody>
</table>
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:

```javascript
> 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,
    "tasks" : [ "run marketing report", "email report" ]
}
```

You can pop an element from an array for processing and update in a single atomic operation:
> task = db.jobs.findAndModify(
  query: {inprogress: false, name: "Next promo"},
  update : {$pop : { tasks:-1}}, fields: {tasks:1},
  new: false
)

{ 
  "_id" : ..., 
  "tasks" : [ 
    "select product",
    "add inventory",
    "do placement"
  ] 
}

> db.jobs.find( { name : "Next promo"} )

{ 
  "_id" : ..., 
  "inprogress" : false, 
  "name" : "Next promo", 
  "priority" : 0, 
  "tasks" : [ "add inventory", "do placement" ]
}

You can also simply remove the object to be returned.

> job = db.jobs.findAndModify( {sort:{priority:-1}, remove:true} );

{ 
  "_id" : ..., 
  "inprogress" : true, 
  "name" : "Biz report", 
  "priority" : 2, 
  "started" : "Mon Oct 25 2010 10:44:15 GMT-0700 (PDT)", 
  "tasks" : [ 
    "run marketing report", 
    "email report"
  ]
}

> db.jobs.find()

{ 
  "_id" : ..., 
  "inprogress" : false, 
  "name" : "Next promo", 
  "priority" : 0, 
  "tasks" : [ "add inventory", "do placement" ]
}

{ 
  "_id" : ..., 
  "name" : "Biz report", 
  "inprogress" : false, 
  "priority" : 1, 
  "tasks" : [ "run sales report", "email report" ]
}

If the client crashes before processing the job or task in the above examples, the data will be lost forever.

See the tests for more examples.

If your driver doesn't provide a helper function for this command, run the command directly with something like this:

job = db.runCommand({ findAndModify : "jobs",
  sort : { priority : -1 },
  remove : true
}).value;
Sharding limitations

`findandmodify` will behave the same when called through a `mongos` as long as the collection it is modifying is unsharded. If the collection is sharded, then the query must contain the shard key.

See Also

- Atomic Operations

Padding Factor

- Overview
- No padding after imports, repairs, compactions and initial replica syncs
- Shrinking Documents
- Manual Padding
- See Also

Overview

When you update a document in MongoDB, the update occurs in-place if the document has not grown in size. If the document did grow in size, however, then it might need to be relocated on disk to find a new disk location with enough contiguous space to fit the new larger document. This can lead to problems for write performance if the collection has many indexes since a move will require updating all the indexes for the document.

Mongo adaptively learns if documents in a collection tend to grow, and if they do, it adds some padding so that the document has room to grow. This helps to prevent excessive movements on subsequent writes. This statistic is tracked separately for each collection.

You can check the collection’s current padding factor by running the `db.<collection-name>.stats()` command helper in the shell. The padding factor indicates what the padding will be for new record allocations (for inserts or updates that grow a document, they cause a move operation to a new location).

```snippet
> db.coll.stats()
{
  "ns" : "...", ...
  "paddingFactor" : 1, ...
  "ok" : 1
}
```

As each document is written at a different point in time the padding for each document will not be the same. Also, as the padding factor is relative to the size of each document you cannot calculate the exact amount of padding for a collection based on the average document size and padding factor.

The padding factor is 1.0 if there is no padding. 1.5 would indicate 50% padding on a new insert/moves.

No padding after imports, repairs, compactions and initial replica syncs

After compaction, repair and import operations, there is (generally) no padding as they were inserted and there were no updates (which would cause the paddingFactor to change). Thus you may see slower update performance after these cases, but the size required for storage will be lower.

Shrinking Documents

If a document gets smaller (e.g., because of an `$unset` or `$pop`), the document does not move but stays at its current location. It thus effectively has more padding. Thus space is never reclaimed if documents shrink by large amounts and never grow again. To reclaim that space run a `compact` operation (or repair).

Manual Padding

Padding in MongoDB is automatic. You should not have to do so manually. In exceptional cases one can do this though. The strategy is to add a faux field that assures allocation of a larger slot size for the document. The faux field is then removed; at this point there is extra room for expansion on a future update. Example below.
> function checkOk() { assert ( !db.runCommand({getlasterror:1,j:true,w:2,wtimeout:5000}).err ); } 

> t = db.zcollection;
> db.setProfilingLevel(2);
> // let variable b be a very long string
> t.insert({q:1})
> t.update({q:1},{$set:{yy:b}})
> checkOk()
>
> // note the "moved:true" in the output below -- indicating not enough
> // padding to avoid moving a document.
> db.system.profile.find()

... {
"ts": ISODate("2011-10-13T02:45:23.062Z"), "op": "insert", "ns": "test.zcollection",
"millis": 0, "client": "127.0.0.1", "user": ""
} {
"ts": ISODate("2011-10-13T02:45:33.560Z"), "op": "update", "ns": "test.zcollection",
"query": { "q": 1 }, "updateobj": { "$set": { "yy": "aaaaaaaaa..." } },
"nskanned": 1, "moved": true, "millis": 3, "client": "127.0.0.1", "user": ""
}

> // not important what value of 'padding' is, only its length:
> t.insert({q:2,padding:ourpaddingstring})
> t.update({q:2},{$unset:{padding:1}})
> t.update({q:2},{$set:{yy:b}})
> checkOk()
>
> // no "moved:true" below (which is good). Note however that the automatic adjustment
> // of a collection's padding factor normally achieves this regardless. Manually padding
> // is rarely necessary.
> db.system.profile.find()

... {
"ts": ISODate("2011-10-13T02:46:34.920Z"), "op": "insert", "ns": "test.zcollection",
"millis": 1, "client": "127.0.0.1", "user": ""
} {
"ts": ISODate("2011-10-13T02:46:42.775Z"), "op": "update", "ns": "test.zcollection",
"query": { "q": 2 }, "updateobj": { "$unset": { "padding": 1 } }, "nskanned": 1,
"millis": 2, "client": "127.0.0.1", "user": ""
} {
"ts": ISODate("2011-10-13T02:46:55.831Z"), "op": "update", "ns": "test.zcollection",
"query": { "q": 2 }, "updateobj": { "$set": { "yy": "aaaaaaaaa..." } },
"nskanned": 1, "millis": 2, "client": "127.0.0.1", "user": ""
}

See Also


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.
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, let's embed a document within an existing document in the collection. We'll keep working with the original `{name: "mongo"}` document for simplicity.

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:

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.

Let's repeat the above example, but create a document and place in a different collection, say `otherthings`, and embed that as a reference in our favorite "mongo" object under the key "otherdata".
// first, save a new doc in the 'otherthings' collection

> var other = { s: "other thing", n: 1};
> db.otherthings.save(other);
> db.otherthings.find();

{ "_id": "497dbcb36b27d59a708e89a4", "s": "other thing", "n": 1 }

// now get our mongo object, and add the 'other' doc as 'otherthings'

> var mongo = db.things.findOne();
> print(tojson(mongo));

{ "_id": "497dab624ee47b3a675d2d9c", "name": "mongo", "type": "database", "data": { "a": 1, "b": 2 } }
> mongo.otherthings = new DBRef( 'otherthings' , other._id );

{ "s": "other thing", "n": 1, "_id": "497dbcb36b27d59a708e89a4" }
> db.things.save(mongo);
> db.things.findOne().otherthings.fetch();

{ "_id": "497dab624ee47b3a675d2d9c", "name": "mongo", "type": "database", "data": { "a": 1, "b": 2, "otherthings": { "_id": "497dbcb36b27d59a708e89a4", "s": "other thing", "n": 1 } } }

// now, let's modify our 'other' document, save it again, and see that when the dbshell
// gets our mongo object and prints it, if follows the dbref and we have the new value

> other.n = 2;

2
> db.otherthings.save(other);
> db.otherthings.find();

{ "_id": "497dbcb36b27d59a708e89a4", "s": "other thing", "n": 2 }
> db.things.findOne().otherthings.fetch();

{ "_id": "497dab624ee47b3a675d2d9c", "name": "mongo", "type": "database", "data": { "a": 1, "b": 2, "otherthings": [ { "_id": "497dbcb36b27d59a708e89a4", "s": "other thing", "n": 2 } } ] }
functions are written in JavaScript and execute on the server.

Command syntax:

```javascript
db.runCommand(
  { mapreduce : <collection>,
    map : <mapfunction>,
    reduce : <reducefunction>,
    out : <see output options below>
    [, query : <query filter object>]  //, sort : <sorts the input objects using this key. Useful for optimization, like sorting by the emit key for fewer reduces>
    [, limit : <number of objects to return from collection, not supported with sharding>]  //, keeptemp : <true|false>
    [, finalize : <finalizefunction>]  //, jsMode : true
    [, verbose : true]
  }
);
```

* finalize - function to apply to all the results when finished
  
  - keeptemp - If true, the generated collection is not treated as temporary. Defaults to false. When out is specified, the collection is automatically made permanent. (MongoDB <=1.6)
  - scope - can pass in variables that can be access from map/reduce/finalize. Note that updates to scope variables' values are not shared among shard members, so in a sharded cluster you should treat scope variables as global constants.
    ```javascript
    example mr5
    ```
    - verbose - provide statistics on job execution time

**Incremental Map-reduce**

If the data set over which you’d like to perform map-reduce aggregations is constantly growing, then you may want to take advantage of incremental map-reduce. The prevents you from having to aggregate over the entire data set each time you want to see your results.

To perform incremental map-reduce, take the following steps:

1. First, run a map-reduce job over an existing collection, and output the data to it's own output collection.
2. When you have more data to process, run a second map-reduce job, but use the query option to the filter the documents to include only new documents.
3. Use the reduce output option. This will use your reduce function to merge the new data into the existing output collection.

**Output options**

**pre-v1.8:** If you do not specify a value for `out`, then the results will be placed into a temporary collection whose name will be given in command's output (see below). Otherwise, you can specify the name of a collection for the `out` option and the results will be placed there.

**v1.8+:** the output options have changed. Map-reduce no longer generates temporary collections (thus, keepTemp has been removed). Now, you must always supply a value for `out`. The `out` directives are:

- "className" - By default the output will be of type "replace".
- `{ replace : "className" }` - the output will be inserted into a collection which will atomically replace any existing collection with the same name.
- `{ merge : "className" }` - 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 : "className" }` - If documents exists for a given key in the result set and in the old collection, then a reduce operation (using the specified reduce function) will be performed on the two values and the result will be written to the output collection. If a finalize function was provided, this will be run after the reduce as well.
- `{ inline : 1 }` - With this option, no collection will be created, and the whole map-reduce operation will happen in RAM. Also, the results of the map-reduce will be returned within the result object. Note that this option is possible only when the result set fits within the 16MB limit of a single document. In **v2.0**, this is your only available option on a replica set secondary.

For example:

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

Additional options within `out` objects are:
• "db" - the db name to output to.

```javascript
out : {replace : "collectionName", db : "otherDB"}
```

• `{ sharded : 1 }` - **MongoDB 1.9+** If true and combined with an output mode that writes to a collection, the output collection will be sharded using the _id field. See details in the sharding section.

• `{ nonAtomic : 1 }` - **MongoDB 2.1+** If true then the post processing step will not be executed under the write lock. This option makes sense only for the "merge" and "replace" output which take time to process and would lock the database. Using this flag means that partial results will be visible as they are processed.

Note: the order of the objects in the out parameter matter.

**Result object**

```javascript
{
[results : <document_array>],
[result : <collection_name> | {db: <db>, collection: <collection_name>}),]
timeMillis : <job_time>,
counts : {
  input : <number of objects scanned>,
  emit : <number of times emit was called>,
  output : <number of items in output collection>
},
ok : <1_if_ok>
[, err : <errmsg_if_error>]
}
```

Either the result or the results field will be present depending on your output type. The results element is only present if the inline output option was used. The value of the results element is an array of embedded documents containing the results. If you chose any other output type the result field will be a string with the name of the collection holding the results, or an embedded document containing the db and collection if you chose to output to another db.

A command helper is available in the MongoDB shell:

```javascript
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).

```javascript
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:

```javascript
{
  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:
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:

```javascript
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:

```javascript
{ 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:

```javascript
{ count: 1, likes: 5 }
```

Then the `reduce` function will be invoked in this way:

```javascript
reduce("jones", [ {count: 10, likes: 247}, { count: 1, likes: 5} ])
```

And the resulting document will be a simple combination (or reduction) of those values:

```javascript
{ 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

```javascript
function reduce(key, array_of_value) -> value
```

OR

```javascript
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:

```javascript
for all k,vals : reduce( k, [reduce(k,vals)] ) == reduce(k,vals)
```

This also means the following is true:

```javascript
emit( this.username, {count: 1, likes: this.likes} );
```
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.

- 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.

- If the map function emits a single document for a given key, the reduce function will not be called.

**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.

```javascript
function finalize(key, value) -> final_value
```

Your reduce function may be called multiple times for the same object. Use finalize when something should only be done a single time at the end; for example calculating an average.

**jsMode flag**

v2.0+

Normally, map/reduce execution follows the steps:

- convert from BSON to JS, execute map, convert from JS to BSON
- convert from BSON to JS, execute reduce, convert from JS to BSON

Thus it requires several translations but it can handle very large datasets during mapping by using a temporary collection. It is possible to make the execution stay in JS by using `{jsMode: true}` which performs the following steps:

- convert from BSON to JS, execute map
- execute reduce, convert from JS to BSON

The execution time may be significantly reduced. Note that this mode is limited by either

- `jsMode` is limited by the JS heap size and a maximum of 500k unique keys. Consequently it is not suitable for large jobs in which case mongo may revert to regular mode.

**Sharded Environments**

There are 2 aspects of sharding with Map/Reduce, input and output.

**Sharded input**

If the input collection is sharded, MongoS will automatically dispatch the map/reduce job to each of the shard, to be executed in parallel. There is no special option required. MongoS will then wait for jobs on all shards to finish.

**Sharded output**

By default the output collection will not be sharded. The process is:

- MongoS dispatches a map/reduce finish job to the shard that will store the target collection.
- that shard will pull results from all other shards, run a final reduce/finalize, and write to the output.

If using the `sharded` option in the `out` object, the output will be sharded using `"_id"` as the shard key.
In version prior to 2.1, the process is:

- MongoS pulls the results from each shard, doing a merge sort to get them ordered.
- on the fly, it does reduce/finalize as needed. Then writes the result to the output collection in sharded mode.
- though MongoS does some processing, only a small amount of memory is required even for large datasets.
- there is currently a limitation in that shard chunks do not get automatically split and migrated during insertion. Doing it manually is required until the chunks are granular and balanced.

⚠️ Sharded output for mapreduce has been overhauled for v2.2. Using it for prior version is not recommended due to many limitations.

For v2.2 and above, the process is:

- if the target collection does not exist, it is created as sharded on _id. Even if empty, its initial chunks are created based on the result of the 1st step of MR.
- MongoS dispatches a map/reduce finish job to every shard that owns a chunk, in parallel.
- each shard will pull results it owns from all other shards, run a final reduce/finalize, and write to the target collection.
- During further MR jobs, chunk splitting will be done as needed.
- Balancing of chunks for the target collection is automatically prevented during post-processing to avoid concurrency issues.

Examples

**Shell Example 1**

The following example assumes we have an *events* collection with objects of the form:

```json
{ 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":

```sh
> m = function() { emit(this.user_id, 1); }
> r = function(k,vals) {
  return 1;
}
> res = db.events.mapReduce(m, r, { query : {type:'sale'} });
> // or in v1.8+:
> // res = db.events.mapReduce(m, r, { query : {type:'sale'}, out : 'example1' });
> db[res.result].find().limit(2)
{ "_id" : 8321073716060 , "value" : 1 }
{ "_id" : 7921232311289 , "value" : 1 }
```

If we also wanted to output the number of times the user had experienced the event in question, we could modify the reduce function like so:

```sh
> 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 Shell Script with Incremental Map-Reduce and Finalize

This example is a JavaScript script file. The map-reduce can be run repeatedly on different dates to incrementally augment the result. The finalize option computes averages.

The output of commands and the queries themselves are saved to variables so that they can be examined after the sample script is run via the load() command in the shell.

```javascript
// work in the map-reduce example db
db = db.getSiblingDB("mrex");

// clean out from previous runs of this sample -- you wouldn't do this in production
db.session.drop();

// simulate saving records that log the lengths of user sessions in seconds
db.session.save({userid:"a", ts: ISODate('2011-11-03 14:17:00'), length: 95 });
db.session.save({userid:"b", ts: ISODate('2011-11-03 14:23:00'), length: 110 });
db.session.save({userid:"c", ts: ISODate('2011-11-03 15:02:00'), length: 120 });
db.session.save({userid:"d", ts: ISODate('2011-11-04 16:45:00'), length: 45 });
```

This script demonstrates how to define a map function and a reduce function for use in a map-reduce operation within a MongoDB shell script. The code snippet shows the insertion of documents into a collection named `things` and how these documents are processed using a map-reduce function. The map function processes the `tags` field of each document, while the reduce function aggregates the counts of each unique tag.
/*
For each user, count up the number of sessions, and figure out the average
session length.

Note that to be able to find the average session length, we need to keep a
total of all the session lengths, and then divide at the end.

We're also going to set this up so that we can repeat the process to get
incremental results over time.
*/

function mapf()
{
    emit(this.userid,
         {userid:this.userid, total_time:this.length, count:1, avg_time:0});
}

function reducef(key, values)
{
    r = {userid:key, total_time:0, count:0, avg_time:0};
    values.forEach(function(v)
    {
        r.total_time += v.total_time;
        r.count += v.count;
    });
    return r;
}

function finalizef(key, value)
{
    if (value.count > 0)
        value.avg_time = value.total_time / value.count;
    return value;
}

/*
Here's the initial run.

The query isn't technically necessary, but is included here to demonstrate
how this is the same map-reduce command that will be issued later to do
incremental adjustment of the computed values. The query is assumed to run
once a day at midnight.
*/
var mrcom1 = db.runCommand( { mapreduce:"session",
               map:mapf,
               reduce:reducef,
               query: {ts: {$gt:ISODate('2011-11-03 00:00:00')}},
               out: { reduce: "session_stat" },
               finalize:finalizef
          });

function saveresults(a)
{
    /* append everything from the cursor to the argument array */
    var statcurs = db.session_stat.find();
    while(statcurs.hasNext())
        a.push(statcurs.next());
}

/* save the results into mrres1 */
var mrres1 = [];
saveresults(mrres1);

/* add more session records (the next day) */
db.session.save({userid:"a", ts: ISODate('2011-11-05 14:17:00'), length: 100});
db.session.save({userid: "b", ts: ISODate('2011-11-05 14:23:00'), length: 115});
db.session.save({userid: "c", ts: ISODate('2011-11-05 15:02:00'), length: 125});
db.session.save({userid: "d", ts: ISODate('2011-11-05 16:45:00'), length: 55});

/*
Run map reduce again.

This time, the query date is the next midnight, simulating a daily job that
is used to update the values in session_stat. This can be repeated daily
(or on other periods, with suitable adjustments to the time).
*/

var mrcom2 = db.runCommand( { mapreduce: "session",
  map:mapf,
  reduce:reducef,
  query: {ts: {gte:ISODate('2011-11-05 00:00:00')}},
  out: { reduce: "session_stat" },
  finalize:finalizef
});

/* save the results into mrres2 */
More Examples

- example mr1
- Finalize example: example mr2

Note on Permanent Collections

Even when a permanent collection name is specified, a temporary collection name will be used during processing. At map/reduce completion, the temporary collection will be renamed to the permanent name atomically. Thus, one can perform a map/reduce job periodically with the same target collection name without worrying about a temporary state of incomplete data. This is very useful when generating statistical output collections on a regular basis.

Parallelism

See info on Concurrency

Presentations

Map/reduce, geospatial indexing, and other cool features - Kristina Chodorow at MongoSF (April 2010)

Troubleshooting

- See Troubleshooting MapReduce

See Also

- Aggregation
- Kyle’s Map/Reduce basics
- Blog post - walkthrough a mongodb map reduce job

Troubleshooting MapReduce

Tips on troubleshooting map/reduce.

Troubleshooting the map function

We can troubleshoot the map function in the shell by defining a test emit function in the shell and having it print out trace information.

For example suppose we have some data:

```javascript
var mrres2 = [];
saveresults(mrres2);
```

> 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:

```javascript
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():

```javascript
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:

```javascript
> 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:

```javascript
> 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:
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.

```javascript
> reduce( { name: 'joe' }, [ { votes: 1 } ] )
{ votes: 1 }
```

The same can also be tested with two values. The format should still be the same.

```javascript
> 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:

```javascript
> reduce( { name: 'joe' }, [ { votes: 1 }, { votes: 3 } ] )
{ votes: 4 }
> reduce( { name: 'joe' }, [ { votes: 3 }, { votes: 1 } ] )
{ votes: 4 }
```
Reduce output can be re-reduced:

```javascript
> reduce( { name : 'joe' }, [
  { votes : 1 },
  reduce ( { name : 'joe' }, [ { votes : 3 } ] )
] )
{ votes : 4 }
```

### Data Processing Manual

By "data processing", we generally mean operations performed on large sets of data, rather than small interactive operations.

#### Import

One can always write a program to load data of course, but the `mongoimport` utility also works for some situations. `mongoimport` supports importing from json, csv, and tsv formats.

A common usage pattern would be to use `mongoimport` to load data in a relatively raw format and then use a server-side script (`db.eval()` or `map/reduce`) to reduce the data to a more clean format.

#### See Also

- Import/Export Tools
- Aggregation Framework
- Map/Reduce

### mongo - The Interactive Shell

- [MongoDB Manual Shell Reference](#)
- [More Information](#)
- [Presentations](#)

The MongoDB distribution includes `bin/mongo`, the MongoDB interactive shell. This utility is a JavaScript shell that allows you to issue commands to MongoDB from the command line. *(it is basically an extended SpiderMonkey shell)*

The shell is useful for:

- inspecting a database's contents
- testing queries
- creating indices
- maintenance scripts
- other administrative functions

When you see sample code in this wiki and it looks like JavaScript, assume it is a shell example. See the [driver syntax table](#) for a chart that can be used to convert those examples to any language.

### MongoDB Manual Shell Reference

- [MongoDB Manual Reference Section](#)
- [mongo Manual Page](#)
- [JavaScript Interface](#)
- [MongoDB Database Commands](#)

### More Information

- [Shell Overview](#)
- [Shell Scripts](#)
- [Shell Reference](#)
- [Shell API Docs](#)

### Presentations

- [Hacking the Shell - MongoSF (May 2011)](#)
- [CRUD and the JavaScript Shell - MongoSF (April 2010)](#)
Scripting the shell

The MongoDB shell is not just an interactive shell, it can also be scripted using JS files. In addition to specifying a Javascript file (*.js) you can also use --eval with a snippet of JS.

Using the shell this way allows for tasks to be performed without the need for any additional drivers or language support; it can be used in cron, or automated administrative tasks. Please be aware there are data format issues in javascript so you should be careful how much you do in Javascript.

Common uses for the scripted shell includes:

- backups
- scheduled Map-Reduce commands
- offline reports
- administration

Running a Script

```bash
./mongo server:27017/dbname --quiet my_commands.js
```

The syntax stems from the interactive shell. This command will execute the my_commands.js as if it had been entered into the shell directly, with some exceptions.

- ./mongo: command to start the interactive shell, may vary on your shell of choice
- server:27017/dbname: basic connection information
- --quiet: this is a flag for the mongo command. This switch removes some header information that is not typically necessary when building unattended scripts.
- my_commands.js: a file containing a series of shell commands to execute

--quiet

This option will remove the header printed at the top when you run the shell normally:

```
$ mongo
MongoDB shell version: 2.1.0
connecting to: test
> ^C
bye
$ mongo --quiet
> ^C
```

--eval

In addition to using a full Javascript file you can also pass in a Javascript fragment:

```
bash-3.2$ ./mongo test --eval "printjson(db.getCollectionNames())"
MongoDB shell version: 1.8.0
connecting to: test
[
  "system.indexes",
  "t1",
  "t2",
  "test.fam",
  "test1",
  "test11",
  "testBinary",
  "testarray"
]
```

Differences between scripted and interactive

Printing
When using the shell interactively, the shell will print returned values and format where possible. This is done as a general convenience from within the shell. However, when building a script, the printing needs to be defined explicitly.

There are two functions commonly used for this:

1. `print()`: works as normal javascript
2. `printjson()`: prints a nicely formatted JSON representation of the given object

**Example**: print JSON for the first 10 objects from a `find`

```
db.foo.find({x:1}).forEach(printjson)
```

**use dbname**

This command does not work in scripted mode. Instead you will need to explicitly define the database in the connection (`/dbname` in the example above).

Alternately, you can also create a connection within the script:

```
> help connect // for more help
> var x = new Mongo('host[:port]');
> var mydb = x.getDB('mydb');
> // or
> var mydb = connect('host[:port]/mydb');
```

Or just reassign the “db” variable.

```
> db = db.getSiblingDB("otherdb") // same as use otherdb
```

**it**

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

**getLastError**

When running an `update/insert` command from the shell, the shell automatically awaits a reply (*i.e. runs a get last error*).

The same is not true when running from a script file. To wait for the status of an operation (such as a write), run the `getLastError` function after `update/insert`.

```
db.getLastErrorObj()
// or
db.getLastError()
```

**Overview - The MongoDB Interactive Shell**

- Running the Shell
  - `.mongorc.js`
- Operations
  - Help
  - Select Database
  - Querying
  - Inserting
  - Updating
  - Deleting
  - Indexes
- Open Additional Connections
- Working from the Prompt
  - Line Continuation
  - Key Shortcuts
  - Custom Prompt
  - Using a real editor
Some Notes on Datatypes in the Shell

- Numbers
- Dates
- BinData

See Also

Running the Shell

The interactive shell is included in the standard MongoDB distribution. To start the shell, go into the root directory of the distribution and type

```
./bin/mongo
```

It might be useful to add `mongo_distribution_root/bin` to your `PATH` so you can just type `mongo` from anywhere.

If you start with no parameters, it connects to a database named "test" running on your local machine on the default port (27017). You can see the `db` to which you are connecting by typing `db`:

```
./mongo
type "help" for help
> db
  test
```

You can pass `mongo` an optional argument specifying the address, port and even the database to initially connect to:

```
./mongo foo
connects to the foo database on your local machine
```

```
./mongo 192.168.13.7/foo
connects to the foo database on 192.168.13.7
```

```
./mongo dbserver.mydomain.com/foo
connects to the foo database on dbserver.mydomain.com
```

```
./mongo 192.168.13.7:9999/foo
connects to the foo database on 192.168.13.7 on port 9999
```

`.mongorc.js`

1.9.1+

When the shell is launched, it checks the user's home directory for a javascript file named `.mongorc.js`. If this file is found, its contents are interpreted and run by the shell prior to displaying the prompt for the first time. This allows the user to define variables, customize the prompt, or update information that they would like updated every time they launch a shell. This functionality can be overridden with the `--norc` flag. It should be noted that if a file is specified to be executed by the shell, the rc file will not be run until after that file has completed.

Operations

Help

```
> help // top level help
> db.help() // help on db-specific methods
> db.mycollection.help() // help on collection methods
> db.mycollection.find().help() // cursor help
```

Select Database

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>show dbs</code></td>
<td>displays all the databases on the server you are connected to</td>
</tr>
<tr>
<td><code>use db_name</code></td>
<td>switches to <code>db_name</code> on the same server</td>
</tr>
<tr>
<td><code>show collections</code></td>
<td>displays a list of all the collections in the current database</td>
</tr>
</tbody>
</table>

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:

```javascript
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:

```javascript
DBQuery.shellBatchSize = #
```

If the shell does not accept the collection name (for example if it starts with a number, contains a space etc), use `db['foo'].find()` instead.

Inserting

In order to insert data into the database, you can simply create a JavaScript object, and call the save() method. For example, to save an object `{ name: "sara" }` in a collection called foo, type:

```javascript
db.foo.save({ name : "sara" });
```

Note that MongoDB will implicitly create any collection that doesn't already exist.

Updating

Let's say you want to change someone's address. You can do this using the following mongo commands:

```javascript
person = db.people.findOne( { name : "sara" });
person.city = "New York";
db.people.save( person );
```

Deleting

```javascript
db.foo.drop()  // drop the entire foo collection
db.foo.remove()  // remove all objects from the collection
db.foo.remove({ name : "sara" })  // remove objects from the collection where name is sara
```

Indexes

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

Open Additional Connections

You can use the following commands to open additional connections (normally you don't need to do this, but might from a script):
conn = new Mongo(host);
db = conn.getDB(dbname);
db.auth(username,password);

where host is a string that contains either the name or address of the machine you want to connect to (e.g. "192.168.13.7") or the machine and port (e.g. "192.168.13.7:9999"). Note that host in an optional argument, and can be omitted if you want to connect to the database instance running on your local machine. (e.g. conn = new Mongo() )

Alternatively you can use the connect helper method:

> db = connect("localhost:27020/mytestdb"); // example with a nonstandard port #

Working from the Prompt

Line Continuation

If a line contains open '(' or '{' characters, the shell will request more input before evaluating:

> function f() {
... x = 1;
... }

You can press Enter twice to escape from "..." mode and terminate line entry.

Key Shortcuts

- up/down array for command history
- in v1.9+ some basic emacs keystrokes work
- ctrl-l to clear the screen
- tab for auto-complete (newer versions only)
- ctrl-c to exit
- Enter twice to break out of line continuation mode

Custom Prompt

1.9.1+

The shell's prompt can be customized by creating variable 'prompt' in the shell. It can be any arbitrary javascript, including a function that returns a string. This flexibility allows for additional information to be displayed in the prompt. For example, to have a prompt that contains the number of commands issued, type:

> cmdCount = 1;
> prompt = function() {
... return (cmdCount++) + "> ";
... }
1> command
2> anothercommand
3>

To make the prompt look a bit more familiar, we can make it database@host$:

admin@mylaptop.local$ use monkeys
switched to db monkeys
monkeys@mylaptop.local$
You could use the prompt to do a bit of database monitoring as well:

```javascript
// prompt = function() {
  ...
  return "Uptime:"+db.serverStatus().uptime+" Files:"+db.stats().objects+" > ";
  ...
}  
Uptime:5897 Files:6 > db.monkeys.save({name : "James"});
Uptime:5948 Files:7 >
```

Using a real editor

We've added a feature to allow you to edit larger values including functions using your editor of choice. Just run `edit` nameOfVariableOrFunction and we will open whatever editor you have defined in your $EDITOR environment variable. Make sure that you save the file when editing. If you wish to discard your changes, you can either not save or make your editor exit with an error (`:cq` in Vim or `kill-emacs 1`) in Emacs.

```bash
$ EDITOR=vim mongo --nodb
MongoDB shell version: 2.1.0-
> function f() {}
> edit f
> f
function f() {
  print("this really works");
}
> f()
this really works
> o = {}  
> edit o
> o
{ "soDoes" : "this" }
>
```

It is possible that the code in functions will be slightly modified by the JavaScript compiler when you try to edit it again. For example it may convert `1+1` to `2` and strip out comments. The actual changes will vary based on the version of JavaScript used, but should not effect the semantics of the code, only its appearance.

Some Notes on Datatypes in the Shell

**Numbers**

By default, the shell treats all numbers as floating-point values. You have the option to work with 64-bit integers by using a class built into the shell called `NumberLong()`. If you have long/integer BSON data from the database you may see something like this:

```javascript
{ "count" : NumberLong("575175") }
```

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:

```javascript
doc = { field: new NumberLong("123212313") }
```

Note that prior to 1.6 long numbers might be displayed like this:
Dates

The `Date()` function returns a string and a `new Date()` will return an object (which is what you should use to store values).

```javascript
> 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

```javascript
> new Date()
ISODate("2010-11-29T19:41:46.730Z")
> ISODate("2010-11-29T19:41:46.730Z")
ISODate("2010-11-29T19:41:46.730Z")
```

As you can see, ISODate is a thin wrapper around the Date constructor to fix some of its shortcomings. It returns a normal Date object with all of the normal methods that JavaScript Date methods support. We have also changed the way that Date objects print to make sure that they don't look like strings and that if you copy and paste the output you get the same object.

BinData

The BSON BinData datatype is represented via class BinData in the shell. Run `help misc` for more information.

```javascript
> new BinData(2, "1234")
BinData(2, "1234")
```

See Also

- MongoDB Shell Reference

**dbshell (mongo) Reference**

- History
- Command Line
- Special Command Helpers
- Basic Shell Javascript Operations
- Queries
- Error Checking
- Administrative Command Helpers
- Opening Additional Connections
- Miscellaneous
- Examples

The dbshell is called mongo (or mongo.exe on Windows). It is an interactive command line utility for administering and querying a MongoDB database.

**History**

You can scroll through previous commands issued to mongo with the up and down arrow keys. Command history is stored in `~/.dbshell`. 
**Auth Commands**
Authentication commands will not be saved in your history.

**Windows Users**
Before version 2.2.0 there was a bug and the file was not stored correctly in your home directory. It was stored in the working directory.

**Command Line**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--help</td>
<td>Show command line options</td>
</tr>
<tr>
<td>--nodb</td>
<td>Start without a db, you can connect later with <code>new Mongo()</code> or <code>connect()</code></td>
</tr>
<tr>
<td>--shell</td>
<td>After running a <code>.js</code> file from the command line, stay in the shell rather than terminating</td>
</tr>
</tbody>
</table>

**Special Command Helpers**
Non-javascript convenience macros:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>Show help</td>
</tr>
<tr>
<td>db.help()</td>
<td>Show help on db methods</td>
</tr>
<tr>
<td>db.myColl().help()</td>
<td>Show help on collection methods</td>
</tr>
<tr>
<td>show dbs</td>
<td>Print a list of all databases on this server</td>
</tr>
<tr>
<td>use dbname</td>
<td>Set the db variable to represent usage of <code>dbname</code> on the server</td>
</tr>
<tr>
<td>show collections</td>
<td>Print a list of all collections for current database</td>
</tr>
<tr>
<td>show users</td>
<td>Print a list of users for current database</td>
</tr>
<tr>
<td>show profile</td>
<td>Print most recent profiling operations that took &gt;= 1ms</td>
</tr>
</tbody>
</table>

**Basic Shell Javascript Operations**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>The variable that references the current database object / connection. Already defined for you in your instance.</td>
</tr>
<tr>
<td>db.auth(user, pass)</td>
<td>Authenticate with the database (if running in secure mode).</td>
</tr>
<tr>
<td>coll = db.collection</td>
<td>Access a specific <code>collection</code> within the database.</td>
</tr>
<tr>
<td>cursor = coll.find();</td>
<td>Find all objects in the collection. See queries.</td>
</tr>
<tr>
<td>coll.remove(objpattern);</td>
<td>Remove matching objects from the collection. objpattern is an object specifying fields to match. E.g.: <code>coll.remove( { name: &quot;Joe&quot; } );</code></td>
</tr>
<tr>
<td>coll.save(object)</td>
<td>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)</td>
</tr>
<tr>
<td>coll.insert(object)</td>
<td>Insert object in collection. No check is made (i.e., no upsert) that the object is not already present in the collection.</td>
</tr>
<tr>
<td>coll.update(...)</td>
<td>Update an object in a collection. See the Updating documentation; update() has many options.</td>
</tr>
<tr>
<td>coll.ensureIndex( { name : 1 } )</td>
<td>Creates an index on <code>tab.name</code>. Does nothing if index already exists.</td>
</tr>
<tr>
<td>coll.update(...)</td>
<td>Update object in a collection.</td>
</tr>
<tr>
<td>coll.drop()</td>
<td>Drops the collection <code>coll</code></td>
</tr>
</tbody>
</table>
**Queries**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db.getSiblingDB(name)</code></td>
<td>Return a reference to another database using this same connection. This allows for cross database queries. Another synonym is &quot;getDbSisterDB&quot;. Usage example: <code>db.getSiblingDB('production').getCollectionNames()</code></td>
</tr>
<tr>
<td><code>coll.find()</code></td>
<td>Find all.</td>
</tr>
<tr>
<td><code>coll.find(criteria);</code></td>
<td>Find objects matching <code>criteria</code> in the collection. E.g.: <code>coll.find( { name: &quot;Joe&quot; } );</code></td>
</tr>
<tr>
<td><code>coll.findOne(criteria);</code></td>
<td>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></td>
</tr>
<tr>
<td><code>coll.find(criteria, fields);</code></td>
<td>Get just specific fields from the object. E.g.: <code>coll.find( {}, {name:true} );</code></td>
</tr>
<tr>
<td><code>coll.find().sort({field:1, field:1});</code></td>
<td>Return results in the specified order (field ASC). Use -1 for DESC.</td>
</tr>
<tr>
<td><code>coll.find(criteria).sort( { field : 1 } )</code></td>
<td>Return the objects matching <code>criteria</code>, sorted by <code>field</code>.</td>
</tr>
<tr>
<td><code>coll.find( ... ).limit(n)</code></td>
<td>Limit result to <code>n</code> rows. Highly recommended if you need only a certain number of rows for best performance.</td>
</tr>
<tr>
<td><code>coll.find( ... ).skip(n)</code></td>
<td>Skip <code>n</code> results.</td>
</tr>
<tr>
<td><code>coll.count()</code></td>
<td>Returns total number of objects in the collection.</td>
</tr>
<tr>
<td><code>coll.find( ... ).count()</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

More information: see queries.

**Error Checking**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db.getLastError()</code></td>
<td>Returns error from the last operation.</td>
</tr>
<tr>
<td><code>db.getPrevError()</code></td>
<td>Returns error from previous operations.</td>
</tr>
<tr>
<td><code>db.resetError()</code></td>
<td>Clear error memory.</td>
</tr>
</tbody>
</table>

**Administrative Command Helpers**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db.cloneDatabase(fromhost)</code></td>
<td>Clone the current database from the other host specified. fromhost database must be in noauth mode.</td>
</tr>
<tr>
<td><code>db.copyDatabase(fromdb, todb, fromhost)</code></td>
<td>Copy fromhost/fromdb to todb on this server. fromhost must be in noauth mode.</td>
</tr>
<tr>
<td><code>db.fromColl.renameCollection(toColl)</code></td>
<td>Rename collection from fromColl to toColl.</td>
</tr>
<tr>
<td><code>db.repairDatabase()</code></td>
<td>Repair and compact the current database. This operation can be very slow on large databases.</td>
</tr>
<tr>
<td><code>db.addUser(user,pwd)</code></td>
<td>Add user to current database.</td>
</tr>
<tr>
<td><code>db.getCollectionNames()</code></td>
<td>get list of all collections.</td>
</tr>
<tr>
<td><code>db.dropDatabase()</code></td>
<td>Drops the current database.</td>
</tr>
</tbody>
</table>
Opening Additional Connections

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db = connect(&quot;&lt;host&gt;:&lt;port&gt;/&lt;dbname&gt;&quot;)</code></td>
<td>Open a new database connection. One may have multiple connections within a single shell, however, automatic getLastError reporting by the shell is done for the 'db' variable only.</td>
</tr>
<tr>
<td><code>conn = new Mongo(&quot;hostname&quot;)</code></td>
<td>Open a connection to a new server. Use <code>getDb()</code> to select a database thereafter.</td>
</tr>
<tr>
<td><code>db = conn.getDB(&quot;dbname&quot;)</code></td>
<td>Select a specific database for a connection</td>
</tr>
</tbody>
</table>

Miscellaneous

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Object.bsonsize(db.foo.findOne())</code></td>
<td>prints the bson size of a db object (mongo version 1.3 and greater)</td>
</tr>
<tr>
<td><code>db.foo.findOne().bsonsize()</code></td>
<td>prints the bson size of a db object (mongo versions predating 1.3)</td>
</tr>
</tbody>
</table>

For a full list of functions, see the [shell API](http://docs.mongodb.org/manual/faq/developers/).

Examples

The MongoDB source code includes a [jstests/](http://docs.mongodb.org/manual/faq/developers/) directory with many mongo shell scripts.

Developer FAQ

⚠️ Redirection Notice
This page should redirect to [http://docs.mongodb.org/manual/faq/developers/](http://docs.mongodb.org/manual/faq/developers/).

How does concurrency work

- mongos
- mongod
- Monitoring and introspection
- Administrative commands and locking
- Operations
- Javascript
  - Group Command
  - MapReduce
- See Also


The documentation below covers MongoDB v2.2. v2.2 has substantial concurrency and parallelism improvements over past releases; for example there is no longer a “global read-write lock” in mongod.

mongos

For [sharded](http://docs.mongodb.org/manual/reference/sharding/) environments, each mongos process can perform any number of operations concurrently. This results in downstream operations to mongod instances. Execution of operations at each mongod is independent; that is, one mongod does not block another.

mongod

mongod utilises two mechanisms to achieve parallelism of operation.

mongod threads yield their lock (read or write) in two classes of situations:

- yield-for-page-fault – v2.2 includes a "PageFaultException" architecture which causes the vast majority of physical i/o (page faults) to occur outside of any lock. This is very helpful for I/O parallelism, and less so for CPU core parallelism. That is to say, PageFaultException makes it easy to saturate a large number of disk spindles simultaneously. The yielding is very granular - "record level" if you will – so this aspect works fine even if you have only a single collection. Not 100% of page faults will be outside of locks, but the vast majority are. Faults within a lock, if any, are reporting in an MMS graph; thus this can be monitored.

- lock per database – in v2.2 each database (analogous to a 'schema' in some products) has its own reader-writer lock. The lock per database is helpful (contrasted with yield-for-page-fault) to achieve good cpu core parallelism (if you have multiple databases). In addition
it sometimes is helpful for administrative/DBA operations (e.g. create index, compact command) – these use this lock so a different db
would be unblocked during the admin operation. Note replication internally has a database – local – so even if you have a single
database this change is helpful.

A good rule of thumb is: if your database is far larger than ram, PageFaultException is key for maximizing disk subsystem utilization. If your
database fits in ram, you may be cpu-bound, in which case the lock per database is important.

More granular locking will be provided post v2.2 (not necessarily v2.4). SERVER-1240.

Monitoring and introspection

- Use `db.currentOp()` to view operations in progress, and `db.killOp()` to terminate an operation.
- `db.serverStatus()` reports information on each lock. `mongotop` and `mongostat` utilities, as well as MMS, present this information in
  a more human-readable format.

Administrative commands and locking

Certain administrative commands can exclusively lock the database in question in a mongod process for extended periods of time. Specifically,
these commands block for long periods:

- foreground index creation
- reindex
- compact command
- repair database
- creating a very large (many gigabytes) capped collection
- validate collection command
- lock and fsync command

On a small collection, the lock time may only be a few seconds. On very large collections, take the mongod instance offline so that clients are not
affected. For example if the server is part of a replica set, let other members service load while maintenance is in progress.

The following commands are fast and will not block the system excessively:

- dropIndex
- getLastError
- isMaster
- replSetGetStatus
- serverStatus
- auth
- addUser

Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Lock type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>db read lock</td>
<td>yields (periodically)</td>
</tr>
<tr>
<td>get more from a cursor</td>
<td>read lock</td>
<td>yields</td>
</tr>
</tbody>
</table>
| insert          | db write lock | Inserts are normally fast and short-lived operations; exception is if the collection has many indexes and they
don't fit in RAM, or if freelists are extremely long. |
| remove          | db write lock | yields (both periodically and when faults would occur) |
| update          | db write lock | yields (both periodically and when faults would occur) |
| map/reduce      | at times locked | Allows substantial concurrent operation but exclusive to other javascript execution. |
| create index    | See notes     | Default "foreground mode" building locks the db in question for extended periods of time. |
| `db.eval()`     | write-lock/none | Substantial blocking without "nolock" option; completely blocking to all javascript – see note below |
| getLastError command | non-blocking |                                          |
| serverStatus command | non-blocking |                                          |
| aggregate command | db read lock  | until $out/$tee is implemented only reads |

Javascript
Only one thread in the mongod process executes Javascript at a time (other database operations are often possible concurrent with this). Note ticket https://jira.mongodb.org/browse/SERVER-4258 will allow multi-threading.

**Group Command**

The group command takes a read lock and does not allow any other threads to execute JavaScript while it is running.

**MapReduce**

The mapreduce operation is composed of many small events: reads from the input collection, executions of map(), executions of reduce(), writes to the output collection, etc.

There is a javascript lock so that only one thread can execute JS code at one point in time. But most JS steps of the MR (e.g. a single map()) are very short and consequently the lock is yielded very often. Note ticket https://jira.mongodb.org/browse/SERVER-4258 will allow multi-threading.

There are also several non-JavaScript operations that MapReduce performs that take locks:

- It reads from a collection (read lock yielded every 100 documents)
- It inserts documents into the temporary collection (write lock for a single write)
- It creates a final collection or merges into or replaces an existing collection (write lock)

The result is that while single-threaded, several MR jobs can be interleaved and appear to run in parallel. Note the problematic lock can be the final write lock during post-processing, which is used to make results appear atomically. This lock can take a long time in “merge” or “reduce” output mode. A flag to disable atomicity has been added as per https://jira.mongodb.org/browse/SERVER-2581 (v2.1.1).

**See Also**


### SQL to Mongo Mapping Chart

<table>
<thead>
<tr>
<th>MySQL executable</th>
<th>Oracle executable</th>
<th>Mongo executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>mysql</td>
<td>oracle</td>
<td>mongod</td>
</tr>
<tr>
<td>mysql</td>
<td>sqlplus</td>
<td>mongo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MySQL term</th>
<th>Mongo term/concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>database</td>
</tr>
<tr>
<td>table</td>
<td>collection</td>
</tr>
<tr>
<td>index</td>
<td>index</td>
</tr>
<tr>
<td>row</td>
<td>BSON document</td>
</tr>
<tr>
<td>column</td>
<td>BSON field</td>
</tr>
<tr>
<td>join</td>
<td>embedding and linking</td>
</tr>
<tr>
<td>primary key</td>
<td>_id field</td>
</tr>
<tr>
<td>group by</td>
<td>aggregation</td>
</tr>
</tbody>
</table>

MongoDB queries are expressed as JSON (BSON) objects. The following chart shows examples as both SQL and in Mongo Query Language syntax.

The query expression in MongoDB (and other things, such as index key patterns) is represented as JSON (BSON). However, the actual verb (e.g. “find”) is done in one’s regular programming language; thus the exact forms of these verbs vary by language. The examples below are Javascript and can be executed from the mongo shell.

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>Mongo Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Query</td>
<td>MongoDB Code</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td>CREATE TABLE USERS (a Number, b Number)</td>
<td>implicit; can also be done explicitly with <code>db.createCollection(&quot;mycoll&quot;)</code></td>
</tr>
<tr>
<td>ALTER TABLE users ADD ...</td>
<td>implicit</td>
</tr>
<tr>
<td>INSERT INTO USERS VALUES(3,5)</td>
<td><code>db.users.insert({a:3,b:5})</code></td>
</tr>
<tr>
<td>SELECT a,b FROM users</td>
<td><code>db.users.find({}, {a:1,b:1})</code></td>
</tr>
<tr>
<td>SELECT * FROM users</td>
<td><code>db.users.find()</code></td>
</tr>
<tr>
<td>SELECT * FROM users WHERE age=33</td>
<td><code>db.users.find({age:33})</code></td>
</tr>
<tr>
<td>SELECT a,b FROM users WHERE age=33</td>
<td><code>db.users.find({age:33}, {a:1,b:1})</code></td>
</tr>
<tr>
<td>SELECT * FROM users WHERE age=33 ORDER BY name</td>
<td><code>db.users.find({age:33}).sort({name:1})</code></td>
</tr>
<tr>
<td>SELECT * FROM users WHERE age&gt;33</td>
<td><code>db.users.find({age:{$gt:33}})</code></td>
</tr>
<tr>
<td>SELECT * FROM users WHERE age!=33</td>
<td><code>db.users.find({age:{$ne:33}})</code></td>
</tr>
<tr>
<td>SELECT * FROM users WHERE name LIKE &quot;%Joe%&quot;</td>
<td><code>db.users.find({name:/Joe/})</code></td>
</tr>
<tr>
<td>SELECT * FROM users WHERE name LIKE &quot;Joe%&quot;</td>
<td><code>db.users.find({name:/^Joe/})</code></td>
</tr>
<tr>
<td>SELECT * FROM users WHERE age&gt;33 AND age&lt;=40</td>
<td><code>db.users.find({&quot;age&quot;:{$gt:33,$lte:40}})</code></td>
</tr>
<tr>
<td>SQL Query</td>
<td>MongoDB Code</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>SELECT * FROM users ORDER BY name DESC</td>
<td><code>db.users.find().sort({name:-1})</code></td>
</tr>
<tr>
<td>SELECT * FROM users WHERE a=1 and b='q'</td>
<td><code>db.users.find({a:1,b:'q'})</code></td>
</tr>
<tr>
<td>SELECT * FROM users LIMIT 10 SKIP 20</td>
<td><code>db.users.find().limit(10).skip(20)</code></td>
</tr>
<tr>
<td>SELECT * FROM users WHERE a=1 or b=2</td>
<td><code>db.users.find( { $or : [ { a : 1 } , { b : 2 } ] } )</code></td>
</tr>
<tr>
<td>SELECT * FROM users LIMIT 1</td>
<td><code>db.users.findOne()</code></td>
</tr>
<tr>
<td>SELECT order_id FROM orders o, order_line_items li WHERE li.order_id=o.order_id AND li.sku=12345</td>
<td><code>db.orders.find({&quot;items.sku&quot;:12345},{_id:1})</code></td>
</tr>
<tr>
<td>SELECT customer.name FROM customers,orders WHERE orders.id=&quot;q179&quot; AND orders.custid=customer.id</td>
<td><code>var o = db.orders.findOne({_id:q179}); var name = db.customers.findOne({_id:o.custid})</code></td>
</tr>
<tr>
<td>SELECT DISTINCT last_name FROM users</td>
<td><code>db.users.distinct('last_name')</code></td>
</tr>
<tr>
<td>SELECT COUNT(*)y FROM users</td>
<td><code>db.users.count()</code></td>
</tr>
<tr>
<td>SELECT COUNT(*)y FROM users WHERE AGE &gt; 30</td>
<td><code>db.users.find({age: {'$gt':30}}).count()</code></td>
</tr>
<tr>
<td>SELECT COUNT(AGE) from users</td>
<td><code>db.users.find({age: {'$exists':true}}).count()</code></td>
</tr>
<tr>
<td>SELECT COUNT(*y) FROM users WHERE AGE &gt; 30</td>
<td><code>db.users.count()</code></td>
</tr>
<tr>
<td>SQL</td>
<td>Mongo Shell</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td><code>CREATE INDEX myindexname ON users(name)</code></td>
<td><code>db.users.ensureIndex({name:1})</code></td>
</tr>
<tr>
<td><code>CREATE INDEX myindexname ON users(name,ts DESC)</code></td>
<td><code>db.users.ensureIndex({name:1,ts:-1})</code></td>
</tr>
<tr>
<td><code>EXPLAIN SELECT * FROM users WHERE z=3</code></td>
<td><code>db.users.find({z:3}).explain()</code></td>
</tr>
<tr>
<td><code>UPDATE users SET a=1 WHERE b='q'</code></td>
<td><code>db.users.update({b:'q'}, {$set:{a:1}}, false, true)</code></td>
</tr>
<tr>
<td><code>UPDATE users SET a=a+2 WHERE b='q'</code></td>
<td><code>db.users.update({b:'q'}, {$inc:{a:2}}, false, true)</code></td>
</tr>
<tr>
<td><code>DELETE FROM users WHERE z='abc'</code></td>
<td><code>db.users.remove({z:'abc'})</code></td>
</tr>
</tbody>
</table>

More examples, specifically aggregation examples, here

**See Also**

- The MongoDB Manual Pages are a good place to learn more.
- SQL to Shell to C++

**SQL to Shell to C++**

MongoDB queries are expressed as JSON (BSON) objects. This quick reference chart shows examples as SQL, Mongo shell syntax, and Mongo C++ driver syntax.

A query expression in MongoDB (and other things, such as an index key pattern) is represented as BSON. In C++ you can use BSONObjBuilder (aka bson::bob) to build BSON objects, or the BSON() macro. The examples below assume a connection c already established:

```cpp
using namespace bson;
DBClientConnection c;
c.connect("somehost");
```

Several of the C++ driver methods throw `mongo::DBException`, so you will want a try/catch statement as some level in your program.
INSERT INTO USERS VALUES(1,1)

```
// GENOID is optional. if not done by client, server will add an _id
c.insert("mydb.users",
   BSON(GENOID<<"a"<<1<<
   "b"<<1));
// then optionally:
   string err =
   c.getLastError();
```

```js
db.users.insert({a:1,b:1})
```
CREATE INDEX myindexname ON users(name)

db.users.ensureIndex({name:1})
c.ensureIndex({
"mydb.users", BSON(
"name"<<1)));

SELECT * FROM users LIMIT 10 SKIP 20
db.users.find().limit(10).skip(20)

auto_ptr<DBClientCursor> cursor = c.query("mydb.users", Query(), 10, 20);

SELECT * FROM users LIMIT 1

db.users.findOne()

bo obj = c.findOne("mydb.users", Query());

SELECT DISTINCT last_name FROM users WHERE x=1

db.users.distinct('last_name',{x:1})

// no helper for distinct yet in c++ driver, so send command manually
bo cmdResult;
bool ok = c.runCommand("mydb", BSON( "distinct" "users" "key" "last_name"
"query" BSON("x"<<1)),
cmdResult);
list<bo> results;
results[0].Obj().Vals(results);

SELECT COUNT(*) FROM users where AGE > 30

db.users.find({age: {'$gt': 30}}).count()

unsigned long long n = c.count("mydb.users", BSON("age"<<GT<<30));

UPDATE users SET a=a+2 WHERE b='q'

db.users.update({b:'q'}, {$inc:{a:2}},
false, true)
c.update("mydb.users", QUERY("b""q"), BSON( "$inc"<BSON("a"<<2)),
false, true);
// then optionally:
string err = c.getLastError();
bool ok = err.empty();
DELETE FROM users WHERE z='abc';

db.users.remove({z:'abc'});

c.remove('mydb.users', QUERY("z"=='abc'));
// then optionally:
string err =
c.getLastError();

See Also

- Several more examples (in shell syntax) are on the SQL to Mongo Mapping Chart page.
- C++ Language Center

Admin Zone

- Journaling
- MongoDB Monitoring Service
- The Database and Caching
- Production Notes
- Replication
- Sharding
- Hosting Center
- Monitoring and Diagnostics
- Backups
- Durability and Repair
- Security and Authentication
- Admin Uls
- Starting and Stopping Mongo
- GridFS Tools
- DBA Operations from the Shell
- 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

- Disabling/Enabling
- Journal Files
- Recovery
- The journal Subdirectory
- Group Commits
- Commit Acknowledgement
- FAQ
  - If I am using replication, can some members use journaling and others not?
  - How's performance?
  - Can I use the journaling feature to perform safe hot backups?
  - 32 bit nuances?
  - When did the --journal option change from --dur?
  - Will the journal replay have problems if entries are incomplete (like the failure happened in the middle of one)?
  - How many times is data written to disk when replication and journaling are both on?

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

**Disabling/Enabling**

In version 1.9.2+, journaling is enabled by default for 64-bit platforms. You can disable journaling with the `mongod --nojournal` command line option. For versions < 1.9.2 or 32-bit platforms, you can enable journaling with the `--journal` command line option.

It is OK to disable journaling after running with journaling by simply shutting down `mongod` cleanly and restarting with `--nojournal`. The reverse is also OK; shutdown cleanly and restart without `--nojournal`.

MongoDB may determine that it is faster to preallocate journal files than to create them as needed. If MongoDB decides to preallocate the files, it will not start listening on port 27017 until this process completes, which can take a few minutes. This means that your applications and the shell will not be able to connect to the database immediately on initial startup. Check the logs to see if MongoDB is busy preallocating. It will print the standard "waiting for connections on port whatever" when it has finished.

**Journal Files**

With journaling enabled, journal files will be created in a `journal/` subdirectory under your chosen db path. These files are write-ahead redo logs. In addition, a last sequence number file, `journal/lsn`, will be created. A clean shutdown removes all files under `journal/`.

The Mongo data files (`database.ns`, `database.0`, `database.1`, ...) have the same format as in previous releases. Thus, the upgrade process is seamless, and a rollback would be seamless too. (If you roll back to a pre v1.7.5 release, try to shut down cleanly first. Regardless, remove the `journal/` directory before starting the pre v1.7.5 version of `mongodb`.)

**Recovery**

On a restart after a crash, journal files in `journal/` will be replayed before the server goes online. This will be indicated in the log output. You do not need to run a `repair`.

**The journal Subdirectory**

You may wish, before starting `mongod` to symlink the `journal/` directory to a dedicated hard drive to speed the frequent (fsynced) sequential writes which occur to the current journal file.

**Group Commits**

MongoDB performs group commits (batch commits) when journaling. This means that a series of operations over many milliseconds are committed all at once. This is done to achieve high performance.

Group commits are performed approximately every 100ms by default. In version 1.9.2+, you can set this interval yourself using the `--journalCommitInterval` command line option. The allowed range is 2 to 300 milliseconds.

**Commit Acknowledgement**

You can wait for group commit acknowledgement with the `getLastError` command. In versions before 1.9.0 using `getLastError + fsync` would do this, in newer versions the `j` option has been specifically created for this purpose.

In version 1.9.2+ the group commit delay is shortened when a commit acknowledgement (`getLastError + j`) is pending; this can be as little as 1/3 of the normal group commit interval.

**FAQ**

*If I am using replication, can some members use journaling and others not?*

Yes.

*How's performance?*

Read performance should be the same. Write performance should be very good but there is some overhead over the non-durable version as the journal files must be written. If you find a case where there is a large difference in performance between running with and without journaling, please let us know so we can tune it.

*Can I use the journaling feature to perform safe hot backups?*

Yes, see Backups with Journaling Enabled.

*32 bit nuances?*

There is extra memory mapped file activity with journaling. This will further constrain the limited db size of 32 bit builds. Thus, for now journaling
by default is disabled on 32 bit systems.

**When did the --journal option change from --dur?**

In 1.8 the option was renamed to --journal, but the old name is still accepted for backwards compatibility; please change to --journal if you are using the old option.

**Will the journal replay have problems if entries are incomplete (like the failure happened in the middle of one)?**

Each journal (group) write is consistent and won't be replayed during recovery unless it is complete.

**How many times is data written to disk when replication and journaling are both on?**

In v1.8, for an insert, four times. The object is written to the main collection, and also the oplog collection (so that is twice). Both of those writes are journaled as a single mini-transaction in the journal file (the files in /data/db/journal). Thus 4 times total.

There is an open item in to reduce this by having the journal be compressed. This will reduce from 4x to probably ~2.5x.

The above applies to collection data and inserts which is the worst case scenario. Index updates are written to the index and the journal, but not the oplog, so they should be 2X today not 4X. Likewise updates with things like $set, $addToSet, $inc, etc. are compactly logged all around so those are generally small.

**See Also**

- Journaling Administration Notes
- getLastError command for controlling writes per operation
- Durability Internals
- Presentations
  - Recorded Webinar on Journaling (April 2011)
  - Journaling and the Storage Engine (May 2011)

**Journaling Administration Notes**

- Journal Files (e.g. journal/j._0)
- Prealloc Files (e.g. journal/prealloc.0)
- serverStatus command
- journalLatencyTest Command

**Journal Files (e.g. journal/j._0)**

Journal files are append-only and are written to the journal/ directory under the dbpath directory (which is /data/db/ by default).

Journal files are named j._0, j._1, etc. When a journal file reached 1GB in size, a new file is created. Old files which are no longer needed are rotated out (automatically deleted). Unless your write bytes/second rate is extremely high, you should have only two or three journal files.

Note: in more recent versions, the journal files are only 128MB apiece when using the --smallfiles command line option.

**Prealloc Files (e.g. journal/prealloc.0)**

mongod will create prealloc files in the journal directory under some circumstances to minimize journal write latency. On some filesystems, appending to a file and making it larger can be slower than writing to a file of a predefined size. mongod checks this at startup and if it finds this to be the case will use preallocated journal files. If found to be helpful, a small pool of prealloc files will be created in the journal directory before startup begins. This is a one time initiation and does not occur with future invocations. Approximately 3GB of files will be preallocated (and truly prewritten, not sparse allocated) - thus in this situation, expect roughly a 3 minute delay on the first startup to preallocate these files.

If you don't want to wait three minutes on startup, you can preallocate the files using another instance of mongod and then move them to your normal dbpath before starting with journaling. For example, if you had an instance of mongod running on port 27017 with a dbpath of /data/db (the defaults), you could preallocate journal files for it with:
$ mkdir ~/tmpDbpath
$ mongod --port 10000 --dbpath ~/tmpDbpath --journal
# startup messages
# .
# .
# .
# wait for prealloc to finish
Thu Mar 17 10:02:52 [initandlisten] preallocating a journal file ~/tmpDbpath/journal/prealloc.0
Thu Mar 17 10:03:03 [initandlisten] preallocating a journal file ~/tmpDbpath/journal/prealloc.1
Thu Mar 17 10:03:14 [initandlisten] preallocating a journal file ~/tmpDbpath/journal/prealloc.2
Thu Mar 17 10:03:25 [initandlisten] flushing directory ~/tmpDbpath/journal
Thu Mar 17 10:03:25 [initandlisten] flushing directory ~/tmpDbpath/journal
Thu Mar 17 10:03:25 [initandlisten] waiting for connections on port 10000
Thu Mar 17 10:03:25 [websvr] web admin interface listening on port 11000
# then Ctrl-C to kill this instance
^C
$ mv ~/tmpDbpath/journal /data/db/
$ # restart mongod on port 27017 with --journal

prealloc files do not contain data, but are rather simply preallocated files that are ready to use that are truly preallocated by the file system (i.e. they are not "sparse"). It is thus safe to remove them, but if you restart mongod with journaling, it will create them again if they are missing.

**serverStatus command**

The `serverStatus` command now includes some statistics regarding journaling.

**journalLatencyTest Command**

You can use the `journalLatencyTest` command to measure how long it takes on your volume to write to the disk (including fsyncing the data) in an append-only fashion.

```plaintext
> use admin
> db.runCommand("journalLatencyTest")
```

You can run this command on an idle system to get a baseline sync time for journaling. In addition, it is safe to run this command on a busy system to see the sync time on a busy system (which may be higher if the journal directory is on the same volume as the data files).

In version 1.9.2+ you can set the group commit interval, using `--journalCommitInterval` command-line option, to between 2 and 300 milliseconds (default is 100ms). The actual interval will be the maximum of this setting and your disk latency as measured above.

`journalLatencyTest` is also a good way to check if your disk drive is buffering writes in its local cache. If the number is very low (e.g., less than 2ms) and the drive is non-ssd, the drive is probably buffering writes. In that case, you will want to enable cache write-through for the device in your operating system. (Unless you have a disk controller card with battery backed ram, then this is a good thing.)

**MongoDB Monitoring Service**

MongoDB Monitoring Service is a free SaaS solution for proactive monitoring of your MongoDB cluster(s). MMS’s web interface features charts, custom dashboards, and automated alerting; and since it runs in the cloud, MMS requires minimal setup and configuration. Within minutes your devops and systems administration teams can manage and optimize your MongoDB deployment, and derive valuable insights from key operational metrics.

**Getting Started**

MMS is free and available to everyone in the MongoDB community. To get started with MMS now, visit the MMS setup page.

- Create an account at [mms.10gen.com](http://mms.10gen.com).
- Download and install the MMS agent on your cluster.
- Within minutes, data will be visible on [mms.10gen.com](http://mms.10gen.com).

**Docs**

Documentation for MMS is available at:

- [https://mms.10gen.com/help/](https://mms.10gen.com/help/)
The Database and Caching

With relational databases, object caching is usually a separate facility (such as memcached), which makes sense as even a RAM page cache hit is a fairly expensive operation with a relational database (joins may be required, and the data must be transformed into an object representation). Further, memcached type solutions are more scaleable than a relational database.

Mongo eliminates the need (in some cases) for a separate object caching layer. Queries that result in file system RAM cache hits are very fast as the object’s representation in the database is very close to its representation in application memory. Also, the MongoDB can scale to (almost) any level and provides an object cache and database integrated together, which is very helpful as there is no risk of retrieving stale data from the cache. In addition, the complex queries a full DBMS provides are also possible.

Production Notes

- Backups
- TCP Port Numbers
- Linux
  - General Unix Notes
  - Linux Kernel Versions
  - Checking Disk IO
  - Checking Network IO
- Storage
  - Solid State Disks (SSDs)
  - RAID
  - Linux File Systems
  - Remote mounts
  - Swap
- What Hardware?
- Tips
- iostat
- MongoDB cluster config best practices
- NFS
- NUMA
- SSD
- Virtualization

Below are some MongoDB production devops recommendations and best practices.

Backups

- Backups Doc Page
- Import Export Tools

TCP Port Numbers

Default TCP port numbers for MongoDB processes:

- Standalone `mongod`: 27017
- `mongos`: 27017
- shard server (`mongod --shardsvr`): 27018
- config server (`mongod --configsvr`): 27019
- `web stats page` for `mongod`: add 1000 to port number (28017, by default)

Firewall rules for these ports and connections are available here.

Linux

General Unix Notes

- Turn off `atime` for the data volume
- Set file descriptor limit and user process limit to 4k+ (see etc/limits and ulimit)
- Do not use large/huge virtual memory pages (the default is small pages which is what you want)
- Use dmesg to see if box is behaving strangely
- Try to disable NUMA in your BIOS. If that is not possible see NUMA
- Ensure that readahead settings for the block devices that store the dbpath are acceptable Readahead
- Minimize clock skew between your hosts by using ntp; linux distros usually include this by default, but check and install the ntpd package if it isn’t already installed

**Linux Kernel Versions**

Some have reported skepticism on behavior of Linux 2.6.33-31 and 2.6.32 kernel, at least on EC2. 2.6.36 is given a thumbs up by the community.

For those running databases on ext4 filesystems, a 2.6.23 kernel is required for efficient filesystem preallocation, 2.6.25 is required for XFS support of the same feature. High filesystem I/O following the allocation of new database files is one symptom of this problem.

**Checking Disk IO**

```
iostat -txm 2
```

If the timestamp feature is not available, `iostat -xm 2`

**Checking Network IO**

- `atop` - Holistic system top
- Munin
- `ethtool eth0` - check network port speed
- `bwm-ng`
- `iptraf`

**Storage**

**Solid State Disks (SSDs)**

- See SSD page

**RAID**

Typically we recommend using RAID-10.

RAID-5 and RAID-6 can be slow, and are not suggested.

RAID-0 will provide good write performance but provides limited availability, and reduced performance on reads (particularly on EBS). It is not suggested.

See also the ec2 page for comments on EBS striping.

**Linux File Systems**

MongoDB uses large files for storing data, and preallocates these. These filesystems seem to work well:

- `ext4 ( kernel version >= 2.6.23 )`
- `xfs ( kernel version >= 2.6.25 )`

In addition to the file systems above you might also want to (explicitly) disable file/directory modification times by using these mount options:

- `noatime` (also enables nodiratime)

We have found `ext3` to be very slow in allocating files (or removing them) as well as access within large files is also poor.

**Remote mounts**

We have found that some versions of NFS perform very poorly and do not recommend using NFS. See the NFS page for more information.

Amazon elastic block store (EBS) seems to work well up to its intrinsic performance characteristics, when configured well.

**Swap**

It is useful for the linux kernel to have swap space to use in emergencies. Because of the way MongoDB memory maps the database files none of this data will ever end up in swap; this means that on a healthy system the swap space will rarely be used on a system only running MongoDB.
Having swap can keep the kernel from killing MongoDB when physical memory limits are reached.

You may also want to look at using something which compresses swap/memory like compcache.

**What Hardware?**

MongoDB tends to run well on virtually all hardware. In fact it was designed specifically with commodity hardware in mind (to facilitate cloud computing); that said it works well on very large servers too. That said if you are about to buy hardware here are a few suggestions:

- Fast CPU clock speed is helpful.
- Many cores helps but does not provide a high level of marginal return, so don't spend money on them. (This is both a consequence of the design of the program and also that memory bandwidth can be a limiter; there isn't necessarily a lot of computation happening inside a database).
- NUMA is not very helpful as memory access is not very localized in a database. Thus non-NUMA is recommended; or configure NUMA as detailed elsewhere in this document.
- RAM is good.
- SSD is good. We have had good results and have seen good price/performance with SATA SSDs; the (typically) more upscale PCI SSDs work fine too.
- Commodity (SATA) spinning drives are often a good option as the speed increase for random I/O for more expensive drives is not that dramatic (only on the order of 2x) – spending that money on SSDs or RAM may be more effective.

**Tips**

- Handling Halted Replication
- Starting and Stopping the Database

**iostat**

On Linux, use the iostat command to check if disk I/O is a bottleneck for your database.

We generally find the form:

```
iostat -xm 2
```

to work well. (Use a number of seconds with iostat, otherwise it will display stats since server boot, which is not very useful.)

Use the mount command to see what device your data/db directory resides on.

**Fields**

- %utl - this is the most useful field for a quick check, it indicates what percent of the time the device/drive is in use. If the number if near 100%, your server may be physical disk I/O bound. (There are some volume situations where this statistic overstates, but most often it is correct.)
- r/s - reads per second.
- w/s - writes per second
- rMB/s - read megabytes per second
- wMB/s - write megabytes per second
- avgreq-sz - average request size. The smaller this number, the more random your IO operations are. This is in sectors : typically sectors are 512 bytes, so multiply by 0.5 to see average request size in kilobytes.

On Windows Server use the performance monitor utility.

**MongoDB cluster config best practices**

**NFS**

We have found that some versions of NFS perform very poorly, or simply don't work, and do not suggest using NFS. (We'd love to hear from you if you are using NFS and what results you are getting, either great or not great.)

**NUMA**

- numacl
- proc settings
- Testing
- References
Linux, NUMA and MongoDB tend not to work well together. If you are running MongoDB on numa hardware, we recommend turning it off (running with an interleave memory policy). Problems will manifest in strange ways, such as massive slow downs for periods of time or high system cpu time.

**numactl**

To turn off NUMA, start mongod with

```
numactl --interleave=all ${MONGODB_HOME}/bin/mongod --config conf/mongodb.conf
```

**proc settings**

```
echo 0 > /proc/sys/vm/zone_reclaim_mode
```


**Testing**

On Linux, `mongod` v2.0+ checks these settings on startup and prints a warning if they do not match the recommendations.

**References**

The MySQL “swap insanity” problem and the effects of the NUMA architecture describes the effects of NUMA on databases. This blog post was aimed at problems NUMA created for MySQL, but the issues are the same. The posting describes the NUMA architecture and goals, and how these are incompatible with the working of databases.

**SSD**

We are not experts on solid state drives, but tried to provide some information here that would be helpful. Comments very welcome.

- **Write Endurance**
  - Reserve some unpartitioned space
  - `smartctl`
- **Speed**
- **Reliability**
- **Random reads vs. random writes**
- **PCI vs. SATA**
- **RAM vs. SSD**
- **FlashCache**
- **OS scheduler**
- **Run mongoperf**
- **Helpful links**

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

**Write Endurance**

Write endurance with solid state drives vary. SLC drives have higher endurance but newer generation MLC (and eMLC) drives are getting better. As an example, the MLC Intel 320 drives specify endurance of 20GB/day of writes for five years. If you are doing small or medium size random reads and writes this is sufficient. The Intel 710 series is the enterprise-class models and have higher endurance.

If you intend to write a full drive’s worth of data writing per day (and every day for a long time), this level of endurance would be insufficient. For large sequential operations (for example very large map/reduces), one could write far more than 20GB/day. Traditional hard drives are quite good at sequential I/O and thus may be better for that use case.

- **Blog post on SSD lifespan**

**Reserve some unpartitioned space**

Some users report good results when leaving 20% of their drives completely unpartitioned. In this situation the drive knows it can use that space as working space. Note formatted but empty space may or may not be available to the drive depending on TRIM support which is often lacking.
**smartctl**

On some devices, "smartctl -A" will show you the Media_Wearout_Indicator.

```
$ sudo smartctl -A /dev/sda | grep Wearout
 233 Media_Wearout_Indicator 0x0032 099 099 000 Old_age Always - 0
```

**Speed**

A paper in ACM Transactions on Storage (Sep2010) listed the following results for measured 4KB peak random direct IO for some popular devices:

<table>
<thead>
<tr>
<th>Device</th>
<th>Read IOPS</th>
<th>Write IOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel X25-E</td>
<td>33,400</td>
<td>3,120</td>
</tr>
<tr>
<td>FusionIO ioDrive</td>
<td>98,800</td>
<td>75,100</td>
</tr>
</tbody>
</table>

Intel's larger drives seem to have higher write IOPS than the smaller ones (up to 23,000 claimed for the 320 series). [More info here.](http://www.facebook.com/note.php?note_id=388112370932)

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

**Reliability**

Some manufacturers specify reliability stats indicating failure rates of approximately 0.6% per year. This is better than traditional drives (2% per year failure rate or higher), but still quite high and thus mirroring will be important. (And of course manufacture specs could be optimistic.)

**Random reads vs. random writes**

Random access I/O is the sweet spot for SSD. Historically random reads on SSD drives have been much faster than random writes. That said, random writes are still an order of magnitude faster than spinning disks.

Recently new drives have released that have much higher random write performance. For example the Intel 320 series, particular the larger capacity drives, has much higher random write performance than the older Intel X25 series drives.

**PCI vs. SATA**

SSD is available both as PCI cards and SATA drives. PCI is oriented towards the high end of products on the market.

Some SATA SSD drives now support 6Gbps sata transfer rates, yet at the time of this writing many controllers shipped with servers are 3Gbps. For random IO oriented applications this is likely sufficient, but worth considering regardless.

**RAM vs. SSD**

Even though SSDs are fast, RAM is still faster. Thus for the highest performance possible, having enough RAM to contain the working set of data from the database is optimal. However, it is common to have a request rate that is easily met by the speed of random IO's with SSDs, and SSD cost per byte is lower than RAM (and persistent too).

A system with less RAM and SSDs will likely outperform a system with more RAM and spinning disks. For example a system with SSD drives and 64GB RAM will often outperform a system with 128GB RAM and spinning disks. (Results will vary by use case of course.)

One helpful characteristic of SSDs is they can facilitate fast "preheat" of RAM on a hardware restart. On a restart a system's RAM file system cache must be repopulated. On a box with 64GB RAM or more, this can take a considerable amount of time – for example six minutes at 100MB/sec, and much longer when the requests are random IO to spinning disks.

**FlashCache**

FlashCache is a write back block cache for Linux. It was created by Facebook. Installation is a bit of work as you have to build and install a kernel module. Sep2011: If you use this please report results in the mongo forum as it's new and everyone will be curious how well it works.


**OS scheduler**

One user reports good results with the noop IO scheduler under certain configurations of their system. As always caution is recommended on nonstandard configurations as such configurations never get as much testing...
Run mongoperf

mongoperf is a disk performance stress utility. It is not part of the mongo database, simply a disk exercising program. We recommend testing your SSD setup with mongoperf. Note that the random writes it are a worst case scenario, and in many cases MongoDB can do writes that are much larger.

Helpful links

- Benchmarks of SSD drives from various manufacturers
  - http://techreport.com/articles.x/20653/5
- Intel SSD Models Comparison (scroll down)
- Intel 710 and 720 series info

Virtualization

Generally MongoDB works very well in virtualized environments, with the exception of OpenVZ.

EC2

Compatible. No special configuration requirements.

VMWare

Some suggest not using overcommit as they may cause issues. Otherwise compatible.

Cloning a VM is possible. For example you might use this to spin up a new virtual host that will be added as a member of a replica set. If Journaling is enabled, the clone snapshot will be consistent. If not using journaling, stop mongod, clone, and then restart.

OpenVZ

Issues have been reported here.

Replication

MongoDB supports asynchronous replication of data between servers for failover and redundancy. Only one server (in the set/shard) is active for writes (the primary, or master) at a given time – this is to allow strong consistent (atomic) operations. One can optionally send read operations to the secondaries when eventual consistency semantics are acceptable.

Two forms of replication are available, Replica Sets and Master-Slave. Use Replica Sets – replica sets are a functional superset of master/slave and are handled by much newer, more robust code.

- Replica Set Documentation
- Replication Documentation Index
- Replication Fundamentals
- Replication Architectures
- Replication Internals
- Replication Tutorials
- Replication Reference

WriteConcern and getlasterror
A client can block until a write operation has been replicated to N servers -- read more here.

With Sharding

```
+----------------+----------------+  ...  
|     Shard1     |     Shard2     |
+----------------+----------------+
  Replicaset     |  Replicaset     |
     Members     |     Members     |
```

Each shard in a shardedMongo cluster consists of a replica set.

Video Presentation

A MongoDB Replication Primer: Replica Sets in Practice - MongoSV December 2011

About the local database

mongod reserves the database local for special functionality. It is special in that its contents are never replicated.

When authentication is used, authenticating against the local database is equivalent to authenticating against the admin database: it gives you permissions across all databases, not just local.

Replica Sets

Replica sets use the following collections in local:

- local.system.replset the replica set's configuration object is stored here. (View via the rs.conf() helper in the shell – or query it directly.)
- local.oplog.rs is a capped collection that is the oplog. You can use the --oplogSize command line parameter to set the size of this collection.
- local.replset.minvalid sometimes contains an object used internally by replica sets to track sync status

Master/Slave Replication

- Master
  - local.oplog.$main the "oplog"
  - local.slaves
- Slave
  - local.sources
- Other
  - local.me
  - local.pair.* (replica pairs, which are deprecated)

Replica Sets

- Overview
- Getting started
- Operation
- Advanced and More
- How-tos
- Troubleshooting
- See Also

Overview

Replica sets are a form of asynchronous master/slave replication, adding automatic failover and automatic recovery of member nodes.

- A replica set consists of two or more nodes that are copies of each other. (i.e.: replicas)
- The replica set automatically elects a primary (master). No one member is intrinsically primary; that is, this is a share-nothing design.
- Drivers (and mongos) can automatically detect when a replica set primary changes and will begin sending writes to the new primary. (The mongos sharding process does this too.)
Replica sets have several common uses:

- Data Redundancy
- Automated Failover / High Availability
- Distributing read load
- Simplify maintenance (compared to "normal" master-slave)
- Disaster recovery

Getting started

- Why use replica sets?
- The basics
- Replica Set Design Concepts
- Upgrading your client code
  - Reading from secondary (slaveOkay)
- How does replication work?
  - Oplog
  - Voting
  - Priorities
- Limitations
- Tutorial: single server replica set (dev-only)

Operation

- Configuration
- Administrative Commands

Advanced and More

- Data center awareness and tagging
- Authentication
- The http admin UI
- Cluster wide commit concepts

How-tos

- Sample Replica Set Config Session (PDF)
- Migrating to replica sets from master/slave replication
- Adding a New Set Member
- Adding an Arbiter
- Forcing a Member to be Primary
- Moving or Replacing a Member (also the same procedure for restoring a member)
- Reconfiguring when members are up

Troubleshooting

- Troubleshooting
- Resyncing a Very Stale Replica Set Member
- Reconfiguring when members are down

See Also

- Practical Replication Video
- Replica Sets Slides
- Webcast Demo of Replica Sets
- Replica set internals

Data Center Awareness

- Examples
  - One primary data center, one disaster recovery site
  - Multi-site with local reads
- Confiming propagation of writes with getLastError
- Replicating from nearby members
- Tagging (version 2.0+)
  - Server X should have a copy.
  - Make n backups
  - Make sure there are at least three copies of the data and it is present on at least two continents.
  - Make sure at least two servers across at least two racks in nyc have it.
- Notes
Examples

One primary data center, one disaster recovery site

Multiple set members can be primary at the main data center. Have a member at a remote site that is never primary (at least, not without human intervention).

```json
{_id: 'myset',
 members: [
  {_id:0, host:'sf1', priority:1 },
  {_id:1, host:'sf2', priority:1 },
  {_id:2, host:'ny1', priority:0 }
 ]
}
```

Multi-site with local reads

The following example shows one set member in each of three data centers. At election time, any healthy update to date node, arbitrarily, can become primary. The others are then secondaries and can service queries locally if the client uses slaveOk mode.

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

Refer to your driver's documentation for more information about read routing.

Confirming propagation of writes with getLastError

Calling `getLastError` (called “write concern" in some drivers) with `w: "majority"` (v2.0+) assures the write reaches a majority of the set before acknowledgement. For example, if you had a three-member replica set, calling `db.runCommand({getLastError : 1, w : "majority"})` would make sure the last write was propagated to at least 2 servers.

Once a write reaches a majority of the set members, the cluster wide commit has occurred (see Replica Set Design Concepts).

Replicating from nearby members

In v2.0+, secondaries automatically sync data from members which are nearby. You can see the latencies that the `mongod` process is observing to its peers in the `replSetGetStatus` command's output. If nearby members are not healthy, more distant members will be used for syncing.

Example output, highlighting new ping time and sync target fields:
Tagging (version 2.0+)

Tagging gives you fine-grained control over where data is written. It is:

- Customizable: you can express your architecture in terms of machines, racks, data centers, PDUs, continents, etc. (in any combination or level that is important to your application).
- Developer/DBA-friendly: developers do not need to know about where servers are or changes in architecture.

Each member of a replica set can be tagged with one or more physical or logical locations, e.g., `{"dc" : "ny", "rack" : "rk1", "ip" : "192.168", "server" : "192.168.4.11"}`. **Modes** can be defined that combine these tags into targets for `getLastError`'s `w` option.

For example, suppose we have 5 servers, A, B, C, D, and E. A and B are in New York, C and D are in San Francisco, and E is in the cloud somewhere.

Our replica set configuration might look like:

```
{  
    _id : "someSet",
    members : [
        {_id : 0, host : "A", tags : {"dc": "ny"}},
        {_id : 1, host : "B", tags : {"dc": "ny"}},
        {_id : 2, host : "C", tags : {"dc": "sf"}},
        {_id : 3, host : "D", tags : {"dc": "sf"}},
        {_id : 4, host : "E", tags : {"dc": "cloud"}}
    ],
    settings : {
        getLastErrorModes : {
            veryImportant : {"dc" : 3},
            sortOfImportant : {"dc" : 2}
        }
    }
}
```

Now, when a developer calls `getLastError`, they can use any of the modes declared to ensure writes are propagated to the desired locations, e.g.:

```
> db.foo.insert({x:1})
> db.runCommand({getLastError : 1, w : "veryImportant"})
```
"veryImportant" makes sure that the write has made it to at least 3 tagged "regions", in this case, "ny", "sf", and "cloud". Once the write has been replicated to these regions, getLastError will return success. (For example, if the write was present on A, D, and E, that would be a success condition).

If we used "sortOfImportant" instead, getLastError would return success once the write had made it to two out of the three possible regions. Thus, A and C having the write or D and E having the write would both be "success." If C and D had the write, getLastError would continue waiting until a server in another region also had the write.

Below are some common examples and how you'd specify tags and w modes for them.

Server X should have a copy.

Suppose you want to be able to specify that your backup server (B) should have a copy of a write. Then you'd use the following tags:

<table>
<thead>
<tr>
<th>Server</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>{&quot;backup&quot; : &quot;B&quot;}</td>
</tr>
</tbody>
</table>

To define a mode for "server B should have a copy," create the mode:

```
backedUp : {"backup" : 1}
```

You want one server with a "backup" tag to have the write.

So, your config would look like:

```json
{
  _id : replSetName,
  members : [
    {
      "_id" : 0,
      "host" : B,
      "tags" : {"backup" : "B"}
    },
    ...
  ],
  settings : {
    getLastErrorModes : {
      backedUp : {backup : 1}
    }
  }
}
```

To use this mode in your application, you'd call getLastError with w set to backedUp:

```
> db.runCommand({getLastError : 1, w : "backedUp"})
```

In the following examples, we will skip the configuration and the usage for brevity. Tags are always added to a member's configuration, modes are always added to getLastErrorModes.

Make n backups

Suppose you have three backup servers (B1, B2, B3) and you want at least two of them to have a copy. Then you'd give each of them a unique "backup" tag:

<table>
<thead>
<tr>
<th>Server</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{&quot;backup&quot; : &quot;B1&quot;}</td>
</tr>
<tr>
<td>B2</td>
<td>{&quot;backup&quot; : &quot;B2&quot;}</td>
</tr>
<tr>
<td>B3</td>
<td>{&quot;backup&quot; : &quot;B3&quot;}</td>
</tr>
</tbody>
</table>

Then you would create the mode:
backedUp : {"backup" : 2}

**Make sure there are at least three copies of the data and it is present on at least two continents.**

All of the rules up until now have only had one condition, but you can include as many and-conditions as you want. Suppose we have the following:

<table>
<thead>
<tr>
<th>Server</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>{&quot;continent&quot; : &quot;nAmerica&quot;, &quot;copies&quot; : &quot;S1&quot;}</td>
</tr>
<tr>
<td>S2</td>
<td>{&quot;continent&quot; : &quot;nAmerica&quot;, &quot;copies&quot; : &quot;S2&quot;}</td>
</tr>
<tr>
<td>S3</td>
<td>{&quot;continent&quot; : &quot;nAmerica&quot;, &quot;copies&quot; : &quot;S3&quot;}</td>
</tr>
<tr>
<td>S4</td>
<td>{&quot;continent&quot; : &quot;Africa&quot;, &quot;copies&quot; : &quot;S4&quot;}</td>
</tr>
<tr>
<td>S5</td>
<td>{&quot;continent&quot; : &quot;Asia&quot;, &quot;copies&quot; : &quot;S5&quot;}</td>
</tr>
</tbody>
</table>

Then create a mode like:

```
level : {copies : 3, continent : 2}
```

Note that modes can contain as many clauses as you need.

**Make sure at least two servers across at least two racks in nyc have it.**

This is a complication of our original example. The key concept here is that not all tags need to be present on all servers. For example, some servers below are tagged with "nyc", others are not.

<table>
<thead>
<tr>
<th>Server</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>{&quot;nycRack&quot; : &quot;rk1&quot;, &quot;nyc&quot; : &quot;S1&quot;}</td>
</tr>
<tr>
<td>S2</td>
<td>{&quot;nycRack&quot; : &quot;rk2&quot;, &quot;nyc&quot; : &quot;S2&quot;}</td>
</tr>
<tr>
<td>S3</td>
<td>{&quot;nycRack&quot; : &quot;rk2&quot;, &quot;nyc&quot; : &quot;S3&quot;}</td>
</tr>
<tr>
<td>S4</td>
<td>{&quot;sfRack&quot; : &quot;rk1&quot;, &quot;sf&quot; : &quot;S4&quot;}</td>
</tr>
<tr>
<td>S5</td>
<td>{&quot;sfRack&quot; : &quot;rk2&quot;, &quot;sf&quot; : &quot;S5&quot;}</td>
</tr>
</tbody>
</table>

Now our rule would look like:

```
customerData : {"nycRack" : 2}
```

**Notes**

The examples above generally use hostnames (e.g., "nyc" : "S1"). This isn't required, it's just and convenient way to specify a server-unique tag. You could just as well use "foo", "bar", "baz" or "1", "2", "3", or any other identifiers.

Do not use "*" or "$" in tags, these characters are reserved for future use.

**Replica Set Sample Configurations**

- Single Data Center
  - Single data center, two servers
- Two Data Centers
- Three Data Centers
- Other tips
- See Also

Below are some example configurations of replica sets with various numbers of servers and numbers of data centers. In a sharded cluster this would be the configuration inside each shard and typically they are the same for all shards (unless using tag-aware sharding).
Single Data Center

Note: the arrows below are illustrative of a particular point-in-time state of the replica set. In most configurations it is possible for different members of the set to be primary over time. The member who is currently Primary is underlined.
Single data center replica set example

```
// 3 member replica set, 1 data center
// 3 copies of the data (replication factor of 3)
{
    _id: <setname>,
    members:
    {
        _id: 0, host: "h0"),
        _id: 1, host: "h1"),
        _id: 2, host: "h2"),
    }
```

...  

Clients write to the Primary, and can read from the primary.
Clients in "slaveOk" mode can read from secondaries.

...  

If the current primary fails, slaveOk reads are still possible from secondaries.

...  

After a short time interval (tens of seconds), if a majority of members are up, a new primary will be elected. At this point automatic failover has completed.
A configuration where one server is smaller than the others

```
// 3 member replica set, 1 data center
// server h2 is a small server, compared to the others, so
// we never want it to be primary.
{
  _id: <setname>,
  members:
  [
    {_id: 0, host: "h0"},
    {_id: 1, host: "h1"},
    {_id: 2, host: "h2", priority: 0},
  ]
}
```

Two Data Centers

Below are some sample configurations using two data centers. Note that three data centers is better than two so that the system can, in the event of a network partition, determine which data center is still "online" and visible to the world (i.e. handle the so-called the "split-brain" problem). The first configuration (recommended) shows "cheats" and uses a VM at a third party provider as an arbiter for this purpose.

---

Single data center, two servers

Replica sets perform consensus elections. If only two servers are available a proper majority can not be formed by a single server. In this case consider manual failover with one member configured to always be primary. For example:

```
{
  _id: <setname>,
  members:
  [
    {_id: 0, host: "h0"},
    {_id: 1, host: "h1", priority: 0, votes: 0}
  ]
}
```
Three Data Centers

One of the examples below uses the `votes` option. In general, do not configure number of votes, it should be used with extreme care and diligence. (If you get it wrong, failover will not work as you expect.)
Other tips

Consider having a server running somewhere with the `slaveDelay` option to have a rolling backup from a certain number of hours in the past. This can be very useful if a “fat finger” type of situation occurs.

See Also

- Sharding
- Tag Aware Sharding
- Reconfiguring a replica set when members are down for information on how to cut over manually to a disaster recovery data center

Replica Sets - Basics

- Basic Configuration
  - Example: 3 full servers
- Getting Started - A sample session
  - Step 1: Start mongod with --replSet
  - Step 1a: Check the Replication UI (optional)
  - Step 2: Initiate the replica set
  - Step 3: Add nodes to the replica set
- Changing Client Code

For production use you will want a minimum of three nodes in the replica set.

Either:

- 2 full nodes and 1 arbiter
- 3 full nodes
To avoid a single point of failure, these nodes must be on different computers.

It is standard to have at least 2 nodes equipped to handle primary duties.

Basic Configuration

- Replica sets typically operate with 2 to 7 full nodes and possibly an arbiter.
- Within a given set, there should be an odd number of total nodes.
  - If full nodes are only available in even numbers, then an arbiter should be added to provide an odd number.
  - The arbiter is a lightweight mongod process whose purpose is to break ties when electing a primary.
  - The arbiter does not typically require a dedicated machine.

Example: 3 full servers

In this example, three servers are connected together in a single cluster. If the primary fails any of the secondary nodes can take over.

### 3-node Replica Set

![Diagram of a 3-node Replica Set]

Getting Started – A sample session

The following is a simple configuration session for replica sets.

For this example assume a 3-node Replica set with 2 full nodes and one arbiter node. These servers will be named sf1, sf2 and sf3.

**Step 1: Start mongod with \(--replSet\)**

On each of the servers start an instance of the mongo daemon:

```
sf1$ mongod --replSet myset
sf2$ mongod --replSet myset
sf3$ mongod --replSet myset
```

The \(--replSet\) parameter has the same value myset on all three instances.

**Step 1a: Check the Replication UI (optional)**

Visit `http://sf1:28017/_replSet`, this will give you an idea on the current status of the replica set. As you proceed through the remaining steps, refreshing this dashboard to see changes. See the docs [here](#) for more details.

**Step 2: Initiate the replica set**

Connect to mongo on sf1.
$ mongo --host sf1
> rs.initiate()
{
  "info2" : "no configuration explicitly specified -- making one",
  "info" : "Config now saved locally. Should come online in about a minute.",
  "ok" : 1
}
>

Initializing the replica set this way will cause the replica set to use the hostname of the current server in the replica set configuration. If your hostnames are not known to all mongo and application servers, you may need to initialize the hosts explicitly - see Replica Set Configuration for more details.

Step 3: Add nodes to the replica set

$ mongo --host sf1
> rs.add("sf2")
{ "ok" : 1 }
> rs.addArb("sf3")
{ "ok" : 1 }

Any operations that change data will now be replicated from sf1 to sf2.

If sf1 is shut down, you will see sf2 take over as primary

Changing Client Code

- To leverage replica sets from your client code, you will need to modify your client connection code.
- The details for doing this will vary with each driver (language).

Replica Sets - Rollbacks

Overview

The classic example of a rollback occurs when you have two replicas (a primary-A and secondary-B) and the secondary (B) is not up to date (replication is behind). If the primary (A) fails (or is shutdown) before B is up-to-date and B becomes primary, then there is data which B does not have but A does.

When this happens, MongoDB cannot automatically merge the old data to the new/current primary, but don't worry, the data is not lost. The rollback operation will ensure that all data is consistent based on the current primary's data.

In order to roll back the data on the original primary (A), the oplog is traversed from the point in time that the server B took over as primary. For each delete or update in the oplog on server A the modified documents are re-fetched from B any new documents after that point are deleted from server A.

This rolled back data, which only A had) is stored in the 'rollback' directory. The data is stored in a BSON file, the name is date-encoded like this foo.bar.2011-05-09T18-10-04.0.bson. In newer versions of mongodb (>2.1) that data may be stored in the local database as well so that restoring it will be easier.

TODO: Add diagram.

Rolled back data

- To view the contents of this file in human-readable format, use the bsondump utility.
- To restore this information to the DB, use the mongorestore utility.

One common strategy for reconciling these conflicts is restore the rollback data to a new collection. Then reconcile between that rollback collection and the master with a custom program/script.

Rollback Limitations

MongoDB will not rollback more than 300MB of data. In this situation, the rollback will halt with the log message: [replica set sync] replSet syncThread: 13410 replSet too much data to roll back
To recover data from the member that has failed to rollback, you will need to manually intervene. If you do not care about this data, remove the data directory and fully resync (or restore from a backup) to resume normal operation.

**Why Replica Sets**

Replica sets provide five distinct benefits over the use of a single node.

A system requiring one or more of the following features should consider using replica sets.

- **Data Redundancy**
- **Automated Failover**
- **Read Scaling**
- **Maintenance**
- **Disaster Recovery**

**Data Redundancy**

- Replica sets provide an automated method for storing multiple copies of your data.
- Supported drivers allow for the control of "write concerns". This allows for writes to be confirmed by multiple nodes before returning a success message to the client.

**Automated Failover**

- Replica sets will coordinate to have a single primary in a given set.
- Supported drivers will recognize the change of a primary within a replica set.
  - In most cases, this means that the failure of a primary can be handled by the client without any configuration changes.
- A correctly configured replica set basically provides a "hot backup". Recovering from backups is typically very time consuming and can result in data loss. Having an active replica set is generally much faster than working with backups.

**Read Scaling**

- The primary means to scale in MongoDB is sharding. Replicas are foremost for data safety, high availability, and redundancy. That said, it is possible to perform queries on secondaries.
- By default, the primary node of a replica set is accessed for all reads and writes.
- Most drivers provide a `slaveOkay` method for identifying that a specific operation can be run on a secondary node. When using `slaveOkay`, a system can share the read load amongst several nodes.
- Unless there is an explicit need, we recommend simply querying the primaries. This keeps things simple and assures immediate consistency for the data read.

**Maintenance**

- When performing tasks such as upgrades, backups and compaction, it is typically required to remove a node from service.
- Replica sets allow for these maintenance tasks to be performed while operating a production system. As long as the production system can withstand the removal of a single node, then it’s possible to perform a “rolling” upgrade on such things.

**Disaster Recovery**

- Replica sets allows for a “delayed secondary” node.
- This node can provide a window for recovering from disastrous events such as:
  - bad deployments
  - dropped tables and collections

Next : Setup Basics

**Replica Set Versions and Compatibility**

**Features**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave delay</td>
<td>v1.6.3+</td>
</tr>
<tr>
<td>Hidden</td>
<td>v1.7+</td>
</tr>
<tr>
<td>replSetFreeze and replSetStepDown</td>
<td>v1.7.3+</td>
</tr>
<tr>
<td>Replicated ops in mongostat</td>
<td>v1.7.3+</td>
</tr>
<tr>
<td>Syncing from secondaries</td>
<td>v1.8.0</td>
</tr>
</tbody>
</table>
Syncing

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

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

See the upgrade documentation for advice on upgrading.

Replica Set Authentication

v1.8+

Replica set authentication works a little differently from single-server authentication, so that each member can automatically authenticate itself to the other members of the set. See the main docs on authentication for details.

Example

If we had a two-member replica set with members a and b, we could start them up with authentication enabled by running:

```bash
a$ echo "this is my super secret key" > mykey
a$ chmod 600 mykey
a$ mongod --keyFile mykey # other options...
```

```bash
b$ echo "this is my super secret key" > mykey
b$ chmod 600 mykey
b$ mongod --keyFile mykey # other options...
```

Then run `rs.initiate()` and so on.

Using the Database with Replica Set Authentication On

From the client's perspective, authentication works the same way with replica sets as it does with single servers.

For example, suppose you create a new replica set and start the members with `--keyFile`. Connect to the master locally to add users:

```bash
master$ mongo
MongoDB shell version: x.y.z
connecting to: test
> db.addUser("foo", "bar")
```

Clients should authenticate as usual when they make connections.

```bash
any-member$ mongo -u foo -p
MongoDB shell version: x.y.z
Enter password: <bar>
```

Upgrading to Replica Sets

- Upgrading From a Single Server
- Upgrading From Replica Pairs or Master/Slave
  - Resyncing the Slaves
  - Adding An 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();
```

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.

### Sharded Cluster

If you are upgrading the member(s) from a sharded cluster then you must change the shard host information in the config database through a mongos instance connection:

```
db.getSiblingDB("config").shards.save({_id:"<name>", host:"<rsName>/member1,member2,..."})
```

### 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.
Now, start up the master with the `--replSet` option, and initialize a one-member replica set.

```bash
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).

### Resynching the Slaves

To resync, clear the data directory:

```bash
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.

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

In the database shell, add the slave as a new member in the replica set.

```bash
m> // still in the mongo shell on the master
m> rs.add("s") // "s" is your slave host name
m> rs.status(); // see also http://localhost:28017/_replSet
```

### Adding An Arbiter

If there are an even number of replica set members, we should add an arbiter to break ties on elections and know who is up in a network partition. An arbiter is very lightweight and can run on virtually any server (including 32 bit servers). We use different directories and ports here so that the server is still available as a "normal" mongod server if that is desired and also to avoid confusion. The `/data/arb` directory will be very small in content size.

```bash
arb$ mkdir /data/arb
arb$ mongod --rest --replSet mysetname --dbpath /data/arb --port 30000
```

Then add the arbiter to your replica set:

```bash
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.

**See Also**

- **Http Interface**

**Replica Set Commands**

- **Shell Helpers**
- **Commands**
  - `isMaster`
  - `replSetGetStatus`
    - `state`
    - `health`
    - `pingMs`
    - `optime`, `optimeDate`
    - `errmsg`
  - `replSetInitiate`
  - `replSetReconfig`
    - `force` option
  - `replSetStepDown`
  - `replSetFreeze`

**Shell Helpers**

```
rs.help()          show help
rs.status()        { replSetGetStatus : 1 }
rs.initiate()      { replSetInitiate : null } initiate
                   with default settings
rs.initiate(cfg)   { replSetInitiate : cfg }
rs.add(hostportstr) add a new member to the set
rs.add(membercfgobj) add a new member to the set
rs.addArb(hostportstr) add a new member which is arbiterOnly: true
rs.remove(hostportstr) remove a member (primary, secondary, or arbiter) from the set
rs.stepDown()      { replSetStepDown : true }
rs.conf()          return configuration from local.system.replset
db.isMaster()      check who is primary
```
rs.conf() can be particularly useful when using commands that manipulate a replica set, such as replSetReconfig, because it can be used to get a copy of the current configuration which can be modified and then put back. For an example of this, see Reconfiguring when Members are Up.

**Commands**

**isMaster**

Checks if the node to which we are connecting is currently primary. Most drivers do this check automatically and then send queries to the current primary.

Returns an object that looks like:

```javascript
> db.adminCommand( { isMaster : 1 } )
{
    "setName" : "florbble",
    "ismaster" : false,
    "secondary" : true,
    "hosts" : [
        "sf1.example.com",
        "sf4.example.com",
        "ny3.example.com"
    ],
    "passives" : [
        "sf3.example.com",
        "sf2.example.com",
        "ny2.example.com"
    ],
    "arbiters" : [
        "ny1.example.com"
    ],
    "primary" : "sf4.example.com",
    "me" : "ny3.example.com",
    "maxBsonObjectSize" : 16777216,
    "ok" : 1
}
```

The **hosts** array lists primary and secondary servers, the **passives** array lists passive servers, and the **arbiters** array lists arbiters.

If the "ismaster" field is false, there will be a "primary" field that indicates which server is primary.

**replSetGetStatus**

Get status on the replica set from this node's point of view. rs.status() is the mongo shell helper for this command. The output looks like:
> db.adminCommand( { replSetGetStatus : 1 } )
{
  "set" : "Foo",
  "date" : ISODate("2012-01-06T00:30:47Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "localhost:27018",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "optime" : {
        "t" : 1325809572000,
        "i" : 1
      },
      "optimeDate" : ISODate("2012-01-06T00:26:12Z"),
      "self" : true
    },
    {
      "_id" : 1,
      "name" : "localhost:27019",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 274,
      "optime" : {
        "t" : 1325809572000,
        "i" : 1
      },
      "optimeDate" : ISODate("2012-01-06T00:26:12Z"),
      "lastHeartbeat" : ISODate("2012-01-06T00:30:47Z"),
      "pingMs" : 0
    },
    {
      "_id" : 2,
      "name" : "localhost:27020",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 274,
      "optime" : {
        "t" : 1325809572000,
        "i" : 1
      },
      "optimeDate" : ISODate("2012-01-06T00:26:12Z"),
      "lastHeartbeat" : ISODate("2012-01-06T00:30:47Z"),
      "pingMs" : 1
    }
  ],
  "ok" : 1
}

state

The state, myState, and stateStr fields indicates the state of this server. Possible state values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Starting up, phase 1 (parsing configuration)</td>
</tr>
<tr>
<td>1</td>
<td>Primary</td>
</tr>
<tr>
<td>2</td>
<td>Secondary</td>
</tr>
<tr>
<td>3</td>
<td>Recovering (initial syncing, post-rollback, stale members)</td>
</tr>
<tr>
<td>4</td>
<td>Fatal error</td>
</tr>
<tr>
<td>5</td>
<td>Starting up, phase 2 (forking threads)</td>
</tr>
</tbody>
</table>
health

The `health` field indicates the health of the server/member. Typical values are zero if a server is down (from this server's POV) and 1 if it is up.

pingMs

The `pingMs` field (v2.0+) reports how long heartbeat commands take from this member to the other member. When members are performing well the pingMs value should approach the round trip ping times of the network between the nodes. If a server is overloaded, this value could be significantly larger (which would indicate a problem). Also if the network has problems this value will intermittently report a high value. The metric is an ongoing average of the last several ping requests (representative of average ping time for roughly say, the last minute).

optime, optimeDate

The `optime`, `optimeDate` fields specify the timestamp of the last operation on the node. This value could be a bit behind realtime as heartbeat requests between servers only occur every few seconds. Thus the real replication lag might be very low even if this statistic shows a few seconds.

errmsg

The `errmsg` field contains informational messages, as shown above.

### replSetInitiate

```javascript
> db.adminCommand( { replSetInitiate : <config> } )
```

Initiate a replica set. Run this command at one node only, to initiate the set. Whatever data is on the initiating node becomes the initial data for the set. This is a one time operation done at cluster creation. `rs.initiate(<cfg>)` is the mongo shell helper for this command. See also Configuration.

### replSetReconfig

Adjust configuration of a replica set (just like initialize).

```javascript
db._adminCommand({replSetReconfig: cfg })
```

**Note:** `db._adminCommand` is short-hand for `db.getSisterDB("admin").runCommand();`

**Note:** as of v1.7.2, `replSetReconfig` closes all connections, meaning there will be no database response to this command.

#### force option

As of 2.0 you can force a reconfigure when you have less than a majority of the replica set up using the `{force:true}` option.

### replSetStepDown

```javascript
db.adminCommand( { replSetStepDown : <seconds> } )
```

Manually tell a member to step down as primary. Node will become eligible to be primary again after the specified number of seconds. (Presumably, another node will take over by then if it were eligible.)

If the primary cannot see anyone who has synced to within 10 seconds of its latest op, the primary will reject the step down request. You can force the request by passing a `force : true` option.

For example:
v1.7.2+: replSetStepDown closes all connections, meaning there will be no database response to this command.

v1.7.3+: the seconds parameter above can be specified. In older versions, the step down was always for one minute only.

v1.9.0+: Added ability for primary to reject step down and added force option.

replSetFreeze

```javascript
db.adminCommand( { replSetFreeze : <seconds> } )
```

v1.7.3+

‘Freeze’ state of this member to the extent we can do that. What this really means is that this node will not attempt to become primary until the time period specified expires.

You can call again with {replSetFreeze:0} to unfreeze sooner. A process restart unfreezes the member also.

If the node is already primary, you need to use replSetStepdown instead.

**Forcing a Member to be Primary**

Replica sets automatically negotiate which member of the set is primary and which are secondaries. If you want a certain member to be primary, there are a couple ways to force this.

v2.0+

In v2.0+, you can set the priority of the preferred primary to be higher than the priorities of the other nodes.

For example, if we had members A, B, and C and A is the current primary and we want B to be primary, we could give B a higher priority like so:

```javascript
> config = rs.conf()
{ 
  "_id" : "foo",
  "version" : 1,
  "members" : [ 
    { 
      "_id" : 0,
      "host" : "A",
    },
    { 
      "_id" : 1,
      "host" : "B",
    },
    { 
      "_id" : 2,
      "host" : "C",
    }
  ]
} 
> config.version++ 
> // the default priority is 1 
> config.members[1].priority = 2 
> rs.reconfig(config)
```
Assuming B is synced to within 10 seconds of A, A will step down, B (and C) will catch up to where A is, and B will be elected primary.

If B is far behind A, A will not step down until B is within 10 seconds of its optime. This minimizes the amount of time there will be no primary on failover. If you do not care about how long the set is primary-less, you can force A to step down by running:

```javascript
> db.adminCommand({replSetStepDown:1000000, force:1})
```

This prevents A from being primary for 1000000 seconds, even if there is no other member that can become primary. B will sync until it is caught up with A and then become primary.

If you change your mind, you can make A start seeking election again by running:

```javascript
> db.adminCommand({replSetFreeze:0})
```

**Older versions**

If you want to force a node to be primary at a given point in time, use the replSetFreeze and replSetStepdown commands (v1.8+). If we have members A, B, and C, and A is current primary, and we want B to become primary, we would send freeze to C so that it does not attempt to become primary, and then stepDown to A.

See the Commands page for more information.

```javascript
$ mongo --host C
> // first check that everyone is healthy and in the states we expect:
> rs.status()
> // C : not eligible to be primary for 120 seconds
> rs.freeze(120)
> exit

$ mongo --host A
> // A : step down as primary and ineligible to be primary for 120 seconds
> rs.stepDown(120)
> // B will now become primary. for this to work B must be up to date.
```

Note that during transitions of primary, there is a short window when no node is primary.

**Command replSetStepDown**

```javascript
> db.adminCommand( { replSetStepDown : <seconds> } )
```

Step down as primary. Will not try to reelect self for the specified time period (1 minute if no numeric secs value specified). (If another member with same priority takes over in the meantime, that member will stay primary.)

Note: seconds parameter is new to v1.8. Old versions default to 60 seconds regardless of param value.

**Command replSetFreeze**

v1.8+

```javascript
> db.adminCommand( { replSetFreeze : <seconds> } )
```

'freeze' state of member to the extent we can do that. What this really means is that this node will not attempt to become primary until the time period specified expires. You can call again with {replSetFreeze:0} to unfreeze sooner. A process restart unfreezes the member also.

**Connecting to Replica Sets from Clients**

Most drivers have been updated to provide ways to connect to a replica set. In general, this is very similar to how the drivers support connecting to a replica pair.
Instead of taking a pair of hostnames, the drivers will typically take a comma separated list of host[:port] names. This is a seed host list; it need not be every member of the set. The driver then looks for the primary from the seeds. The seed members will report back other members of the set that the client is not aware of yet. Thus we can add members to a replica set without changing client code.

With Sharding

With sharding, the client connects to a `mongo` process. The `mongo` process will then automatically find the right member(s) of the set.

See Also

- Driver authors should review Connecting Drivers to 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()
```

Replication halts with "objects in a capped ns cannot grow" (assertion 10003)

Generally this happens if you have a capped collection without an `_id` index and you are using a custom `_id`.

To fix, make sure that any capped collections you are using have a unique index on the `_id` field and resync the halted slave.

```
"couldn't initiate : can't find self in the replset config my port: 27017" under Mac OS X
```

Generally this happens because your hostname and computer name do not match. The safest solution is to supply a config object to `rs.initiate`. Another solution is to:

Open System Preferences, select the Sharing page, and set your Computer Name at the top. Then open Terminal and run

```
$ sudo hostname (name chosen above including the .local at the end)
```

It will prompt you for your password. Once the command has finished, `rs.initiate` should work.

"not electing self, not all members up and we have been up less than 5 minutes"

The idea here is that if a bunch of nodes bounce all at once, we don't want to drop data if we don't have to – we'd rather be offline and wait a little longer instead. Once a node has been up for five minutes, it is eligible to be primary as long as it can achieve a majority. In addition, if all members are up, a member can become primary immediately.

Reconfiguring a replica set when members are down

One can easily reconfigure a replica set when some members are down as long as a majority is still up. In that case, simply send the reconfig command to the current primary.

If there is no majority (i.e. a majority of nodes are not up or reachable and will not be anytime soon), you must manually intervene to change the set configuration. Reconfiguring a minority partition can be dangerous, as two sides of a network partition won't both be aware of the reconfiguration. Thus, this should not be done in scripts, but only by an operator after careful consideration.

In v2.0+, the `force : true` option was added for reconfiguring the set when a majority is down. `force` is for manually reconfiguring the set when there is a serious problem, such as a disaster recovery failover. Do not use it every time you reconfigure, or put it into a script that automatically runs, or use it when there is still a primary.

If you use `force` and look at your replica set config, you will notice that your version number has jumped a lot (by tens or hundreds of thousands). This is normal and designed to prevent set version collisions if network partitioning ends.

Suggestions on how to deal with majority outages are outlined below.
In newer versions we are able to recover when a majority of nodes are down or on the wrong side of a network partition. We will connect to a surviving member (any one is fine) and "force" a reconfiguration of the set without the down hosts.

1. Consider doing a backup on a surviving member, both for safety and to bring up more new set members more easily.
2. Save the current config somewhere so that you can switch back to the old configuration once the down/unreachable members are healed.

```plaintext
> config = rs.config()
> printjson(config) # store this somewhere
```

3. Remove the down members from the `config.members` array.

```plaintext
> // remove the unreachable members of the set. this is just an example, look
> // at config to determine which members to keep
> config.members = [config.members[1], config.members[3], config.members[4]]
```

4. Reconfigure the set using the `force` option on the surviving member.

```plaintext
> rs.reconfig(config, {force : true})
```

You should now have a new primary, most likely the node you are connected to.

If the failure or partition was only temporary, when healed the nodes will detect that they have been removed from the set and enter a special state where they are up but refuse to answer requests as they are no longer syncing changes. You can now re-add them to the config object and do a reconfig (without `force`). **Be sure that each host has the same _id it had before.** If it does not, it will not become an active member of the set and complain verbosely in the logs until you restart it (at which point it will be fine).

Once you add the removed hosts back into the set, they will detect that they have been added and synchronize to the current state of the set. Be aware that if the original master was one of the removed nodes, these members may need to be aware that if the original master was one of the removed nodes, these members may need to [rollback].

**Pre v2.0:**

Versions before 1.9.1 do not support the `force` flag to reconfig. This means that we cannot cleanly reconfigure an existing set to remove down nodes unless we can reach the primary, even if no primary exists. Instead, we need to abandon the old replica set. There are two options for moving forward and using the data on surviving nodes.

**Option 1:** Turn off replication

One option is to make a surviving mongod a standalone server and not a set member:

1. stop a surviving mongod
2. consider doing a backup...
3. delete the `local.*` datafiles in the data directory. This will prevent potential future confusion if it is ever restarted with `--replSet` in the future.
4. restart `mongod` without the `--replSet` parameter.

We are now back online with a single node that is not a replica set member. Clients can use it for both reads and writes.

**Option 2:** "Break the mirror"

This option will use the "break the mirror" technique. One of the surviving members will be selected to be the new master and be the "seed" for a new replica set. Unfortunately all slaves will need to completely resync from this new master.

1. stop the surviving `mongod`
2. consider doing a backup...
3. delete the `local.*` datafiles on the new master
4. delete (ideally just move to a backup location) all the datafiles from any other surviving members
5. restart all `mongod` with the new replica set name either on the command line or in a config file
6. initiate this new set on the new master
7. then, add each of the slaves to this new master

**See Also**

- Reconfiguring when Members are Up
Resyncing a Very Stale Replica Set Member

Error RS102

MongoDB writes operations to an oplog. For replica sets this data is stored in collection local.oplog.rs. This is a capped collection and wraps when full "RRD"-style. Thus, it is important that the oplog collection is large enough to buffer a good amount of writes when some members of a replica set are down. If too many writes occur, the down nodes, when they resume, cannot catch up. In that case, a full resync would be required.

In v1.8+, you can run db.printReplicationInfo() to see the status of the oplog on both the current primary and the overly stale member. This should show you their times, and if their logs have an overlapping time range. If the time ranges don't overlap, there is no way for the stale secondary to recover and catch up (except for a full resync).

There is also a MMS graph of the oplog time length.

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:

```
> db.printReplicationInfo()
configured oplog size:   47.6837158203125MB
log length start to end: 132secs (0.04hrs)
oplog first event time:  Wed Apr 13 2011 02:58:08 GMT-0400
oplog last event time:   Wed Apr 13 2011 03:00:20 GMT-0400
now:                     Wed Apr 13 2011 14:09:08 GMT-0400
```

What to do on a RS102 sync error

If one of your members has been offline and is now too far behind to catch up, you will need to resync. There are a number of ways to do this.

- **Perform a full resync**: If you stop the failed mongod, delete all data in the dbpath (including subdirectories), and restart it, it will automatically resynchronize itself. Obviously it would be better/safer to back up the data first. If disk space is adequate, simply move it to a backup location on the machine if appropriate. Resyncing may take a long time if the database is huge or the network slow – even idealized one terabyte of data would require three hours to transmit over gigabit ethernet.

  or

- **Copy data from another member**: You can copy all the data files from another member of the set IF you have a snapshot of that member's data file's. This can be done in a number of ways. The simplest is to stop mongod on the source member, copy all its files, and then restart mongod on both nodes. The Mongo fsync and lock feature is another way to achieve this if you are using EBS or a SAN. On a slow network, snapshotting all the datafiles from another (inactive) member to a gziped tarball is a good solution. Also similar strategies work well when using SANs and services such as Amazon Elastic Block Service snapshots.

  or

- **Find a member with older data**: *Note: this is only possible (and occurs automatically) in v1.8+. If another member of the replica set has a large enough oplog or is far enough behind that the stale member can sync from it, the stale member can bootstrap itself from this member.

See Also

- Adding a New Set Member
- Moving or Replacing a Member

Replica Sets Limits

- A set can contain
  - A maximum of 12 members
  - A maximum of 7 members that can vote

- Typically the set configuration can be changed only when a majority can be established. Limits on config changes to sets at first.
Especially when a lot of set members are down.

v1.6

- Authentication is supported only in v1.8+.
- Map/reduce writes new collections to the server. Because of this, for now it may only be used on the primary. This will be enhanced later.

Replica Set Internals

This page contains notes on the original MongoDB replica set design. While the concepts still apply, this page is not kept perfectly up-to-date; consider this page historical rather than definitive.

- Design Concepts
- 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
- Idempotence
- Future Versions
- See Also

Design Concepts

Check out the Replica Set Design Concepts for some of the core concepts underlying MongoDB Replica Sets.

Configuration

Command Line

We specify --replSet set_name/seed_hostname_list on the command line. seed_hostname_list is a (partial) list of some members of the set. The system then fetches full configuration information from the collection local.system.replset. set_name is specified to help the system catch misconfigurations. In current versions of MongoDB (1.8+) seed_hostname_list is not required; --replSet set_name will suffice.

Node Types

Conceptually, we have some different types of nodes:

- Standard - a standard node as described above. Can transition to and from being a primary or a secondary over time. There is only one primary (master) server at any point in time.
- Passive - a server can participate as if it were a member of the replica set, but be specified to never be Primary.
- Arbiter - member of the cluster for consensus purposes, but receives no data. Arbiters cannot be seed hosts.

Each node in the set has a priority setting. On a resync (see below), the rule is: choose as master the node with highest priority that is healthy. If multiple nodes have the same priority, pick the node with the freshest data. For example, we might use 1.0 priority for Normal members, 0.0 for passive (0 indicates cannot be primary no matter what), and 0.5 for a server in a less desirable data center.

local.system.replset

This collection has one document storing the replica set's configuration. See the configuration page for details.

Set Initiation (Initial Setup)
For a new cluster, on negotiation the max OpOrdinal is zero everywhere. We then know we have a new replica set with no data yet. A special command

```json
{replSetInitiate:1}
```

is sent to a (single) server to begin things.

**Design**

**Server States**

- **Primary** - Can be thought of as "master" although which server is primary can vary over time. Only 1 server is primary at a given point in time.
- **Secondary** - Can be thought of as a slave in the cluster; varies over time.
- **Recovering** - getting back in sync before entering Secondary mode.

**Applying Operations**

Secondaries apply operations from the Primary. Each applied operation is also written to the secondary's local oplog. We need only apply from the current primary (and be prepared to switch if that changes).

**OpOrdinal**

We use a monotonically increasing ordinal to represent each operation.

These values appear in the oplog (local.oplog.$main). maxLocalOpOrdinal() returns the largest value logged. This value represents how up-to-date we are. The first operation is logged with ordinal 1.

Note two servers in the set could in theory generate different operations with the same ordinal under some race conditions. Thus for full uniqueness we must look at the combination of server id and op ordinal.

**Picking Primary**

We use a consensus protocol to pick a primary. Exact details will be spared here but that basic process is:

1. get maxLocalOpOrdinal from each server.
2. if a majority of servers are not up (from this server's POV), remain in Secondary mode and stop.
3. if the last op time seems very old, stop and await human intervention.
4. else, using a consensus protocol, pick the server with the highest maxLocalOpOrdinal as the Primary.

Any server in the replica set, when it fails to reach master, attempts a new election process.

**Heartbeat Monitoring**

All nodes monitor all other nodes in the set via heartbeats. If the current primary cannot see half of the nodes in the set (including itself), it will fall back to secondary mode. This monitoring is a way to check for network partitions. Otherwise in a network partition, a server might think it is still primary when it is not.

Heartbeats requests are sent out every couple of seconds and can either receive a response, get an error, or time out (after ~20 seconds).

**Assumption of Primary**

When a server becomes primary, we assume it has the latest data. Any data newer than the new primary’s will be discarded. Any discarded data is backed up to a flat file as raw BSON, to allow for the possibility of manual recovery (see this case for some details). In general, manual recovery will not be needed - if data must be guaranteed to be committed it should be written to a majority of the nodes in the set.

**Failover**

We renegotiate when the primary is unavailable, see Picking Primary.

**Resync (Connecting to a New Primary)**

When a secondary connects to a new primary, it must resynchronize its position. It is possible the secondary has operations that were never committed at the primary. In this case, we roll those operations back. Additionally we may have new operations from a previous primary that never replicated elsewhere. The method is basically:

- for each operation in our oplog that does not exist at the primary, (1) remove from oplog and (2) resync the document in question by a query to the primary for that object. update the object, deleting if it does not exist at the primary.

We can work our way back in time until we find a few operations that are consistent with the new primary, and then stop.
Any data that is removed during the rollback is stored offline (see Assumption of Primary, so one can manually recover it. It can't be done automatically because there may be conflicts.

Reminder: you can use $w=$ to ensure writes make it to a majority of slaves before returning to the user, to ensure no writes need to be rolled back.

**Consensus**

Fancier methods would converge faster but the current method is a good baseline. Typically only ~2 nodes will be jockeying for primary status at any given time so there isn't be much contention:

- query all others for their maxappliedoptime
- try to elect self if we have the highest time and can see a majority of nodes
  - if a tie on highest time, delay a short random amount first
  - elect (selfid,maxoptime) msg -> others
- if we get a msg and our time is higher, we send back NO
- we must get back a majority of YES
- if a YES is sent, we respond NO to all others for 1 minute. Electing ourself counts as a YES.
- repeat as necessary after a random sleep

**Increasing Durability**

We can trade off durability versus availability in a replica set. When a primary fails, a secondary will assume primary status with whatever data it has. Thus, we have some desire to see that things replicate quickly. Durability is guaranteed once a majority of servers in the replica set have an operation.

To improve durability clients can call getlasterror and wait for acknowledgement until replication of 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,a4,a5)
server-b: secondary oplog: (a1,a2,a3,c4)
server-c: primary oplog: (a1,a2,a3,c4)
...
server-a: secondary oplog: (a1,a2,a3,c4,c5,c6,c7,c8)
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

Idempotence
Replica set oplog operations are idempotent and need to be for a few cases:
- initial sync
- post-rollback catch-up
- sharding chunk migrations?

Future Versions
- add support for replication trees / hierarchies
- replicating to a slave that is not a member of the set (perhaps we do not need this given we have the Passive set member type)

See Also
- About the local database

Master Slave

Use Replica Sets rather than this – replica sets are a functional superset of master/slave, and newer, more robust code.

- Configuration and Setup
- 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
  - Slave chaining
  - 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 try this setup on a single node, they are supplied in the interest of safety.
As a result, the master server process will create a \texttt{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 \texttt{local.sources} collection. Instead of specifying the \texttt{--source} parameter, one can add an object to \texttt{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> } );
```

- \texttt{host: masterhostname} is the IP address or FQDN of the master database machine. Append \texttt{:port} to the server hostname if you wish to run on a nonstandard port number.
- \texttt{only: databasename} (optional) if specified, indicates that only the specified database should replicate. NOTE: A bug with \texttt{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 \texttt{resync:1} command. Alternatively, the command line option \texttt{--autoresync} causes a slave to restart replication automatically (after ten second pause) if it becomes out of sync. If the \texttt{--autoresync} option is specified, the slave will not attempt an automatic resync more than once in a ten minute period.

The \texttt{--oplogSize} command line option may be specified (along with \texttt{--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 \texttt{--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

- \texttt{--master} master mode
- \texttt{--oplogSize arg} size limit (in MB) for op log

#### Slave

- \texttt{--slave} slave mode
- \texttt{--source arg} arg specifies master as \texttt{<server:port>}
- \texttt{--only arg} arg specifies a single database to replicate
- \texttt{--slavedelay arg} arg specifies delay (in seconds) to be used when applying master ops to slave
- \texttt{--autoresync} automatically resync if slave data is stale

```
$ bin/mongod --slave --source <masterhostname>[:<port>] [--dbpath /data/masterdb/]
```

#### Diagnostics

Check master status from the \texttt{mongo} shell with:
// inspects contents of local.oplog.$main on master and reports status:
// db.printReplicationInfo()

Check slave status from the mongo shell with:

// inspects contents of local.sources on the slave and reports status:
// db.printSlaveReplicationInfo()

(Note you can evaluate the above functions without the parenthesis above to see their javascript source and a bit on the internals.)

As of 1.3.2, you can do this on the slave

```
db._adminCommand( { serverStatus : 1 , repl : N } )
```

$N$ is the level of diagnostic information and can have the following values:
- 0: none
- 1: local (doesn't have to connect to other server)
- 2: remote (has to check with the master)

Security

When security is enabled, one must configure a user account for the local database that exists on both servers.

The slave-side of a replication connection first looks for a user repl in local.system.users. If present, that user is used to authenticate against the local database on the source side of the connection. If repl user does not exist, the first user object in local.system.users is tried.

The local database works like the admin database: an account for local has access to the entire server.

Example security configuration when security is enabled:

```
$ mongo <slavehostname>/admin -u <existingadminusername> -p<adminpassword>
> use local
> db.addUser('repl', <replpassword>);
^c
$ mongo <masterhostname>/admin -u <existingadminusername> -p<adminpassword>
> use local
> db.addUser('repl', <replpassword>);
```

Master Slave vs. Replica Sets

Master/slave and replica sets are alternative ways to achieve replication with MongoDB.

Replica sets are newer (v1.6+) and more flexible, although a little more work to set up and learn at first.

The following replica set configuration is equivalent to a two node master/slave setup with hosts M (master) and S (slave):

```
$ # run mongod instances with "--replSet mysetname" parameter
$ # then in the shell:
$ mongo --host M
> cfg = {
>   _id : 'mysetname',
>   members : [
>     { _id : 0, host : 'M', priority : 1 },
>     { _id : 1, host : 'S', priority : 0, votes : 0 }
>   ]
> }
> rs.initiate(cfg);
```

Administrative Tasks
Failing over to a Slave (Promotion)

To permanently fail over from a down master (A) to a slave (B):

- shut down A
- stop mongod on B
- backup or delete local." datafiles on B
- restart mongod on B with the --master option

Note that is a one time cutover and the "mirror" is broken. A cannot be brought back in sync with B without a full resync.

Inverting Master and Slave

If you have a master (A) and a slave (B) and you would like to reverse their roles, this is the recommended sequence of steps. Note the following assumes A is healthy, up-to-date and up.

1. Halt writes on A (using the fsync command)
2. Make sure B is caught up
3. Shut down B
4. Wipe local." on B to remove old local.sources
5. Start up B with the --master option
6. Do a write on B (primes the oplog to provide a new sync start point).
7. Shut down B. B will now have a new set of local." files.
8. Shut down A and replace A's local." files with a copy of B's new local." files. Remember to compress the files before/while copying them – they can be quite large.
9. Start B with the --master option
10. Start A with all the usual slave options plus --fastsync

If A is not healthy but the hardware is okay (power outage, server crash, etc.):

- Skip the first two steps
- Replace all of A's files with B's files in step 8.

If the hardware is not okay, replace A with a new machine and then follow the instructions in the previous paragraph.

Creating a slave from an existing master’s disk image

If you can stop write operations to the master for an indefinite period, you can copy the data files from the master to the new slave, and then start the slave with --fastsync.

---

Be careful with --fastsync. If the data is not perfectly in sync, a discrepancy will exist forever.

---

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

Slave chaining

Slaves cannot be "chained", they must all connect to the master directly. If a slave is chained to another slave you may see the following in the logs: assertion 13051 tailable cursor requested on non capped collection ns:local.oplog.$main

Correcting a slave's source

If you accidentally type the wrong host for the slave's source or wish to change it, you can do so by manually modifying the slave's local.sources collection. For example, say you start the slave with:

```bash
$ mongod --slave --source prod.mississippi
```
Restart the slave without the `--slave` and `--source` arguments.

```bash
$ mongod
```

Now start the shell and update the `local.sources` collection.

```bash
> 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).

```bash
$ ./mongod --slave --source prod.mississippi
$ # or
$ ./mongod --slave
```

Now your slave will be pointing at the correct master.

### See Also

- [Replica Sets](#)

### Halted Replication

> These instructions are for master/slave replication. For replica sets, see [Resyncing a Very Stale Replica Set Member](#) instead.

If you're running mongod with master-slave replication, there are certain scenarios where the slave will halt replication because it hasn't kept up with the master's oplog.

The first is when a slave is prevented from replicating for an extended period of time, due perhaps to a network partition or the killing of the slave process itself. The best solution in this case is to resync the slave. To do this, open the mongo shell and point it at the slave:

```bash
$ mongo <slave_host_and_port>
```

Then run the resync command:

```bash
> use admin
> db.runCommand({resync: 1})
```

This will force a full resync of all data (which will be very slow on a large database). The same effect can be achieved by stopping `mongod` on the slave, delete all slave datafiles, and restarting it.

### Increasing the OpLog Size

Since the oplog is a capped collection, it's allocated to a fixed size; this means that as more data is entered, the collection will loop around and overwrite itself instead of growing beyond its pre-allocated size. If the slave can't keep up with this process, then replication will be halted. The solution is to increase the size of the master's oplog.

> Resync warning
> At present, the only way to increase the oplog's size is to delete the oplog and create a new one; this has the side-effect that the new oplog's oldest entry will be newer than any slaves' last replication timestamp, and so slaves will need to be resynced after allocating the new oplog. (This warning does not apply to replica sets.)
There are a couple of ways to do this, depending on how big your oplog will be and how much downtime you can stand. But first you need to figure out how big an oplog you need. If the current oplog size is wrong, how do you figure out what's right? The goal is not to let the oplog age out in the time it takes to clone the database. The first step is to print the replication info. On the master node, run this command:

```bash
> db.printReplicationInfo();
```

You'll see output like this:

- **configured oplog size:** 1048.576MB
- **log length start to end:** 7200secs (2hrs)
- **oplog first event time:** Wed Mar 03 2010 16:20:39 GMT-0500 (EST)
- **oplog last event time:** Wed Mar 03 2010 18:20:39 GMT-0500 (EST)
- **now:** Wed Mar 03 2010 18:40:34 GMT-0500 (EST)

This indicates that you're adding data to the database at a rate of 524MB/hr. If an initial clone takes 10 hours, then the oplog should be at least 5240MB, so something closer to 8GB would make for a safe bet.

The standard way of changing the oplog size involves stopping the `mongod` master, deleting the local.* datafiles, and then restarting with the oplog size you need, measured in MB:

```bash
$ # Stop mongod - killall mongod or kill -2 or ctrl-c) - then:
$ rm /data/db/local.*
$ mongod --oplogSize=8038 --master
```

Once you've changed the oplog size, restart with slave with `--autoresync`:

```
mongod --slave --autoresync
```

This method of oplog creation might pose a problem if you need a large oplog (say, > 10GB), since the time it takes `mongod` to pre-allocate the oplog files may mean too much downtime. If this is the case, read on.

### Manually Allocating OpLog Files

An alternative approach is to create the oplog files manually before shutting down `mongod`. Suppose you need an 20GB oplog; here's how you'd go about creating the files:

1. Create a temporary directory, /tmp/local.
2. You can either create the files yourself or let MongoDB allocate them. If you'd like to create them yourself, here's a shell script for doing just that:

   ```bash
cd /tmp/local
   for i in {0..9} do
     echo $i
     head -c 2146435072 /dev/zero > local.$i
   done
   done
```

   Note that the datafiles aren't exactly 2GB due MongoDB's max int size.

If you'd like MongoDB to preallocate them for you, you can do:

```
$ mongod --dbpath /tmp/local --port 27099 --master --oplogSize=20000
```

Set the port to be something that is different than the other `mongod` running on the machine. Once this instance has finished allocating oplog files (watch the log), shut it down. If you are allocating these files for a replica set, remove the local.ns file:

```
$ rm /tmp/local/local.ns
```

3. Shut down the `mongod` master (kill -2) and then replace the oplog files:
$ mv /data/db/local.* /safe/place
$ mv /tmp/local/* /data/db/

4. Restart the master with the new oplog size:

$ mongod --master --oplogSize=20000

5. Finally, resync the slave. This can be done by shutting down the slave, deleting all its datafiles, and restarting it.

Replica Pairs

Replica pairs will be removed for 1.8
Replica pairs should be migrated to their replacement, Replica Sets.

- Setup of Replica Pairs
- 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:

```bash
$ ./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.

   ```bash
   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

   ```bash
   n1> ./mongod --pairwith n3 --arbiter <arbiterserver>
   ```

4. Start n3 paired with n1.

   ```bash
   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:

```javascript
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.

**Sharding**

MongoDB scales horizontally via an auto-sharding (partitioning) architecture. MongoDB sharding provides:

- Automatic balancing for changes in load and data distribution
- Easy addition of new machines without down time
- Scaling to one thousand nodes
- No single points of failure
- Automatic failover

**Getting Started**

- Introduction. Philosophy, use cases, and its core components.
- Simple Initial Sharding Architecture
- Configuration. Setting up your cluster.
- Administration

**Additional Info**

- Failover How failover/HA works.
- Sharding Internals Implementation details.
- Restrictions and Limitations
- FAQ Frequently asked questions.
- HOWTO
  - Changing Config Servers

**Presentations and Further Materials**

- How Sharding Works - O’Reilly Webcast (February 2011)
- How queries work with sharding (PDF)
- Illustration of chunks and migration (PDF) http://www.10gen.com/video/mongosf2011/sharding
- Scaling MongoDB - O’Reilly ebook
- Schema design at scale (video)
- Mongo Sharding Architecture, Implementation, Internals (video)

**See Also**

- Replication

**Video - How and When to Scale MongoDB with Sharding**

Presentation at MongoSV (December 2011)

**Bulk Inserts**

This page details a few notes on bulk / batch insert operations into collections.

**No data distribution until the amount of data is not small**

Sharding uses a "chunk" concept and chunks are on the order of 50 megabytes. Thus until you reach significantly more data volume in the collection than that, you won’t see a large amount of data distribution.

**Sharded collections and bulk insert**

If the collection is not yet populated, the database cannot yet have "learned" what the key distribution is and how to distribute it. The sudden bulk
insert will happen quickly and the system will need time to learn and adapt (at least in <=v2.2). To avoid this learning time you can **predefine some splits (key ranges)** and then the early stages of the bulk import will run more efficiently.

**Parallel import**

You could parallel import by sending inserts to multiple mongos instances. You may want to **pre-split** first (see above) if the collection is empty at the beginning.

**ObjectIds and other monotonically increasing shard key values**

If your shard key, during inserts, monotonically increases, this will mean that all inserts are going to the last chunk in the collection. The system will adjust the metadata to keep balance, but at a given time t all writes will be going to a single shard, which is undesirable if insert rate is extremely large. To avoid this, consider using a shard key that is not increasing in value. For example in some cases you could reverse all the bits of your shard key, which is information preserving yet then avoids the increasing sequence of values.

Note that **BSON ObjectIds** have this property. You might wish at generation time to reverse the bits of the ObjectIds, or swap the first and last 16 bit words, to “shuffle” the inserts. Alternatively you might use UUIDs instead (but check that your UUID generator does not generate increasing UUIDs consistently or you would get the same behavior).

Note that shard key values that are strictly increasing is fine if the insert volume is within the range that a single shard can process at a given point in time.

Future versions of the database will provide an option to hash shard keys for distribution when that is desired by the operator/developer.

**Example**

The following example, in C++, swaps the leading and trailing 16 bit word of object IDs generated so that they are no longer monotonically increasing:

```cpp
using namespace mongo;
OID make_an_id() {
    OID x = OID::gen();
    const unsigned char *p = x.getData();
    swap( (unsigned short&) p[0], (unsigned short&) p[10] );
    return x;
}

void foo() {
    // create an object
    BSONObj o = BSON("_id" << make_an_id() << "x" << 3 << "name" << "jane");
    // now we might insert o into a sharded collection...
}
```

**See Also**

- Import Export Tools

**Changing Config Servers**

**Sections:**

- Best Practices
- Upgrading from one config server to three
- Moving your config servers - same host name or virtual ip
- Renaming a config server - different host name
- Replacing a dead config server

The config server data is the most important data in your entire cluster. Back it up before doing config server maintenance.

Adding and changing config servers is a bit tricky right now. This will be improved in a future release, see [http://jira.mongodb.org/browse/SERVER-1658](http://jira.mongodb.org/browse/SERVER-1658).

Note that config servers are **not** a replica set; instead they use a two phase commit protocol to keep their data synchronous. Thus each config
server (if you have 3) has exactly the same data. When one or more config servers are down, the others are available for reading, but not for writing. During that window of time, sharding metadata will be static (which is fine for a while – fine for a couple hours, for example).

See also the Sharding and Failover page for more information on HA when various mongo processes go down in a cluster.

**Best Practices**

- Use three config servers.
- Give them logical (DNS) names. Do not refer to them (on the mongos command line) by IP address.
- The DNS name should be logical (e.g. "mycluster_config2") not a name that is VM-specific.
- Use a short DNS TTL (20 seconds or example).
- Act slowly when replacing a config server – your system should be fine with one down. Better to not have an operational typo and to let one be down for a while. If one is down, take some time and read this page, consider opening a support ticket, etc., before taking action. The metadata for the cluster is very important.
- When replacing a config server, make sure the old process/machine is truly down before starting up the new one. There may be legacy sockets open to the old config server process otherwise, and that would be bad.

**Upgrading from one config server to three**

Unfortunately you will need to shutdown the entire system.

1. Shutdown all processes (mongod, mongos, config server).
2. Copy the data subdirectories (dbpath tree) from the config server to the new config servers.
3. Start the config servers.
4. Restart mongod processes.
5. Restart mongos processes with the new --configdb parameter.

**Moving your config servers - same host name or virtual ip**

If you are using hostnames or virtual ips for your config, this is pretty simple.

1. Shutdown config server you want to move
2. Change dns entry to new machine
3. Move data to new machine
4. Start new config server

**Renaming a config server - different host name**

If you wish to use a different name or ip address in the --configdb option then this applies to you.

1. Shutdown config server you want to move
2. Move data to new machine
3. Start new config server
4. Shutdown all processes (mongod, mongos, config server).
5. Restart mongod processes.
6. Restart mongos processes with the new --configdb parameter.

**Replacing a dead config server**

Let's assume we have been running with mongos commands lines of:

```bash
mongos --configdb hosta,hostb,hostc
```

and that hostb has died. We want to replace it.

> You cannot change the name/ip used in the `mongos --configdb` line; if you wish to do this you must follow the directions above for renaming or moving your servers.

1. Provision a new machine. Give it the same hostname (hostb) that the host to be replaced had.
2. Shut down (only) one of the other config server processes – say, hostc. Then copy its data files to hostb. (We shut down so that we know we have a consistent image of the datafiles. See also the backups page for alternatives.)
3. Restart the config server on hostc.
4. Start the config server for the first time on hostb.

That's it. The key above is that we reused the logical (DNS) name of the host. That way we do not have to tell the other (many) processes in the system where the config servers are. Check the length of your DNS ttl using the `dig` command.
Simple Initial Sharding Architecture

- **Overview**
- **Goal**
- **Machines Locations**
  - Datacenter Roles
- **Replica sets**
- **Config Servers**
- **MongoS (routers)**
  - Suggested
  - Alternative
  - Startup Options
- **Everything is running**
  - Notes

**Overview**

This is a sample sharding architecture that is a good starting point for building your cluster.

**Goal**

- Two datacenters (East=primary, West=backup/DR)
- Data Tier (MongoDB)
  - 3 shards
  - 3 nodes per shard
  - 9 hosts total
- Application Tier
  - 4 application servers

**Machines Locations**

- e1-e6 are in 1 datacenter (East)
- w1-w3 are in another datacenter (West)
**Datacenter Roles**

We'll use datacenter East as the primary, and data center West as disaster recovery.

**Replica sets**

The replica set nodes in West will be priority 0 so they don't become master automatically.

The first thing we need to do is setup the 3 replica sets

- **Replica Set A**:
  - e1.acme.com: priority=1
  - e2.acme.com: priority=1
  - w1.acme.com: priority=0

Start the mongod process on each node, e.g.

```
e1.acme.com# mongod --shardsvr --replSet rs_a
e2.acme.com# mongod --shardsvr --replSet rs_a
w1.acme.com# mongod --shardsvr --replSet rs_a
```

Note: --shardsvr will default the port to 27018

In the mongo shell, create the replica set
```json
> cfg = {
    _id : "rs_a",
    members : [
        {_id : 0, host : "e1.acme.com:27018", priority : 1},
        {_id : 1, host : "e2.acme.com:27018", priority : 1},
        {_id : 2, host : "w1.acme.com:27018", priority : 0}
    ]
}>
rs.initiate(cfg)
```

Repeat for each replica set as follows

- **Replica Set B:**
  - e3.acme.com : priority=1
  - e4.acme.com : priority=1
  - w2.acme.com : priority=0

- **Replica Set C:**
  - e5.acme.com : priority=1
  - e6.acme.com : priority=1
  - w3.acme.com : priority=0

**Config Servers**

The next thing we need is to choose 3 config server locations.

We'll pick 2 random nodes in E (primary) and 1 in W (backup/DR). Since we have multiple MongoDB process on a node we need to ensure that different ports are being used, so we will use 27019 for the Config Servers.

- **Config Servers**
  - e1 - c1.acme.com:27019
  - e4 - c2.acme.com:27019
  - w1 - c3.acme.com:27019

**DNS Aliases**

We should make dns aliases for these so its easy to change later

Start up the config servers, e.g.

```
e1.acme.com> mongod --configsvr
e4.acme.com> mongod --configsvr
w1.acme.com> mongod --configsvr
```

*Note: --configsvr will default the port to 27019*

**MongoS (routers)**

The last question is where to put the mongos.

**Suggested**

The suggested configuration is to run an instance on each **app-server** (as shown by a1-a4 in diagram above).

**Alternative**

These are the other common options:

- On each server (all 9)
- Create a (sticky) load-balanced cluster (independent of client/shards)

**Startup Options**

When we start the mongos, we'll use
> mongos --configdb c1.acme.com:27019,c2.acme.com:27019,c3.acme.com:27019

Note: be sure to use DNS names for your configdb names, not IP addresses. Otherwise moving a config server later will be quite difficult.

Then we'll need to add the 3 replica sets as shards

```language
> db.adminCommand( { addShard : "rs_a/e1.acme.com:27018,e2.acme.com:27018,w1.acme.com:27018" } )
> db.adminCommand( { addShard : "rs_b/e3.acme.com:27018,e4.acme.com:27018,w2.acme.com:27018" } )
> db.adminCommand( { addShard : "rs_c/e5.acme.com:27018,e6.acme.com:27018,w3.acme.com:27018" } )
```

Everything is running

At this point your basic architecture is ready to go. You've got 3 shards for scalability, and 3 copies of each piece of data (with one ready for DR). There are obviously many ways to configure this, but this is a pretty simple way to get started.

Your next step is to enable sharding for any database you would like to use sharded, and to then enable sharding for any collection you want sharded. Databases and collections by default will be "unsharded" and simply reside in the first shard.

✅ To save yourself time, don't bother sharding tiny collections, just do the big ones.

**Notes**

- Names (DNS) should be used everywhere, and consistently
- All client writes/reads will be isolated to the Primary Datacenter (East)
- SlaveOk can be used to allow stale/eventually-consistent reads
  - If this is done, hidden should be used with the West nodes e.g.

```language
> cfg = {
  _id : "rs_a",
  members : [
    { _id : 0, host : "e1.acme.com:27018", priority : 1},
    { _id : 1, host : "e2.acme.com:27018", priority : 1},
    { _id : 2, host : "w1.acme.com:27018", priority : 0, hidden : true}
  ]
}
> rs.initiate(cfg)
```

Sharding Introduction

MongoDB supports an automated sharding/partitioning architecture, enabling horizontal scaling across multiple nodes. For applications that outgrow the resources of a single database server, MongoDB can convert to a sharded cluster, automatically managing failover and balancing of nodes, with few or no changes to the original application code.

This document explains MongoDB's auto-sharding approach to scalability in detail and provides an architectural overview of the various components that enable it.

Be sure to acquaint yourself with the current limitations.

- **MongoDB's Auto-Sharding**
  - Sharding in a Nutshell
  - Balancing
  - Failover
  - Scaling Model
- **Architectural Overview**
  - Shards
    - Shard Keys
    - Chunks
  - Config DB Processes
  - Routing Processes (mongos)
  - Operation Types
  - Server Layout
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:

<table>
<thead>
<tr>
<th>Machine 1</th>
<th>Machine 2</th>
<th>Machine 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama Arizona</td>
<td>Colorado Florida</td>
<td>Arkansas California</td>
</tr>
<tr>
<td>Indiana Kansas</td>
<td>Idaho Illinois</td>
<td>Georgia Hawaii</td>
</tr>
<tr>
<td>Maryland Michigan</td>
<td>Kentucky Maine</td>
<td>Minnesota Missouri</td>
</tr>
<tr>
<td>Montana Montana</td>
<td>Nebraska New Jersey</td>
<td>Ohio Pennsylvania</td>
</tr>
<tr>
<td>New Mexico North Dakota</td>
<td>Rhode Island South Dakota</td>
<td>Tennessee Utah</td>
</tr>
<tr>
<td>Vermont West Virgina</td>
<td>Wisconsin Wyoming</td>
<td></td>
</tr>
</tbody>
</table>

Note that each machine stores multiple "chunks" of data, based on state of residence. MongoDB distributes these chunks evenly across all of the machines available.

The chunking mechanism would only kick in if the amount of data you were storing reached the threshold where sharding is advantageous.

Our application connects to the sharded cluster through a `mongos` process, which routes operations to the appropriate shard(s). In this way, the sharded MongoDB cluster looks like a single logical server to our application. If our `users` collection receives heavy writes, those writes are now distributed across three shard servers. Most queries continue to be efficient, as well, because they too are distributed. And since the documents are organized in an order-preserving manner, any operations specifying the state of residence will be routed only to those nodes containing that state.

Sharding is performed on a per-collection basis. Small collections need not be sharded. For instance, if we were building a service like Twitter, our collection of tweets would likely be several orders of magnitude larger than the next biggest collection. The size and throughput demands of such a collection would be prime for sharding, whereas smaller collections would still live on a single server. Non-sharded collections reside on just one shard.

Balancing

Balancing is necessary when the load on any one shard node grows out of proportion with the remaining nodes. In this situation, the data must be redistributed to equalize load across shards.

Failover

Failover is also quite important since proper system functioning requires that each logical shard be always online. In practice, this means that each shard consists of more than one machine in a configuration known as a `replica set`. A replica set is a set of `n` servers, typically two or three, each of which contains a replica of the entire data set for the given shard. One of the `n` servers in a replica set will always be primary. If the primary fails, the remaining replicas are capable of electing a new master.

Scaling Model

MongoDB's auto-sharding scaling model shares many similarities with Yahoo's PNUTS and Google's BigTable. Readers interested in detailed discussions of distributed databases using order-preserving partitioning are encouraged to look at the PNUTS and BigTable white papers.

Architectural Overview

A MongoDB shard cluster consists of two or more shards, one or more config servers, and any number of routing processes to which the application servers connect. Each of these components is described below in detail.
**Shards**

Each shard consists of one or more servers and stores data using `mongod` processes (`mongod` being the core MongoDB database process). In a production situation, each shard will consist of multiple servers to ensure availability and automated failover. The set of servers/mongod processes within the shard comprise a **replica set**.

In MongoDB, **sharding is the tool for scaling a system, and replication is the tool for data safety, high availability, and disaster recovery.** The two work in tandem yet are orthogonal concepts in the design.

For testing, you can use sharding with a single mongod instance per shard. Production databases typically need redundancy, so they use replica sets.

**Shard Keys**

To partition a collection, we **specify a shard key pattern**. This pattern is similar to the key pattern used to define an index; it names one or more fields to define the key upon which we distribute data. Some example shard key patterns include the following:

```json
{ state : 1 }
{ name : 1 }
{ _id : 1 }
{ lastname : 1, firstname : 1 }
{ tag : 1, timestamp : -1 }
```

MongoDB’s sharding is order-preserving; adjacent data by shard key tend to be on the same server. The config database stores all the metadata indicating the location of data by range:

<table>
<thead>
<tr>
<th>collection</th>
<th>minkey</th>
<th>maxkey</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>users</td>
<td>{ name : 'Miller' }</td>
<td>{ name : 'Nessman' }</td>
<td>shard2</td>
</tr>
<tr>
<td>users</td>
<td>{ name : 'Nessman' }</td>
<td>{ name : 'Ogden' }</td>
<td>shard4</td>
</tr>
</tbody>
</table>

**Chunks**

A chunk is a contiguous range of data from a particular collection. Chunks are described as a triple of `collection, minKey, and maxKey`. Thus, the shard key K of a given document assigns that document to the chunk where minKey <= K < maxKey.

Chunks grow to a maximum size, usually 64MB. Once a chunk has reached that approximate size, the chunk **splits** into two new chunks. When a particular shard has excess data, chunks will then **migrate** to other shards in the system. The addition of a new shard will also influence the
migrations of chunks.

When choosing a shard key, the values must be of high enough cardinality (granular enough) that data can be broken into many chunks, and thus distribute-able. For instance, in the above example, where we are sharding on `name`, if a huge number of users had the same name, that could be problematic as all documents involving that name would be in a single undivided chunk. With names that typically does not happen and thus `name` is a reasonable choice as a shard key.

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

**Config DB Processes**

The config servers store the cluster's metadata, which includes basic information on each shard server and the chunks contained therein.

Chunk information is the main data stored by the config servers. Each config server has a complete copy of all chunk information. A two-phase commit is used to ensure the consistency of the configuration data among the config servers. Note that config server use their own replication model; they are not run in as a replica set.

If any of the config servers is down, the cluster's meta-data goes read only. However, even in such a failure state, the MongoDB cluster can still be read from and written to.

**Routing Processes (mongos)**

The `mongos` process can be thought of as a routing and coordination process that makes the various components of the cluster look like a single system. When receiving client requests, the `mongos` process routes the request to the appropriate server(s) and merges any results to be sent back to the client.

`mongos` processes have no persistent state; rather, they pull their state from the config server on startup. Any changes that occur on the 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 }`.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db.foo.find( { x : 300 } )</code></td>
<td>Targeted</td>
<td>Queries a single shard.</td>
</tr>
<tr>
<td><code>db.foo.find( { x : 300, age : 40 } )</code></td>
<td>Targeted</td>
<td>Queries a single shard.</td>
</tr>
<tr>
<td><code>db.foo.find( { age : 40 } )</code></td>
<td>Global</td>
<td>Queries all shards.</td>
</tr>
<tr>
<td><code>db.foo.find()</code></td>
<td>Global</td>
<td>sequential</td>
</tr>
<tr>
<td><code>db.foo.find(...).count()</code></td>
<td>Variable</td>
<td>Same as the corresponding find() operation</td>
</tr>
<tr>
<td><code>db.foo.find(...).sort( { age : 1 } )</code></td>
<td>Global</td>
<td>parallel</td>
</tr>
<tr>
<td><code>db.foo.find(...).sort( { x : 1 } )</code></td>
<td>Global</td>
<td>sequential</td>
</tr>
<tr>
<td><code>db.foo.count()</code></td>
<td>Global</td>
<td>parallel</td>
</tr>
<tr>
<td><code>db.foo.insert( &lt;object&gt; )</code></td>
<td>Targeted</td>
<td></td>
</tr>
<tr>
<td><code>db.foo.update( { x : 100 }, &lt;object&gt; )</code></td>
<td>Targeted</td>
<td></td>
</tr>
<tr>
<td><code>db.foo.remove( { x : 100 } )</code></td>
<td>Targeted</td>
<td></td>
</tr>
<tr>
<td><code>db.foo.update( { age : 40 }, &lt;object&gt;, false, true )</code></td>
<td>Global</td>
<td>Updates without the shard key must be multi-updates</td>
</tr>
<tr>
<td><code>db.foo.remove( { age : 40 } )</code></td>
<td>Global</td>
<td></td>
</tr>
</tbody>
</table>
Server Layout

Machines may be organized in a variety of fashions. For instance, it's possible to have separate machines for each config server process, `mongos` process, and `mongod` process. However, this can be overkill since the load is almost certainly low on the config servers. Here is an example where some sharing of physical machines is used to lay out a cluster. The outer boxes are machines (or VMs) and the inner boxes are processes.

In the picture about a given connection to the database simply connects to a random `mongos`. `mongos` is generally very fast so perfect balancing of those connections is not essential. Additionally the implementation of a driver could be intelligent about balancing these connections (but most are not at the time of this writing).

Yet more configurations are imaginable, especially when it comes to `mongos`. Alternatively, as suggested earlier, the `mongos` processes can exist on each application server. There is some potential benefit to this configuration, as the communications between app server and `mongos` then can occur over the localhost interface.

Exactly three config server processes are used in almost all sharded mongo clusters. This provides sufficient data safety; more instances would increase coordination cost among the config servers.

Next: Configuring Sharding

Configuring Sharding

This document describes the steps involved in setting up a basic sharding cluster. A sharding cluster has three components:

1. One to 1000 shards. Shards are partitions of data. Each shard consists of one or more `mongod` processes which store the data for that shard. When multiple `mongod`'s are in a single shard, they are each storing the same data – that is, they are replicating to each other.
2. Either one or three config server processes. For production systems use three.
3. One or more `mongos` routing processes.

For testing purposes, it's possible to start all the required processes on a single server, whereas in a production situation, a number of server configurations are possible.
Once the shards (mongod’s), config servers, and mongos processes are running, configuration is simply a matter of issuing a series of commands to establish the various shards as being part of the cluster. Once the cluster has been established, you can begin sharding individual collections.

This document is fairly detailed; for a terse, code-only explanation, see the sample shard configuration. If you’d like a quick script to set up a test cluster on a single machine, we have a python sharding script that can do the trick.

- Sharding Components
  - 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
  - Examples
  - See Also

**Sharding Components**

First, start the individual shards (mongod’s), config servers, and mongos processes.

**Shard Servers**

A shard server consists of a mongod process or a replica set of mongod processes. For production, use a replica set for each shard for data safety and automatic failover. To get started with a simple test, we can run a single mongod process per shard, as a test configuration doesn’t demand automated failover.

**Config Servers**

Run a mongod --configsvr process for each config server. If you’re only testing, you can use only one config server. For production, use three.

Note: Replicating data to each config server is managed by the router (mongos); they have a synchronous replication protocol optimized for three machines, if you were wondering why that number. Do not run any of the config servers with --replSet; replication between them is automatic.

Note: As the metadata of a MongoDB cluster is fairly small, it is possible to run the config server processes on boxes also used for other purposes.

**mongos Router**

Run mongos on the servers of your choice. Specify the --configdb parameter to indicate location of the config database(s). Note: use dns names, not ip addresses, for the --configdb parameter’s value. Otherwise moving config servers later is difficult.

Note that each mongos will read from the first config server in the list provided. If you’re running config servers across more than one data center, you should put the closest config servers early in the list.

**Configuring the Shard Cluster**

Once the shard components are running, issue the sharding commands. You may want to automate or record your steps below in a .js file for replay in the shell when needed.

Start by connecting to one of the mongos processes, and then switch to the admin database before issuing any commands.

The mongos will route commands to the right machine(s) in the cluster and, if commands change metadata, the mongos will update that on the config servers. So, regardless of the number of mongos processes you’ve launched, you’ll only need run these commands on one of those processes.

You can connect to the admin database via mongos like so:

```bash
./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:
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.

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:

```javascript
> db.runCommand( { addshard : "sf103", maxSize:100000/*MB*/ } );
```

**Listing shards**

To see current set of configured shards, run the `listshards` command:

```javascript
> db.runCommand( { listshards : 1 } );
```

This way, you can verify that all the shard have been committed to the system.

**Removing a shard**

See the `removeshard` command.

**Enabling Sharding on a Database**

In versions prior to v2.0, dropping a sharded database causes issues - see SERVER-2253 for workaround.

Once you've added one or more shards, you can enable sharding on a database. Unless enabled, all data in the database will be stored on the same shard. After enabling you then need to run `shardCollection` on the relevant collections (i.e., the big ones).

```javascript
> db.runCommand( { enablesharding : "<dbname>" } );
```

Once enabled, `mongos` will place new collections on the primary shard for that database. Existing collections within the database will stay on the original shard. To enable partitioning of data, we have to shard an individual collection.

**Sharding a Collection**

When sharding a collection, "pre-splitting", that is, setting a seed set of key ranges, is recommended. Without a seed set of ranges, sharding works, however the system must learn the key distribution and this will take some time; during this time performance is not as high. The presplits do not have to be particularly accurate; the system will adapt to the actual key distribution of the data regardless.
Use the `shardcollection` command to shard a collection. When you shard a collection, you must specify the shard key. If there is data in the collection, mongo will require an index to be created upfront (it speeds up the chunking process); otherwise, an index will be automatically created for you.

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

Running the "shardcollection" command will mark the collection as sharded with a specific key. Once called, there is currently no way to disable sharding or change the shard key, even if all the data is still contained within the same shard. It is assumed that the data may already be spread around the shards. If you need to "unshard" a collection, drop it (of course making a backup of data if needed), and recreate the collection (loading the backup data).

For example, let's assume we want to shard a GridFS chunks collection stored in the test database. We'd want to shard on the files_id key, so we'd invoke the `shardcollection` command like so:

```javascript
> db.runCommand( { shardcollection : "test.fs.chunks", key : { files_id : 1 } } )
```

You can use the `unique: true` option to ensure that the underlying index enforces uniqueness so long as the unique index is a prefix of the shard key. (note: prior to version 2.0 this worked only if the collection is empty).

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

If the "unique: true" option is not used, the shard key does not have to be unique.

```javascript
db.runCommand( { shardcollection : "test.products", key : { category : 1, _id : 1 } } );
```

You can shard on multiple fields if you are using a compound index.

In the end, picking the right shard key for your needs is extremely important for successful sharding. Choosing a Shard Key.

**Examples**

- **Sample configuration session**
- The following example shows how to run a simple shard setup on a single machine for testing purposes: Sharding JS Test.

**See Also**

- Sharding Administration
- Notes on TCP Port Numbers

**A Sample Configuration Session**

The following example uses two shards (each with a single mongod process), one config db, and one mongos process, all running on a single test server. In addition to the script below, a python script for starting and configuring shard components on a single machine is available.

**Creating the Shards**

First, start up a couple mongods to be your shards.

```bash
$ mkdir /data/db/a /data/db/b
$ ./mongod --shardsvr --dbpath /data/db/a --port 10000 > /tmp/sharda.log &
$ cat /tmp/sharda.log
$ ./mongod --shardsvr --dbpath /data/db/b --port 10001 > /tmp/shardb.log &
$ cat /tmp/shardb.log
```

Now you need a configuration server and mongos:
$ mkdir /data/db/config
$ ./mongod --configsvr --dbpath /data/db/config --port 20000 > /tmp/configdb.log &
$ cat /tmp/configdb.log
$ ./mongos --configdb localhost:20000 > /tmp/mongos.log &
$ cat /tmp/mongos.log

mongos does not require a data directory, it gets its information from the config server.

In a real production setup, mongod's, mongos's and configs would live on different machines. The use of hostnames or IP addresses is mandatory in that case. 'localhost' appearance here is merely illustrative – but fully functional – and should be confined to single-machine, testing scenarios only.

You can toy with sharding by using a small `--chunkSize`, e.g. 1MB. This is more satisfying when you're playing around, as you won't have to insert 64MB of documents before you start seeing them moving around. It should not be used in production.

$ ./mongos --configdb localhost:20000 --chunkSize 1 > /tmp/mongos.log &

Setting up the Cluster

We need to run a few commands on the shell to hook everything up. Start the shell, connecting to the mongos process (at localhost:27017 if you followed the steps above).

To set up our cluster, we'll add the two shards (a and b).

$ ./mongo
MongoDB shell version: 1.6.0
connecting to: test
> use admin
switched to db admin
> db.runCommand( { addshard : "localhost:10000" } )
{ "shardadded" : "shard0000", "ok" : 1 }
> db.runCommand( { addshard : "localhost:10001" } )
{ "shardadded" : "shard0001", "ok" : 1 }

Now you need to tell the database that you want to spread out your data at a database and collection level. You have to give the collection a key (or keys) to partition by.
This is similar to creating an index on a collection.

> db.runCommand( { enablesharding : "test" } )
{"ok" : 1}
> db.runCommand( { shardcollection : "test.people", key : {name : 1} } )
{"ok" : 1}

Administration

To see what's going on in the cluster, use the config database.
> use config
switched to db config
> show collections
chunks
databases
lockpings
locks
mongos
settings
shards
system.indexes
version

These collections contain all of the sharding configuration information.

**Changing a Shard Key**

There is no automatic support for changing the shard key for a collection. In addition this would fundamentally be a very expensive operation most of the data in the cluster for the collection likely moves from machine to machine.

Thus it is very important to choose the right shard key up front.

If you do need to change a shard key, an export and import is likely the best solution. Create a new pre-sharded collection, and then import the exported data to it. If desired use a dedicated mongos for the export and the import.

https://jira.mongodb.org/browse/SERVER-4000

**Choosing a Shard Key**

*It is important to choose the right shard key for a collection.* If the collection is gigantic it is difficult to change the key later. When in doubt please ask for suggestions in the support forums or IRC.

The basic rule of thumb is: whatever field is common to most of your queries, that field is likely the right one to use for the shard key. If a query involves the shard key, it can (typically) be sent to only one shard.

- Cardinality
- Write scaling
- Query isolation
- Sorting
- Reliability
- Index optimization
- GridFS

The discussion below considers a structured event logging system. Documents in the events collection have the form:

```json
{
    mon_node : "ny153.example.com",
    application : "apache",
    time : "2011-01-02T21:21:56.249Z",
    level : "ERROR",
    msg : "something is broken"
}
```

**Cardinality**

Documents in a sharded collection are grouped together in *chunks*.

All data in a sharded collection is broken into *chunks*. A chunk consists of all the documents who have a shard key within the chunk's specified (key_low..key_high) key range.

Using the logging example above, if you chose

```json
{mon_node:1}
```

as the shard key, then all data for a single mon_node is in one chunk, and thus on one shard. One can easily imagine having a lot of data for a
mon_node and wanting to split it among shards. In addition, chunks are approximately 64MB in size. If the chunk is unsplitable because it represents a single mon_node value, the balancer migrating that chunk from server to server would be slow.

If one were to shard on the key

```
{mon_node:1,time:1}
```

we can then split data for a single mon_node down to the millisecond. No chunk will ever be too large. (Note there will not be a chunk for every mon_node,time value – rather on ranges such as

```
{mon_node:"nyc_app_24",time:"2011-01-01"}..{mon_node:"nyc_app_27",time:"2011-07-12"}
```

or if a lot of data for one mon_node:

```
{mon_node:"nyc_app_24",time:"2011-01-01"}..{mon_node:"nyc_app_24",time:"2011-01-07"}
```

Keeping chunks to a reasonable size is very important so data can be balanced and moving chunks isn't too expensive.

**Write scaling**

One of the primary reasons for using sharding is to distribute writes. To do this, it's important that writes hit as many chunks as possible.

Again using the example above, choosing

```
{ time : 1 }
```

as the shard key will cause all writes to go to the newest chunk. If the shard key were

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

then each mon_node,application pair can potentially be written to different shards. Thus there were 100 mon_node,application pairs, and 10 shards, then each shard would get 1/10th the writes.

Note that because the most significant part of an ObjectID is time-based, using ObjectID as the shard key has the same issue as using time directly.

**Query isolation**

Another key consideration is how many shards any query has to hit. Ideally a query would go directly from mongos to the shard (mongod) that owns the document requested. Thus, if you know that most common queries include a condition on mon_node, then starting the shard key with mon_node would be efficient.

All queries work regardless of the shard key, but if mongos cannot determine which shard that owns the data, it will send the operation to all shards in parallel. This will increase the response time latency and increase the volume of network traffic and load.

**Sorting**

When a query includes a sort criteria, the query is sent to the same shards that would have received the equivalent query without the sort expression. Each shard queried performs a sort on its subset of the data locally (potentially utilizing an appropriate index locally if one is available). mongos merges the ordered results coming from each shard and returns the merged stream to the client. Thus the mongos process does very little work and requires little RAM for request of this nature.
Reliability

One important aspect of sharding is how much of the system will be affected in case an entire shard becomes unavailable (even though it is usually a reliable replica set).

Say you have a twitter-like system, with comment entries like:

```json
{
    _id: ObjectId("4d084f78a4c8707815a601d7"),
    user_id : 42 ,
    time : "2011-01-02T21:21:56.249Z" ,
    comment : "I am happily using MongoDB",
}
```

One might use `id` or `user_id` as shard key. `id` will give you better granularity and spread, but in case a shard goes down, it will impact almost all your users (some data missing). If you use `user_id` instead, a lower percentage of users will be impacted (say 20% for a 5 shard setup), although these users will not see any of their data.

Index optimization

As mentioned in other sections about indexing, it is usually much better performance-wise if only a portion of the index is read/updated, so that this "active" portion can stay in RAM most of the time. Most shard keys described above result in an even spread over the shards but also within each mongod’s index. Instead, it can be beneficial to factor in some kind of timestamp at the beginning of the shard key in order to limit the portion of the index that gets hit.

Say you have a photo storage system, with photo entries like:

```json
{
    _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:
This custom id would be the key used for sharding, and also the id used to retrieve a document. It gives both a good spread across all servers, and it reduces the portion of the index hit on most queries.

Further notes:

- at the beginning of each month, only one shard would be hit for some time until the balancer starts splitting buckets. To avoid this potential slowdown and migration of data, it is advised to create ranges ahead of time (e.g. 5 or more ranges for each month if you have 5 servers).
- As a further improvement, you can incorporate the user id into the photo id. This way it will keep all documents belonging to same user within the same shard. For example: "2011-01_42_4d084f78a4c8707815a601d7"

GridFS

There are different ways that GridFS can be sharded, depending on the need. One common way to shard, based on pre-existing indexes, is:

- "files" collection is not sharded. All file records will live in 1 shard. It is highly recommended to make that shard very resilient (at least 3 node replica set)
- "chunks" collection should be sharded using the index "files_id: 1". The existing "files_id, n" index created by the drivers cannot be used to shard on "files_id" (this is a sharding limitation that will be fixed), so you have to create a separate index on just "files_id". The reason to use "files_id" is so that all chunks of a given file live on the same shard, which is safer and allows "filemd5" command to work (required by certain drivers).

Run the commands:

```shell
$ db.fs.chunks.ensureIndex({files_id: 1});
$ db.runCommand({ shardcollection : "test.fs.chunks", key : { files_id : 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.

Tag Aware Sharding

In 2.2, MongoDB adds additional support for custom partitioning in sharded clusters – specifically sharded collections. By using this “tag aware” sharding and balancing, you can (automatically) ensure that data in a sharded database system is always on specific shards. This can be used to ensure data is geographically close to the systems which use it. You can also restrict sharded collections to only a few shards, therefore federating a set of shards for different uses.

Shard tagging controls data location, and is complementary but separate from replica set tagging – which can be used to enforce writing (and reading with Read Preferences within the replica set) within the replica set.

For example you can tag all “USA” data (by defining a range of your shard key which contains the "USA" data) to one or more shards (by tagging those shards with "USA").

Typically this is used to make assignment of documents to shards geographically aware. One marks a shard key range as "homed" on a certain set of shards. Those shards would then be configured such that their primary defaults to the desired data center / geography.
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
  - Version 2.2+
  - Tags on the Wrong Shard(s)
- Balancer window
  - Removing balancing window
- Chunk Size Considerations

Identifying a Shard Cluster
// Test if we're speaking to a mongos process or straight to a mongod process.
if connected to a mongod this will return a "no such cmd" message.
> db.runCommand({ isdbgrid : 1});

// If connected to mongos, this command returns { "ismaster": true,
// "msg": "isdbgrid", "maxBsonObjectSize": XXX, "ok": 1 }
> db.runCommand({ismaster:1});

List Existing Shards

> db.runCommand({ listShards : 1});
{"servers": [{"_id": ObjectId("4a9d40c981ba1487ccfaa634"),
"host": "localhost:10000"},
{"_id": ObjectId("4a9d40df81ba1487ccfaa635"),
"host": "localhost:1001"}]
,"ok" : 1
}

List Which Databases are Sharded

Here we query the config database, albeit through mongos. The getsisterdb command is used to return the config database. This will list all databases in the cluster. Databases with partitioned : true have sharding enabled.

> config = db.getsisterdb("config")
> config.databases.find()
{ "_id": "admin", "partitioned": false, "primary": "config" }
{ "_id": "MyShardedDatabase", "partitioned": true, "primary": "localhost:30001" }
{ "_id": "MyUnshardedDatabase", "partitioned": false, "primary": "localhost:30002" }

View Sharding Details

> use admin
> db.printShardingStatus();

// A very basic sharding configuration on localhost
sharding version: { "_id" : 1, "version" : 2 }
shards:
{ "_id" : ObjectId("4bd9ae3e0a2e26420e556876"),
"host": "localhost:30001" }
{ "_id" : ObjectId("4bd9ae420a2e26420e556877"),
"host": "localhost:30002" }
{ "_id" : ObjectId("4bd9ae460a2e26420e556878"),
"host": "localhost:30003" }
databases:
{ "name": "admin", "partitioned": false,
"primary": "localhost:20001",
"_id": ObjectId("4bd9add2c0302e394c6844b6") }
my chunks

{ "name": "foo", "partitioned": true,
"primary": "localhost:30002",
"sharded": { "foo.foo": { "key": { "_id" : 1 }, "unique": false } },
"_id": ObjectId("4bd9aee50c0302e394c6844b7") }
my chunks
foo.foo { "_id": { $minKey : 1 } } -->> { "_id": { $maxKey : 1 } }
on : localhost:30002 { "t": 1272557259000, "i": 1 }

Notice the output to the printShardingStatus command. First, we see the locations of the three shards comprising the cluster. Next, the various databases living on the cluster are displayed.

The first database shown is the admin database, which has not been partitioned. The primary field indicates the location of the database, which, in the case of the admin database, is on the config server running on port 20001.
The second database is partitioned, and it's easy to see the shard key and the location and ranges of chunks comprising the partition. Since there's no data in the `foo` database, only a single chunk exists. That single chunk includes the entire range of possible shard keys.

**Balancing**

The balancer is a background task that tries to keep the number of chunks even across all servers of the cluster. The activity of balancing is transparent to querying. Your application doesn't need to know or care that there is any data-moving activity ongoing.

To make that so, the balancer is careful about when and how much data it would transfer. Let's look at how much to transfer first. The unit of transfer is a chunk. With a steady state, the size of chunks should be roughly 64MB of data. This has shown to be the sweet spot of how much data to move at once. More than that, and the migration would take longer and the queries might perceive that in a wider difference in response times. Less than that, and the overhead of moving wouldn't pay off as highly.

Regarding when to transfer load, the balancer waits for a threshold of uneven chunk counts to occur before acting. In the field, having a difference of 8 chunks between the least and most loaded shards showed to be a good heuristic. (This is an arbitrary number, granted.) The concern here is not to incur overhead if -- 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 beween any two shards is down to 2 chunks.

A common source of questions is why a given collection is not being balanced. By far, the most probable cause is: it doesn't need to. If the chunk difference is small enough, redistributing chunks won't matter enough. The implicit assumption here is that you actually have a large enough collection and the overhead of the balancing machinery is little compared to the amount of data your app is handling. If you do the math, you'll find that you might not hit "balancing threshold" if you're doing an experiment on your laptop.

Another possibility is that the balancer is not making progress. The balancing task happens at an arbitrary mongos (query router) in your cluster. Since there can be several query routers, there is a mechanism they all use to decide which mongos will take the responsibility. The mongos acting as balancer takes a "lock" by inserting a document into the 'locks' collection of the config database. When a mongos is running the balancer the 'state' of that lock is 1 (taken).

To check the state of that lock

```bash
// connect to mongos
> use config
> db.locks.find({ _id: "balancer" })
```

A typical output for that command would be

```
```

There are at least three points to note in that output. One, the `state` attribute is 1 (2 in v2.0), which means that lock is taken. We can assume the balancer is active. Two, that balancer has been running since Monday, December the 20th. That's what the attribute "when" tells us. And, the third thing, the balancer is running on a machine called "guaruja". The attribute "who" gives that away.

To check what the balancer is actually doing, we'd look at the mongos log on that machine. The balancer outputs rows to the log prefixed by "[Balancer]."

```
Mon Dec 20 11:53:00 [Balancer] chose [shard0001] to [shard0000] { _id: "test.foo-_id_52.0", lastmod: Timestamp 2000|1, ns: "test.foo", min: { _id: 52.0 }, max: { _id: 105.0 }, shard: "shard0001" }
```

What this entry is saying is that the balancer decided to move the chunk _id:[52..105) from shard0001 to shard0000. Both mongod's log detailed entries of how that migrate is progressing.
In MongoDB versions 1.7 and earlier, the lock will always appear in `db.locks.find` even if the balancer is deactivated:

```
{ 
  "_id" : "balancer", 
  "process" : "guaruja:1292810611:1804289383", 
  "state" : 1, 
  "ts" : ObjectId("4d0f872630c42d1978be8a2e"), 
  "when" : "Mon Dec 20 2010 11:41:10 GMT-0500 (EST)" 
  "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](https://jira.mongodb.org/browse/SERVER-2632).

In newer versions of MongoDB the balancer does not create new lock entries when disabled. However, a pre-existing lock entry will continue to exist with its state set to 0.

If you want to pause the balancer temporarily for maintenance, you can by modifying the settings collection in the config db.

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

As a result of that command, one should stop seeing "[Balancer]" entries in the mongos that was running the balancer. If, for curiosity, you're running that mongos in a more verbose level, you'd see an entry such as the following:

```
Mon Dec 20 11:57:35 "[Balancer]" skipping balancing round because balancing is disabled
```

You would just set `stopped` to false to turn on again.

For more information on chunk migration and commands, see: [Moving Chunks](https://docs.mongodb.org/v3.4/reference/migration/migration-overview/)

**Version 2.2+**

Since tags were introduced to the balancing algorithm things have changed a bit. The policy now sets a threshold as follows (per collection):

<table>
<thead>
<tr>
<th>Number of chunks</th>
<th>Threshold to Initiate Migrations</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 80</td>
<td>8</td>
</tr>
<tr>
<td>80 - 21</td>
<td>4</td>
</tr>
<tr>
<td>&lt; 20 (and previously balanced)</td>
<td>2</td>
</tr>
</tbody>
</table>

Each tags follows the rules above; if no tags exist the policy still applies.

**Tags on the Wrong Shard(s)**

If chunks are found with a tag which are not appropriate to the tags on that shard they currently reside then they will be moved to an appropriate shards (containing that tag) – this happens independent of the thresholds listed above since it supersedes that algorithm.

**Balancer window**

By default the balancer operates continuously, but it is also possible to set an active "window" of time each day for balancing chunks. This can often be useful when a cluster adds or removes only a few GB of data per day and traffic would be less disrupted if balancing occurred during low-traffic times. This time interval is also specified in the `config.settings` balancer document. For example, to balance chunks only from 9AM to 9PM:

```
// connect to mongos
> use config
> db.settings.update({ _id : "balancer" }, { $set : { activeWindow : { start : "9:00", stop : "21:00" } } }, true )
```

To specify a time range spanning midnight, just swap the order of the times:
This enables balancing from 9PM to 9AM. If enabling the active window feature, it is important to check periodically that the amount of data inserted each day is not more than the balancer can handle in the limited window.

**Limitations**

- Currently only time of day is supported for automatic scheduling.
- The times are evaluated in local time on each mongos instance.

### Removing balancing window

```javascript
> use config
> db.settings.update({ _id : "balancer" }, { $set : { activeWindow : { start : "21:00", stop : "9:00" } } }, true)
```

### Chunk Size Considerations

mongoDB sharding is based on "range partitioning". Chunks are split automatically when they reach a certain size threshold. The threshold varies, but the rule of thumb is, expect it to between a half and the maximum chunk size in the steady state. The default maximum chunk size is 64MB (sum of objects in the collection, not counting indices), though in older versions it was 200MB.

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

When you move data, there's some state resetting in mongos. Queries that used to hit a given shard for a migrated chunk, now need to hit a new shard. This state resetting isn't free, so we want to move chunks infrequently (pressure to move a lot of keys at once). However, the actual moving has a cost that's proportional to the number of keys you're moving (pressure to move few keys).

If you opt to change the default chunk size for a production site, you can do that by changing the value of the chunksize setting on the config database by running

```javascript
> use config
> db.settings.save({_id:"chunksize", value:<new_chunk_size_in_mb>})
```

You will need to restart all mongos instances for them to pick up the change. Note though that for an existing cluster, it may take some time for the collections to split to that size, if smaller than before, and currently autosplitting is only triggered if the collection gets new documents or updates.

For more information on chunk splitting and commands, see: [Splitting Shard Chunks](#)

### Backing Up Sharded Cluster

- `mongos->mongodump (small clusters only)`
- **Backing up via shards**
  - Snapshooting
  - Large cluster backup (v2.0+)
  - Large cluster backup (v1.8 and earlier)
  - Restoring the entire cluster
  - Restoring a single shard
  - Other Suggestions
  - See Also

See also Backups and Import Export Tools for more information on backups, particularly on the step of backing up each individual server. If you are on EC2 you should look at EC2 Backup & Restore for how to take advantage of EBS snapshots.

`mongos->mongodump (small clusters only)`

⚠️ You do not need to turn off the balancer when running mongodump against mongos.
If you have a small sharded cluster, you can use `mongodump`, connected to a `mongos`, to dump the entire cluster's data, or a subset of it.

This is easy, but only works if one machine can dump all the data in a reasonable period of time. The default dump, when you don't specify a database or collection to dump, will contain both the sharding config metadata (the data on the config servers) and the actual database content.

The `mongodump --oplog` cannot be used when dumping from a `mongos`. This option is only available when running directly against a replica member.

**Backing up via shards**

The balancer **must** be turned off during a backup of the cluster (when done directly from shards)

**Snapshotting**

In MongoDB, documents move from shard to shard as the system continually rebalances the data and load. Thus care is needed to assure getting a good backup of a large cluster. The diagram at the bottom of this page illustrates a bit how data is distributed and some of the nuances of a cluster backup.

**Large cluster backup (v2.0+)**

- Turn off the balancer

```
// connect to a mongos (not a config server!)
> use config
> db.settings.update( { _id: "balancer" }, { $set : { stopped: true } }, true );
> // wait for any migrations that were in progress to finish
> // "state" field is zero if no migrations in progress
> while( db.locks.findOne({_id: "balancer"}).state ) { print("waiting..."); sleep(1000); }
```

- `mongodump` the config db from one of the config servers to back up the cluster's metadata (either through mongos or by direct connection to the config server). You only need to back up one config server, as they all have replicas of the same information.

- Backup each shard. Use the standard practices for backing up a replica set (e.g. fsync+lock then snapshot, or mongodump). Shards can be backed up one at a time or in parallel.

- Turn the balancer back on

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

**Large cluster backup (v1.8 and earlier)**

- Turn off the balancer

```
// connect to mongos (not a config server!)
> use config
> db.settings.update( { _id: "balancer" }, { $set : { stopped: true } }, true );
> // wait for any migrations that were in progress to finish
> // "state" field is zero if no migrations in progress
> while( db.locks.findOne({_id: "balancer"}).state ) { print("waiting..."); sleep(1000); }
```

**v1.6:** in this older release you must check the config.changelog collection rather than config.locks to see if migrates are still in-flight.

- stop one (and only one) config server (of your three config servers). This will make the configuration database read only. The db cluster is still fully readable and writable. Do not lock+fsync config servers; that can would block write operations on the cluster.

- Backup datafiles from the the stopped config server.
• Backup each shard. Use the **standard practices** to backup a replica sets (e.g. fsync+lock then snapshot, or mongodump). Shards can be backed up one at a time or in parallel.

• Restart the config server that was stopped.

• Turn the balancer back on

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

**Restoring the entire cluster**

• Stop all processes. No server should be running mongod, mongos, or a config server mongod processes.
• Restore data files for each server in each shard and also for each config server. Normally each shard is comprised of a replica set. You must restore all the members of the replica set, or use the other standard approaches for restoring a replica set from backup.
• If shard IPs/hosts are different, you have to manually update config.shards to use the new servers:
  1. Start up the three config servers
  2. Load/restore the config database to each config server
  3. Start one mongos instance for the rest
  4. Update the config database collection named "shards" to reflect the new shards' ip addresses; the contents of config.shards are a replica set seed list, so use the replica set name style "replName/seedAddress1,..."
• Restart the mongos instances
• Restart the mongod instances (the replica set members)
• Connect to a mongos from the shell. Run "printShardingStatus()" and "show collections" to make sure all shards are correctly seen.

**Restoring a single shard**

Clusters are designed to be restored as a whole. Since the last backup, data (chunks) may have moved from shard to shard, making restoration of a single shard difficult.

That said, we can with sufficient work restore a single shard from backup.

• Restore the shard which was lost
• Chunks that were migrated away from this shard are fine. Those documents do not need to be deleted from this shard because they are automatically filtered out from queries by mongos. (See diagram below.)
• Chunks that were migrated to this shard since the last backup are now empty. Those documents must be recovered manually from the other shard backups. You can see a changelog of chunk migrations in the config.changelog collection.

For a "mature" cluster that has been running for a while, if desired we can mitigate the above scenario by:

• keeping the auto-balancer off for significant time periods. If the shards are well balanced this is not a problem. If you do this, occasionally check the balance between the shards using either MMS or the mongo shell.
• if there has been no migrations since the last backup, we can restore the lost shard in a straightforward fashion
• at set times, turn the balancer on let it run for a time to rebalance the cluster if any rebalancing is needed. afterwards, a new backup would be warranted.

The above mitigation is not necessarily recommended. With the balancer off the cluster could become unbalanced unless we are monitoring it. Keeping full cluster backups and using a good degree of replication (perhaps including slaveDelay replication) is often sufficient.

The following illustration shows a five shard cluster and what might happen on a single shard restoration from a backup.
### Sharding Limits

- **Security**
- **Differences from Unsharded Configurations**
  - `$where`
  - `db.eval`
  - `group`
  - `getPrevError`
  - **Unique Indexes**
- **Scale Limits**
  - Query speed
- **Steps to shard an existing collection**

### Security

Authentication mode will be available with sharding as of v2.0. See SERVER-921 for details.

### Differences from Unsharded Configurations

- Prior to v2.0, sharding must be run in trusted security mode, without explicit security.
- Shard keys are immutable in the current version.
- All (non-multi)updates, upserts, and inserts must include the current shard key. This may cause issues for anyone using a mapping library since you don’t have full control of updates.

---

Other Suggestions

Note that unsharded collections are always on shard a single shard (the first shard, or the one marked as the primary shard in the databases collection in the config db) and do not move. Thus you might want to back up that shard more frequently to have fresher backups of those small collections.

**See Also**

- Single server backups documentation

---

### Table Examples

**at time of backup**

<table>
<thead>
<tr>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
<th>s5</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a,b]</td>
<td>[e,j]</td>
<td>[c,e]</td>
<td>[j,n]</td>
<td>[p,z]</td>
</tr>
<tr>
<td>[b,c]</td>
<td>[n,p]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**just before failure**

<table>
<thead>
<tr>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
<th>s5</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a,b]</td>
<td>[e,i]</td>
<td>[c,e]</td>
<td>[j,n]</td>
<td>[p,s]</td>
</tr>
<tr>
<td>[b,c]</td>
<td>[n,p]</td>
<td>[u,z]</td>
<td>[i,j]</td>
<td></td>
</tr>
</tbody>
</table>

**s5 has been lost, and needs restoration from backup. the s5 backup image has the old s5 backup image contains**

<table>
<thead>
<tr>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
<th>s5</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a,b]</td>
<td>[e,i]</td>
<td>[c,e]</td>
<td>[j,n]</td>
<td>[p,s]</td>
</tr>
<tr>
<td>[b,c]</td>
<td>[n,p]</td>
<td>[u,z]</td>
<td>[i,j]</td>
<td></td>
</tr>
</tbody>
</table>

**after restoring the s5 backup we have the [p,s] data. the older [s,z] data has also been restored to s3 but will be ignored. [i,j] is missing as it was not in the s5 backup image.**

<table>
<thead>
<tr>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
<th>s5</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a,b]</td>
<td>[e,i]</td>
<td>[c,e]</td>
<td>[j,n]</td>
<td>[p,s]</td>
</tr>
<tr>
<td>[b,c]</td>
<td>[n,p]</td>
<td>[u,z]</td>
<td>[i,j]</td>
<td></td>
</tr>
</tbody>
</table>

**we manually find and restore the [i,j] data, which in this case was present in the s2 backup.**

<table>
<thead>
<tr>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
<th>s5</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a,b]</td>
<td>[e,i]</td>
<td>[c,e]</td>
<td>[j,n]</td>
<td>[p,s]</td>
</tr>
<tr>
<td>[b,c]</td>
<td>[n,p]</td>
<td>[u,z]</td>
<td>[i,j]</td>
<td></td>
</tr>
</tbody>
</table>
$where works with sharding. However, do not reference the db object from the $where function (one normally does not do this anyway).

db.eval

db.eval() may not be used with sharded collections. However, you may use db.eval() if the evaluation function accesses unsharded collections within your database. Use map/reduce in sharded environments.

group

Currently, one must use MapReduce instead of group() on sharded collections.

getPrevError

getPrevError is unsupported for sharded databases, and may remain so in future releases (TBD). Let us know if this causes a problem for you.

Unique Indexes

For a sharded collection, you may (optionally) specify a unique constraint on the shard key. You also have the option to have other unique indices if and only if the shard key is a prefix of their attributes. In other words, MongoDB does not enforce uniqueness across shards. You may specify other secondary, non-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).

Query speed

Queries involving the shard key should be quite fast and comparable to the behavior of the query in an unsharded environment.

Queries not involving the shard key use a scatter/gather method which sends the query to all shards. This is fairly efficient if one has 10 shards, but would be fairly inefficient on 1000 shards (although still ok for infrequent queries).

Sharding an existing collection

It is possible to shard an existing collection, but there are some limitations. Put differently, if you have an existing single node (or single replica set) and you want to upgrade that data to a sharded configuration, this is possible.

The current limitations are size and time.

1. **Size**: In v1.6 we put a cap on the maximum size of the original collection of 25GB (increased to 256GB in v1.8). This limit is going to be pushed up and may eventually disappear. If you are above that limit and you shard an existing collection it will work, but all of your data will start out in one chunk, making initial distribution slower. In practice, if your collection contains many large documents, this limit may be slightly higher (due to the statistical way in which split points are calculated). One workaround is to increase the default chunk size in db.config.settings to a higher value (say 512MB or 1GB), which will enable the initial split and some migration. Then the large chunks will be naturally split over time as data is inserted.

2. **Time**: When sharding an existing collection, please be aware that this process will take some time. This will happen in the background, so operations are not significantly impacted typically. However, it will take quite a while for the data to migrate/balance on a large collection. For example, on a system with ten shards, 90% of the data for the collection will need to transfer to elsewhere to attain balance. Note that only large collections rebalance. If the collection is small (less than say, 400MB) we recommend not bothering to shard it.

Steps to shard an existing collection

1. Start your 3 config servers required for a sharded cluster - if your cluster will have large numbers of mongoses or shards (more than 10 of either, for example) moderately powerful machines will be needed for the config servers.

2. Start a mongos process pointing at these config servers to perform cluster administration.

3. Using this mongos process, add the existing database server or replica set as a shard to the cluster, as well as any empty servers or shards you would like to add to the cluster (more info [here](#)).

⚠️ Once the shards have been added, any application mongoses can be started and the cluster can start handling application reads and writes.
4. Choose a shard key for all existing collections you would like to shard, and create an index in the background over the shard key fields (more info here). Depending on the size of the collection and the index, this can take a long time. Note that background index creation becomes foreground index creation on secondaries, which will block other operations.

Choosing a shard key index is a complex topic, and is covered here

5. Using the administrative mongos, run the enableSharding and shardCollection commands on any existing collections you want distributed across the cluster (see link) with the shard key chosen above. At this point the collections are considered “sharded,” but enabling sharding is just a metadata operation and no data has been touched. However, once chosen, your shard key cannot be changed without significant manual intervention.

All mongoses should be restarted at this stage for mongos versions < 2.1

6. If any of the existing collections to be sharded were greater than 256GB in size, you will need to manually generate split points for these collections. The procedure for doing so will vary depending on your data and shard key, but is documented here. It is not necessary to do the splits in this case before the data is inserted because the data is already present.

7. Wait for the balancer operations from mongos to migrate excess data from the full shard to the empty shards. This can take significant time, longer if competing for resources with current traffic. Estimating the time migrations will take is difficult, as there are a number of factors which affect the speed:
   - the relationship between shard key and organization on disk (disk seeks)
   - the current requests to the chunks being moved
   - the in-memory state of the documents in the chunks being moved (page faults)

8. Once most of the data is balanced, the cluster will be in steady state, and as more data is inserted chunks will be split and moved automatically. If the shard key was chosen appropriately, traffic should be balanced across all servers for the existing collection.

**Sharding Internals**

This section includes internal implementation details for MongoDB auto sharding. See also the main sharding documentation.

Note: some internals docs could be out of date -- if you see that let us know so we can fix.

**Internals**

- Sharding Config Schema
- Sharding Design
- Sharding Use Cases
- Shard Ownership

**Unit Tests**

```
./mongo --nodb jstests/sharding/*js
```

**Sharding Config Schema**

Sharding configuration schema. This lives in the config servers.

The config server data is the most important data in your entire cluster. Back it up before doing config server maintenance.

**Collections**

version

This is a singleton that contains the current meta-data version number.
> db.getCollection("version").findOne()
    { "_id" : 1, "version" : 3 }

settings

Key/Value table for configurable options (chunkSize/balancer)

> db.settings.find()
    { "_id" : "chunksize", "value" : 64 }
    { "_id" : "balancer", "stopped" : false } //you can stop the balancer

shards

Stores information about the shards (possibly replicases in rsname/list-of-host:port,... ). The addshard command creates these documents, and mongos processes periodically update the seed lists as the replica sets evolve over time.

> db.shards.findOne()
    { "_id" : "shard0", "host" : "localhost:30001" }
    { "_id" : "shard1", "host" : "shard1/localhost:30002" } //shard1 is a replicaset (think of this as a seed list)

databases

{  
    "_id" : "admin",
    "partitioned" : false,
    "primary" : "shard0"
}
collections

Stores meta information about each sharded collection. One entry per sharded collection.

mongos> db.collections.find{}
    {  
        "_id" : "test.foo",
        "dropped" : false,
        "key" : { "_id" : 1 },
        "unique" : false
    }

chunks
mongo

Record of all mongos affiliated with this cluster. mongos will ping every 30 seconds so we know who is alive. This is not used by the system for anything other than reporting.

changelog

Human readable log of all meta-data changes. Capped collection that defaults to 10mb.
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

general concepts

- **config database** - the top level database that stores information about servers and where things live.
- **shard**. this can be either a single server or a replica pair.
- **database** - one top level namespace. a database can be partitioned or not
- **chunk** - a region of data from a particular collection. A chunk can be though of as (collectionname,fieldname,lowvalue,highvalue). The range is inclusive on the low end and exclusive on the high end, i.e., [lowvalue,highvalue).

components and database collections

- **config database**
- **config.servers** - this contains all of the servers that the system has. These are logical servers. So for a replica pair, the entry would be 192.168.0.10,192.168.0.11
- **config.databases** - all of the databases known to the system. This contains the **primary server** for a database, and information about whether its partitioned or not.
  - **config.shards** - a list of all database **shards**. Each shard is a db pair, each of which runs a db process.
  - **config.homes** - specifies which shard is **home** for a given client db.
- **shard databases**
  - **client.system.chunklocations** - the home shard for a given client db contains a **client.system.chunklocations** collection. this
collection lists where to find particular chunks; that is, it maps chunk->shard.

- mongos process
  - "routes" request to proper db's, and performs merges. can have a couple per system, or can have 1 per client server.
  - gets chunk locations from the client db's home shard. load lazily to avoid using too much mem.
    - chunk information is cached by mongos. This information can be stale at a mongos (it is always up to date at the owning shard; you cannot migrate an item if the owning shard is down). If so, the shard contacted will tell us so and we can then retry to the proper location.

**db operations**

- moveprimary - move a database's primary server
- migrate - migrate a chunk from one machine to another.
  - lock and migrate
  - shard db's coordinate with home shard to atomically pass over ownership of the chunk (two phase commit)
  - split - split a chunk that is growing too large into pieces. as the two new chunks are on the same machine after the split, this is really just a metadata update and very fast.
- reconfiguration operations
  - add shard - dbgrid processes should lazy load information on a new (unknown) shard when encountered.
  - retire shard - in background gradually migrate all chunks off

**minimizing lock time**

If a chunk is migrating and is 50MB, that might take 5-10 seconds which is too long for the chunk to be locked.

We could perform the migrate much like Cloner works, where we copy the objects and then apply all operations that happened during copying. This way lock time is minimal.

**Sharding Use Cases**

What specific use cases do we want to address with db 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 shards ownership

**mongod**

The mongod processes maintain a cache of shards the mongod instance owns:
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:

```plaintext
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:

```plaintext
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.

Hosting Center

Database-as-a-Service

- MongoOd.com
- MongoHQ
- MongoLab
- HostedMongo.com
- MongoGrid
- ObjectRocket - MongoDB service provider specializing in high performance and availability.
- MongoIC by GrandCloud (China)

Infrastructure-as-a-Service

- Amazon EC2
- Joyent
- Rackspace Cloud
- Windows Azure VM

Platform-as-a-Service

- alwaysdata
- cloudControl offers a fully managed platform-as-a-service solution with MongoDB as one of their powerful add-ons. Read the blog post MongoDB Setup at cloudControl for more information.
- dotCloud
- Heroku has add-on connectors to allow you to use from MongoDB from Heroku applications
- NodeGrid
- RedHat OpenShift supports MongoDB
- VMware CloudFoundry supports MongoDB
- Windows Azure - see the MongoDB on Azure Worker Roles page

Dedicated Servers

MongoDB runs well on both virtualized and non-virtualized servers.

- ServerBeach offers preconfigured, dedicated MongoDB servers. Blog

VPS

- (mt) Media Temple's (ve) server platform is an excellent choice for easy MongoDB deployment.
- A2 Hosting has a quick installer to add MongoDB to your VPS hosting account. Instructions for running the installer are on A2's wiki
- Dreamhost offers instant configuration and deployment of MongoDB
- LOCUM Hosting House is a project-oriented shared hosting and VDS. MongoDB is available for all customers as a part of their subscription plan.

More

- Linode
- Webfaction
- Presentations

Amazon EC2

- Getting Started on EC2
- Backup, Restore, Verify
- Deployment Notes
  - MongoDB via AWS Marketplace
  - Automating Deployment with CloudFormation
  - Instance Types
    - Installing MongoDB
    - Operating System
  - Networking
    - Port Management
    - Keepalive
  - Storage Configuration
    - EBS Snapshots
  - Securing instances
  - Communication across regions
  - Presentations

MongoDB runs well on Amazon EC2. This page includes some notes in this regard.

Getting Started on EC2

This guide is intended to provide instructions on using the MongoDB AMI to set up production instances of MongoDB across Amazon's Web Services (AWS) EC2 infrastructure.
First, we’ll step through deployment planning (instance specifications, deployment size, etc.) and then we’ll set up a single production node. We’ll use those setup steps to deploy a three node MongoDB replica set for production use. Finally, we’ll briefly cover some advanced topics such as multi-region availability and data backups.

See the Amazon EC2 Quickstart guide for more information.

**Backup, Restore, Verify**

Depending upon the configuration of your EC2 instances, there are a number of ways to conduct regular backups of your data. For specific instructions on backing up, restoring and verifying refer to EC2 Backup & Restore.

**Deployment Notes**

**MongoDB via AWS Marketplace**

If you installed MongoDB via the AWS Marketplace refer to the MongoDB via AWS Marketplace guide to get a development instance up and running. If you are interested in creating a production deployment, refer to the Amazon EC2 Quickstart. Start with the section on Configuring Storage to set up a place for your data to be stored. After that refer to the Starting MongoDB section to get your MongoDB instance running. If you're interested in scaling your deployment, check out the sections on Replica Sets and Sharding.

**Automating Deployment with CloudFormation**

CloudFormation from Amazon Web Services provides an easy mechanism to create and manage a collection of AWS resources. To use CloudFormation you create and deploy a template which describes the resources in your stack via the AWS Management Console. We have created a series of reference templates that you can use as a starting point to build your own MongoDB deployments using CloudFormation. Check out Automating Deployment with CloudFormation for a walkthrough and the template files.

**Instance Types**

MongoDB works on most EC2 types including Linux and Windows. We recommend you use a 64 bit instance as this is required for all MongoDB databases of significant size. Additionally, we find that the larger instances tend to be on the freshest ec2 hardware.

**Installing MongoDB**

One can download a binary or build from source. Generally it is easier to download a binary. We can download and run the binary without being root. For example on 64 bit Linux:

```
[-]$ curl -O http://downloads.mongodb.org/linux/mongodb-linux-x86_64-1.0.1.tgz
[-]$ tar -xzf mongodb-linux-x86_64-1.0.1.tgz
[-]$ cd mongodb-linux-x86_64-1.0.1/bin
[bin]$ ./mongod --version
```

Before running the database one should decide where to put datafiles. Run df -h to see volumes. On some images /mnt will be the many locally attached storage volume. Alternatively you may want to use Elastic Block Store which will have a different mount point.

If you mount the file-system, ensure that you mount with the noatime and nodiratime attributes, for example

```
/dev/mapper/my_vol /var/lib/mongodb xfs noatime,noexec,nodiratime 0 0
```

Create the mongodb datafile directory in the desired location and then run the database:

```
mkdir /mnt/db
./mongod --fork --logpath ~/mongod.log --dbpath /mnt/db/
```

**Operating System**

Occasionally, due to the shared and virtualized nature of EC2, an instance can experience intermittent I/O problems and low responsiveness compared to other similar instances. Terminating the instance and bringing up a new one can in some cases result in better performance.

Some people have reported problems with ubuntu 10.04 on ec2. Please read https://bugs.launchpad.net/ubuntu/+source/linux-ec2/+bug/614853 and https://bugzilla.kernel.org/show_bug.cgi?id=16991
Networking

Port Management

By default the database will now be listening on port 27017. The web administrative UI will be on port 28017.

Keepalive

Change the default TCP keepalive time to 300 seconds. See our troubleshooting page for details.

Storage Configuration

For production use, we recommend raid 10 across 4-8 ebs drives for best performance. Local ec2 drives may be faster but are generally not suitable as they are ephemeral. Multiple ebs drives increase the potential number of random IO's per second (iops), but not necessarily sequential read/write throughput much. In most database applications random iops are important.

For more information, refer to EBS info at Amazon Web Services.

EBS Snapshots

If your datafiles are on a single EBS volume, you can snapshot them for backups.

If you are using journaling, simply take a snapshot (including the journal/ directory).

If not using journaling, you need to use the lock+fsync command (v1.3.1+).

Use this command to lock the database to prevent writes. Then, snapshot the volume. Then use the unlock command to allow writes to the database again. See the full EC2 Backup, Verify & Recovery guide for more information. This method may also be used with slaves / secondaries.

Securing instances

Secure your instances from direct external access by the use of Security Groups. A common approach is to create a MongoDB security group that contains the nodes of your cluster (replica set members or sharded cluster members). Then create a separate group for your app servers or clients.

Create a Rule in your MongoDB security group with the Source field set to the Security Group name containing your app servers and the port set to 27017 (or whatever port you use for your MongoDB). This will ensure that only your app servers have permission to connect to your MongoDB instances.

Communication across regions

Every EC2 instance will have a private IP address that can be used to communicate within the EC2 network. It is also possible to assign a public "elastic" IP to communicate with the servers from another network. If using different EC2 regions, servers can only communicate via public IPs.

To set up a cluster of servers that spans multiple regions, it is recommended to cname the server hostname to the "public dns name" provided by EC2. This will ensure that servers from a different network use the public IP, while the local servers use the private IP, thereby saving costs. This is required since EC2 security groups are local to a region.

For example one solution is following, on each server:

1. set the hostname of the server

   sudo hostname server1

2. install "bind", it will serve as local resolver

3. add a zone for your domain, say "myorg.com", and add the CNAMEs for all your servers

   ... server1 IN CNAME ec2-50-19-237-42.compute-1.amazonaws.com.

4. restart bind and modify /etc/resolv.conf to use the local bind

   search myorg.conf
   nameserver 127.0.0.1

Then:
- verify that you can properly resolve server1, server2, ... using a tool like dig.
- when running mongod, db.serverStatus() should show the correct hostname, e.g. "server1:27017".
- you can then set up replica sets or shards using the simple hostname. For example connect to server1 and run "rs.initiate()", then "rs.add('server2:27017')".

**Presentations**

**MongoDB & AWS** - Free Webinar on January 20, 2012

Presentation from [MongoSV](December 2011)

**More Presentations**

- Running MongoDB in the Cloud - MongoSF (May 2011)
- MongoDB on Amazon EC2 - Webinar (March 2011)

**AWS Marketplace**

- Starting MongoDB
- Security Setup
- Instance Configurations
  - MongoDB AMI
  - MongoDB with EBS RAID AMI
- More Information

If you installed MongoDB via the AWS Marketplace, refer to the instructions below for Starting MongoDB. For information on how the MongoDB with instance-storage AMI was configured, see [MongoDB AMI](MongoDB AMI). For information on how the MongoDB with EBS RAID storage was configured, see [MongoDB with EBS RAID AMI](MongoDB with EBS RAID AMI). The Security Setup section contains information about the security group settings for these AMIs.

**Starting MongoDB**

The instance has been mostly configured, the only steps left are to set MongoDB to start at system boot and then start up:

```
$ sudo chkconfig mongod on
$ sudo /etc/init.d/mongod start
Starting mongod: [ OK ]
forked process: 1234
all output going to: /log/mongod.log
```

In a few moments MongoDB will finish starting up and then you can connect to MongoDB using the mongo client:

```
$ mongo
MongoDB shell version: 2.0.4
connecting to: test
>
```

If you have trouble connecting to the MongoDB daemon, check the log file (either `/var/log/mongo/mongod.log` or `/log/mongod.log`) for mongod console output.

**Security Setup**

By default, the instance starts up with a newly created security group and that group specifies that only access via port 22 (SSH) is allowed. In order to make MongoDB accessible to other instances, you'll need to add an additional rule to your security group specifying access to port 27017 along with a source (this can be a specific IP address, instances from within another security group or from an IP). Start by logging in to the **AWS EC2 Management Console** and navigate to the Security Groups section. Next, find the group that was created when you created your instance and add an Inbound Rule:
After the rule is applied, you'll be able to access MongoDB running in your instance from the source you specified. Consult the AWS EC2 Security Groups documentation for more information about configuring access to your instance with security groups.

**Instance Configurations**

MongoDB AMI

MongoDB was installed onto Amazon Linux using a subset of steps found in the Install MongoDB section of the EC2 Quickstart guide. Specifically, the 10gen repository was created and MongoDB (along with sysstat tools) were then installed via yum. This AMI uses a single EBS-backed volume as storage and does not employ RAID10 or LVM. The data directory was set to `/data` in the MongoDB configuration file `/etc/mongodb.conf`.

MongoDB with EBS RAID AMI

Using the Amazon EC2 Quickstart guide as a base, the EBS RAID AMI was configured using a subset of the steps found in the Deploying a Single Node section. This AMI was configured with block device mapping and uses 4 EBS volumes, each with 100GiB of storage. The volumes were then setup as a single RAID10 device using `mdadm` and 3 logical volumes (data, log, journal) were created using LVM. With RAID10 the total available storage is 200GiB and within that 180GiB is for `/data`, 10GiB for `/log` and 10GiB for `/journal`. The MongoDB configuration file `/etc/mongod.conf` reflects the updated locations for these elements. For more information on the specific steps, refer to the Amazon EC2 Quickstart guide.

**More Information**

For more information on deploying MongoDB on AWS (including advanced storage setups, backups, etc.), refer to the Amazon EC2 page or the Amazon EC2 Quickstart guide.

**Automating Deployment with CloudFormation**

- Template Walkthrough
  - Security
  - Storage Configuration
  - Instance Configuration
- Replica Set Stack
- Customizing Templates
  - Operating System
  - Storage
  - Instances
- Sample Template Usage

CloudFormation from Amazon Web Services provides an easy mechanism to create and manage a collection of AWS resources. To use CloudFormation you create and deploy a template which describes the resources in your stack via the AWS Management Console. CloudFormation templates are simple JSON formatted text files that can be placed under your normal source control mechanisms, stored in private or public locations such as Amazon S3.

We have created a series of reference templates (see attachments below) that you can use as a starting point to build your own MongoDB deployments using CloudFormation. The sample templates show how to build a single node MongoDB deployment and a MongoDB replica set deployment. Refer to the following sections for information on how these sample templates were developed and instructions on how to customize them for your own deployment onto AWS. The documentation provided below focuses on the sections specific to deploying MongoDB on AWS, for background on standard template sections (e.g input parameters) refer to the AWS CloudFormation User Guide.

The sample templates we created are as follows (click to download):

- **MongoDB_SingleNode.template** is an example single-node MongoDB deployment with 4 EBS volumes
- **MongoDB_ReplicaSetStack.template** sets up a single node and references the...
- **MongoDB_ReplicaSetMember.template** twice to create the additional nodes (each node has 4 EBS volumes)
Template Walkthrough

Security

EC2 instances in AWS must have security groups designated during creation that specify firewall rules for each instance. In our templates, we create a simple security group `MongoSecurityGroup` that opens up port 22 (SSH) to the outside world and a separate rule (`AWS::EC2::SecurityGroupIngress`) that’s used to open up the standard `mongodb` port (27017) to other instances within that group:

```json
"MongoSecurityGroup" : {
  "Type" : "AWS::EC2::SecurityGroup",
  "Properties" : {
    "GroupDescription" : "MongoDB security group",
    "SecurityGroupIngress" : [
      {
        "IpProtocol" : "tcp",
        "FromPort" : "22",
        "ToPort" : "22",
        "CidrIp" : "0.0.0.0/0"
      }
    ]
  }
},
"MongoSecurityGroupIngress" : {
  "Type" : "AWS::EC2::SecurityGroupIngress",
  "Properties" : {
    "GroupName" : { "Ref" : "MongoSecurityGroup" },
    "IpProtocol" : "tcp",
    "FromPort" : "27017",
    "ToPort" : "27017",
    "SourceSecurityGroupName" : { "Ref" : "MongoSecurityGroup" }
  }
},
```

Depending on the type of deployment you are creating, you may need to add additional security group ingress rules to open up additional ports, available to your instances or to the outside world (e.g. port 27018 for sharding).

Storage Configuration

When using MongoDB on AWS, we recommend using multiple EBS volumes configured as a single RAID10 storage device for your data. Configuring storage via EBS volumes using CloudFormation requires multiple steps. First the EBS volume must be created, using the `VolumeSize` input value and the same availability zone that we’ll use for our EC2 instance (see Instance Configuration).

```json
"MongoVolume1" : {
  "Type" : "AWS::EC2::Volume",
  "Properties" : {
    "Size" : { "Ref" : "VolumeSize" },
    "AvailabilityZone" : { "Fn::GetAtt" : [ "MongoInstance", "AvailabilityZone" ]}
  }
},
```

The next step is to attach the volume to an EC2 instance. By referencing the instance name in "InstanceId" we ensure that the EBS volumes will be created after the instance is created.

```json
"MongoVolumeMount1" : {
  "Type" : "AWS::EC2::VolumeAttachment",
  "Properties" : {
    "InstanceId" : { "Ref" : "MongoInstance" },
    "VolumeId" : { "Ref" : "MongoVolume1" },
    "Device" : "/dev/sdh1"
  }
},
```

In the attached sample templates we used 4 EBS volumes as the basis for our RAID10 configuration. If you are interested in increasing the
number of EBS volumes, you will need to add additional AWS::EC2::Volume and AWS::EC2::VolumeAttachment resources inside of your template. Refer to the Customizing Storage section below for more information on the steps required.

**Instance Configuration**

The centerpiece of the CloudFormation template is the creation of the EC2 instances that will be used as the MongoDB server. There are two main sections to be configured for your instance: the instance metadata and instance properties. In the sample provided, the metadata contains information about the packages to be installed (mdadm and sysstat) and any files to be created within the instance. In our sample, the only file to be created is a yum repository entry to facilitate MongoDB being installed via yum after the instance has booted.

```json
"MongoInstance" : {
    "Type" : "AWS::EC2::Instance",
    "Metadata" : {
        "AWS::CloudFormation::Init" : {
            "config" : {
                "packages" : {
                    "yum" : {
                        "mdadm" : [],
                        "sysstat" : []
                    },
                },
                "files" : {
                    "/etc/yum.repos.d/10gen.repo" : {
                        "content" : [ "Fn::Join" : ["", [ "[10gen]\n", "name=10gen Repository\n", "baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64\n", "gpgcheck=0\n" ] ] ],
                        "mode" : "000644",
                        "owner" : "root",
                        "group" : "root"
                    }
                }
            }
        }
    }
}
```

For more information about the possible options for the metadata section, refer to the CloudFormation documentation for the AWS::EC2::Instance resource. The properties section in the template is used to specify things like the instance type, AMI, security groups and a script to be run after boot (found in the "UserData" section):

```json
"Properties" : {
    "InstanceType" : { "Ref" : "InstanceType" },
    "ImageId" : { "Fn::FindInMap" : [ "RegionImageScheme", { "Ref" : "AWS::Region" } ],
                  { "Fn::FindInMap" : [ "InstanceTypeArch", { "Ref" : "InstanceType" }, "Arch" ] } },
    "KeyName" : { "Ref" : "KeyName" },
    "UserData" : { "Fn::Base64" : { "Fn::Join" : ["", [ 
        "/bin/bash
", ...]]}
}
```

The instance type is determined by the InstanceType input parameter. The ImageId specifies the AMI to use, which is determined by the chosen instance type (e.g. m1.large) and region in which the instance is launched (e.g. us-east-1). Refer to the "Mappings" section inside the sample templates for a list of the available AMIs.

The "UserData" section shown above contains a bash script that is executed once the instance is launched and available. The first step is install the aws-cfn-bootstrap tool which are used in the script to initialize the instance, signal when errors have occurred and when the the instance creation is complete:
"yum update -y aws-cfn-bootstrap"

"## Error reporting helper function"
"function error_exit"
"{"
"/opt/aws/bin/cfn-signal -e 1 -r "$1" ' "Ref" : "WaitHandleMongoInstance" }, "'\n", "exit 1\n"}, "}\n",

"## Initialize CloudFormation bits"
"/opt/aws/bin/cfn-init -v -s "$ref" AWS::StackName }, " -r MongoInstance",
" --access-key "$ref" HostKeys",
" --secret-key "["Fn::GetAtt": ["HostKeys", "SecretAccessKey"]],
" --region "$ref" AWS::Region }, " > /tmp/cfn-init.log 2>&1 || error_exit $(</tmp/cfn-init.log)"
"
Next we include a series of sleep conditions in case our EBS volumes are not yet available. If you plan to use more than 4 EBS volumes in your CloudFormation template, you should add additional sleep conditions here:

"## Waiting EBS mounts to become available"
"while [ ! -e /dev/sdh1 ]; do echo waiting for /dev/sdh1 to attach; sleep 10; done",
"while [ ! -e /dev/sdh2 ]; do echo waiting for /dev/sdh2 to attach; sleep 10; done",
"while [ ! -e /dev/sdh3 ]; do echo waiting for /dev/sdh3 to attach; sleep 10; done",
"while [ ! -e /dev/sdh4 ]; do echo waiting for /dev/sdh4 to attach; sleep 10; done",

Then we install MongoDB and create the RAID10 device:

"yum -y install mongo-10gen-server > /tmp/yum-mongo.log 2>&1"

"## Create RAID10 and persist configuration"
"mdadm --verbose --create /dev/md0 --level=10 --chunk=256 --raid-devices=4 /dev/sdh1 /dev/sdh2 /dev/sdh3 /dev/sdh4 > /tmp/mdadm.log 2>&1",
"echo "mdadm --detail --scan" | tee -a /etc/mdadm.conf",

With the RAID10 created, we can set some block device attributes (read-ahead) for each storage device:

"## Set read-ahead on each device"
"blockdev --setra 128 /dev/md0"
"blockdev --setra 128 /dev/sdh1"
"blockdev --setra 128 /dev/sdh2"
"blockdev --setra 128 /dev/sdh3"
"blockdev --setra 128 /dev/sdh4"

Now we use LVM to create a series of logical volumes for data, journal and logs. The values used below for each volume are specified as percentages, those may need to be changed for your deployment. After creating the volumes, we create the filesystem, mount points and entries in the filesystem table. The last storage-related step is to set the user:group ownership of each mount point to mongod:mongod.
## Create physical and logical volumes

```
dd if=/dev/zero of=/dev/md0 bs=512 count=1
pvcreate /dev/md0
vgcreate vg0 /dev/md0
lvcreate -l 90%vg -n data vg0
lvcreate -l 5%vg -n log vg0
lvcreate -l 5%vg -n journal vg0
```

## Create filesystems and mount point info

```
mke2fs -t ext4 -F /dev/vg0/data > /tmp/mke2fs1.log 2>&1
mke2fs -t ext4 -F /dev/vg0/log > /tmp/mke2fs2.log 2>&1
mke2fs -t ext4 -F /dev/vg0/journal > /tmp/mke2fs3.log 2>&1
mkdir /data
mkdir /log
mkdir /journal
```

```
echo '/dev/vg0/data /data ext4 defaults,auto,noatime,noexec 0 0' | tee -a /etc/fstab
echo '/dev/vg0/log /log ext4 defaults,auto,noatime,noexec 0 0' | tee -a /etc/fstab
echo '/dev/vg0/journal /journal ext4 defaults,auto,noatime,noexec 0 0' | tee -a /etc/fstab
```

```
mount /data > /tmp/mount1.log 2>&1
mount /log > /tmp/mount2.log 2>&1
mount /journal > /tmp/mount3.log 2>&1
```

```
ln -s /journal /data/journal
```

```
chown -R mongod:mongod /data
chown -R mongod:mongod /log
chown -R mongod:mongod /journal
```

Next we proceed to creating a MongoDB configuration file, specifying the logpath and data directory (among others), and start MongoDB:

```
## Update mongod configuration

```
cat <<EOF > /etc/mongod.conf
logpath=/data/log/mongod.log
logappend=true
fork=true
dbpath=/data
rest=true
EOF
```

## Start mongod

```
/etc/init.d/mongod start > /tmp/mongod-start.log 2>&1
```

The final step is to signal our previously created WaitCondition that setup is complete:

```
## CloudFormation signal that setup is complete

```
/opt/aws/bin/cfn-signal -e 0 -r "MongoInstance setup complete" , { "Ref" : "WaitHandleMongoInstance" }, "\n"
```

Once this script has completed executing, the instance and associated resources have been created and our CloudFormation stack is ready to go.

### Replica Set Stack

The “ReplicaSetStack” sample template first creates two “ReplicaSetMember” instances (complete with storage configuration) and then creates the overall replica set. The “ReplicaSetMember” template is modeled very closely after the “SingleNode” template except it includes additional input parameters and adds additional commands to the instance setup script in “UserData” specific to creating replica set members (adding in the replSet parameter to the MongoDB configuration file).
"## Update mongod configuration

```bash
cat <<EOF > /etc/mongod.conf
logpath=/data/log/mongod.log
logappend=true
fork=true
dbpather=/data
rest=true
replset=, {"Ref" : "ReplicaSetName"}, 
EOF
```

The "ReplicaSetStack" template also closely follows the "SingleNode" template but adds the following replica set specific steps: (1) it creates a "replicaSetConfigInit.js" file containing the replica set configuration (with hostnames for the additional members) and (2) initiates the replica set. These steps are executed just prior to signaling that the instance setup has been completed:

```bash
"cat <<EOF > /tmp/replicaSetConfigInit.js"
config = {_id: "", "_id": 0, host:"$HOSTNAME:27017"},
     {_id: 1, host:"", { "Fn::GetAtt" : ["ReplicaSetMember1", "Outputs.ReplicaSetMemberName"] }},
     {_id: 2, host:"", { "Fn::GetAtt" : ["ReplicaSetMember2", "Outputs.ReplicaSetMemberName"] }},
"rs.initiate(config);"
EOF

"/usr/bin/mongo /tmp/replicaSetConfigInit.js > /tmp/replica-setup.log 2>&1"
```

The child "ReplicaSetMember" instances are created from within the "ReplicaSetStack" template using the following resource definition. The inputs for the "ReplicaSetMember" instances are taken from the "ReplicaSetStack" template:

```json
"ReplicaSetMember1": {
  "Type": "AWS::CloudFormation::Stack",
  "Properties": {
    "TemplateURL": "http://S3URL/MongoDB_ReplicaSetMember.template",
    "Parameters": {
      "KeyName": { "Ref": "KeyName" },
      "InstanceType": { "Ref": "InstanceType" },
      "VolumeSize": { "Ref": "VolumeSize" },
      "AccessKeyId": { "Ref": "AccessKey" },
      "SecretAccessKey": { "Fn::GetAtt": ["AccessKey", "SecretAccessKey"] },
      "ReplicaSetName": { "Ref": "ReplicaSetName" },
      "SecurityGroupName": { "Ref": "MongoSecurityGroup" },
      "InstanceZone": { "Fn::FindInMap": ["RegionZone", { "Ref": "AWS::Region" }, "AZ1"] }
    }
  }
}
```

### Customizing Templates

#### Operating System

In the sample templates provided, we used Amazon Linux as the base operating system. If you are interested in using a recent release of Ubuntu (9.10 or greater), please use the following steps to customize the templates. The steps refer to the line numbers for the MongoDB_ReplicaSetStack.template but changes should also be made in MongoDB_ReplicaSetMember.template.

First, update the "files" content of the EC2 instance Metadata section to use the 10gen apt source:
Then in the UserData section of the EC2 instance configuration, make the following changes:

- Line 220: change `yum -y install mongo-10gen-server...` to `apt-get install mongodb-10gen`
- Line 221: add `service mongodb stop`
- Line 260, 261, 262: update to use `mongodb:mongodb` as the owner:group
- Line 265: update to `cat <<EOF > /etc/mongodb.conf`
- Line 275: change `/etc/init.d/mongod start...` to `service mongodb start`

Storage

In the sample templates provided, we used 4 EBS volumes as the basis for our RAID10 configuration. If you are interested in using additional volumes you will need to update the following items for each new volume you want to add:

- Add an "AWS::EC2::Volume" resource
- Add an "AWS::EC2::VolumeAttachment" resource and mount point
- Add an additional sleep condition
- Update the call to `mdadm` to include your additional volumes

Instances

The sample "ReplicaSetStack" template creates three instances, one from the "stack" template and two additional "ReplicaSetMember" instances, via the "AWS::CloudFormation::Stack" resource in the "ReplicaSetStack" template. If you are interested in adding additional replica set members, you'll need to create an additional member instances and edit the `replicaSetConfigInit.js` found within the "ReplicaSetStack" template. Refer to the Replica Set Stack for information about the additional resources and config file to be updated. When creating the replica set the templates spread the created instances across multiple availability zones (e.g. `us-east-1a` or `us-east-1b`). When adding additional instances be sure to specify your desired Availability Zone for increase redundancy.

Sample Template Usage

If you are interesting in launching a single MongoDB node in AWS, download the `MongoDB_SingleNode.template` file and edit it for your specific deployment. Once you have a completed template, login to the AWS Management Console and navigate to the "AWS CloudFormation" and "Create New Stack". There you'll be prompted to upload your template and input the necessary parameters.

If instead you are interested in launching a multi-node replica set, download the `MongoDB_ReplicaSetStack.template` and `MongoDB_ReplicaSetMember.template`. In order for a parent template ("ReplicaSetStack") to refer to child templates ("ReplicaSetMember"), the child template must be uploaded to S3 and the S3 URL of the child template must be specified in the parent template. Once you've uploaded the child template to S3, update the "TemplateURL" parameter in each "ReplicaSetMember" resource in the "ReplicaSetStack" template:
After updating the "TemplateURL" parameters, login to the AWS Management Console and navigate to the "AWS CloudFormation" and "Create New Stack". There you'll be prompted to upload your template and input the necessary parameters.

For more information on deploying MongoDB on AWS, refer to the Amazon EC2 page and the Amazon EC2 Quickstart guide.

Amazon EC2 Quickstart

- Prerequisites
- Planning Your Deployment
  - Instance Specifications
  - Storage Configuration
  - Topology
  - Security
- Securing Your Deployment
- Deploying a Single Node
  - Launch Instance
  - Configure Storage
  - Install and Configure MongoDB
  - Starting MongoDB
- Deploying a Multi-node Replica Set
  - Replica Set Background
  - Create and Configure Instances
  - Configure Replica Set
- Deploying a Sharded Configuration
  - Simple Sharding Architecture
- Backup and Restore

This guide is intended to provide instructions on setting up production instances of MongoDB across Amazon's Web Services (AWS) EC2 infrastructure.

First, we'll step through deployment planning (instance specifications, deployment size, etc.) and then we'll set up a single production node. We'll use those setup steps to deploy a three node MongoDB replica set for production use. Finally, we'll briefly cover some advanced topics such as multi-region availability and data backups.

If you installed MongoDB via the AWS Marketplace this guide can be used to get your instance up and running quickly. Start with the section on Configuring Storage to set up a place for your data to be stored. After that refer to the Starting MongoDB section to get your instance started and allow you to get started using MongoDB. If you're interested in scaling your deployment, check out the sections on Replica Sets and Sharding below.

Prerequisites

Generally, there are two ways to work with EC2 - via the command line tools or the AWS Management Console. This guide will use the EC2 command line tools to manage security and storage and launch instances. Use the following steps to setup the EC2 tools on your system:

- Download the EC2 command line tools
- Next, refer to Prerequisites and Setting Up the Tools from Amazon's Getting Started with the Command Line Tools

Planning Your Deployment

Before starting up your EC2 instances, it's best to sketch out a few details of the planned deployment. Regardless of the configuration or the number of nodes in your deployment, we'll configure each one in roughly the same manner.

Instance Specifications

Amazon has several instance choices available ranging from low to high (based on CPU and memory) throughput. Each instance available serves a different purpose and plays a different role when planning your deployment. There are several roles to consider when deploying a MongoDB production cluster. Consider a situation where your deployment contains an even number of replicated data (mongod) instances, an arbiter participates in electing the primary but doesn't hold any data. Therefore a Small instance may be appropriate for the arbiter role but for data nodes you'll want to use 64-bit (standard Large or higher) instances, which have greater CPU and memory scaling. For the purposes of this guide we'll be focused on deploying mongod instances that use the standard Large instance. The AMI (ID: ami-41814f28) is the 64-bit base Amazon Linux, upon which we'll install MongoDB.

Storage Configuration

For storage we recommend using multiple EBS volumes (as opposed to instance-based storage which is ephemeral) in a RAID-based configuration. Specifically for production deployments you should use RAID 10 across 4-8 EBS volumes for the best performance. When deploying RAID 10, you'll need enough volume storage to be twice that of the desired available storage for MongoDB. Therefore for 8 GiB of available storage you'll need to have 16 GiB of allocated storage space across multiple EBS volumes.

Topology

For the purposes of this guide, our topology will be somewhat simple: one to three EC2 instances, each with multiple EBS volumes attached, all
located in the same availability zone (and by extension, within the same region). If you are interested in creating a deployment that spans availability zones or regions, it's best to do that planning up front and take into account security group designations (they cannot span regions) and hostname/DNS access (AWS internal IP addresses can only be used within a zone).

An example 3 node replica set with RAID 10 storage, spanning multiple availability zones would like similar to the following. Availability zones within EC2 are similar to different server racks, therefore it is recommended that you deploy your replica set across multiple zones.

![Diagram of 3 node replica set spanning multiple availability zones]

For even greater redundancy and failover, you could also deploy your replica set across multiple regions (and go further with multiple zones in each region):

![Diagram of replica set across multiple regions]

Refer to the AWS documentation on Using Regions and Availability Zones for more information.

**Security**

The recommended approach to securing your instances is to use multiple security groups for your MongoDB deployment, one for each type of interaction. For example, you could use one group to manage communication amongst the nodes in your cluster, another group that allows your application to communicate with the database and optionally, a group for tools and maintenance tasks.

An example setup with two security groups might look like this:

![Diagram of security groups in a MongoDB deployment]

Refer to the AWS documentation on Using Regions and Availability Zones for more information.
Securing Your Deployment

Before starting up instances we want to get the security groups created. As previously discussed, we recommend using multiple groups, one for each type of interaction. The following steps will show you how to create two groups (one for your app and another for your database) and provide the authorizations necessary for communication between them.

From the command line, create the database group and authorize SSH:

```
$ ec2-create-group database --description "security group for database"
GROUP   sg-08146660 database    security group for database

$ ec2-authorize database -p 22
GROUP   database
PERMISSION   database    ALLOWS  tcp 22 22 FROM CIDR 0.0.0.0/0 ingress
```

Authorize communication within the group of MongoDB instances by adding the group to itself. Note you'll need to provide the user account number (using the -u flag) when authorizing groups:

```
$ ec2-authorize database -o database -u [AWS_ACCOUNT_NUMBER]
GROUP   database
PERMISSION   database    ALLOWS  tcp 0 65535 FROM USER AWS_ACCOUNT_NUMBER NAME database ingress
PERMISSION   database    ALLOWS  udp 0 65535 FROM USER AWS_ACCOUNT_NUMBER NAME database ingress
PERMISSION   database    ALLOWS  icmp -1 -1 FROM USER AWS_ACCOUNT_NUMBER NAME database ingress
```

Optionally, for testing you could also enable the port for the MongoDB web-based status interface (port 28017):

```
$ ec2-authorize database -p 28017
GROUP   database
PERMISSION   database    ALLOWS  tcp 28017 28017 FROM CIDR 0.0.0.0/0 ingress
```

Now create a group that will hold application servers, which will communicate with the database cluster:

```
$ ec2-create-group application --description "security group for application servers"
GROUP   sg-081bf960 application security group for application servers
```

Finally, authorize communication from the application servers (group application) to the MongoDB instances (group database):

```
$ ec2-authorize database -o application -u [AWS_ACCOUNT_NUMBER]
GROUP   database
PERMISSION   database    ALLOWS  tcp 0 65535 FROM USER AWS_ACCOUNT_NUMBER NAME application ingress
PERMISSION   database    ALLOWS  udp 0 65535 FROM USER AWS_ACCOUNT_NUMBER NAME application ingress
PERMISSION   database    ALLOWS  icmp -1 -1 FROM USER AWS_ACCOUNT_NUMBER NAME application ingress
```

Refer to the AWS guide Using Security Groups for more information on creating and managing security groups.

The next step is to generate an SSH key-pair that we'll use to connect to our running EC2 instances. Amazon's tools provide a mechanism to quickly generate a public-private key pair. Once generated, we'll need to save the private key so that we can use it to connect via SSH later (click here for more info on key pairs and AWS).

First, generate the key pair:
Save the contents of the key to a file (including the BEGIN and END lines) and make sure that file is only readable by you:

```bash
$ chmod 600 private-key-file
```

Optionally, you can also the key to the SSH agent to ease connecting to our instances later:

```bash
$ ssh-add private-key-file
```

We're finished with the pre-deployment steps; we've covered the storage and security considerations that's necessary to setup and deploy our instances.

## Deploying a Single Node

We'll start our deployment by setting up single node because later on we'll use the same steps to set up a larger cluster. The first step is to launch the instance and then setup the EBS-backed RAID 10 storage for the instance. Setting up the storage requires creating, attaching, configuring and formatting the volumes where our data will be stored.

**Note:** If you created a MongoDB instance via the AWS Marketplace, skip ahead to Configure Storage below.

### Launch Instance

From the command line we can launch the instance. We'll need to supply an ID for an Amazon Machine Image (AMI) that we'll build our node from. We recommend using a 64-bit Amazon Linux AMI as the base of your MongoDB nodes. In this example, we are using `ami-e565ba8c` with the number of nodes (1), security group (`database`), authentication keypair (`cluster-keypair`), type of instance (`m1.large`) and availability zone (`us-east-1a`). Depending on the region you deploy into, a different AMI ID may be needed:

```bash
$ ec2-run-instances ami-e565ba8c -n 1 -g database -k cluster-keypair -t m1.large -z us-east-1a
```

Next, let's add some tags to the instance so we can identify it later. Tags are just metadata key-value pairs:

```bash
$ ec2-create-tags i-11eee072 --tag Name=QuickstartTestNode --tag Owner=Bob
```

Now we can ascertain some status information about running instances at AWS (includes EBS volumes as well):

```bash
$ ec2-create-tags i-11eee072 --tag Name=QuickstartTestNode --tag Owner=Bob
```
Configure Storage

Now that the instance is running, let's create the EBS volumes we'll use for storing our data. In this guide we'll set up 4 volumes with 4 GiB of storage each (configured that's 16 GiB but because we're using a RAID 10 configuration that will become 8 GiB of available storage).

First off, create the EBS volumes supplying the size (4) and zone (us-east-1a) and save the results into a temporary file that we'll read from for the next command. Here's the command we'll use:

```bash
$ for x in {1..4}; do \
  > ec2-create-volume -s 4 -z us-east-1a; \
  > done > vols.txt
```

Here's the output of that command:

```
$ more vols.txt
VOLUME  vol-e796108a    4               us-east-1a      creating        2011-12-07T04:37:21+0000
VOLUME  vol-c39610ae    4               us-east-1a      creating        2011-12-07T04:37:30+0000
VOLUME  vol-a19610cc    4               us-east-1a      creating        2011-12-07T04:37:39+0000
VOLUME  vol-b19610dc    4               us-east-1a      creating        2011-12-07T04:37:47+0000
```

Now, let's attach those newly created volumes to our previously launched running instance from above. From the command line we'll start with the temp file (vols.txt), the running instance ID (i-11eee072), and a prefix for each attached device (/dev/sdh):

```bash
$ (i=0; \
  > for vol in $(awk '{print $2}' vols.txt); do \
  >  i=$((i+1)); \
  >  ec2-attach-volume $vol -i i-11eee072 -d /dev/sdh$i; \
  >  done)
```

Assuming the volumes attached successfully, you should see something like this:

```
ATTACHMENT  vol-e796108a    i-11eee072  /dev/sdh1   attaching   2011-12-07T04:48:22+0000
ATTACHMENT  vol-c39610ae    i-11eee072  /dev/sdh2   attaching   2011-12-07T04:48:29+0000
ATTACHMENT  vol-a19610cc    i-11eee072  /dev/sdh3   attaching   2011-12-07T04:48:37+0000
ATTACHMENT  vol-b19610dc    i-11eee072  /dev/sdh4   attaching   2011-12-07T04:48:44+0000
```

Now we'll need to connect to the running instance via SSH and configure those attached volumes as a RAID array. If you added the private key to your running SSH agent, you should be able to connect with something like (substituting your instance's hostname):

```
$ ssh ec2-user@ec2-a-b-c-d.amazonaws.com
```

And now create the RAID array using the built-in mdadm. You'll need the level (10), number of volumes (4), name of the new device (/dev/md0) and the attached device prefix (/dev/sdh*):
Once `mdadm` is done and we've persisted the storage configuration, we'll need to tune the EBS volumes to achieve desired performance levels. This tuning is done by setting the "read-ahead" on each device. For more information refer to the `blockdev` man page.

```bash
$ sudo blockdev --setra 128 /dev/md0
$ sudo blockdev --setra 128 /dev/sdh1
$ sudo blockdev --setra 128 /dev/sdh2
$ sudo blockdev --setra 128 /dev/sdh3
$ sudo blockdev --setra 128 /dev/sdh4
```

With the RAID10 created we now turn to the Logical Volume Manager (`lvm`) which we'll use to create logical volumes for the data, log files and journal for MongoDB. The purpose of using `lvm` is to (1) safely partition different objects from each other and (2) provide a mechanism that we can use to grow our storage sizes later. First we start by zeroing out our RAID, creating a physical device designation and finally a volume group for that device.

```bash
$ sudo dd if=/dev/zero of=/dev/md0 bs=512 count=1
$ sudo pvcreate /dev/md0
$ sudo vgcreate vg0 /dev/md0
```

Once the volume group has been created, we now create logical volumes for the data, logs and journal. Depending upon the amount of available storage you may want to designate specific sizes vs. volume group percentages (as shown below). We recommend approximately 10GB for log storage and 10GB for journal storage.

```bash
$ sudo lvcreate -l 90%vg -n data vg0
$ sudo lvcreate -l 5%vg -n log vg0
$ sudo lvcreate -l 5%vg -n journal vg0
```

At this point we have three volumes to configure (/dev/vg0/...). For each volume we'll create a filesystem, mount point and an entry in the filesystem table. In the example below we used the `ext4` filesystem however you could instead elect to use `xfs`, just be sure to edit the `mke2fs` and `sed` commands accordingly. The `/etc/fstab` entries require the partition (e.g. `/dev/vg0/data`), a mount point for the filesystem (`/data`), the filesystem type (`ext4` or `xfs`) and the mount parameters (defaults, auto, noatime, noexec, nodiratime 0 0, refer to the `mount` man page for more information on these parameters:

```bash
$ sudo mke2fs -t ext4 -F /dev/vg0/data
$ sudo mke2fs -t ext4 -F /dev/vg0/log
$ sudo mke2fs -t ext4 -F /dev/vg0/journal
$ sudo mkdir /data
$ sudo mkdir /log
$ sudo mkdir /journal
$ echo '/dev/vg0/data /data ext4 defaults,auto,noatime,noexec 0 0' | sudo tee -a /etc/fstab
$ echo '/dev/vg0/log /log ext4 defaults,auto,noatime,noexec 0 0' | sudo tee -a /etc/fstab
$ echo '/dev/vg0/journal /journal ext4 defaults,auto,noatime,noexec 0 0' | sudo tee -a /etc/fstab
```

Now mount all of the storage devices. By adding the entry to `/etc/fstab`, we've shortened the call to mount because it will look in that file for the extended command parameters.

```bash
$ sudo mount /data
$ sudo mount /log
$ sudo mount /journal
```
With the devices mounted we issue one last call to set the MongoDB journal files to be written to our new journal device, via a symbolic link to the new device:

```bash
$ sudo ln -s /journal /data/journal
```

**Install and Configure MongoDB**

**Note**: If you created a MongoDB instance via the AWS Marketplace, skip ahead to Starting MongoDB below.

Now that the storage has been configured, we need to install and configure MongoDB to use the storage we've set up, then set it to start up on boot automatically. First, add an entry to the local `yum` repository for MongoDB:

```bash
$ echo "[10gen]
name=10gen Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64
gpgcheck=0" | sudo tee -a /etc/yum.repos.d/10gen.repo
```

Next, install MongoDB and the `sysstat` diagnostic tools:

```bash
$ sudo yum -y install mongo-10gen-server
$ sudo yum -y install sysstat
```

Set the storage items (data, log, journal) to be owned by the user (`mongodb`) and group (`mongodb`) that MongoDB will be starting under:

```bash
$ sudo chown mongodb:mongodb /data
$ sudo chown mongodb:mongodb /log
$ sudo chown mongodb:mongodb /journal
```

Now edit the MongoDB configuration file and update the following parameters:

```bash
$ sudo nano /etc/mongod.conf
...
logpath=/log/mongodb.log
logappend=true
fork=true
dbpath=/data
...
```

**Starting MongoDB**

Set the MongoDB service to start at boot and activate it:

```bash
$ sudo chkconfig mongod on
$ sudo /etc/init.d/mongod start
```

When starting for the first time, it will take a couple of minutes for MongoDB to start, setup it's storage and become available. Once it is, you should be able to connect to it from within your instance:

```bash
$ mongo
MongoDB shell version: 2.0.4
connecting to: test
>
```
Just to confirm the system is working correctly, try creating a test database, test collection and save a document:

```mongodb
> use testdb
switched to db testdb
> db.createCollection("testCollection")
{ "ok" : 1 }
> db.testCollection.save({"name":"bob"})
> db.testCollection.find()
{ "_id" : ObjectId("4edfda1c86176ab8e27ee976"), "name" : "bob" }
```

Now that we’ve got a single node up and running with EBS backed RAID storage, let’s move on and create a multi-node replica set.

### Deploying a Multi-node Replica Set

**Replica Set Background**

Replica sets are a form of asynchronous master/slave replication, adding automatic failover and automatic recovery of member nodes. A replica set consists of two or more nodes that are copies of each other (i.e.: replicas). Refer to the [MongoDB Replica Set](https://docs.mongodb.com/manual/core/repl-set) documentation for more information.

**Create and Configure Instances**

For this guide, we’ll set up a three node replica set. To set up each node use the instructions from Deploying a Single Node above. Once that’s completed, we’ll update the configurations for each node and get the replica set started.

First we’ll need to edit the MongoDB configuration and update the `replSet` parameter:

```
$ sudo nano /etc/mongod.conf
...
replSet=exampleReplicaSetName
...
```

Save the configuration file and restart `mongod`:

```
$ sudo /etc/init.d/mongod restart
```

**Configure Replica Set**

Once MongoDB has started and is running on each node, we’ll need to connect to the desired primary node, initiate the replica set and add the other nodes. First connect to the desired primary via SSH and then start mongo to initiate the set:

```
$ mongo
MongoDB shell version: 2.0.4
connecting to: test
> rs.initiate()
{ 
  "info2" : "no configuration explicitly specified -- making one",
  "me" : "ip-10-127-127-91:27017",
  "info" : "Config now saved locally. Should come online in about a minute.",
  "ok" : 1
}
```

Next, add the other nodes to the replica set:
The 3 node replica set is now configured. You can confirm the setup by checking the health of the replica set:

```
> rs.status()
{
  "set" : "exampleReplicaSet",
  "date" : "Tue Dec 06 2011 11:39:08 GMT-0500 (CDT)",
  "myState" : 1,
  "members" : [
    {
      "name" : "ip-10-127-127-91:27017",
      "self" : true,
    },
    {
      "name" : "ec2-107-21-46-145.compute-1.amazonaws.com:27017",
      "health" : 1,
      "uptime" : 101,
      "lastHeartbeat" : "Tue Dec 06 2011 11:39:07 GMT-0500",
    },
    {
      "health" : 1,
      "uptime" : 107,
      "lastHeartbeat" : "Tue Dec 06 2011 11:39:07 GMT-0500",
    }
  ],
  "ok" : 1
}
```

What we’ve completed here is a simple replica set; there are additional configurations out there that may make more sense for your deployment, refer to the MongoDB documentation for more information. If you intend to use your replica set to help scale read capacity, you’ll also need to update your application’s code and add the appropriate `slaveOk=true` where necessary so that read results can be returned from additional nodes more quickly.

**Deploying a Sharded Configuration**

MongoDB scales horizontally via a partitioned data approach known as sharding. MongoDB provides the ability to automatically balance and distribute data across multiple partitions to support write scalability. For more information, refer to the MongoDB sharding docs.

**Simple Sharding Architecture**

To build our simple sharded configuration, we’ll be building upon the replica set steps we just worked on. To get started you’ll need to create two additional replica set configurations, just the same from above. When configuring each server instance we’ll set the `shardsvr` parameter inside the `mongod` configuration file. Next we’ll take one node from each replica set and set it to run as a config server as well. The config server maintains metadata about the sharded data storage cluster. Finally, we’ll add instances for the `mongos` router, which handles routing requests from your app to the correct shard. The recommended approach is to run this component on your application servers. The following image shows a recommended topology for use with sharding:
Use the instructions from Deploying a Single Node above to create the required nodes for each replica set (3 instances per set, for a total of 9 instances). Now start configuring the replica set but before saving /etc/mongod.conf, add this parameter:

```
shardsvr = true
```

Save the configuration file and restart mongod:

```
$ sudo /etc/init.d/mongod restart
```

Once /etc/mongod.conf has been updated, initiate the replica set and add the members as described in Configure Replica Set. Once that's complete, choose one instance from each replica set and start an additional mongod process those instances, this time as the config server component:

```
$ mongod --configsvr
```

Now that we've got N config servers running (where N is the number of running replica sets) we can set up the request router mongos. This process typically runs on your application servers and handles routing database requests from your app to the correct database shard. Assuming you already have your application configured and deployed, use ssh to connect to each instance and install MongoDB using the steps from Install and Configure MongoDB.

Before we continue, it is important to consider the role DNS plays in a sharding setup. Generally we recommend using DNS hostnames for configuring replica sets, which Amazon handles appropriately, as opposed to using specific IP addresses. Essentially, AWS knows the mapping between public and private addresses and hostnames and manages inter-region domain name resolution. Therefore, by using the public DNS name for our servers we can ensure that whether our servers are in a single region or across multiple regions, AWS will correctly route our requests. When it comes to setting up sharding, we recommend an additional step of using DNS aliases for the instances that will be acting as config servers. The routers must know the hostnames of the config servers so by using DNS aliases we gain additional flexibility if config servers ever need to change. All it takes is pointing the DNS alias to another instance and no additional update to the router configuration is needed. For more information on this topic, refer to docs on changing config servers.

Once the DNS settings have taken effect, we can proceed with the mongos setup. Go back to the command line on each of the instances you'll be using for mongos and start the service and point it to the instances running the config server using their DNS aliases (ex: alias1.yourdomain.com, alias2.yourdomain.com and alias3.yourdomain.com) along with the config server port 27019:

```
```

With the mongos routers running, we can now complete the setup for the sharding architecture. The last step is to add the previously created replica sets to the overall system. Choose one of the instances that is running mongos, start the mongo client using the hostname and port (27017?) and connect to the admin database. You'll need to have the name for each replica set (ex: replicaSetName1) and the hostnames for each member of the set (e.g: replicaSetHost1)
The `addShard` command will need to be repeated for each replica set that is part of the sharded setup:

```sql
> db.adminCommand({addShard:
  "replicaSetName2/replicaSetHost4:27018,replicaSetHost5:27018,replicaSetHost6:27018"})
> db.adminCommand({addShard:
  "replicaSetName3/replicaSetHost7:27018,replicaSetHost8:27018,replicaSetHost9:27018"})
```

Once these steps have been completed, you’ll have a simple sharded configuration. The architecture we used includes 3 database shards for write scalability and three replicas within each shard for read scalability and failover. This type of setup deployed across multiple regions (ex: one node from each replica located in `us-west-1`) would also provide some degree of disaster recovery as well.

In order to utilize this newly created configuration, you’ll need to specify which databases and which collections are to be sharded. See Enabling Sharding on a Database and Sharding a Collection for more information.

**Backup and Restore**

There are several ways to backup your data when using AWS, refer to the EC2 Backup & Restore guide for more information.

**dotCloud**

**Running MongoDB on dotCloud**

MongoDB can run on dotCloud. It supports replica sets, and has alpha support for sharding.

The whole point of dotCloud is to run your apps and your databases in the same place, to optimize for latency and reliability. However, you can also deploy MongoDB on dotCloud and use it to power an app running anywhere else.

If you don’t have a dotCloud account yet...

Well, what are you waiting for? 😊

Go ahead and create one (it’s free!) and install the CLI:

```sql
sudo easy_install pip ; sudo pip install dotcloud
```

If you need help to get the CLI running, check the dotCloud install docs and don’t hesitate to ask for help.

**With a dotCloud account**

The following snippet will deploy MongoDB on dotCloud for you in no time:

```sql
mkdir mongodb-on-dotcloud
cat >mongodb-on-dotcloud/dotcloud.yml <<EOF
db:
  type: mongodb
EOF
dotcloud pushmongorocks mongodb-on-dotcloud
dotcloud info mongorocks.db
```

The last command will show you the host, port, and credentials to be used to connect to your database.
Scaling

Assuming you followed the instructions of the previous section, if you want to get a replica sets of 3 servers:

```bash
dotcloud scale mongorocks db=3
```

Advanced use

If you want to have a closer look at your MongoDB server, nothing beats SSH access:

```bash
dotcloud ssh mongorocks.db
```

Moar docs

- dotCloud documentation for the MongoDB service
- generic introduction to dotCloud (in case you want to run not only MongoDB, but also Node.js, Python, Ruby, Perl, Java, RabbitMQ, Redis, MySQL, PostgreSQL, CouchDB, Riak, Erlang, or something else, on dotCloud)

Ready-to-use apps

All you need to do to run them is a `git clone` and a `dotcloud push`:

- Django setup using MongoDB to store objects
- MongoDB + Node.js sample app

Getting help

dotCloud has a Q&A site, and the dotCloud team can be reached through the FreeNode IRC network on #dotcloud.

Joyent

For the quickest start, you can use the Joyent SmartMachine for MongoDB Appliance

For installing MongoDB on a Joyent Node Smart Machine, see this article

The prebuilt MongoDB Solaris 64 binaries work with Joyent accelerators.

Some newer gcc libraries are required to run -- see sample setup session below.

```
$ # assuming a 64 bit accelerator
$ /usr/bin/isainfo -kv
64-bit amd64 kernel modules

$ # get mongodb
$ # note this is 'latest' you may want a different version
$ curl -O http://downloads.mongodb.org/sunos5/mongodb-sunos5-x86_64-latest.tgz
$ gzip -d mongodb-sunos5-x86_64-latest.tgz
$ tar -xf mongodb-sunos5-x86_64-latest.tar
$ mv "mongodb-sunos5-x86_64-2009-10-26" mongo
$ cd mongo

$ # get extra libraries we need (else you will get a libstdc++.so.6 dependency issue)
$ curl -O http://downloads.mongodb.org/s3.amazonaws.com/sunos5/mongo-extra-64.tgz
$ gzip -d mongo-extra-64.tgz
$ tar -xf mongo-extra-64.tar
$ # just as an example - you will really probably want to put these somewhere better:
$ export LD_LIBRARY_PATH=mongo-extra-64
$ bin/mongod --help
```
Rackspace Cloud

- Deployment Planning
  - Instance sizes
  - Topology
  - Security
  - Monitoring
- Deploying a Single Node
  - Creating an Instance
  - Securing Your Instance
  - Installing MongoDB
  - Configuring MongoDB
- Replica Sets
- Backups
  - Cloud Servers Images
  - Built-in Tools

This guide is intended to provide instructions on getting started with MongoDB using Rackspace Cloud Servers.

First we'll step through deployment planning (instance sizes, topology, security) and then we'll set up a single MongoDB node. We'll use the same steps to create a multi-node replica set and cover the steps needed to backup your database.

**Deployment Planning**

**Instance sizes**

Rackspace Cloud offers instances with RAM ranging from 256 MB up to 64 GB. When considering instances for initial development purposes, those with 256 or 512 MB are an appropriate starting point. As development progresses towards production-level deployments we recommend moving to higher memory instances. The top-end Cloud instances are appropriate for pre-production and some production deployments. If you need to grow your deployment beyond the Cloud instances, Rackspace also offers managed dedicated servers that can be scaled to increase database performance and throughput.

When planning your deployment, it’s important to account for your resource needs for today and into the future. If you plan on growing your production systems into Rackspace’s dedicated services then it may be useful to consider using RackConnect. This is a service that bridges Rackspace’s public cloud infrastructure with their private dedicated servers. It makes migrating from Cloud to dedicated much easier and is also useful when creating a hybrid deployment solution combining resources from both. For more information refer to Rackspace’s RackConnect site.

**Topology**

The following are example deployment scenarios for MongoDB built on Rackspace servers.

The single node example is the simplest; a single MongoDB instance can be deployed in Rackspace Cloud or Managed hosting. This deployment is useful for development and early app testing.

For production-level deployments, we'll move to a three node Replica Set running in Rackspace Cloud.
Depending on the size of your database deployment or if your app requires greater levels of database performance, another option is a hybrid deployment. In this case, a Replica Set which spans Rackspace Cloud and Managed hosting can be deployed. Cloud and Managed hosting are “connected” via RackConnect, providing a faster interconnect between the two hosting services.

Security

When deploying onto Rackspace Cloud, you'll need to secure access to your instances so that MongoDB is only available to known safe systems. To secure your instances we'll use the `iptables` command-line tool to control access to the system with a firewall. Once we've completed securing our instance, it will be using the following firewall configuration:

- port 22 (SSH) will accept any incoming TCP connections
- All other ports will reject incoming TCP connections

Later on when we deploy multiple instances we'll open up port 27017 (for mongod) to other specific IP addresses in order to facilitate a working replica set. The ports required for sharding (27018 and 27019) aren't required immediately since we'll be setting up a single node and replica set however just remember to open them up later if necessary.

Monitoring

Monitoring is a critical component of all database servers. MongoDB includes several tools to help gather statistics about data sizes, index sizes and analyze queries. In addition, there are several third-party tools that exist to help integrate MongoDB information into other monitoring solutions. 10gen also offers MongoDB Monitoring Service, a free hosted SaaS monitoring solution that provides custom dashboards and reporting for your MongoDB instances. For more information, refer to Monitoring and Diagnostics.

Deploying a Single Node

Creating an Instance

Log in to the Rackspace Cloud control panel and navigate to Hosting > Cloud Servers > Add Server. In this example we used Ubuntu 11.10 (Oneiric Ocelot) as the OS image for our instance. Once selected, enter a server name, select a server size and continue. Be sure to record your
root password or you'll have to reset it later. Once the instance is available, connect to it via SSH to secure it. By default, you'll be connecting to the server as root. If you'd like to create additional users/groups to use with your instance, consult the documentation for the chosen OS. For our configuration, we used the admin group and added an additional user, per the Ubuntu 11.10 server guide.

First we added a new user (admin) to the admin group:

```
root# adduser admin --ingroup admin
```

After stepping through the prompts to set a password (and other account information), the admin user will be created and added to the admin group.

At this point, log out of the server as root and login as admin.

Securing Your Instance

As discussed above in Security, we'll use a simple firewall setup to secure access to the instances running MongoDB. The built-in ufw command provides a simple mechanism to configure the system firewall. First off, enable the firewall and add a rule to allow SSH access:

```
$ sudo ufw enable
$ sudo ufw allow 22
```

Now the only port that will allow connections is port 22, attempts to connect on any other port will be dropped. With this firewall setup, the instance is secure and we can proceed with configuring MongoDB. For more information, refer to the Rackspace Introduction to IPTables and the Ubuntu firewall guide.

Installing MongoDB

Using the instructions from the MongoDB documentation on Ubuntu and Debian packages we added an additional apt-get source and installed MongoDB.

First add the 10gen GPG key to apt-get to create a "trusted" source:

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

Next, add the following line to /etc/apt/sources.list:

```
$ sudo nano /etc/apt/sources.list
... 
... 
deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen 
... 
```

Now update apt-get to pickup the new packages:

```
$ sudo apt-get update
```

Finally, install MongoDB:

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

At this point MongoDB will be installed and started, which we can confirm via the service utility:

```
$ sudo service mongodb status
mongodb start/running, process 2308
```
You can also check the status by using `tail` to examine the MongoDB log:

```
$ sudo tail -f /var/log/mongodb/mongodb.log
```

### Configuring MongoDB

By default, MongoDB is set to use `/var/lib/mongodb` as the data path. If you'd like to amend that, or update the log path, shutdown the MongoDB instance and update `/etc/mongodb.conf` with any changes.

If you change the MongoDB `dbpath` or `logpath`, be sure to set the proper ownership, etc. For example, to change MongoDB to use `/data` as the data path, use the following steps.

First, shutdown the MongoDB service:

```
$ sudo service mongodb stop
```

Next, update the MongoDB configuration file:

```
$ sudo nano /etc/mongodb.conf
...
  dbpath=/data
...
```

Now create the `/data` directory and update it's ownership:

```
$ sudo mkdir /data
$ sudo chown mongodb:mongodb /data
```

Restart MongoDB with

```
$ sudo service mongodb start
```

### Replica Sets

MongoDB Replica Sets are a form of asynchronous data replication between primary/secondary instances of MongoDB, adding automatic failover and automatic recovery of secondary nodes. In addition, Replica Sets can also provide the ability to distribute the read workload of your data.

To create a Replica Set, first create three instances of MongoDB using the instructions above in [Creating an Instance]. Next follow the instructions to [install] and [configure] MongoDB on each instance. Just before starting MongoDB, edit the `/etc/mongodb.conf` configuration file adding the `replSet` parameter:

```
$ sudo nano /etc/mongodb.conf
...
  replSet=replicaSetName
...
```

Before proceeding, note that our current security setup has blocked access to all incoming ports (other than port 22 for SSH) therefore the MongoDB instances won't be accessible. We'll need to add rule to each instance that allows access to port 27017 (the standard `mongod` port) from the other instances in our replica set.

Collect the IP addresses of the instances in your Replica Set and execute the following `ufw` commands on each instance:
$ sudo ufw allow proto tcp from [IP Address 1] to any port 27017
$ sudo ufw allow proto tcp from [IP Address 2] to any port 27017
$ sudo ufw allow proto tcp from [IP Address 3] to any port 27017

Then start MongoDB with

$ sudo service mongodb start

Once MongoDB has started and is running on each instance we'll connect to the desired primary node, initialize the Replica Set and add the secondary nodes. First connect to the desired primary node and launch the mongo shell:

$ mongo
MongoDB shell version: 2.0.4
connecting to: test
>

Now initialize the Replica Set:

> rs.initiate()
{
   "info2" : "no configuration explicitly specified -- making one",
   "me" : "67.207.133.237:27017",
   "info" : "Config now saved locally. Should come online in about a minute.",
   "ok" : 1
}

Next add the other nodes to the Replica Set. If you're executing this on the first configured instance, add the other 2 instances here:

> rs.add("[IP Address 2]");
{ "ok" : 1 }
PRIMARY> rs.add("[IP Address 3]");
{ "ok" : 1 }
PRIMARY>

Finally, you can check the status of the Replica Set from the Mongo shell with the following:

PRIMARY> rs.status()
...

In the output of the `rs.status()` command, you should see three entries, one for each instance. Each one should provide information about it's current status within the Replica Set. For more information, refer to the [Replica Sets](#) documentation.

**Backups**

To facilitate backups there are two main options: Cloud Servers images or MongoDB built-in backup tools.

**Cloud Servers Images**

Cloud Servers images are snapshots of an entire instance which are saved in Rackspace Cloud Files. Images can be created from any running instance and the process can be completed via the Rackspace Cloud control panel. When creating backups from instances in a Replica Set, be sure to image a SECONDARY instance to avoid any interruptions for your applications or data. To image a SECONDARY, first lock it against database writes from within the Replica Set, create the image and then unlock it to rejoin the Replica Set.

To lock the SECONDARY database, connect with the Mongo shell and use the following command:
While the database is locked it will not accept any incoming writes from the PRIMARY instance. Once locked, go to the Rackspace Cloud control panel and create a "New On-Demand Image". Once the image has been created and is complete, go back to the Mongo shell and issue the "unlock" command:

```plaintext
SECONDARY> db.fsyncUnlock()
{ "ok" : 1, "info" : "unlock completed" }
```

Once unlocked, any data that was written to the PRIMARY while the SECONDARY was locked will asynchronously replicate to the newly unlocked SECONDARY.

**Built-in Tools**

MongoDB ships with several built-in command line tools to help with database administration. One of those tools mongodump can be used to create file-based backups of the running database. Mongodump can be used on a live database instance however for the sake of data consistency, it is wise to use the --oplog option when calling mongodump. This allows for point-in-time backups that catch any data that was written while mongodump was working. For more information about live data backups, refer to the mongodump documentation.

**RedHat OpenShift**

- Getting Started
- Documentation
- Sample MongoDB Apps
- Additional Information

OpenShift is a Platform as a Service (PaaS) offering from RedHat, which provides support for rapid deployment and automatic scalability support for web applications developed with Java EE, Node.js, Python, PHP, Perl and Ruby, and several databases including MongoDB.

**Getting Started**

To get started with OpenShift and MongoDB, check out the OpenShift Quickstart guide. The guide reviews the steps necessary to create, deploy and manage your MongoDB-backed apps. The guide also covers things like application snapshots and database backups.

**Documentation**

- https://openshift.redhat.com/community/developers/mongodb

**Sample MongoDB Apps**

- Python Twitter Clone on Github
- PHP Twitter Clone on Github

**Additional Information**

- Getting Started with MongoDB Shell on OpenShift (video)
- Getting Started with MongoDB Monitoring Service (MMS) on OpenShift (video)
- Deploying Python Apps in the Cloud with MongoDB & OpenShift (video)
- Deploying a PHP Twitter App in the Cloud with MongoDB & OpenShift (video)
- How to Manage MongoDB on OpenShift with Your Favorite Admin Tool (video)

**OpenShift Quickstart**

- Cloning the App
- Deploying onto Express
- Testing the Deployment
- Advanced Functionality
  - App Administration
OpenShift is a Platform as a Service (PaaS) offering from RedHat, which provides support for rapid deployment and automatic scalability support for web applications developed with Java EE, Node.js, Python, PHP, Perl and Ruby. OpenShift Express is a free shared-cloud solution for deploying your applications. With Express, you simply push your app using the command line tools to OpenShift and the platform takes care of the hosting infrastructure.

In this guide, we'll take a simple Java web application and deploy it onto OpenShift Express. We'll walk through the steps required to setup and administer MongoDB as well as backing up your data.

### Cloning the App

We'll be deploying an existing Java web application onto OpenShift. The app is a simple REST-style checkin mechanism with a single endpoint. The app supports POST-ing new checkins (by supplying a comment and location) and GET-ing checkins near a given location (by supplying a pair of coordinates). The app was built using Java SE 6 and Maven. Make sure to have those components installed, along with Git for your platform, before continuing.

First, start by cloning the repository for the web application::

```bash
$ git clone https://github.com/crcsmnky/openshift-checkins.git
$ cd openshift-checkins
```

Next, make sure you can build the app as-is using Maven::

```bash
$ mvn package
```

If that completes successfully, you're ready to move on. To prepare our app for deployment, we'll need to setup our OpenShift Express account.

### Deploying onto Express

To deploy apps onto Express you'll need to create an OpenShift account from the [OpenShift sign up page](https://www.openshift.com/) and install the OpenShift command line tools.

First, follow the steps from the [OpenShift Express getting started guide](https://www.openshift.com/) in the *Install the client tools* section for your platform. Once the tools have been installed we'll create a domain name, app name and then push the sample app to OpenShift Express (the URL for the app will be something like **http://appname-domainname.rhcloud.com**).

First, create the domain name::

```bash
$ rhc domain create -n your-domain -l your-openshift-login
```

This command will prompt you for your account password then for an SSH passphrase (it generates a public/private keypair to use for connecting to the OpenShift service). Next, create an entry for the app that we'll be deploying. Here you'll need to supply the app name (example, expresscheckin) and jbossas-7.0 as the app type::

```bash
$ rhc app create -a expresscheckin -t jbossas-7
```

After the app is created the command will also clone the app's repository locally. Before continuing, we need to set up MongoDB on OpenShift and also update our app's MongoDB connection information::
$ rhc app cartridge add -a expresscheckin -c mongodb-2.0

RESULT:

MongoDB 2.0 database added. Please make note of these credentials:

  Root User: admin
  Root Password: 1wbLGYAmPgDM
  Database Name: expresscheckin
  Connection URL: mongodb://127.6.85.129:27017/

You can manage your new MongoDB by also embedding rockmongo-1.1

Now that we've added support for MongoDB to our app (on OpenShift) we'll need to take the credentials returned and update our openshift-checkins app configuration. Go back to the openshift-checkins directory and edit CheckinServlet.java and add/update the following lines in the init function using the MongoDB details provided by OpenShift (don't forget to uncomment the if statement with the authentication statement):

```java
$ cd openshift-checkins
$ nano src/main/java/CheckinServlet.java
...
conn = new Mongo("127.6.85.129", 27017);
db = conn.getDB("expresscheckin");
if (db.authenticate("admin", "1wbLGYAmPgDM".toCharArray())) {
  throw new MongoException("unable to authenticate");
}
...
```

Then build the openshift-checkins WAR file:

```bash
$ mvn package
```

Now, return to the cloned repository, remove the sample code generated from the expresscheckin app repo that was cloned to our system and copy the checkins.war file into the deployments as ROOT.war:

```bash
$ cd ../expresscheckin
$ rm -rf pom.xml src
$ cp ../openshift-checkins/target/checkins.war deployments/ROOT.war
```

As part of the deployment process, we also need to flag our WAR file to be deployed (expanded and copied to the right places):

```bash
$ touch deployments/ROOT.war.dodeploy
```

Now we can add it to the repository, commit and push:

```bash
$ git add -A
$ git commit -m "initial deployment of expresscheckin app onto OpenShift Express"
$ git push origin
```

After pushing the app, it will take a few minutes for the app to become available at http://expresscheckin-your-domain.rhcloud.com/. That's it, you've deployed a simple Java app that uses MongoDB to OpenShift Express. Refer to Testing the Deployment below for some notes on testing the app's functionality.

**Testing the Deployment**
Once you've deployed your app onto OpenShift Express, the URL for the app will be something of the form http://expresscheckin-your-domain.rhcloud.com. We'll now use that URL to conduct some tests on our deployed app.

Since the app is a simple RESTish mechanism we can use `curl` to test it out. Let's start by posting a new comment and location to the URL (ex. http://appurl.rhcloud.com/checkin)::

```bash
$ curl -X POST -d "comment=hello&x=1&y=1" http://appurl.rhcloud.com/checkin
```

Now let's see if we can find the comment we just posted at that location (x = 1, y = 1)::

```bash
$ curl "http://appurl.rhcloud.com/checkin?x=1&y=1"
```

```json
{  "_id" : { "$oid" : "4f0e068c3004bfd40822840b"} ,  "comment" : "hello" ,  "location" : [ 1.0 , 1.0]}
```

If these worked, then it looks like we've got a functional app deployed onto OpenShift.

**Advanced Functionality**

Once you've deployed your app, you'll probably need to connect to your server at some point so you can do things like reviewing app logs or query data in MongoDB. The following steps will cover how you can do that (and if available, set up some advanced functionality).

**App Administration**

Using the command line tools, we can connect to the server our app is deployed on. Run the following command to get information about the current running apps::

```bash
$ rhc domain show
Password:

User Info
==========
Namespace: checkins
RHLogin: sandeep@clusterbeep.org

Application Info
------------------
expresscheckin
  Framework: jbossas-7
  Creation: 2012-04-14T14:04:54-04:00
  UUID: 485daa768043454d9ccdb9343018eb6e
  Git URL: ssh://485daa768043454d9ccdb9343018eb6e@expresscheckin-checkins.rhcloud.com/~/git/expresscheckin.git/
  Public URL: http://expresscheckin-checkins.rhcloud.com/

  Embedded:
  mongodb-2.0 - Connection URL: mongodb://127.6.85.129:27017/
```

The **UDID** above shows the user ID we can use to SSH into our server::
$ ssh 429827960cbf4518b1785ed928db9be7@expresscheckin-your-domain.rhcloud.com

Welcome to OpenShift shell

This shell will assist you in managing openshift applications.

!!! IMPORTANT !!! IMPORTANT !!! IMPORTANT !!!
Shell access is quite powerful and it is possible for you to accidentally damage your application. Proceed with care!
If worse comes to worse, destroy your application with rhc-ctl-app and recreate it
!!! IMPORTANT !!! IMPORTANT !!! IMPORTANT !!!
type "help" for more info.

Now at the prompt type help to see the available commands (in addition to normal shell commands):

[openshift]$ help
Help menu: The following commands are available to help control your openshift application and environment.

  ctl_app         control your application (start, stop, restart, etc)
  ctl_all         control application and deps like mysql in one command
  tail_all        tail all log files
  export          list available environment variables
  rm              remove files / directories
  ls              list files / directories
  ps              list running applications
  kill            kill running applications
  mongo           interactive MongoDB shell

To connect to MongoDB, we'll need to use the credentials provided to us from OpenShift above when we set it up initially. Once you have those in hand, here's how you should connect to the database:

[openshift]$ mongo expresscheckin -u admin -p lwbLGYAmPgDM
MongoDB shell version: 2.0.2-rc1
connecting to: 127.1.13.1:27017/expresscheckin
>

From here you can query your data as needed for your application. Finally, to backup your app's data, follow the instructions found on the MongoDB Backup docs. Specifically you'll need to use the mongodump command to do a live backup of your data.

Backing up your app is a simple single step where we create a "snapshot" of the entirety of the application including data, which we can do locally on our development machine:

$ rhc app snapshot save -a expresscheckin
Password:
Running extra dump: mongodb_dump.sh
MongoDB already running
Stopping application...
Done
Creating and sending tar.gz
Running extra cleanup: mongodb_cleanup.sh
Starting application...
Done
**Additional Information**

For additional information, refer to the following:
- OpenShift Express User Guide or
- OpenShift Flex User Guide

**VMware CloudFoundry**

MongoDB is a supported service on VMware's Cloud Foundry.

Cloud Foundry is a platform-as-a-service solution. At this time, CloudFoundry.com (the public instance of Cloud Foundry operated by VMware) supports applications written in Spring Java, Rails and Sinatra for Ruby, Node.js. Scala and other JVM languages/frameworks including Groovy and Grails.*

**Starting a MongoDB service**

```
vmc create-service mongodb --name MyMongoDB
```

Once you create a MongoDB service, you can bind and use it inside of Cloud Foundry applications.

**Java**

- MongoDB Java Language Center

**Node.js**

- [http://docs.cloudfoundry.com/services/mongodb/nodejs-mongodb.html](http://docs.cloudfoundry.com/services/mongodb/nodejs-mongodb.html)
- Blog post: Getting started with VMware Cloud Foundry, MongoDB and Node.js
- MongoDB Node.js language center

**Ruby**

- [http://docs.cloudfoundry.com/services/mongodb/ruby-mongodb.html](http://docs.cloudfoundry.com/services/mongodb/ruby-mongodb.html)
- Getting started with VMware Cloud Foundry, MongoDB and Rails
- MongoDB Ruby language center
- Using GridFS with Ruby and Cloud Foundry

**See Also**

- VMware Cloud Foundry with MongoDB webinar

**Monitoring and Diagnostics**

- mongostat
- mongotop
- Query Profiler
- Http Console
- mongo Shell Diagnostic Commands
- Trending/Monitoring Adaptors
- Other Tools
- Hosted Monitoring
- Database Record/Replay (diagLogging command)
- Additional Resources
- Admin UIs

- Checking Server Memory Usage
- collStats Command
- Database Profiler
- mongoperf
- Munin configuration examples
- Http Interface
- mongosniff

- MMS (MongoDB Monitoring Service) a free hosted monitoring tool for MongoDB by 10gen
mongostat

mongostat is a great utility which exposes many internal MongoDB metrics. For any MongoDB related issues it is a good start for the analysis of performance issues.

mongotop


Query Profiler

Use the Database Profiler to analyze slow queries.

db.currentOp() is another way to get a snapshot of what is currently happening.

Http Console


mongo Shell Diagnostic Commands

- db.serverStatus()
- See the serverStatus Command page.
- db.stats()
  - Stats on the current database. Command takes some time to run, typically a few seconds unless the .ns file is very large (via use of --nssize). While running other operations may be blocked.
  - fileSize is the total size of all files allocated for the db.
- db.foo.find().explain()
- explain plan
- help
- db.help()
- db.foo.help()

Trending/Monitoring Adaptors

- munin
  - Server stats: this will retrieve server stats (requires python; uses http interface)
  - Collection stats, this will display collection sizes, index sizes, and each (configured) collection count for one DB (requires python; uses driver to connect)
- Ganglia:
  - ganglia-gmond
  - mongdb-ganglia
- cacti
- Mikoomi provides a MongoDB plugin for Zabbix
- Nagios
- Motop - Realtime monitoring tool for several MongoDB servers. Shows current operations ordered by durations every second.
- mtop - A top like utility for Mongo
- Mongo Live - A Chrome extension that provides a real-time server status view (uses the rest interface).

Chris Lea from (mt) Media Temple has made an easy to install Ubuntu package for the munin plugin.

Other Tools

- Dex analyzes logs and/or system.profile collections for (slow) queries and suggests indexes which might help based on a few simple heuristics.

Hosted Monitoring

- [MongoDB Monitoring Service (MMS)](http://www.mongodb.com/products/mms) is a free hosted monitoring tool for MongoDB provided by 10gen
- [Server Density](http://www.serverdensity.com) provides hosted monitoring for your hardware and software infrastructure, and supports a number of status checks for MongoDB.
- Cloudkick
- scout app slow queries
- AppFirst

Database Record/Replay (diagLogging command)

Recording database operations, and replaying them later, is sometimes a good way to reproduce certain problems in a controlled environment.
To enable logging:

```javascript
db._adminCommand( { diagLogging : 1 } )
```

To disable:

```javascript
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:

```bash
nc ''database_server_ip'' 27017 < ''somelog.bin'' | hexdump -c
```

**Additional Resources**

- Monitoring MongoDB
- Server Status Reference
- Database Stats Reference
- Collection Stats Reference

**Admin UIs**

See the Admin UIs page

**Checking Server Memory Usage**

- How Caching Works
- Memory Mapped Files
- Windows
- Virtual Memory Size
- Swap
  - Always Have Swap
- Commands
- Working Set Size
  - Eatmem utility
  - Asymmetry
- Unix Utilities
- Historical Memory Leak Bugs
  - See Also

**How Caching Works**

See Caching

**Memory Mapped Files**

Depending on the platform you may see the mapped files as memory in the process (see the Virtual Memory section below), but this is not strictly correct. If mapped files are counted as process memory, the `top` utility may show way more memory for `mongod` than is really appropriate for what is in physical memory.

The operating system manages the memory where the memory mapped files reside (depending on the OS, this is usually done by the virtual memory manager). You can usually see mapped files using a program like `free -lmt`.

Memory mapped files are shown under "cached" memory:
```bash
skot@stump:~$ free -tm

            total   used   free  shared   buffers  cached
    Mem:       3962   3602    359      0       411     2652
  -/+ buffers/cache:     538   3423
Swap:         1491     52    1439
Total:        5454   3655   1799
```

**Windows**

By bringing up the Task Manager you can see the process memory allocation for mongod.

In addition in the Performance tab the "cached" memory which represents the memory allocated by the memory mapped (data) files.

<table>
<thead>
<tr>
<th>Physical Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
</tr>
<tr>
<td>Cached</td>
</tr>
<tr>
<td>Available</td>
</tr>
<tr>
<td>Free</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paged</td>
</tr>
</tbody>
</table>

**Virtual Memory Size**

Virtual Memory includes the mapped memory files which can be much larger than the actual physical (or swap) available on your host. If journaling is enabled then the database files will be mapped twice leading to much higher virtual memory allocation than might be obvious.

It is important to remember that only a portion of the virtual memory number is really in physical memory. There are many utilities which will report memory usage as the virtual memory number which can be misleading.

Read below for more metrics about virtual memory allocation and how to detect related problems.

**Swap**

Since the database files are memory mapped and should constitute most of your mongodb memory usage it is very unlikely that mongod will ever use any swap space. The reason for this is because any of the memory mapped files can simply either be released from memory (without going to swap) to free memory for other programs or written back to the database files which means they never need to be "swapped" out to disk in the swap space (since they are already backed by files).

If you see continuous swap usage then it most likely means one of these things:

- Your system is extremely constrained by memory (see free -ltm "cached" to confirm)
- There is a memory leak (like the javascript engine, or lots of no_timeout cursors never being closed, or other internal data structures piling up)
- Some other program is stealing lots of memory

**Always Have Swap**

Because of these possible conditions it is always good to have some swap space available on your system. Think of the swap space as something like a steam release valve which allows excess pressure to release without blowing the system up.

**Commands**

The `serverStatus()` command provides memory usage information. Shell example:
One can verify there is no memory leak in the mongod process by comparing the mem.virtual and mem.mapped values (these values are in megabytes). If you are running with journaling disabled, the difference should be relatively small compared to total RAM on the machine. If you are running with journaling enabled, compare mem.virtual to 2*mem.mapped. Also watch the delta over time; if it is increasing consistently, that could indicate a leak.

The mem.mapped value reflects the size of all databases currently open. When replication is on, this includes the size of the local database which includes the oplog. When journaling is enabled each file is mapped twice, once as a writable memmapped view and once as a protected view. The total amount of RAM used is roughly the same; the larger virtual memory size should not be a cause for concern.

One large component of the difference between mem.virtual and mem.mapped (or 2*mem.mapped when journaling is enabled) can be stack memory. In particular, each connection that you have open has a stack frame. The size of each stack frame is determined by the stack size; in Linux this typically defaults to 8MB, which means that each connection will use 8MB on the server. If you are using many connections and are concerned with memory usage you should reduce the stack size to 1MB (this is automatic in the upcoming v2.0 release).

On Linux, extra_info will contain information about the total heap size in a heap bytes field.

You can also see the virtual size and mapped values in the mongostat utility’s output.

While increasing virtual size can indicate a memory leak, increasing resident size (ie, what is reflected as RES in the output of top) indicates that the operating system is using a larger portion of available memory to hold mongodb data, which often occurs under normal operations as a system warms up.

Note: OS X includes the operating system image in virtual size (~2GB on a small program). Thus interpretation of the values on OS X is a bit harder.

**Working Set Size**

In MongoDB it is fine if databases (and thus virtual size) are much larger than ram (terabytes for example); however, you will want your working set to stay in memory to achieve good performance. Otherwise lots of random disk IO's will occur, and unless you are using SSD, this can be quite slow. One area to watch specifically in managing the size of your working set is index access patterns. If you are inserting into indexes at random locations (for example, with id’s which are effectively randomly generated by hashes), you will continually be updating the whole index. If instead you are able to create your id’s in approximately ascending order (for example, day concatenated with a random id), all the updates will occur at the right side of the b-tree and the working set size for index pages will be much smaller.

**Eatmem utility**

Measuring working set size can be difficult; even if it is much smaller than total RAM, if the db has been up for a while and the db is much larger than RAM in total, all memory will be indicated as in use for the cache. Thus we need some other way to estimate our working set size.

One technique is to use a utility which reserves a certain amount of system memory for itself. So one could run this with a certain amount specified and see if the server continues to perform well. If not, the working set is largely than (total_ram - eaten_ram). Note this test will eject some data from the file system cache which may take time to page back in after the eatmem utility is terminated.

Running eatmem continuously with a small percentage of total RAM (say, 20%) is a good technique to get an “early warning” of memory being too low. If disk i/o activity increases significantly, terminate eatmem to mitigate the problem for the moment until further steps can be taking.

**Asymmetry**

In replica sets if one server is underpowered this could one again help as an early warning mechanism for server capacity. Of course the server must be receiving representative traffic to get an indication here.

**Unix Utilities**

mongod uses memory-mapped files; thus the memory stats in top require interpretation in a special way. On a large database, virtual bytes/VSIZE will tend to be the size of the entire database, and if the server doesn’t have other processes running, resident bytes/RSIZE will be the total memory of the machine (as this counts file system cache contents).

vmstat can be useful – try running vmstat 2. on OS X, just vm_stat.

**Historical Memory Leak Bugs**

<table>
<thead>
<tr>
<th>Key</th>
<th>Status</th>
<th>FixVersion</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(31 issues)</td>
</tr>
<tr>
<td>Server ID</td>
<td>Status</td>
<td>Version</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SERVER-2497</td>
<td>Closed</td>
<td>1.7.6</td>
<td>memory leak in ModSet::prepare()</td>
</tr>
<tr>
<td>SERVER-3522</td>
<td>Closed</td>
<td>2.1.0</td>
<td>Memory Leak in dbtests/perftests.cpp</td>
</tr>
<tr>
<td>SERVER-2522</td>
<td>Closed</td>
<td>1.7.6</td>
<td>State::reduceInMemory leaks InMemory on exception</td>
</tr>
<tr>
<td>SERVER-4008</td>
<td>Closed</td>
<td>2.1.0</td>
<td>A memory leak per line of typed text in the shell</td>
</tr>
<tr>
<td>SERVER-5589</td>
<td>Resolved</td>
<td></td>
<td>Possible memory leak in Linux 64-bit server</td>
</tr>
<tr>
<td>SERVER-2520</td>
<td>Closed</td>
<td>1.9.1</td>
<td>add comments in code about intentional memory leak in ReplSetImpl::initFromConfig()</td>
</tr>
<tr>
<td>SERVER-1827</td>
<td>Closed</td>
<td></td>
<td>Memory leak when there's multiple query plans with empty result</td>
</tr>
<tr>
<td>SERVER-2511</td>
<td>Closed</td>
<td>1.9.0</td>
<td>memory leak of _MultiCommandJob</td>
</tr>
<tr>
<td>SERVER-6785</td>
<td>Resolved</td>
<td></td>
<td>Possible mongos memory leak</td>
</tr>
<tr>
<td>SERVER-2122</td>
<td>Closed</td>
<td></td>
<td>memory leak of shard + replication</td>
</tr>
<tr>
<td>SERVER-3911</td>
<td>Closed</td>
<td></td>
<td>memory leak with journaling in windows</td>
</tr>
<tr>
<td>SERVER-5843</td>
<td>Open</td>
<td></td>
<td>debugging with submitter</td>
</tr>
<tr>
<td>SERVER-1897</td>
<td>Closed</td>
<td></td>
<td>admin page plugins and handlers leak memory</td>
</tr>
<tr>
<td>SERVER-6810</td>
<td>Resolved</td>
<td></td>
<td>Memory leak in mongos when using auth</td>
</tr>
<tr>
<td>SERVER-6652</td>
<td>Open</td>
<td>2.3.x</td>
<td>memory leak on Matcher validation failure after _where allocated</td>
</tr>
<tr>
<td>SERVER-768</td>
<td>Closed</td>
<td>1.3.4</td>
<td>Memory leak and high memory usage from snapshots thread</td>
</tr>
<tr>
<td>SERVER-774</td>
<td>Closed</td>
<td></td>
<td>MessagingPorts are leaking</td>
</tr>
<tr>
<td>SERVER-4089</td>
<td>Closed</td>
<td></td>
<td>Memory leak</td>
</tr>
<tr>
<td>SERVER-6168</td>
<td>Open</td>
<td></td>
<td>Memory leak in BSONIteratorSorted</td>
</tr>
<tr>
<td>SERVER-2558</td>
<td>Closed</td>
<td>1.8.0-rc0</td>
<td>memory and cursor leak in FindingStartCursor</td>
</tr>
<tr>
<td>SERVER-6656</td>
<td>Open</td>
<td>2.3.x</td>
<td>Memory Leak - SSL Enabled Build</td>
</tr>
<tr>
<td>SERVER-4354</td>
<td>Closed</td>
<td></td>
<td>Waiting mongo process leaks memory.</td>
</tr>
<tr>
<td>SERVER-2521</td>
<td>Open</td>
<td>Planning Bucket A</td>
<td>1300 byte memory leak in PiggyBackData on socket exception</td>
</tr>
<tr>
<td>SERVER-2498</td>
<td>Open</td>
<td>Planning Bucket B</td>
<td>small memory leak when closing a database</td>
</tr>
</tbody>
</table>
See Also

- The Linux Out of Memory OOM Killer

collStats Command

Statistics on a collection. The command name is "collStats" and the shell helper is database.collection.stats().

```javascript
> db.commandHelp("collStats")
help for: collStats { collStats:"blog.posts", scale : 1 } scale divides sizes e.g. for KB use 1024
```

In the shell:

```javascript
> // to see params etc.:
> db.foo.stats
function (scale) {
  return this._db.runCommand({collstats:this._shortName, scale:scale});
}

// to run:
> db.foo.stats()
{
  "ns" : "test.foo",
  "count" : 9,  // number of documents
  "size" : 432,  // collection size
  "avgObjSize" : 48,  // average object size in bytes
  "storageSize" : 3840,  // (pre)allocated space for the collection
  "numExtents" : 1,  // extents are contiguously allocated
  // chunks of datafile space
  "indexes" : 2,
  "lastExtentSize" : 3840,
  "paddingFactor" : 1,  // padding can make updates faster
  // if documents grow
  "flags" : 1,
  "totalIndexSize" : 16384,
  "indexSizes" : {
    "_id_" : 8192,
    "x_1" : 8192
  },
  "ok" : 1
}
```
Database Profiler

Mongo includes a profiling tool to analyze the performance of database operations.

- **Enabling Profiling**
  - Through the profile command
  - Through the command-line/config-file
- **Using with Sharding**
- **Viewing the Data**
  - Filtering example
  - View stats for only one collection example
  - View slow operations only
  - To see newest information first
  - To view information from a certain time range
  - The `show profile` shell helper
- **Understanding the Output**
- **Optimizing Query Performance**
- **Optimizing Update Performance**
- **Profiler Overhead**
- **Profiling and Replication**
- **Changing the system.profile Collection Size**
- **Alternatives to Profiling**
- **See Also**

See also the `currentOp` command.

**Enabling Profiling**

**Through the profile command**

You can enable and disable profiling from the mongo shell, or through a driver, via the profile command.

```bash
> db.commandHelp("profile") // see how to run from drivers
```

When using the profile command, profiling is enabled or disabled per database. A system.profile collection will be created for the database.

To enable profiling, from the mongo shell invoke:

```bash
> db.setProfilingLevel(2);
{"was": 0, "slowms": 100, "ok": 1} // "was" is the old setting
> db.getProfilingLevel()
2
```

Profiling levels are:

- 0: off
- 1: write slow operations (by default, >100ms is considered slow) to the system.profile collection
- 2: write all operations to the system.profile collection

In addition to the default levels you can also specify a `slowms` option:

```bash
> db.setProfilingLevel(1,20) // log slow operations, slow threshold=20ms
> db.getProfilingStatus() // new shell helper method as of v1.7+
{ "was" : 1, "slowms" : 20 }
```

Note: the profiling level controls which operations get written to the system.profile collection. However, even if the profiler is off, queries slower than the "slowms" level will get written to the logs.

**Through the command-line/config-file**

You can also enable profiling on the command line; for example:
Using with Sharding

Enabling and aggregating profile data globally is not yet available. For now, connect directly to a mongod of interest with the shell, and follow the instructions on this page. You can then repeat the same procedure on other shards if necessary.

Viewing the Data

Profiling data is recorded in the database's system.profile collection. Query that collection to see the results.

Because profile data is written to system.profile, you will see write activity on the database, even if only reading, because of the writes to system.profile.

`> db.system.profile.find()
{"ts": "Thu Jan 29 2009 15:19:32 GMT-0500 (EST)", "info": "query test.$cmd ntoreturn:1 reslen:66 nscanned:0 <br>query: { profile: 2 } nreturned:1 bytes:50", "millis": 0}
...

Filtering example

As an example, to see output without $cmd (command) operations, invoke:

`db.system.profile.find( function() { return this.info.indexOf('$cmd')<0; } )`

Likewise we could query for indexOf(...)>=0 to see only those lines.

View stats for only one collection example

To view operations for a particular collection:

`> db.system.profile.find( { info: /test.foo/ } )
{"ts": "Thu Jan 29 2009 15:19:40 GMT-0500 (EST)", "info": "insert test.foo", "millis": 0}
{"ts": "Thu Jan 29 2009 15:19:42 GMT-0500 (EST)", "info": "insert test.foo", "millis": 0}
{"ts": "Thu Jan 29 2009 15:19:45 GMT-0500 (EST)", "info": "query test.foo ntoreturn:0 reslen:102 nscanned:2 <br>query: {} nreturned:2 bytes:86", "millis": 0}
{"ts": "Thu Jan 29 2009 15:21:17 GMT-0500 (EST)", "info": "query test.foo ntoreturn:0 reslen:36 nscanned:2 <br>query: { $not: { x: 2 } } nreturned:0 bytes:20", "millis": 0}

View slow operations only

To view operations slower than a certain number of milliseconds:

`> db.system.profile.find( { millis : { $gt : 5 } } )

To see newest information first

`> db.system.profile.find().sort({$natural:-1})`

To view information from a certain time range
In the next example we look at the time range, suppress the `user` field from the output to make it easier to read, and sort the results by how long each operation took to run.

```javascript
> db.system.profile.find(
...{ts:{$gt: ISODate("2011-07-12T03:00:00Z"),
...
$lt: ISODate( "2011-07-12T03:40:00Z" )}
...})
...
...,{user:0}).sort({millis:-1})
```

The `show profile` shell helper

The mongo shell includes a helper to see the most recent 5 profiled events that took at least 1ms to execute. Type

```
show profile
```

at the command prompt to use this feature.

**Understanding the Output**

The output reports the following values:

- `ts` Timestamp of the profiled operation.
- `millis` Time, in milliseconds, to perform the operation. This time does not include time to acquire the lock or network time, just the time for the server to process.
- `info` Details on the operation.
  - `query` A database query operation. The query info field includes several additional terms:
    - `ntoreturn` Number of objects the client requested for return from a query. For example, `<code>findOne()</code>` sets `ntoreturn` to 1. `<code>limit()</code>` sets the appropriate limit. Zero indicates no limit.
    - `query` Details of the query spec.
    - `nscanned` Number of objects scanned in executing the operation.
    - `reslen` Query result length in bytes.
    - `nreturned` Number of objects returned from query.
  - `update` A database update operation. `<code>save()</code>` calls generate either an update or insert operation.
    - `fastmod` Indicates a fast modify operation. See `Updates`. These operations are normally quite fast.
    - `fastmodinsert` - indicates a fast modify operation that performed an upsert.
    - `upsert` Indicates an upsert performed.
    - `moved` Indicates the update moved the object on disk (not updated in place). This is slower than an in place update, and normally occurs when an object grows.
    - `key updates` How many index keys changed during the update. Key updates are a little bit expensive since the db needs to remove the old key and insert a new key into the b-tree index.
  - `insert` A database insert.
  - `getmore` For large queries, the database initially returns partial information. `getmore` indicates a call to retrieve further information.

**Optimizing Query Performance**

- If `nscanned` is much higher than `nreturned`, the database is scanning many objects to find the target objects. Consider creating an index to improve this.
- `reslen` A large number of bytes returned (hundreds of kilobytes or more) causes slow performance. Consider passing `<code>find()</code>` a second parameter of the member names you require.

*Note:* There is a cost for each index you create. The index causes disk writes on each insert and some updates to the collection. If a rare query, it may be better to let the query be "slow" and not create an index. When a query is common relative to the number of saves to the collection, you will want to create the index.

**Optimizing Update Performance**

- 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.
Examine the `ns_canned` info field. If it is a very large value, the database is scanning a large number of objects to find the object to update. Consider creating an index if updates are a high-frequency operation. Use fast modify operations when possible (and usually with these, an index). See Updates.

**Profiler Overhead**

When enabled, profiling affects performance, although not severely.

Profile data is stored in the database's `system.profile` collection, which is a Capped Collection. By default it is set to a very small size and thus only includes recent operations.

**Profiling and Replication**

In v1.9+, you can use profiling on secondaries in addition to the current primary. In older versions of MongoDB, use profiling on the primary only.

**Changing the system.profile Collection Size**

Profiling information is written to the `system.profile` capped collection. There is a separate profile collection per database. By default the collection is very small and like all capped collections works in a rotating RRD-like style. To make it bigger you can create it explicitly. You will need to drop it first; you may need to disable profiling before the drop/recreate. Example in the shell:

```bash
> db.system.profile.drop()
> db.createCollection("system.profile", {capped:true, size:4000000})
> db.system.profile.stats()
```

**Alternatives to Profiling**

The profiler can generate write locks as it writes to the profile collection. Thus other tools to consider for optimizing queries are:

1. Running `db.currentOp()`, perhaps many times in a row to get a good sample;
2. Using the `explain()` helper in the shell

**See Also**

- Optimization
- `explain()`
- Viewing and Terminating Current Operation

**mongoperf**

- Prebuilt Binaries
- Building
- Running
- mmf:false mode (direct i/o test)
- mmf:true mode
- Additional Options
- See Also

mongoperf is a utility for checking disk i/o performance of a server independent of MongoDB. It performs simple timed random disk i/o's. The utility is new and will likely be more sophisticated in the future.

mongoperf can be used to check disk subsystem performance even when MongoDB will not be used at all: its `mmf:false` mode is completely generic. In that mode it is somewhat analogous to tools such as bonnie++ (albeit mongoperf is simpler).

**Prebuilt Binaries**

- Win32 24Jul2012 `mongoperf.exe`

**Building**

Run `scons mongoperf` or `scons mongoperf.exe` to build. Note: mongoperf is new, so will need a recent version of the mongodb source code.

**Running**
mmf:false mode (direct i/o test)

In this default mode, mongoperf performs random 4KB direct (physical) disk i/o's (i.e., O_DIRECT is used on Linux). Thus this is a physical disk i/o test.

- Output form an example run

mmf:true mode

If mmf:true is specified as an option, tests are performed using memory-mapped files. These files are opened in a "normal" fashion and thus caching is allowed. This sometimes can be used to test file system cache behavior with memory mapped files.

Additional Options

```
{
   nThreads:<n>,   // number of threads (default 1)
   fileSizeMB:<n>, // test file size (default 1MB)
   sleepMicros:<n>, // pause for sleepMicros/nThreads between each operation (default 0)
   mmf:<bool>,   // if true do i/o's via memory mapped files (default false)
   r:<bool>,     // do reads (default false)
   w:<bool>,     // do writes (default false)
   syncDelay:<n>  // secs between fsyncs, like --syncdelay in mongod. (default 0/never)
}
```

- **nThreads** Number of test threads. You will need several threads to test a disk subsystem to saturation. For example, try nThreads:16.
- **sleepMicros** Pause for sleepMicros/nThreads between each operation.
- **r** Do read operations while testing.
- **w** Do write options while testing. w and r may be used at the same time.
- **syncDelay** For use with mmf:true mode only. The syncDelay:secs option instructs mongoperf to perform an asynchronous fsync of the test mmap file at the specified interval. mongod does something similar every 60 seconds, thus this can be useful to test basic system behavior in a simpler setting. This option is applicable only when using mmf:true mode. Currently in mongoperf this defaults to zero, which means off.
- **fileSizeMB**. Specifies the size in megabytes of the test data file. The file will be placed in the current directory. **Specify a very large test file size to create a realistic simulation**. A very small file (1MB) could be cached by your disk controller completely. A file that is say, 100MB in size would involve a small number of disk cylinders, and adjacent track-to-track seeks are much faster than a drive's average seek time. The file size is particularly important when mmf:true is specified, as the file system cache is then involved. A file size much larger than RAM will result in much different performance results than a file size smaller than RAM.

See Also

- The utility is quite simple, so you might wish to also take a look at the source code.
- iostat

Munin configuration examples

Overview
Munin can be used for monitoring aspects of a running system. The following is a mini tutorial to help you set up and use the MongoDB plugin with munin.

**Setup**

Munin is made up of two components

- agent and plugins that are installed on the system you want to monitor
- server which polls the agent(s) and creates the basic web pages and graphs to visualize the data

**Install**

You can download from SourceForge, but prebuilt packages are also available. For example on Ubuntu you can do the following:

**Agent install**

To install the agent, repeat the following steps on each node you want to monitor.

```
shell> sudo apt-get install munin-node
```

**Server install**

The server needs to be installed once. It relies on apache2, so you will need to ensure that it is installed as well.

```
shell> apt-get install apache2
shell> apt-get install munin
```

**Configuration**

Both the agent(s) and server need to be configured with the IP address and port to contact each other. In the following examples we will use these nodes:

- **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]
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-* /etc/munin/plugins/
```

Check your setup

After installing the plugin and making the configuration changes, force the server to update the information to check your setup is correct using the following

```
shell> sudo -u munin /usr/share/munin/munin-update
```

If everything is set up correctly, you will get a chart like this

![MongoDB ops - by day](chart.png)
**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.

```plaintext
/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 delete for the cluster

```plaintext
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.update.label update
cluster_ops.delete.label 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 on the key for the chart
  - cluster_ops.insert : represents a single line on the chart, in this case the "insert"
  - cluster_ops.insert.sum : indicates the values are summed
    - db1-ec2-174-129-52-161.compute-1.amazonaws.com : indicates the node to aggregate
    - mongo_ops.insert : indicates the chart (mongo_ops) and the counter (insert) to aggregate

And this is what it looks like
Http Interface

- REST Interfaces
  - DrowsyDromedary (Ruby)
  - MongoDB Rest (Node.js)
  - Mongodb Java REST server
- HTTP Interfaces
  - Sleepy Mongoose (Python)
- HTTP Console
  - HTTP Console Security
  - Simple REST Interface
  - JSON in the simple REST interface
- See Also

REST Interfaces

**DrowsyDromedary (Ruby)**

DrowsyDromedary is a REST layer for Mongo based on Ruby.

**MongoDB Rest (Node.js)**

MongoDB Rest is an alpha REST interface to MongoDB, which uses the MongoDB Node Native driver.

**Mongodb Java REST server**

Mongodb Java REST server based on Jetty.

HTTP Interfaces

**Sleepy Mongoose (Python)**

Sleepy Mongoose is a full featured HTTP interface for MongoDB.

HTTP Console

MongoDB provides a simple http interface listing information of interest to administrators. This interface may be accessed at the port with numeric value 1000 more than the configured mongod port; the default port for the http interface is 28017. To access the http interface an administrator may, for example, point a browser to http://localhost:28017 if mongod is running with the default port on the local machine.
Here is a description of the informational elements of the http interface:

<table>
<thead>
<tr>
<th>element</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db version</td>
<td>database version information</td>
</tr>
<tr>
<td>git hash</td>
<td>database version developer tag</td>
</tr>
<tr>
<td>sys info</td>
<td>mongod compilation environment</td>
</tr>
<tr>
<td>dblocked</td>
<td>indicates whether the primary mongod mutex is held</td>
</tr>
<tr>
<td>uptime</td>
<td>time since this mongod instance was started</td>
</tr>
<tr>
<td>assertions</td>
<td>any software assertions that have been raised by this mongod instance</td>
</tr>
<tr>
<td>replInfo</td>
<td>information about replication configuration</td>
</tr>
<tr>
<td>currentOp</td>
<td>most recent client request</td>
</tr>
<tr>
<td># databases</td>
<td>number of databases that have been accessed by this mongod instance</td>
</tr>
<tr>
<td>curclient</td>
<td>last database accessed by this mongod instance</td>
</tr>
<tr>
<td>Cursors</td>
<td>describes outstanding client cursors</td>
</tr>
<tr>
<td>master</td>
<td>whether this mongod instance has been designated a master</td>
</tr>
<tr>
<td>slave</td>
<td>whether this mongod instance has been designated a slave</td>
</tr>
<tr>
<td>initialSyncCompleted</td>
<td>whether this slave or repl pair node has completed an initial clone of the mongod instance it is replicating</td>
</tr>
<tr>
<td>DBTOP</td>
<td>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</td>
</tr>
<tr>
<td>dt</td>
<td>Timing information about the primary mongod mutex</td>
</tr>
</tbody>
</table>

**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 REST interface, with no support for insert/update/remove operations, as a convenience – it is generally used for monitoring/alerting scripts or administrative tasks. For full REST capabilities we recommend using an external tool such as Sleepy.Mongoose.

v1.4+: This interface is disabled by default. Use --rest on the command line to enable.

To get the contents of a collection (note the trailing slash):

```
http://127.0.0.1:28017/databaseName/collectionName/
```

To add a limit:

```
http://127.0.0.1:28017/databaseName/collectionName/?limit=-10
```

To skip:

```
http://127.0.0.1:28017/databaseName/collectionName/?skip=5
```

To query for \{a : 1\}:

```
http://127.0.0.1:28017/databaseName/collectionName/?filter_a=1
```

Separate conditions with an &:

```
http://127.0.0.1:28017/databaseName/collectionName/?filter_a=1&limit=-10
```

Same as `db.$cmd.findOne({listDatabase:1})` on the "admin" database in the shell:

```
http://localhost:28017/admin/$cmd/?filter_listDatabases=1&limit=1
```

To count documents in a collection:

```
http://host:port/db/$cmd/?filter_count=collection&limit=1
```

JSON in the simple REST interface

The simple ReST interface uses strict JSON (as opposed to the shell, which uses Dates, regular expressions, etc.). To display non-JSON types, the web interface wraps them in objects and uses the key for the type. For example:

```
ObjectIds just become strings
"_id" : "4a8acf6e7fbadc242de5b4f3"

dates
"date" : { "$date" : 1250609897802 }

regular expressions
"match" : { "$regex" : "foo", "$options" : "ig" }
```

The code type has not been implemented yet and causes the DB to crash if you try to display it in the browser.

See Mongo Extended JSON for details.

See Also

- Replica Set Admin UI
- Diagnostic Tools
mongosniff

Unix releases of MongoDB include a utility called mongosniff. This utility is to MongoDB what tcpdump is to TCP/IP; that is, fairly low level and for complex situations. The tool is quite useful for authors of driver tools.

$ ./mongosniff --help
Usage: mongosniff [--help] [--forward host:port] [--source NET <interface> | FILE <filename>]
[<port0> <port1> ...]
--forward       Forward all parsed request messages to mongod instance at specified host:port
--source        Source of traffic to sniff, either a network interface or a file containing perviously captured packets, in pcap format. If no source is specified, mongosniff will attempt to sniff from one of the machine’s network interfaces.
<port0>...      These parameters are used to filter sniffing. By default, only port 27017 is sniffed.
--help          Print this help message.

Building

mongosniff is including in the binaries for Unix distributions. As mongosniff depends on libpcap, the MongoDB SConstruct only builds mongosniff if libpcap is installed.

$ # Red Hat
$ sudo yum install libpcap-devel
$
$ # Ubuntu/Debian
$ sudo apt-get install libpcap-dev
$
$ scons mongosniff

Example

To monitor localhost:27017, run ifconfig to find loopback's name (usually something like lo or lo0). Then run:

mongosniff --source NET lo

If you get the error message "error opening device: socket: Operation not permitted" or "error finding device: no suitable device found", try running it as root.

Other Tools

If you want to use a GUI with more detailed introspection, there is Wireshark support for MongoDB Protocol.

Wireshark Support for MongoDB Protocol

Wireshark, an advanced interactive network traffic sniffer, has full support for the MongoDB Wire protocol.

You can visually inspect MongoDB traffic, do complex filters on specific values of MongoDB wire messages and dig into individual documents both sent and received.

Note: Wireshark looks for port 27017 and infers MongoDB protocol from this. If you are running on a different port number, go to Preferences...Protocols...Mongo and set your port number and it should then interpret the data.
Backups

- **Back up Sharded Clusters**
- **Methods**
  - Filesystem Snapshot (Journaling must be Enabled)
  - mongodump
  - Shutdown and Backup
  - Lock, Fsync, and Backup
  - Secondary Backup
- **Incremental Backups**
- **See Also**

It is not safe to simply copy the mongod data files (by default in /data/db/) while the database is running and writes are occurring. In particular if journaling is disabled (it is on by default for recent 64-bit builds), this is not safe. Even with journaling on you would have to copy the journal files last and be careful none rotate out (are deleted by mongod itself) while you are backing up.

With all these backup methods we suggest that you be careful how this could affect your live system when choosing when and where you backups are run. Try not to mix too many distinct workloads on the same resource, like doing backup and analytics or reporting for example. This could overwhelm those resources and negatively affect performance for your users.

**Back up Sharded Clusters**

This page covers how to back up a single server or a single replica set or a single shard. To back up an entire cluster, see the Backing Up Sharded Cluster documentation.

**Methods**

- **Filesystem Snapshot (Journaling must be Enabled)**

  If the storage infrastructure (SAN, lvm, etc.) supports snapshots, it is safe to snapshot the entire dbpath directory of a mongod while it is running,
as long as MongoDB journaling is enabled (journaling defaults to on in v2.0+ for 64 bit MongoDB). Take an lvm/ebs snapshot of the entire dbpath directory of a mongod running with journaling. All files and directories (start from the dbpath directory) must be included (especially the journal/subdirectory). As long as all files are snapshotted at the same point in time, you don't need to lock+fsync (see below) the database.

Amazon EBS qualifies if you are not RAIDing the volumes yourself. When RAIDed, the snapshot would be separate for each embedded volume. Thus in this case one should still use lock+fsync instead of snapshots.

**mongodump**

*mongodump* can be used to do a live backup of your data, or can work against an inactive set of database files. The *mongodump* utility may be used to dump an entire cluster/server/database/collection (or part of a collection with a query), even when the database is running and active.

**--oplog option**

If you are backing up a replica (from a replica set, or master/slave) you can use the

```
--oplog
```

to do a point in time backup; that point in time will be at the end of the backup. When you restore you will need to use the corresponding

```
--oplogReplay
```

to use this extra backup information.

**Shutdown and Backup**

A simple approach is just to stop the database, back up the data files, and resume. This is safe but of course requires downtime. This can be done on a secondary without requiring downtime, but you must ensure your oplog is large enough to cover the time the secondary is unavailable so that it can catch up again when you restart it.

**Lock, Fsync, and Backup**

MongoDB supports an lock and fsync command with which we can lock the database to prevent writing, flush writes, and then backup the datafiles.

While in this locked mode, all writes will block (including replication, for secondaries). If this is a problem consider one of the other methods.

> A write attempt will request a lock and may block new readers. This will be fixed in a future release. Thus currently, fsync and lock works best with storage systems that do quick snapshots.

For example, you could use LVM2 to create a snapshot after the fsync+lock, and then use that snapshot to do an offsite backup in the background. This means that the server will only be locked while the snapshot is taken. Don't forget to unlock after the backup/snapshot is taken.

**Secondary Backup**

Another good technique for backups is to backup from one of the secondaries of your replica set. This keeps the backup load off your primary and also facilitates some strategies that might involve having the member offline.

Note: be sure your replicas are reasonably up to date and not lagging far behind the primary. This is important anyway: if lagging they would not be very suitable for failover purposes.

We then have a few options for backing up from the secondary:

1. Lock, fsync, backup the slave, unlock, as described above. After unlocking the secondary will resume replication and "catch up".
2. Shut down the mongod process of the secondary, backup its data directory, and then restart. Given the node is not primary, this would not be disruptive to production if one is using default read preferences. After restarting the mongod should resume replication and "catch up".
   - Note: be sure the backup doesn't take so long that your oplog rolls over. (Typically the oplog is very large; MMS shows its effective duration.)
   - Note: be sure the node to back up is reasonably up to date. This can be checked in various ways one of which is by viewing the MMS graph for replication lag.
3. Any other method listed on this page, e.g., mongodump.

**Incremental Backups**

MongoDB does not yet include an official incremental backup mechanism.
See also the slaveDelay replica set feature; while different than an incremental backup, many users find slaveDelay as a way to keep a rolling backup from a certain delta into the past.

See Also

- http://github.com/micahwedemeyer/automongobackup
- Video - Backing up your MongoDB Cluster - MongoSF (May 2011)

EC2 Backup & Restore

Overview

This article describes how to backup, verify & restore a MongoDB running on EC2 using EBS Snapshots.

Backup

How you backup MongoDB will depend on whether you are using the --journal option in 1.8 (or above) or not.

**Backup with --journal**

The journal file allows for roll forward recovery. The journal files are located in the dbpath directory so will be snapshotted at the same time as the database files.

If the dbpath is mapped to a single EBS volume then proceed to the Backup the Database Files section.

If you dbpath is mapped to multiple EBS volumes, in order to guarantee the stability of the file-system then you will need to Flush and Lock the Database section.

![Note that snapshotting with the journal is only possible if the journal resides on the same volume as the data files, so that one snapshot operation captures the journal state and data file state atomically.](image)

**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:

1. `db.runCommand( { fsync: 1, lock: 1 } )`
2. `ec2-create-snapshot`
3. `db.$cmd.sys.unlock.findOne()`
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-eb09a81 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 /opt/mongodb/current/bin/mongod --port 27000
--shardsvr --dbpath /var/lib/mongodb/tokyo0 --fork --logpath /var/opt/mongodb/log/server.log
--logappend --rest
```

dbpath is set to `/var/lib/mongodb/tokyo0` in this example.

Mapping the dbpath to the physical devices

Using the `df` command, determine what the --dbpath directory is mapped to

```
root> df /var/lib/mongodb/tokyo0
Filesystem 1K-blocks Used Available Use% Mounted on
/dev/mapper/data_vg-data_vol 104802308 4320 104797988 1% /var/lib/mongodb
```
Next determine the logical volume associated with this device, in the example above /dev/mapper/data_vg-data_vol

```
root> lvdisplay /dev/mapper/data_vg-data_vol
--- Logical volume ---
LV Name /dev/data_vg/data_vol
VG Name data_vg
LV UUID fixOyX-6Aiw-PnBA-i2bp-ovUc-u9uu-TGvjxl
LV Write Access read/write
LV Status available
# open 1
LV Size 100.00 GiB
```

This output indicates the volume group associated with this logical volume, in this example data_vg. Next determine how this maps to the physical volume.

```
root> pvscan
PV /dev/md0 VG data_vg lvm2 [100.00 GiB / 0 free]
Total: 1 [100.00 GiB] / in use: 1 [100.00 GiB] / in no VG: 0 [0]
```

From the physical volume, determine the associated physical devices, in this example /dev/md0.

```
root> mdadm --detail /dev/md0
/dev/md0:
  Version : 00.90
  Creation Time : Thu Nov  4 02:17:11 2010
  Raid Level : raid10
  Array Size : 104857472 (100.00 GiB 107.37 GB)
  Used Dev Size : 52428736 (50.00 GiB 53.69 GB)
  Raid Devices : 4
  ...  
    UUID : 07552c4d:6c11c875:e5a1de64:a9c2f2fc (local to host ip-10-204-215-62)
    Events : 0.19
  Number Major Minor RaidDevice State
  0     8     80     0  active sync   /dev/sdf
  1     8     96     1  active sync   /dev/sdg
  2     8    112     2  active sync   /dev/sdh
  3     8    128     3  active sync   /dev/sdi
```

We can see that block devices /dev/sdf through /dev/sdi make up this physical devices. Each of these volumes will need to be snapped in order to complete the backup of the file-system.

**Create the EBS Snapshot**

Create the snapshot for each devices. Using the `ec2-create-snapshot` command, use the Volume Id for the device listed by the `ec2-describe-instances` command.

```
shell> ec2-create-snapshot -d backup-20101103 vol-96e8f0ff
SNAPSHOT snap-417af82b vol-96e8f0ff pending 2010-11-04T05:57:29+0000  289727918005 50 backup-20101103
shell> ec2-create-snapshot -d backup-20101103 vol-90e8f0ff
SNAPSHOT snap-5b7af831 vol-90e8f0ff 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.
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.

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.
**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/sss vol-caaab2a3
ATTACHMENT vol-caaab2a3 i-cad26ba7 /dev/sss 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
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.
Validate Command

Restore

Restore uses the same basic steps as the verification process.

1. `db.shutdownServer()`
2. `ec2-create-volume`
3. `mount file-system`
4. `mongod`

After the file-system is mounted you can decide to

- Copy the database files from the backup into the current database directory
- Startup mongod from the new mount point, specifying the new mount point in the `--dbpath` argument

After the database is started, it will be ready to transact. It will be at the specific point in time from the backup, so if it is part of a master/slave or replica set relationship, then the instance will need to synchronize itself to get itself back up to date.

How to do Snapshotted Queries in the Mongo Database

This document refers to query snapshots. For backup snapshots of the database's datafiles, see the `fsync lock` page.

MongoDB does not support full point-in-time snapshotting. However, some functionality is available which is detailed below.

Cursors

A MongoDB query returns data as well as a cursor ID for additional lookups, should more data exist. Drivers lazily perform a "getMore" operation as needed on the cursor to get more data. Cursors may have latent `getMore` accesses that occurs after an intervening write operation on the database collection (i.e., an insert, update, or delete).

Conceptually, a cursor has a current position. If you delete the item at the current position, the cursor automatically skips its current position forward to the next item.

MongoDB cursors do not provide a snapshot: if other write operations occur during the life of your cursor, it is unspecified if your application will see the results of those operations. In fact, it is even possible (although unlikely) to see the same object returned twice if the object were updated and grew in size (and thus moved in the datafile). To assure no update duplications, use `snapshot()` mode (see below).

Snapshot Mode

`snapshot()` mode assures that objects which update during the lifetime of a query are returned once and only once. This is most important when doing a find-and-update loop that changes the size of documents that are returned ($inc does not change size).
> // mongo shell example
> var cursor = db.myCollection.find({country:'uk'}).snapshot();

Even with snapshot mode, items inserted or deleted during the query may or may not be returned; that is, this mode is not a true point-in-time snapshot.

Because snapshot mode traverses the _id index, it may not be used with sorting or explicit hints. It also cannot use any other index for the query.

You can get the same effect as snapshot by using any unique index on a field(s) that will not be modified (probably best to use explicit hint() too). If you want to use a non-unique index (such as creation time), you can make it unique by appending _id to the index at creation time.

**Import Export Tools**

- **Data Import and Export**
  - **mongoimport**
    - Example: Import file format
    - Example: Importing with upsert
    - Example: Importing Interesting Types
  - **mongodbump and mongorestore**
    - **mongodbump**
      - Example: Dumping Everything
      - Example: Dumping a Single Collection
      - Example: Dumping a Single Collection to Stdout
      - Example: Dumping a Single Collection with a query
      - Example: Using --oplog to get a point-in-time backup
      - Performance Tips
      - If mongodump seems to skip documents...
    - **mongorestore**
  - **bsondump**
  - **See Also**

If you just want to copy a database from one mongod server to another, use the copydb Command instead of these tools.

These tools work with the raw data (the BSON documents in the collections, both user and system); they do not save, or load certain metadata such as (capped) collection properties. You will need to (re)create those yourself in a separate step, before loading that data. Vote for SERVER-808 to change this. (Consider using the copydb command which does preserve these properties.)

**Default Write Mode**

These tools do fire-n-forget writes. This means that errors which happen during those inserts/updates will only be recorded on the server (in the logs) and not reported in the tool, nor will it cause the process to stop. See SERVER-2611 for possible improvements.

**Data Import and Export**

**mongoimport**

This utility takes a single file that contains 1 JSON/CSV/TSV string per line and inserts it. You have to specify a database and a collection.
options:
--help                  produce help message
-v [ --verbose ]        be more verbose (include multiple times for more verbosity e.g. -vvvvv)
-h [ --host ] arg       mongo host to connect to ("left,right" for pairs)
--port arg              server port. (Can also use --host hostname:port)
--ipv6                  enable IPV6 support (disabled by default)
-d [ --db ] arg         database to use
-c [ --collection ] arg collection to use (some commands)
-u [ --username ] arg   username
-p [ --password ] arg   password
--dbpath arg            directly access mongod data files in the given path, instead of connecting to a mongod instance - needs to lock the data directory, so cannot be used if a mongod is currently accessing the same path
--directoryperdb        dbpath specified, each db is in a separate directory
--fields ] arg          comma seperated list of field names e.g. -f name,age
--fieldFile arg         file with fields names - 1 per line
--ignoreBlanks          if given, empty fields in csv and tsv will be ignored
--type arg              type of file to import. default: json (json,csv,tsv)
--file arg              file to import from; if not specified stdin is used
--drop                  drop collection first
--headerline            CSV, TSV only - use first line as headers
--upsert                insert or update objects that already exist
--upsertFields arg      comma-separated fields for the query part of the upsert. You should make sure this is indexed.
--stopOnError           stop importing at the first error rather than continuing (for network errors, not server insert/update errors)
--jsonArray             load a json array, not one item per line. Currently limited to 4MB.

Note that the following options are only available in 1.5.3+: upsert, upsertFields, stopOnError, jsonArray

Example: Import file format

The import file should contain one document per line (with a few exceptions: if using --jsonArray, if importing a CSV then one document may span multiple lines if it contains multi-line string, if importing a CSV with --headerline then the first line doesn't correspond to a document but instead specifies which fields are being imported).

When using the standard JSON import format, each line in input file must be one JSON document which will be inserted directly into the database.

For example, if you imported a file that looks like this:

```
{ "_id" : { "$oid" : "4e5bb37258200ed9aabc5d65" }, "name" : "Bob", "age" : 28, "address" : "123 fake street" }
```

by running

```
mongoimport -d test -c foo importfile.json
```

you'd get this imported:

```
> db.foo.find()
{ "_id" : ObjectId("4e5bb37258200ed9aabc5d65"), "name" : "Bob", "age" : 28, "address" : "123 fake street" }
```

Example: Importing with upsert

The following command will import data from temp.csv into database foo, collection bar on localhost. Additionally it will perform an upsert of the data. By default the upsert will use the field marked as `_id` as the key for updates.
If the file does not have an _id field, you can update on alternate fields by using upsertFields. Note that when using this with sharding, the upsertField must be the shardkey.

Even though using --upsert may result in an update, every document in the input file must be formatted in a way that is compatible to insert. Therefore, no update modifiers are allowed.

**Example: Importing Interesting Types**

MongoDB supports more types that JSON does, so it has a special format for representing some of these types as valid JSON. For example, JSON has no date type. Thus, to import data containing dates, you structure your JSON like:

```json
{"somefield" : 123456, "created_at" : {"$date" : 1285679232000}}
```

Then mongoimport will turn the created_at value into a Date.

Note: the $-prefixed types must be enclosed in double quotes to be parsed correctly.

**mongoexport**

mongoexport takes a collection and exports to either JSON or CSV. You can specify a filter for the query, or a list of fields to output. See the mongoexport page for more information.

**mongodump and mongorestore**

There are many ways to do backups and restores. (Here are some other backup strategies)

**mongodump**

This takes a database and outputs it in a binary representation. This is used for doing (hot) backups of a database.

If you're using sharding and trying to migrate data this way, this will dump shard configuration metadata information (from the config db) and overwrite configurations upon restore. This is true because without any options mongodump will dump all dbs/collections, including the config db where this information is kept.

options:
---
--help                  produce help message
-v [ --verbose ]        be more verbose (include multiple times for more
verbosity e.g. -vvvv)
-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
--out [ --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)               force a table scan (do not use $snapshot)
**Example: Dumping Everything**

To dump all of the collections in all of the databases, run `mongodump` with just the `--host`:

```bash
$ ./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:

```bash
$ ./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.

```bash
$ ./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`:

```bash
$ ./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.

Note that you must use the MongoDB Extended JSON format when specifying a query.

```bash
$ ./mongodump --db blog --collection posts
- q  '{"created_at" : { "$gte" : 1293868800000},
        "$lt" : 1296460800000}
```

**Example: Using --oplog to get a point-in-time backup**

If data is changed over the course of a backup then the resulting dump may wind up in an inconsistent state that doesn't correspond to how the data looked in the DB at any one moment. This can be avoided by using `--oplog` in `mongodump` and `--oplogReplay` in `mongorestore`. If you use
When the backup is started, mongodump will note the time of the most recent entry in the oplog. When the dump is finished, mongodump will then find all the oplog entries since the dump started and will dump those as well. When you run mongorestore with --oplogReplay, after it finishes restoring the main dump, it will replay the entries in the oplog dump so that the data restored ends up in a consistent state corresponding to the moment that the original dump finished.

It is not valid to use --oplog when dumping from a mongos. You can use it to dump an individual shard though – see the Backing Up Sharded Cluster page. Likewise, --oplog can only be used on the master in a Master Slave configuration because the slave does not store an oplog.

**Performance Tips**

The default dump mode is to do a "snapshot" query. This results in the dump query walking through the _id index and returning results in that order. If you use a custom _id value, not the default ObjectId type, then this could cause much more disk activity to do a dump; it could (dramatically) slow down things.

In 1.9.1+ you can force a walk of the data without using an index:

```
$ ./mongodump --forceTableScan ...
```

In earlier versions you can cause this to behavior with a special query (one that cannot use the index):

```
$ ./mongodump -q "{xxxx : { $ne : 0 } }" ...
```

Note: In some shells (like bash) you must escape the "$" in this command like so "\$".

If mongodump seems to skip documents...

There is a maximum key size in the indexes, currently approximately 800 bytes. This limit also applies to the default index on _id. Any document with an _id key larger than 800 bytes will not be indexed on _id. By default mongodump walks the _id index and will skip documents with keys too large to index.

```
> use bigid
> db.foo.count()
3

$ mongodump -d bigid -c foo
connected to: 127.0.0.1
DATABASE: bigid to dump/bigid
bigid.foo to dump/bigid/foo.bson
0 objects
```

You can work around this issue with either of the options listed in the previous section:

```
$ mongodump -d bigid -c foo -q "{xxxx : { \$ne : 0 } }"
connected to: 127.0.0.1
DATABASE: bigid to dump/bigid
bigid.foo to dump/bigid/foo.bson
3 objects
```
mongorestore

mongorestore takes the output from mongodump and restores it. Indexes will be created on a restore. mongorestore just does inserts with the data to restore; if existing data (like with the same _id) is there it will not be replaced. This can be done with an existing database, or mongorestore will create a new one if the database does not exist. Mongorestore is mostly non-blocking (it just calls a series of normal inserts), though if the dump included indexes it may cause the DB to block as the indexes are rebuilt.

If you do not wish to create indexes you can remove the system.indexes.bson file from your database(s) dump directory before restoring. (The default _id indexes will always be created.)

usage: ./mongorestore [options] [directory or filename to restore from]
options:
  --help                  produce help message
  -v [ --verbose ]        be more verbose (include multiple times for more
  verbosity e.g. -vvvvv)
  --version               print the program's version and exit
  -h [ --host ] arg       mongo host to connect to ( <set name>/s1,s2 for sets)
  --port arg              server port. Can also use --host hostname:port
  --ipv6                  enable IPv6 support (disabled by default)
  -u [ --username ] arg   username
  -p [ --password ] arg   password
  --dbpath arg            directly access mongod database files in the given
  path, instead of connecting to a mongod server -
  needs to lock the data directory, so cannot be used
  if a mongod is currently accessing the same path
  --directoryperdb        if dbpath specified, each db is in a separate
  directory
  --journal               enable journaling
  -d [ --db ] arg         database to use
  -c [ --collection ] arg collection to use (some commands)
  --objcheck              validate object before inserting
  --filter arg            filter to apply before inserting
  --drop                  drop each collection before import
  --oplogReplay           replay oplog for point-in-time restore
  --keepIndexVersion      don't upgrade indexes to newest version

The above --help output is from v2.0.3. Check your version and output for the relevant options for your version of mongorestore.

bsondump

bsondump v1.6+

This takes a bson file (typically from mongodump) and converts it to json/debug output. Passing type=debug outputs an indented format that shows the type and size for each object.

usage: bsondump [options] <bson filename>
options:
  --help                  produce help message
  -v [ --verbose ]        be more verbose (include multiple times for more
  verbosity e.g. -vvvvv)
  --version               print the program's version and exit
  --objcheck              validate object before inserting
  --filter arg            filter to apply before inserting
  --type arg (=json)      type of output: json,debug

The debug format displays extra debug information for each field, including the type and size, in an indented form. The debug option also tries to validate strings are valid utf-8.

See Also
mongoexport

The mongoexport utility is a command line tool that takes a collection and exports to either JSON or CSV. You can specify a filter for the query, or a list of fields to output.

Neither JSON nor TSV/CSV can represent all data types. Please be careful not to lose or change data (types) when using this. For full data fidelity, or backups, please use `mongodump`.

If you want to output CSV, you have to specify the fields in the order you want them output.

**Command Line**

```plaintext
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 dbpath specified, each db is in a separate directory
  -q [ --query ] arg      query filter, as a JSON string
  -f [ --fields ] arg     comma seperated list of field names e.g. -f name,age
  --csv                   export to csv instead of json, requires -f
  -o [ --out ] arg        output file; if not specified, stdout is used
```

**Dates**

To pass a date in a query on the command line, use the syntax:

```plaintext
Date(<millis_since_epoch>)
```

You can get the numeric value for a date in the mongo shell:

```plaintext
$ mongo --nodb
> Date("2011-01-12T12:00:00Z").valueOf()
129483600000
```

and then exit the shell and use that value in your mongoexport `-q` parameter:

```plaintext
mongoexport -q "(when:new Date(129483600000))" ...
```

**Sorting**

Example with the `$orderby` meta query operator:
$ mongoexport  -q '{ $query: {age: {$lte:40}}, $orderby: {age:-1} }'

Escaping your command line

If you are running a shell such as bash, it will interpret certain characters, so be sure to escape them for example the dollar sign. The echo command is a good way to test that the escaping is correct:

```bash
$ echo "{'facebookName':{"$ne":null}}"
{"facebookName":null}
$ echo "{"facebookName":{"\$ne":null}}"
{"facebookName":{"$ne":null}}
```

Durability and Repair

MongoDB (specifically, the mongod process) is normally ran with journaling enabled. This makes it crash-safe.

Journaling is on by default in v2.0+ for 64 bit builds. (Use journaling, leave it enabled.)

**Journaling Enabled**

If you are running with Journaling you should not do a repair to recover to a consistent state. When you start with journaling files they will automatically be replayed to a consistent state.

⚠️ The --dur option was used before 1.8; now the option is --journal, and is on by default in version 1.9.2+ on 64-bit platforms

When using journaling, you may see the message:

```
**************
old lock file: mongod.lock. probably means unclean shutdown,
but there are no journal files to recover.
this is likely human error or filesystem corruption.
found 23 dbs.
see: http://dochub.mongodb.org/core/repair for more information
**************
```

You may want to check:

- If someone moved the journal files
- The integrity of your disk.

Replication without Journaling

If you have a replica set then it is favorable to wipe the corrupt/suspect data and re-sync the failed node from scratch or a backup than to do a repair. Repair makes a best effort attempt to put the database files in a useable state, but if corruption or bad data is found then it is skipped (and will be lost after the repair).

No Replication nor Journaling

Recent Backup

If you have a recent backup then it makes sense to use that instead of repair if you are concerned with application data consistency.

Repair Command

When not using journaling (--nojournal), after a machine crash or kill -9 termination, run the repairDatabase command. This command will check all data for corruption, remove any corruption found, and compact data files a bit. Repair is analogous to running fsck for a file system.

When journaling is enabled, it should not be necessary to run repair. However one could still use the repair command to compact a database.
From the command line:

```
mongod --repair
```

From the shell (you have to do for all dbs including local if you go this route):

```
> db.repairDatabase();
```

During a repair operation, `mongod` must store temporary files to disk. By default, `mongod` creates temporary directories under the dbpath for this purpose. Alternatively, the `--repairpath` command line option can be used to specify a base directory for temporary repair files.

Note that repair is a slow operation which inspects the entire database.

After running with `--repair`, `mongod` will start up normally.

⚠️ When running the repairDatabase command on a non-primary server (replica set secondary), you will get an error stating that the server is not master. In order to run the repair, restart the server without the `--replSet` option so that the server is in single server mode, and run the repair. When you restart, make sure to do it on a different port, so as not to confused the other members. Then restart one more time with the `--replSet` option on. This may put the replica server back in a consistent state, but it is highly recommended to check the data validity by comparing a dump of the master/primary and repaired-replica. If there is a suspicion of data being corrupted, it is safer to resync the replica from scratch.

⚠️ Because `mongod` rewrites all of the database files during the repair routine, if you do not run `--repair` under the same user account as `mongod` usually runs, you will need to run `chown` on your database files to correct the permissions before starting `mongod` again.

`mongod.lock`

Do not remove the `mongod.lock` file. If `mongod` is unable to start, use one of the methods above to correct the situation.

Removing the lock file will allow the database to start when its data may be corrupt. In general, you should never force the database to start with possibly corrupt data. In an emergency situation, you may want to remove the lock file to pull whatever data you can off the server. If you have ever manually removed the lock file and started the server back up, you should not consider that server "healthy."

**Checking Data Integrity**

You can use the `validate` command on to check if the contents of a collection are valid.

For example, here we validate the `users` collection:
> db.users.validate();
{
"ns" : "test.users",
"result" : "validate",

details: 0x1243dbbc offset: 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 w/o headers
deletedList: 0000000000000000
deleted: n: 1 size: 7956
nIndexes: 2

test.users.$_id_ keys: 1


test.users.$username_1 keys: 1",
"ok" : 1,
"valid" : true,
"lastExtentSize" : 8192
}

This is a slow command, as it has to check every document in a collection.

**If journaling is disabled**

If the databases exited uncleanly and you attempt to restart the database, `mongod` will

**************
old lock file: /data/db/mongod.lock. probably means unclean shutdown
recommend removing file and running --repair
see: http://dochub.mongodb.org/core/repair for more information
**************

Then it will exit.

**See Also**

- What About Durability? (MongoDB Blog)
- `fsync` Command
- MongoDB (Single-Server) Data Durability Guide

**Security and Authentication**

- Running in a Trusted Environment (without `--auth`)
- Securing the Network
  - Firewall Rules
  - IP Address Binding
  - TCP Port Numbers
- Running in Secure Mode (with `--auth` or `--keyFile`)
  - Logging in as an admin user
  - Enabling Secure Mode
  - Configuring
  - Viewing Users
  - Changing Passwords
  - Deleting Users
- Secure Mode with Sharded Clusters and Replica Sets
  - About the Key File
  - Security and Sharding
- FAQ
MongoDB includes authentication and coarse-grained (to date) access control functionality. By default, security mode is off. When off, it is then required that one runs the database in a trusted (network restricted) environment.

Secure mode requires v2.0+ when using sharding. Older sharded environments must run with the database in a trusted environment.

Please also see the replica set authentication documentation.

Running in a Trusted Environment (without --auth)

Trusted environment / network is the default option and is simplest. It is often valid to run the database in a trusted environment with no in-database security and authentication (much like how one would use, say, memcached). Of course, in such a configuration, one must be sure only trusted machines can access database TCP ports - that is, the network must be secured or locked down. If dedicated servers are used for the database, one option would be to simply make them completely inaccessible externally.

Securing the Network

Firewall Rules

Given the default ports here are the basic firewall rules needed.

<table>
<thead>
<tr>
<th>Process</th>
<th>Direction</th>
<th>Ports</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mongod</td>
<td>Incoming</td>
<td>27017/*</td>
<td>client connections</td>
</tr>
<tr>
<td>mongod</td>
<td>Outgoing</td>
<td>*/27017</td>
<td>replication from other members, shard migrations</td>
</tr>
<tr>
<td>mongos</td>
<td>Incoming</td>
<td>27017/*</td>
<td>client connections</td>
</tr>
<tr>
<td>mongos</td>
<td>Outgoing</td>
<td>*/[27019, 27018]</td>
<td>connections to shards and config servers</td>
</tr>
</tbody>
</table>

Notes:
- In a sharded environment:
  - All the mongo processes (mongos, mongod, mongod --configsvr) in the cluster should be able to connect to each other.
  - Clients must be able to connect to the mongos processes; however, they can be blocked from the mongod's.
- In a non-sharded replica set environment:
  - All clients need to be able to connect to all non-hidden replica set members.
  - All members of a replica set (the mongod processes that is) need to be able to communicate with each other.

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.

For example this could be set to 127.0.0.1, the loopback interface, to make mongod only listen to requests from the same machine (localhost). Or, on a machine with two interfaces, we might want to listen only on the private network.

To enable listening on all interfaces, remove the bind_ip option from your server configuration file.

TCP Port Numbers

MongoDB's default TCP port numbers are as follows:

- Standalone mongod: 27017
- mongos: 27017
- shard server (mongod --shardsvr): 27018
- config server (mongod --configsvr): 27019
- web stats page for mongod: add 1000 to port number (28017, by default). Most stats pages in the HTTP UI are unavailable unless the --rest option is specified. To disable the "home" stats page use --nohttpinterface. (This port should be secured, if used, however, the information on the stats home page is read only regardless.)

You can change the port number using the 'port' option when starting mongod. This is not recommended.
Running in Secure Mode (with --auth or --keyFile)

MongoDB supports authentication and simple coarse-grained access control in “Secure Mode”. This is enabled with the --auth and --keyFile command line parameters.

You authenticate a username and password in the context of a particular database. Once authenticated, a normal user has full read and write access to the database. You can also create read-only users that only have read access.

The admin database is special. Several administrative commands can only run on the admin database (so can only be run by an admin user). Also, users with access to the admin database also have read and write access to all other databases on the server/cluster.

Logging in as an admin user

Although admin user accounts can access any database, you must log into the admin database. For example, if someAdminUser has an admin account, this login will succeed:

```bash
> use admin
> db.auth("someAdminUser", password)
> // and then if desired switch databases:
> use test
```

and this one will fail:

```bash
> use test
> db.auth("someAdminUser", password)
```

Enabling Secure Mode

To enable secure mode, run the mongod process with the --auth option (or --keyFile for replica sets and sharding). You must either (1) have added a user to the admin db before starting the server with --auth, or (2) add the first user from a connection (you cannot add the first user from a connection that is not local with respect to the mongod process).

Configuring

We should first create an administrator user for the entire db server process. This user is stored under the special admin database.

If there are no admin users, one may access the database from the localhost interface without authenticating. Thus, from the machine running the database (and thus on localhost), run the database shell and configure an administrative user:

```bash
$ mongo localhost/admin
> // we are using database admin
> db.addUser("theadmin", "anadminpassword")
```

We now have a user created for database admin. Note that if we have not previously authenticated, we now must if we wish to perform further operations, as there is now an admin user.

```bash
> db.auth("theadmin", "anadminpassword")
```

Now, let's configure a "regular" user for another database.
> // give joe read/write access to the projectx database
> use projectx
> db.addUser("joe", "passwordForJoe")

Finally, let's add a readonly user (v1.4+).

> use projectx
> db.addUser("guest", "passwordForGuest", true)

### Viewing Users

User information is stored in each database's `system.users` collection. For example, on a database `projectx`, the `projectx.system.users` collection contains user information.

We can view existing users for the current database with the query:

```
> db.system.users.find()
```

### Changing Passwords

The shell `addUser` command may also be used to update a password: if the user already exists, the password simply updates.

Some Mongo drivers provide a helper function equivalent to the `db` shell's `addUser` method.

### Deleting Users

To delete a user from the shell:

```
> db.removeUser( username )
> // or
> db.system.users.remove( { user: username } )
> // check it worked:
> db.system.users.find()
> // check we are on the db we intended:
> db
```

### Secure Mode with Sharded Clusters and Replica Sets

**The keyFile is required!**

You are required to use the `keyFile` option for sharding or replica sets in secure mode. Just using the `--auth` option will not work.

Replica sets and sharding can use authentication in v2.0+ (replica sets without sharding in v1.8). From the client's perspective, it is identical to using single-server authentication: connect to your mongos (if using sharding), create users, authenticate when you make connections.

**Read-only users**

In version 2.0, sharded clusters do not enforce read-only limitations. Do not use read-only users in sharded clusters in version 2.0. This has been fixed in 2.2+

The only difference is that the servers use a `key file` to authenticate internal communication. A key file is basically a plaintext file which is hashed and used as an internal password.

To set up replica sets and/or sharding with authentication:

1. Create a key file that can be copied to each server in the set. A key file is composed of characters in the Base64 set, plus whitespace and newlines (see `About the Key File` for details).
2. Modify this file's permissions to be only readable by the current user.
3. Start each server in the cluster (including all replica set members, all config servers, and all mongos processes) with the `--keyFile`
4. Each client connection to the database must be authenticated before it can be used, as with single-server authentication.

You do not need to use the `--auth` option, too (although there's no harm in doing so), `--keyFile` implies `--auth`. `--auth` does not imply `--keyFile`.

**About the Key File**

A key file must contain at least six Base64 characters and be no larger than 1KB (whitespace included). Whitespace characters are stripped (mostly for cross-platform convenience), so the following keys are identical to the database:

```bash
$ 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
```

As the key is stored in a file, you won't have to type it in much, so make it long and with a good degree of randomness within it – 32 characters long for example. Perhaps generate with `mkpasswd` plus extra appended characters from the administrator setting up the cluster.

If you run `mongod` with `-v`, the key will be printed in the log.

---

**Security and Sharding**

To secure a shard cluster using the Secure Mode, you must enable secure mode on all components of the shard cluster. Use the `--keyFile` option on all `mongod`, config database instances, and shard `mongod` instances.

There are two classes of security credentials in a shard cluster: credentials for "admin" users (i.e. for the "admin" database) and credentials for all other databases. These credentials reside in different locations within the shard cluster and have different roles:

1. Admin database credentials reside on the config databases, to receive admin access to the cluster you must authenticate a sessions while connected to a `mongos` instance using the "admin" database.
2. Other database credentials reside on the primary shard for the database.

This means that you can authenticate to these users and databases while connected directly to the primary shard for a database; however, for clarity and consistency all interactions between the client and the database should use a `mongos` instance.

---

**FAQ**

- Are passwords sent over the wire encrypted?
  - Yes. (Actually a nonce-based digest is passed.)

- Are database operations, after authenticating, passed over the wire encrypted?
  - Not by default. See the SSL doc page for more information.

- How do I report a potential security issue in MongoDB?
  - Please email security@10gen.com with any and all relevant information.

---

**Admin UIs**

MongoDB does not include a GUI-style administrative interface. Instead most administration is done from command line tools such as the mongo shell. However some UI's are available as separate community projects and are listed below. Some are focused on administration, while some focus on data viewing.

See also:

- The Mongo Monitoring Service (MMS) from 10gen.
- Tim Gourley's blog has a good summary of the tools.
- The built-in replica set admin UI page.
Tools

- Edda
- Fang of Mongo
- UMongo (formerly JMongoBrowser)
- MongoExplorer
- MongoHub
- MongoVision
- MongoVUE
- mViewer
- Opricot
- PHPMoAdmin
- RockMongo
- Genghis
- Mecclipse
- Humongous
- MongoDB ODA plugin for BIRT
- Monad Management for MongoDB

- Commercial
  - Database Master

- Data Viewers
  - mongs

Edda

Edda is a log visualizer. It takes logs as input and creates a timeline of notable events in the set. It can be installed via pip:

```bash
$ pip install edda
```

Screen shot:

See also: Github source

Fang of Mongo

- [http://github.com/Fiedzia/Fang-of-Mongo](http://github.com/Fiedzia/Fang-of-Mongo)

A web-based user interface for MongoDB build with django and jquery.
It will allow you to explore content of mongodb with simple but (hopefully) pleasant user interface.

Features:

- field name autocompletion in query builder
- data loading indicator
- human friendly collection stats
- disabling collection windows when there is no collection selected
- twitter stream plugin
- many more minor usability fixes
- works well on recent chrome and firefox

To track progress on twitter: @fangofmongo

UMongo (formerly JMongoBrowser)

- home page: http://edgytech.com/umongo/
- github: http://github.com/agirbal/umongo
- download: https://github.com/agirbal/umongo/downloads

UMongo is a GUI app that can browse and administer a MongoDB cluster. It is available for Linux, Windows and Mac OSX.

MongoExplorer

- http://mongoexplorer.com/

MongoExplorer is a MongoDB management tool, written in Silverlight (.net – works in windows/osx/?linux?).

Features:

- Easy to use
- Shows all the collections and documents of the database
- Uses a convenient tree view for documents
- Drag’n’drop is fully supported
- Document in-place editing
MongoHub


MongoHub is a native OS X GUI.

MongoVision

- [http://code.google.com/p/mongo-vision/](http://code.google.com/p/mongo-vision/)

MongoVision is a MongoDB management tool, written for Prudence.

Features:

- Extended JSON support
- Tabular view
- Click to sort
- Filter boxes to alter query
- Auto-refresh
MongoVUE

- [http://blog.mongovue.com](http://blog.mongovue.com)

MongoVUE is a .NET GUI for MongoDB.

mViewer

- [http://imaginea.com/mviewer](http://imaginea.com/mviewer)

mViewer is a web-based MongoDB administration tool.

Opricot


Opricot is a hybrid GUI/CLI/Scripting web frontend implemented in PHP to manage your MongoDB servers and databases. Use as a point-and-click adventure for basic tasks, utilize scripting for automated processing or repetitive things.

Opricot combines the following components to create a fully featured administration tool:

- An interactive console that allows you to either work with the database through the UI, or by using custom Javascript.
- A set of simple commands that wrap the Javascript driver, and provide an easy way to complete the most common tasks.
- Javascript driver for Mongo that works on the browser and talks with the AJAX interface.
- Simple server-side AJAX interface for communicating with the MongoDB server (currently available for PHP).
PHPMoAdmin

- http://www.phpmoadmin.com/

PHPMoAdmin is a MongoDB administration tool for PHP built on a stripped-down version of the Vork high-performance framework.

- Nothing to configure - place the moadmin.php file anywhere on your web site and it just works!
- Fast AJAX-based XHTML 1.1 interface operates consistently in every browser!
- Self-contained in a single 95kb file!
- Works on any version of PHP5 with the MongoDB NoSQL database installed & running.
- Super flexible - search for exact-text, text with * wildcards, regex or JSON (with Mongo-operators enabled)
- Option to enable password-protection for one or more users; to activate protection, just add the username-password(s) to the array at the top of the file.
- E_STRICT PHP code is formatted to the Zend Framework coding standards + fully-documented in the phpDocumentor DocBlock standard.
- Textareas can be resized by dragging/stretching the lower-right corner.
- Free & open-source! Release under the GPLv3 FOSS license!
- Option to query MongoDB using JSON or PHP-array syntax
- Multiple design themes to choose from
- Instructional error messages - phpMoAdmin can be used as a PHP-Mongo connection debugging tool

PHPMoAdmin can help you discover the source of connection issues between PHP and Mongo. Download phpMoAdmin, place the moadmin.php file in your web site document directory and navigate to it in a browser. One of two things will happen:

- You will see an error message explaining why PHP and Mongo cannot connect and what you need to do to fix it
- You will see a bunch of Mongo-related options, including a selection of databases (by default, the "admin" and "local" databases always exist) - if this is the case your installation was successful and your problem is within the PHP code that you are using to access MongoDB; troubleshoot that from the Mongo docs on php.net

RockMongo


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

**Genghis**

**Meclipse**

**Humongous**

**MongoDB ODA plugin for BIRT**
The MongoDB ODA plugin for BIRT is an Eclipse based plugin which enables you to connect to a Mongo database and pull out data to display in your BIRT report. The interface is simple and an extensive user guide is also provided with the release.

**Monad Management for MongoDB**
An operations and administration management tool for MongoDB, with dashboards, alerting, and monitoring graphs.

[http://monad.citsoft.net/](http://monad.citsoft.net/)

**Commercial**

**Database Master**
Database Master from Nucleon Software.
Seems to be written in .net for windows (windows installer).
Features:
- Tree view for dbs and collections
• Create/Drop indexes
• Server/DB stats
• Support RDMBS (MySQL, postgres, ...)

Data Viewers

mongs

• http://www.whit537.org/mongs/

Starting and Stopping Mongo

• Starting mongod
  • Starting with Default Data Directory and TCP Port
  • Specifying a Data Directory
  • Specifying a TCP Port
  • Running as a Daemon
• Stopping mongod
  • 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. For information about running mongod or mongos as a service in Windows, see Windows Service.

The following examples assume that you are in the directory where the mongod executable resides. mongod is the primary database process that runs on an individual server. mongos is the sharding process. mongo is the administrative shell. This page discusses mongod.

Starting mongod

Starting with Default Data Directory and TCP 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
```

Specifying a Data Directory

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.

Specifying a TCP Port

You can specify a different port for Mongo to listen on for connections from clients using the --port option.
$ ./mongod --port 12345

Use this when running several mongod processes on a single machine. It is highly recommended to use the default port number whenever possible to avoid any confusion.

Note: if you use the --configsvr or --shardsvr command line options, those parameters will select an appropriate port number automatically (more info). Port 27017 is by convention the port to which client applications talk to the cluster. Thus if you are unsharded, mongod would be listening on 27017, but if sharded, the mongos process would be.

**Running as a Daemon**

Note: these options are only available in MongoDB version 1.1 and later.

This will fork the Mongo server and redirect its output to a logfile. As with --dbpath, you must create the log path yourself, Mongo will not create parent directories for you.

$ ./mongod --fork --logpath /var/log/mongodb.log --logappend

**Stopping mongod**

**Control-C**

If you have Mongo running in the foreground in a terminal, you can simply “Ctrl-C” the process. This will cause Mongo to do a clean exit, flushing and closing it’s data files. Note that it will wait until all ongoing operations are complete.

**Sending shutdownServer() message from the mongo shell**

The shell can request that the server terminate.

$ ./mongo
    > use admin
    > db.shutdownServer()

This command only works from localhost or if one is authenticated.

From a driver (where the helper function may not exist), one can run the command

{ "shutdown" : 1 }

If this server is the primary in a replica set, it will go through the following process (version 1.9.1+):

1. Check how up-to-date the secondaries are.
2. If no secondary within 10 seconds of the primary, return that we won’t shut down (optionally pass the timeoutSecs option to wait for a secondary to catch up. 
3. If there is a secondary within 10 seconds of the primary, the primary will step down and wait for the secondary to catch up. 
4. After 60 seconds or once the secondary has caught up, the primary will shut down.

If there is no up-to-date secondary and you want the primary to shut down anyway, you can use force : true:

> db.adminCommand({shutdown : 1, force : true})
> // or
> db.shutdownServer({force : true})

You can also specify timeoutSecs : N, which will keep checking the secondaries for N seconds if none are immediately up-to-date. If any of the secondaries catch up within N seconds, the primary will shut down. If no secondaries catch up, it will not shut down.
Sending a Unix INT or TERM signal

You can cleanly stop mongod using a SIGINT or SIGTERM signal on Unix-like systems. Either ^C, "kill -2 PID," or kill -15 PID will work.

Sending a KILL signal kill -9 will probably cause damage if mongod is not running with Journaling. (Journaling is on by default for 64 bit mongod.) See also: repairDatabase command.

After a hard crash, when not using --journal, MongoDB will say it was not shutdown cleanly, and ask you to do a repair of the database.

Memory Usage

Mongo uses memory mapped files to access data, which results in large numbers being displayed in tools like top for the mongod process. This is not a concern, and is normal when using memory-mapped files. Basically, the size of mapped data is shown in the virtual size parameter, and resident bytes shows how much data is being cached in RAM.

You can get a feel for the "inherent" memory footprint of Mongo by starting it fresh, with no connections, with an empty /data/db directory and looking at the resident bytes.

getCmdLineOpts command

The getCmdLineOpts command return both raw and formatted versions of the command line options used to start mongod.
use admin
db.runCommand({getCmdLineOpts: 1})
{
    "argv" : [
    "mongod",
    "--replSet",
    "replica-set-foo",
    "--logpath",
    "/data/mongod.log",
    "--oplogSize",
    "512",
    "--nojournal",
    "--dbpath",
    "/data/rs-30000",
    "--port",
    "30000",
    "--fork"
    ],
    "parsed" : {
    "dbpath" : "/data/rs-30000",
    "fork" : true,
    "logpath" : "/data/mongod.log",
    "nojournal" : true,
    "oplogSize" : 512,
    "port" : 30000,
    "replSet" : "replica-set-foo"
    },
    "ok" : 1
}

The "parsed" section is new in version 2.x

Logging

- **Command Line Options**
- **Rotating the log files**
- From the mongo shell
- From the unix shell
- **Accessing Logs via Shell**
  - Get a list of available loggers
  - Get a log

MongoDB outputs some important information to stdout while its running. There are a number of things you can do to control this

**Command Line Options**

- **--quiet** - less verbose output *(more details)*
- **-v** - more verbose output. Use more Vs (such as **--vvvvv**) for higher levels of verbosity. To change the logging verbosity on a running instance, you can use the **setParameter Command**
- **--logpath <file>** - output to file instead of stdout
  - If you use **logpath**, you can rotate the logs by either running the **logRotate command** (1.3.4+) or sending SIGUSR1
  - You should always use **--logappend** with **--logpath** to append to the existing log file, instead of overwriting it

**Rotating the log files**

Log file rotation renames the current log file to the same name with a timestamp file extension appended, then continues logging to a new file with the original name. The timestamp is the time that the logRotate command was executed, expressed in UTC (GMT) and formatted as ISO but with dashes instead of colons for the time portion.

For example:

```
$ ./mongod -v --logpath /var/log/mongodb/server1.log --logappend
```

will start mongod with verbose logging to **/var/log/mongodb/server1.log**, appending to any existing log file.
In another terminal, list the matching files:

```
$ ls /var/log/mongodb/server1.log*
server1.log
```

Rotate the log file using one of the methods described below, then list the files again:

```
$ ls /var/log/mongodb/server1.log*
server1.log server1.log.2011-11-24T23-30-00
```

This indicates a log rotation performed at exactly 11:30 pm on November 24th, 2011 UTC, which will be the local time offset by the local time zone. The original log file is the one with the timestamp, and new log lines are now being written to the `server1.log` file.

If another `logRotate` command is given one hour later, an additional file will appear:

```
$ ls /var/log/mongodb/server1.log*
server1.log server1.log.2011-11-24T23-30-00 server1.log.2011-11-25T00-30-00
```

The `server1.log.2011-11-23T23-30-00` file is unchanged from before, while `server1.log.2011-11-25T00-30-00` is the previous `server1.log` file renamed and `server1.log` is a new empty file that will receive new log output.

**From the mongo shell**

```
> use admin
> db.runCommand("logRotate");
```

**Windows**

The `logRotate` command is available on Windows in version 2.0.3 and higher

**From the unix shell**

**Rotate logs for a single process**

```
shell> kill -SIGUSR1 <mongod process id>
```

**Rotate logs for all mongo processes on a machine**

```
shell> killall -SIGUSR1 mongod
```

**Linux**

There is currently a bug with our handling of the SIGUSR1 signal which can cause the server to deadlock in certain cases. For now we recommend using the `logRotate` command from the mongo shell instead.

**Windows**

Windows does not have an equivalent to the unix `kill -SIGUSR1` feature, but the mongo shell can be used from the Windows command line using a JavaScript command file to issue a `logRotate` command.

```
C:\> type logRotate.js
db.getMongo().getDB("admin").runCommand("logRotate")
C:\> mongo logRotate.js
C:\> rem Log files rotated, still at Windows command prompt
```

**Accessing Logs via Shell**
New in 1.9.x
See the getLog Command for more details

Get a list of available loggers

```
> show logs
> db.runCommand( { getLog : "*" } )
```

Get a log

```
> show log global
> db.runCommand( { getLog : "global" } )
```

Command Line Parameters

MongoDB can be configured via command line parameters in addition to File Based Configuration. You can see the currently supported set of command line options by running the database with `-h [ --help ]` as a single parameter:

```
$ ./mongod --help
```

Information on usage of these parameters can be found in Starting and Stopping Mongo.

The following list of options is not complete; for the complete list see the usage information as described above.

The command line parameters passed to mongod can be viewed while the instance is running via the `getCmdLineOpts` command.

**Basic Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`-h</td>
<td>--help`</td>
</tr>
<tr>
<td>`-f</td>
<td>--config &lt;file&gt;`</td>
</tr>
<tr>
<td><code>--port &lt;portno&gt;</code></td>
<td>Specifies the port number on which Mongo will listen for client connections. Default is 27017</td>
</tr>
<tr>
<td><code>--dbpath &lt;path&gt;</code></td>
<td>Specifies the directory for datafiles. Default is /data/db or c:\data\db</td>
</tr>
<tr>
<td><code>--fork</code></td>
<td>Fork the server process</td>
</tr>
<tr>
<td><code>--bind_ip &lt;ip&gt;</code></td>
<td>Specifies a single IP that the database server will listen for</td>
</tr>
<tr>
<td><code>--directoryperdb</code></td>
<td>Specify use of an alternative directory structure, in which files for each database are kept in a unique directory. (more details) (v1.4+)</td>
</tr>
<tr>
<td><code>--journalCommitInterval</code></td>
<td>How often to group/batch commit (ms). Default 100ms</td>
</tr>
<tr>
<td><code>--nojournal</code></td>
<td>Disable journaling. In v2.0+, journaling is on by default for 64-bit</td>
</tr>
<tr>
<td><code>--quiet</code></td>
<td>Reduces amount of log output (more details)</td>
</tr>
<tr>
<td><code>--nohttpinterface</code></td>
<td>Disable the HTTP interface (localhost:28017)</td>
</tr>
<tr>
<td><code>--rest</code></td>
<td>Allow extended operations at the Http Interface</td>
</tr>
<tr>
<td><code>--logpath &lt;file&gt;</code></td>
<td>File to write logs to (instead of stdout). You can rotate the logs by sending SIGUSR1 to the server.</td>
</tr>
<tr>
<td><code>--logappend</code></td>
<td>Append to existing log file, instead of overwriting</td>
</tr>
<tr>
<td><code>--syslog</code></td>
<td>Send all output to syslog. Added in 2.1.0. Not compatible with <code>--logpath</code></td>
</tr>
<tr>
<td><code>--repairpath &lt;path&gt;</code></td>
<td>Root path for temporary files created during database repair. Default is dbpath value.</td>
</tr>
<tr>
<td><code>--cpu</code></td>
<td>Enables periodic logging of CPU utilization and I/O wait</td>
</tr>
<tr>
<td><code>--noauth</code></td>
<td>Turns off security. This is currently the default</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>--auth</code></td>
<td>Turn on security</td>
</tr>
<tr>
<td><code>-v[v[v[v[v]]]]</code></td>
<td>Verbose logging output (-vvvvv is most verbose, -v == --verbose)</td>
</tr>
<tr>
<td><code>--objcheck</code></td>
<td>Inspect all client data for validity on receipt (useful for developing drivers)</td>
</tr>
<tr>
<td><code>--syncdelay arg (=60)</code></td>
<td>Seconds between disk syncs. Do not use zero. (more details)</td>
</tr>
<tr>
<td><code>--diaglog &lt;n&gt;</code></td>
<td>Set oplogging level where n is 0=off (default) 1=W 2=R 3=both 7=W+some reads</td>
</tr>
<tr>
<td><code>--nocursors</code></td>
<td>Diagnostic/debugging option – limits results to single batch request. Do NOT use in production. This option will be removed in 2.1.0.</td>
</tr>
<tr>
<td><code>--nohints</code></td>
<td>Ignore query hints</td>
</tr>
<tr>
<td><code>--noscripting</code></td>
<td>Turns off server-side scripting. This will result in greatly limited functionality</td>
</tr>
<tr>
<td><code>--notablescan</code></td>
<td>Turns off table scans. Any query that would do a table scan fails</td>
</tr>
<tr>
<td><code>--noprealloc</code></td>
<td>Disable data file preallocation</td>
</tr>
<tr>
<td><code>--smallfiles</code></td>
<td>Use a smaller initial file size (16MB) and maximum size (512MB)</td>
</tr>
<tr>
<td><code>--nssize &lt;MB&gt;</code></td>
<td>Specifies .ns file size for new databases</td>
</tr>
<tr>
<td><code>--slowms &lt;value&gt;</code></td>
<td>Specifies the threshold (in milliseconds) above which long-running queries will appear in the log, and in the system.profiler collection if profiling is enabled.</td>
</tr>
<tr>
<td><code>--sysinfo</code></td>
<td>Print system info as detected by Mongo and exit</td>
</tr>
<tr>
<td><code>--unixsocket</code></td>
<td>disable listening on unix sockets (will not create socket files at /tmp/mongodb-&lt;port&gt;.sock)</td>
</tr>
<tr>
<td><code>--upgrade</code></td>
<td>Upgrade database files to new format if necessary (required when upgrading from &lt;= v1.0 to v1.1+)</td>
</tr>
</tbody>
</table>

**Replica Set Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--replSet</code> <code>&lt;setName&gt;[/&lt;seedlist&gt;]</code></td>
<td>Use replica sets with the specified logical set name. Typically the optional seed host list need not be specified.</td>
</tr>
<tr>
<td><code>--oplogSize &lt;MB&gt;</code></td>
<td>Custom size for replication operation log</td>
</tr>
</tbody>
</table>

**Master/Slave Replication Options**

Please use replica sets instead of master/slave.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--slave</code></td>
<td>Designate this server as a slave in a master-slave setup</td>
</tr>
<tr>
<td><code>--source &lt;server:port&gt;</code></td>
<td>Specify the source (master) for a slave instance</td>
</tr>
<tr>
<td><code>--only &lt;db&gt;</code></td>
<td>Slave only: specify a single database to replicate</td>
</tr>
<tr>
<td><code>--arbiter &lt;server:port&gt;</code></td>
<td>Address of arbiter server</td>
</tr>
<tr>
<td><code>--autoresync</code></td>
<td>Automatically resync if slave data is stale</td>
</tr>
<tr>
<td><code>--oplogSize &lt;MB&gt;</code></td>
<td>Custom size for replication operation log</td>
</tr>
<tr>
<td><code>--fastsync</code></td>
<td>If the node has a completely up-to-date copy of the data, use this option to let it know it can skip the resync. Be careful – the server will assume it is caught up completely and if not so the data will be out of sync permanently.</td>
</tr>
</tbody>
</table>

**--directoryperdb**

By default data files for MongoDB are all created in a single directory. This directory is defined by the `--dbpath` variable and defaults to /data/db.
The "directory per DB" option (--directoryperdb) allows for a separate directory for the files of each database.

**Uses**

The most common use for this option is to enable data to be stored on different physical disks. This is generally accomplished by creating a symbolic link to the appropriate underlying hardware.

**Example**

The following sample demonstrates the difference in directory structures. The `db/` directory is run without "directory per DB", the `db2/` directory is run with "directory per DB".

```plaintext
--quiet

When the `mongod` process is run with `--quiet`, certain messages are suppressed from the logs.

The following log messages are suppressed:

1. Connection accepted & connection closed
2. Commands: like..., drop, dropIndex, diaglogging, validate, clean, ...
3. Replication syncing activity: sleeping, "repl: from... host"

we are not suppressing the actual commands, just suppressing the logging of the execution of those commands

The setting of `quiet` does not influence the setting of `verbose`. A server can be both quiet and verbose.

**Example log entries suppressed**
#1: Connection accepted & closed
[initandlisten] connection accepted from 127.0.0.1:52798 #3
[conn3] end connection 127.0.0.1:52798

#2: Commands
[conn10] CMD: drop test.foo
[conn10] CMD: dropIndexes test.foo

#3: Replication syncing activity
[replslave] repl: sleep 1sec before next pass
[replslave] repl: from host:127.0.0.1:6900

--syncdelay
This command line option specifies the number of seconds between data file flushes/syncs.

The MongoDB journal file flushes almost immediately, but data files are flushed lazily to optimize performance.

The default is 60 seconds.

A 0 setting means never, but is not recommended and should never be used with journaling enabled.

On Linux, longer settings will likely be ineffective unless /proc/sys/vm/dirty_expire_centisecs is adjusted also. Generally, set syncdelay to 4x the desired dirty_expire_centiseconds value.

If backgroundFlushing.average_ms is relatively large (>= 10,000 milliseconds), consider increasing this setting as well as dirty_expire_centiseconds.

Under extremely high write volume situations, a higher syncdelay value may result in more journal files as they cannot be rotated (deleted) as quickly.

File Based Configuration

In addition to accepting Command Line Parameters, MongoDB can also be configured using a configuration file. A configuration file can be specified using the \-f or --config command line options. On some packaged installs of MongoDB (for example Ubuntu & Debian), the default file can be found in /etc/mongodb.conf, which is automatically used when starting and stopping MongoDB from the service.

The following example configuration file demonstrates the syntax to use:

```
# This is an example config file for MongoDB.
dbpath = /var/lib/mongodb
bind_ip = 127.0.0.1
noauth = true # use 'true' for options that don't take an argument
verbose = true # to disable, comment out.
```

Parameters

⚠️ Before 2.0
Unfortunately, flag parameters like "quiet" will register as true no matter what value you put after them. So don’t write this: "quiet=false", just comment it out, or remove the line.

In 2.0, you can use option=false in the config file and it will be handled correctly (the option will be ignored).

Suppose you were running 1.8 and using a config file with the contents:

```
auth = false
```

This would actually enable authentication in 1.8. If you upgrade to 2.0, this would suddenly be parsed correctly, turning off authentication (which you might not want, if you’ve been running with it on forever). Thus, if you start up with 2.0+ and you have any option=false options in your config file, you’ll get a warning that looks like:
This is to let you know that this option is being skipped. If you want this option skipped, feel free to ignore this warning.

### Basic database configuration

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbpath</td>
<td>Location of the database files</td>
<td>dbpath = /var/lib/mongodb</td>
</tr>
<tr>
<td>port</td>
<td>Port the mongod will listen on</td>
<td>port = 27017</td>
</tr>
<tr>
<td>bind_ip</td>
<td>Specific IP address that mongod will listen on</td>
<td>bind_ip = 127.0.0.1</td>
</tr>
<tr>
<td>logpath</td>
<td>Full filename path to where log messages will be written</td>
<td>logpath = /var/log/mongodb/mongodb.log</td>
</tr>
<tr>
<td>logappend</td>
<td>Whether the log file will be appended to or over-written (default) at start-up</td>
<td>logappend = true</td>
</tr>
</tbody>
</table>

### Logging

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu</td>
<td>Enable periodic logging of CPU utilization and I/O wait</td>
<td>cpu = true</td>
</tr>
<tr>
<td>verbose</td>
<td>Verbos logging output (same as v = true)</td>
<td>verbose=true</td>
</tr>
<tr>
<td>v[vv]</td>
<td>Level of verbose logging output</td>
<td>vvvvv = true</td>
</tr>
</tbody>
</table>

### Security

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>noauth</td>
<td>Turn authorization off. Off is currently the default</td>
<td>noauth = true</td>
</tr>
<tr>
<td>auth</td>
<td>Turn authorization on. Off is currently the default</td>
<td>auth = true</td>
</tr>
<tr>
<td>keyFile</td>
<td>Private key for cluster authentication.</td>
<td>keyFile = /var/lib/mongodb/key</td>
</tr>
</tbody>
</table>

### Administration & Monitoring

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>nohttpinterface</td>
<td>Disable the HTTP interface. The default port is 1000 more than the dbport</td>
<td>nohttpinterface = true</td>
</tr>
<tr>
<td>rest</td>
<td>Turn on simple rest API</td>
<td>rest = true</td>
</tr>
<tr>
<td>noscripting</td>
<td>Turns off server-side scripting. This will result in greatly limited functionality</td>
<td>noscripting = true</td>
</tr>
<tr>
<td>notablescan</td>
<td>Turns off table scans. Any query that would do a table scan fails.</td>
<td>notablescan = true</td>
</tr>
<tr>
<td>noprealloc</td>
<td>Disable data file preallocation.</td>
<td>noprealloc = true</td>
</tr>
<tr>
<td>nssize</td>
<td>Specify .ns file size for new databases in MB</td>
<td>nssize = 16</td>
</tr>
</tbody>
</table>

### Replication

#### All

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>oplogSize</td>
<td>Custom size for replication operation log in MB.</td>
<td>oplogSize = 100</td>
</tr>
</tbody>
</table>

### Replica Pairs (Deprecated – do not use)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
</table>
**Master-slave Replication**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>master</td>
<td>In replicated mongo databases, specify here whether this is a slave or master</td>
<td>master = true</td>
</tr>
<tr>
<td>slave</td>
<td>In replicated mongo databases, specify here whether this is a slave or master</td>
<td>slave = true</td>
</tr>
<tr>
<td>source</td>
<td>Specify the</td>
<td>source = master.example.com</td>
</tr>
<tr>
<td>only</td>
<td>Slave only: specify a single database to replicate</td>
<td>only = master.example.com</td>
</tr>
<tr>
<td>autoresync</td>
<td>Automatically resync if slave data is stale</td>
<td>autoresync</td>
</tr>
<tr>
<td>fastsync</td>
<td>Indicate that this instance is starting from a dbpath snapshot of the repl peer</td>
<td></td>
</tr>
</tbody>
</table>

**Replica Sets**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>replSet</td>
<td>Use replica sets with the specified logical set name. Typically the optional seed host list need not be specified.</td>
<td>replSet = &lt;setname&gt;[/&lt;seedlist&gt;]</td>
</tr>
</tbody>
</table>

**Sharding**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>shardsvr</td>
<td>Indicates that this mongod will participate in sharding. * This is optional and just changes the default port</td>
<td>shardsvr = true</td>
</tr>
</tbody>
</table>

**Journaling**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>journal</td>
<td>Whether writes will be journaled or not. This feature enables fast recovery from crashes</td>
<td>journal = true</td>
</tr>
<tr>
<td>nojournal</td>
<td>Turns off journaling if it is enabled by default (1.9.2+)</td>
<td>nojournal = true</td>
</tr>
</tbody>
</table>

**Notes**

- Lines starting with octothorpes (#) are comments
- Options are case sensitive
- The syntax is assignment of a value to an option name
- All command line options are accepted, for example "vvvv=true"

**GridFS Tools**

**File Tools**

`mongofiles` is a tool for manipulating GridFS from the command line.

Example:
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 current database to <db name>

db.addUser (username, password)
db.removeUser(username)

db.cloneDatabase(fromhost)
db.copyDatabase(fromdb, todb, fromhost)
db.createCollection(name, { size : ..., capped : ..., max : ... } )

db.getName()
db.dropDatabase()

// runs the collstats] command on each collection in the database
db.printCollectionStats()

db.currentOp() displays the current operation in the db
db.killOp() kills the current operation in the db

db.getProfilingLevel()
db.setProfilingLevel(level) 0=off 1=slow 2=all

db.getReplicationInfo()
db.printReplicationInfo()
db.printSlaveReplicationInfo()
db.repairDatabase()

db.version() current version of the server

db.shutdownServer()
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 } }

See Also

- collStats (stats) command
- Commands

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).

The Windows 2008+ MongoDB build uses newer features of Windows to enhance performance. Use this build if you are running with 64-bit
Windows Server 2008 R2, Windows 7, or greater.
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 Vista
- Windows XP

⚠️ As of MongoDB Server 2.1.2, Windows XP is no longer supported as a server environment.

Windows Azure

Instructions for running MongoDB on Azure on Azure Worker Role (alpha)
MongoDB Installer for Windows Azure VM

AppHarbor

Instructions for running MongoDB on AppHarbor
Sample ASP.NET MVC app that uses MongoDB from AppHarbor

MongoDB on Azure - Overview

Overview

MongoDB on Azure brings the power of the leading NoSQL database to Microsoft’s flexible, open, and scalable cloud.

Windows Azure is the cloud services operating system that serves as the development, service hosting, and service management environment for the Azure Services Platform. Windows Azure provides developers on-demand compute & storage to create, host and manage scalable and available web applications through Microsoft data centers.

Together, MongoDB and Azure provide customers the tools to build limitlessly scalable applications in the cloud.

Deployment Options

Users interested in deploying MongoDB on Azure can do so using Azure Worker Roles (Azure Platform-as-a-Service) or Azure VM (Azure Infrastructure-as-a-Service). For further information, please see the relevant documentation:

Azure Worker Roles (Platform-as-a-Service)

- Introduction
- Configuration
- Deployment

Azure VM (Infrastructure-as-a-Service)

- Introduction
- Windows Installer
- Linux Tutorial

Users deploying MongoDB on Azure may be interested in the following presentations, as well:

- MongoDB Paris 2012: MongoDB on Azure
- MongoDBNYC 2012: MongoDB on Windows Azure
- Hands-On: Deploying MongoDB on Azure
MongoDB on Azure VM

Windows Azure VMs allow you to deploy both Windows and Linux virtual machines to the Windows Azure cloud service. These VMs are similar to local instances or other IaaS services and hence can be used to run MongoDB instances.

Currently Windows Azure VMs are in preview and you need to sign up for the preview to use the feature and deploy MongoDB to it.

### Signing Up

Sign up for ‘Virtual Machines and Virtual Networks' preview feature from the new Azure portal”

In the ‘Preview' portal, go to Account->Preview Features.

---

+++ – The VM Role beta program from the old Azure portal (shown below) is not the same as the preview feature above
MongoDB on Azure VM - Linux Tutorial

In this tutorial, you will learn to deploy a MongoDB replica set to CentOS VMs on Windows Azure and access it from outside Azure. The following are the tasks to achieve this:

1. Sign up for the feature on the Azure portal
2. Set up affinity group
3. Create and set up the virtual machines
4. Install and run MongoDB
5. Configure the replica set

**Signing Up**

Sign up for ‘Virtual Machines and Virtual Networks’ preview feature from the new Azure portal*. More information can be found [here](#).

**Set Up Affinity Group**

The affinity group needs to be set up from the old portal.

1. If in the new Azure Management preview portal, click ‘Preview’ at the top of the screen and then ‘Take me to the previous portal’ to switch to the old portal

2. Once in the old portal, click ‘Hosted Services, Storage Accounts & CDN’
3. Select ‘Affinity Groups’ and then ‘New Affinity Group’
4. In the “Create a New Affinity Group” dialog, enter an affinity group name such as “mongoaffinitygroup” and choose a location
   - Choose one of the newer Azure data centers such as “West US,” “East US” or “West Europe”
Create and Set Up the Virtual Machines

To create a 3-node MongoDB replica set you will be creating 3 VM instances. For this step, log in to the new preview portal, or from the old portal, click ‘Visit the Previous Portal’ at the bottom of the page.

Create Instances Instance 1

1. In the portal, click New->Virtual Machine->From Gallery

2. On the VM OS Selection page, choose ‘OpenLogic CentOS 6.2’ and click the next arrow to continue.
3. On the VM Configuration page, specify values for the parameters
   • Virtual Machine Name - "mongodbrs1"
   • New User Name – "mongouser." This user will be added to the ‘sudoers’ list file
   • New Password box – type a strong password that conforms to the Azure specifications
   • Retype the password to confirm
   • Size – Choose appropriate size from drop down list. For anything but small test instances, choose a size larger than Medium
   • Leave ‘Upload SSH key for Authentication’ unchecked

   Click the next arrow to continue

4. On the VM Mode page, specify values for the required parameters
   • Select ‘Standalone Virtual Machine’
   • DNS – a valid DNS prefix e.g., “mongodbs”
   • Storage Account box – choose ‘Use Automatically Generated Storage Account’
   • In the ‘Region/Affinity Group/Virtual Network’ box, select the affinity group created previously “mongoaffinitygroup”
5. On the VM Options page select ‘None for Availability Set’

6. Click the check mark to create the VM instance
Instance 2

A similar process to creating instance 1.

1. In the portal, click New -> Virtual Machine -> From Gallery
2. On the VM OS Selection page, choose ‘OpenLogic CentOS 6.2’ and click the next arrow to continue
3. On the VM Configuration page, specify values for the parameters
   - Virtual Machine Name – “mongodbrs1”
   - New User Name – “mongouser.” This user will be added to the ‘sudoers’ list file
   - New Password box – type a strong password that conforms to the Azure specifications
   - Retype the password to confirm
   - Size – Choose appropriate size from drop down list. For anything but small test instances, choose a size larger than Medium
   - Leave ‘Upload SSH key for Authentication’ unchecked

   Click the next arrow to continue
4. On the VM Mode page, specify values for the required parameters
   - Select ‘Standalone Virtual Machine’
   - DNS – a valid DNS prefix e.g., “mongodbrs”
   - Storage Account box – choose ‘Use Automatically Generated Storage Account’
   - In the ‘Region/Affinity Group/Virtual Network’ box, select the affinity group created previously “mongoaffinitygroup

   Click next arrow to continue
5. On the VM Options page select ‘None for Availability Set’
6. Click the check mark to create the VM instance

Instance 3

A similar process for creating instance 2. Choose ‘mongodbrs3’ to be the Virtual Machine Name in step 3.

Configure Endpoints

Once the virtual machines are connected you need to configure the endpoints to:

- Allow remote connection
- Allow mongo traffic

Instance 1:

1. In the management portal, click virtual machines and click the name of instance 1, ‘mongodbrs1’
2. Now click on endpoints.
3. The ssh endpoint should be automatically created. For this endpoint ensure the following are set:
   - Protocol – tcp
   - Public Port – 22
   - Private Port – 22
   - Load Balanced – No
4. Create a new endpoint for MongoDB by clicking on ‘Add Endpoint’ at the bottom of the screen.
5. Ensure ‘Add endpoint’ is selected and click the right arrow to go to the next screen.

6. Specify the endpoint details as below:
   - Name : MongoDB-Port
   - Protocol : TCP
   - Public Port : 27018
   - Private Port : 27018
   - Click on check mark to create endpoint
The instance now should have 2 endpoints, 1 for SSH and 1 for MongoDB

Instance 2:

We need to configure the endpoints for instance 2 similar to instance 1:

1. In the management portal, click ‘virtual machines’ and click the name of instance 2
2. Now click on ‘endpoints’
3. The ssh endpoint should be automatically created. Ensure that:
   - Name – SSH
   - Protocol – TCP
   - Private Port – 22
   - Load Balanced – No
4. Now click on Edit Endpoint at the bottom of the screen and set ‘Public Port’ to 23. Click on the ‘check mark’ to update
5. Create a new endpoint for MongoDB by clicking on ‘Add Endpoint’ at the bottom of the screen
6. Ensure ‘Add Endpoint’ is selected and click the right arrow to go to the next screen
7. Specify the endpoint details as below:
   - Name : MongoDB-Port
   - Protocol : TCP
   - Public Port : 27019
   - Private Port : 27019
   - Click on check mark to create endpoint

The instance now should have 2 endpoints, 1 for SSH and 1 for MongoDB

Instance 3:

Create endpoints for instance 3 similar to instance 2 with the following changes:

1. In step 4, set public port to 24
2. In step 7, set public and private ports to be 27020

The instance now should have 2 endpoints, 1 for SSH and 1 for MongoDB

Update OS

Use this optional step to update the operating system on each of your VM instances. Once the machine endpoints have been configured above, you need to log into the machines to update them. More information on this can be found at ‘How to Log on to a Virtual Machine Running Linux’

Use the username and password you used when creating the virtual machine instances.

Once you are connected to the machine, update the operating system by running: `sudo yum update` and following the prompts. This could take some time.

**Note:** When connecting to instances 2 and 3, remember to use ports 23 and 24 and not the default ssh port of 22.

Set up disks

Once the instances are updated, you can then attach a data disk to each of the instances. The data disks will be storing the actual mongodb data as part of --dbpath. More information on Azure data disks can be found at Data Disk Concepts.

To set up the data disk follow the steps outlined below for each of the instances you created:

1. Attach an empty disk to the instance as described in How to: Attach an empty disk
   a. Create a data disk of at least 10 GB
2. Now initialize the data disk by following the steps described at How to: Initialize a new data disk in Linux
3. Also once mounted, create a mongodb data directory by:
   a. Log on onto the instance
   b. Then run `sudo chown `id -u` /mnt/datadrive/` to make the mongouser the owner of the data directory
   c. Run the following command: `mkdir -p /mnt/datadrive/data`

Install and Run MongoDB

As part of this step, you will be using the official 10gen supplied packages to install, configure and run MongoDB as a service using YUM. You
want to install as a service since this would ensure that mongod is started on machine restart also. More information can be found at Install MongoDB on RedHat Enterprise, CentOS, or Fedora Linux doc page.

Install MongoDB

Repeat the following steps on each instance:

1. Log onto the instance
2. Create a `/etc/yum.repos.d/10gen.repo` file to hold information about your repository using your favorite editor as sudo. Place the following configuration in `/etc/yum.repos.d/10gen.repo` file:

   ```
   [10gen]
   name=10gen Repository
   baseurl=[http://downloads-distro.mongodb.org/repo/redhat/os/x86_64]
   gpgcheck=0
   enabled=1
   ```

3. After saving the new `.repo` file, issue the following command to update the local package database: `sudo yum update`

4. Issue the following command (as root or with sudo) to install the latest stable version of MongoDB and the associated tools:

   ```
   sudo yum install mongo-10gen mongo-10gen-server
   ```

5. When this command completes, you have successfully installed MongoDB.
Configure MongoDB

The packages installed in the previous step configure MongoDB using the /etc/mongod.conf file in conjunction with the control script. You can find the init script at /etc/rc.d/init.d/mongod. As part of this step you will edit the mongod.conf file to set the appropriate parameters. If the parameters are commented, make sure to uncomment them.

Instance 1

1. Connect to the instance using ssh or PuTTY
2. As sudo, edit /etc/mongod.conf to set the following parameters:
   - `port = 27018`
   - `dbpath = /mnt/datadrive/data`
   - `logpath = /mnt/datadrive/mongod.log`
   - `replSet = mongors`
3. Save the config file

Instance 2

1. Connect to the instance using ssh or PuTTY
2. As sudo, edit /etc/mongod.conf to set the following parameter:
   - `port = 27019`
   - `dbpath = /mnt/datadrive/data`
   - `logpath = /mnt/datadrive/mongod.log`
   - `replSet = mongors`
3. Save the config file

Instance 3

1. Connect to the instance using ssh or PuTTY
2. As sudo, edit /etc/mongod.conf to set the following parameter:
   - `port = 27020`
   - `dbpath = /mnt/datadrive/data`
   - `logpath = /mnt/datadrive/mongod.log`
   - `replSet = mongors`
3. Save the config file

Start MongoDB

Once the configuration files have been edited, start the database process mongod on each instance by:

1. Log on onto the instance
2. Run the following command to start the process:
   ```
   mongod --config /etc/mongod.conf
   ```
3. This should start the mongod process
4. Verify that mongod start by tailing the log file using the command
tail --f /mnt/datadrive/mongod.log
5. The ‘waiting for connections’ message in the log file indicates mongod is up and running and waiting for client connections. This may take a while as mongod preallocates its journal files.

Configure the Replica Set

At this point in time, you should have mongod running on all 3 of your instances. You can now configure the 3 instances as a replica set by connecting to 1 of the 3 instances from within Azure or from outside.

Connect to the running mongod process using the mongo shell:

1. If connected to the VM instance type the following command where port is 27018 for instance 1, 27019 for instance 2 and 27020 for instance 3: `mongo --port <port number>`
2. If connecting to the mongod process in Azure from your local machine use the following command: `mongo --host mongodbrs.cloudapp.net --port <port number>`
3. In the mongo shell type the following:

```javascript
> conf = {
  id = "mongors",
  members : [
    \{id:0, host:"mongodbrs.cloudapp.net:27018\}
    \{id:0, host:"mongodbrs.cloudapp.net:27019\}
    \{id:0, host:"mongodbrs.cloudapp.net:27020\}\}
>rs.initiate(conf)
```
4. This will start the initialization of the mongodb replica set.

5. Type the command `rs.status()` to check the status of the replica set. Upon successful initialization, you should see 1 of the 3 instances being the ‘Primary’ of the set and the other 2 being the ‘Secondaries’.

6. You have now successfully initiated the replica set.

**Summary**

In this tutorial you have learned how to create a set of CentOS virtual machines on Windows Azure, deploy MongoDB to them and create a replica set out of them. You can access this set from anywhere using the connection string `mongodb://mongodbrs.cloudapp.net:27018, mongodbrs.cloudapp.net:27019, mongodbrs.cloudapp.net:27020/?replicaSet=mongors`. 
More information on MongoDB can be found at http://docs.mongodb.org/manual/.

To create and deploy a replica set to Windows Virtual Machines on Azure you can use the MongoDB Installer for Windows Azure.

**MongoDB on Azure VM - Windows Installer**

The MongoDB Installer for Windows Azure is a tool that can be used to provision a MongoDB replica set cluster into Window Azure VM instances.

The following are the steps to deploy a replica set using the tool

1. Install the MongoDB Installer for Windows Azure
2. Download publish settings file
3. Run the MongoDB Installer

**Prerequisites**

- You need to be signed for Windows Azure VM Preview
- Windows versions: any 32-bit or 64-bit Windows 7 version
- Accounts: you must have administrator access to run the setup program and installer

**Install and Deployment**

*Install the tool.*

If you haven’t done so already, you can download the MongoDB Installer for Windows Azure here. Double click on the msi and accept the license agreement to complete the installation.

*Download your publish settings file:*

The publish settings file is an xml file containing information about your subscription. This information is used by the actual script to create the VM instances in your account. To download the publish settings file, use the downloader start menu item:
Start -> All Programs -> MongoDB Installer for Windows Azure -> DownloadPublicSettings

Sign in with your Windows Live ID credentials when prompted. Remember the path to where the file was downloaded
Run the MongoDB Installer

The MongDB Installer for Azure is a powershell script. To install and configure your replica launch the installer as an Administrator

Start -> All Programs -> MongoDB Installer for Windows Azure -> MongoDB Installer for Windows Azure

Right click on the above start menu item and choose Run as administrator. This will open a power shell window where you can run the script to
configure MongoDB on Azure VMs. You would deploy as:

.\deploy-mongo.ps1 <node-count> <dns-prefix> <image-name> <password> <location>
<pub-settings-file-path> [replica-set-name]

node-count - The number of instances required in the replica set. Setting this to 1 results in a stand alone instance being deployed and
not a replica set
dns-prefix - The DNS prefix that will be used in the FQDN of an instance . For example, if you specify a DNS prefix of myreplicaset, the
URL is http://myreplicaset.cloudapp.net.
Note that the MongoDB Installer for Windows Azure always deploys to the production environment, so the URL never uses a GUID (as is
used for deployments to the staging environment).
The DNS prefix is not case-sensitive.
image-name - The name of the VHD image in your Windows Azure storage account that will be used to create the virtual machines for
the replica set. There are two possibilities:
You can specify the name of an existing image that you’ve created, which must meet these criteria:
It is based on the Windows Server 2008 R2 operating system
It has WinRM enabled with HTTP listener
It is hosted in your storage account
You can specify the name of a new image that the tool will create for you in your storage account. If the installer finds no existing
image with the name you specified, it assumes that you want to create a new image and prompts you to confirm before
proceeding. The new image will meet the requirements listed earlier for an existing image. Note that for this option, the name you
provide cannot match any of the names of existing Windows Azure platform images
password - The password to the Administrator account on the provisioned virtual machines. The same password is used for all virtual
machines (nodes) in the replica set, and the password must meet Windows password complexity requirements.
You will use this password when you connect to the virtual machines via Remote Desktop Protocol (RDP) to perform system
management and troubleshooting activities. Note that RDP is enabled on all virtual machines provisioned by the MongoDB Installer for
Windows Azure.
location - The Windows Azure data center location where your virtual machines will be provisioned. Note that data center locations are
case sensitive.
For example, you might use “North Central US” as a location. If you don’t know which locations are available for your Windows Azure
subscription, you can use either of these methods to see a list of available locations:
Sign in to the management portal on https://windows.azure.com, click New Hosted Service, and then click Choose a Region to
see a dropdown list of locations. Click Cancel when you’re done.
You can use the azure vm location list command to display a list of available locations. Note that the installer installs the
command-line tools if they are missing; however, if you want to install them before you run the installer you can do so with the
pub-settings-file-path - The path to the publish settings file you downloaded earlier.
replica-set-name - This optional can be used to specify the name of your replica set. The default name is rs. This parameter is ignored if
node-count is set to 1, because a single-node deployment is not a replica set.
Troubleshooting

Qn. Error with unauthorized during deployment as seen below
error: The subscription is not authorized for this feature.
error: vm image create command failed


Ans. This usually indicates you do not have access to the VM preview program. Can you verify in the new preview portal that “Preview Features” shows account is active for Virtual machines and Virtual networks. If you have multiple accounts/subscriptions make sure you download the right pubsettings file.

Qn. I am seeing a timeout error from the installer

```
Error: connect ETIMEDOUT code: 'ETIMEDOUT', errno: 'ETIMEDOUT', syscall: 'connect'
Error: connect ETIMEDOUT
at errnoException (net.js:670:11)
at Object.afterConnect [as oncomplete] (net.js:661:19)
```

Ans. This occasionally occurs due to connectivity issues with the Azure service management server or your actual deployed instances. At this point the work around is to retry. Working on making this more robust.

Qn. In spite of meeting the strong password requirements the installer fails in the password setting

Ans. Ensure you do not have a $ in the password as powershell is parsing it out as a parameter.

---

**MongoDB on Azure Worker Roles**

![Warning](image)

The MongoDB Replica Set Azure wrapper is currently a preview release. Please provide feedback, mongodb-dev, mongodb-user and IRC #mongodb are good places!

- Getting the package
- Components
- Initial Setup
- Deploying and Running
  - Running locally on compute/storage emulator
  - Deploying to Azure
- Additional Information
- FAQ/Troubleshooting
- Known issues/Where do I file bugs?

The MongoDB Wrapper for Azure allows you to deploy and run a MongoDB replica set on Windows Azure. Replica set members are run as Azure worker role instances. MongoDB data files are stored in an Azure Blob mounted as a cloud drive. One can use any MongoDB driver to connect to the MongoDB server instance. The MongoDB C# driver is included as part of the package.

**Getting the package**

The MongoDB Azure Wrapper is delivered as a Visual Studio 2010 solution with associated source files. The simplest way to get the package is by downloading it from GitHub. It is recommended using the latest tagged version.

Alternatively, you can clone the repository run the following commands from a git bash shell:

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

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

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

**Components**

Once you have unzipped the package or cloned the repository, you will see the following directories:

- **Setup** - Contains a file called solutionsetup.cmd. Run this before opening the solution file.
- **src** - Contains all the project’s source code.
- **src/SampleApplications** - Contains sample applications that you can use to demo MongoDB on Azure. See the listing for more info.
- **lib** - Library files. Includes the MongoDB .NET driver and the downloaded MongoDB Windows binaries.
- **Tools** - Contains tools you might want to run.

**Initial Setup**
We assume you’re running Windows x64 and Visual Studio. If not, install those first; Visual Studio 2010 or Visual Web Developer 2010 should work.

1. Install the latest Windows Azure SDK June 2012
2. Enable IIS on your local machine. This can be done by going to the “Turn Windows features on or off” control panel, under “Programs”. Check “Internet Information Services” and also check ASP.NET under World Wide Web Services|Application Development Features.
3. Clone the project.
4. Before opening either solution file, run Setup\solutionsetup.cmd.
5. Open the solution you want, set the "MongoDB.WindowsAzure.(Sample.)Deploy" project as the StartUp Project, and run it!

The setup script does the following:

- Creates the cloud configs for the 2 solutions ServiceConfiguration.Cloud.cscfg
- Downloads the MongoDB binaries to lib\MongoDBBinaries.

**32-bit note:** The setup script downloads the 64-bit version of MongoDB by default. If you are developing with 32-bit Windows, you will need to download the latest 32-bit MongoDB binaries and place them in lib\MongoDBBinaries yourself. Do this after running solutionsetup.cmd so it won’t overwrite your work.

The prerequisites can be found in the Github readme

Once these are installed, you can open either solution (MongoDB.WindowsAzure.sln) for just the replica set; MongoDB.WindowsAzure.Sample.sln for the replica set and a sample IIS app, MvcMovie, to test it.

### Deploying and Running

**Running locally on compute/storage emulator**

The following instructions are for running the sample application.

To start, you can test out your setup locally on your development machine. The default configuration has 3 replica set members running on ports 27017, 27018 and 27019 with a replica set name of ‘rs’.

In Visual Studio, run the solution using F5 or Debug->Start Debugging. This will start up the replica set and the MvcMovie sample application (if you are in the sample solution).

You can verify this by using the MvcMovie application in the browser or by running mongo.exe against the running instances.

### Deploying to Azure

Once you have the application running locally, you can deploy the sample app solution to Windows Azure. **Note** You cannot execute locally against storage in Azure due to the use of Windows Azure Drive.

- Detailed configuration options are outlined [here](#)
- Step-by-step deployment instructions are [here](#)

### Additional Information

The azure package wraps and runs mongod.exe with the following mongod command line options:

```
--dbpath --port --logpath --journal --nohttpinterface --logappend --replSet
```

MongoDB creates the following containers and blobs on Azure storage:

- Mongo Data Blob Container Name - mongoddatadrive(replica set name)
- Mongo Data Blob Name - mongoddblob(instance id).vhd

### FAQ/Troubleshooting

- Can I run mongo.exe to connect?
  - Yes if you set up remote desktop. Then you can connect to the any of the worker role instances and run `\e:\approot\MongoDBBinaries\bin\mongo.exe`
- Role instances do not start on deploy to Azure
  - Check if the storage URLs have been specified correctly.

### Known issues/Where do I file bugs?

[https://jira.mongodb.org/browse/AZURE](https://jira.mongodb.org/browse/AZURE)

**MongoDB on Azure Worker Roles - Configuration**
MongoDBRole configuration

The following are the configuration operations available as part of the MongoDB Replica Sets Azure package.

**MongoDBRole configuration**

- **Configuration**
  - **.Net Trust Level** - Ensure this is set to **Full Trust**
  - **Instance count** - Set to the number of replica set members you require. Default is 3.
    - Setting this to 1 would run a replica set with 1 instance (equivalent to stand alone)
  - **VM Size** - Choose size of Medium or higher. **Note** The I/O characteristics of small or extra small instance make these configurations unsuitable for MongoDB.

- **Settings**
  - **MongoDBDataDir** - Storage for mongo data files --dbpath. Configure for development or deployment. Data files are in a subdirectory called `data`.
  - **MongoDBDataDirSizeMB** - Size of blob (in MegaBytes) allocated for mongodb data directory. Default is 1GB for the emulator and 100GB for deployed.
  - **ReplicaSetName** - Name of the mongodb replica set. Default is `rs`. This also serves as the blob container name
  - **MongoDBLogVerbosity** - Verbosity to start mongod with. Default is `-v`
  - **RecycleOnExit** - This dictates whether the worker role is recycled when MongoDB exits. Default is true. Set to false initially when debugging.

The following image shows the settings required in a local context.

- **Endpoints**
  - **MongodPort** - Port on which mongod listens. Default is 27017. If running in the Azure emulator this is port for the first instance that is started.
Local Storage

- **MongoDBLocalDataDir** - This specifies the size of the local storage used as a cache for the Windows Azure Drive used as the mongod data directory. Larger sizes provide better read performance.
- **MongodLogDir** - Size of storage allocated for mongod.log. Make this large enough depending on verbosity chosen and instance size.

Use the default settings for local storage as specified below when running on the Azure emulator. When deploying to Azure, change local storage size based on instance size chosen. MongoDBLocalData

In a development environment

The solution can be built and run in a development environment using the Azure emulators as is using Visual Studio 2010 (if adequate disk space is available). Since this solution uses Cloud Drive you cannot run from a development environment against Azure cloud storage. Instead, when you run in your development environment it uses development storage:
The mongod log file is at C:\Users\<user>\AppData\Local\dftmp\Resources\<deploymentid>\directory\MongodLogDir. Note - On a development environment the port mongod listens on would be configured port (27017 by default) + instance id.

The mongod data files are at C:\Users\<user>\AppData\Local\dftmp\wadd\devstoreaccount1\mongoddata\drive{replica set name}\mongoddblob{instance id}.vhd\data. Note - Additionally when the app is running you should be able to access the drives on mounting as you would access any mounted drive.

In the Azure environment

Login to the Azure management portal using your azure credentials

**Azure setup (first time only per deployment)**

Affinity Group

Create an affinity group for your deployment. Choose your required location for the group.

![Create a New Affinity Group](image)

Storage account

Create the required number of storage account(s) to store the blobs. For region/affinity choose the affinity group you created earlier. Note The access key for the storage account created is the one you need to specify in your cloud configuration.

![Create a New Storage Account](image)

Service

Create a new hosted service to host your Mongo package. For region/affinity group use the same affinity group as your storage account. Note for cloud drives the compute and storage instances should be in the same azure domain. Choose do not deploy
Deployment configuration for Mvc Role

If deploying the sample app you can use the default settings as is. You would only need to set the storage settings for diagnostics.

Settings

In the Settings tab:

- **ReplicaSetName** - This should be the same as the replica set name specified in the ReplicaSetRole.
Deployment configuration for Replica Set Role

Configuration

- Ensure VM size for ReplicaSetRole is at least Medium. Larger instance sizes provide more RAM and also greater bandwidth. More information on instance sizes can be found here
- Set the Instance Count equivalent to the required number of replica set members. Default is 3.

Settings

Change connection setting from UseDevelopment storage to actual storage account credentials. It is recommended to use different storage accounts for data and diagnostics. This would allow you to give access to external monitors for diagnostics information without giving access to your data.

- **MongoDBDataDir** - Ensure that connection mode is `http`
- **ReplicaSetName** - This is the name of the replica set in the replica set configuration. This is also the suffix to the blob container created in your storage account. **Note** - This needs to be the same as the replica set name in the client application.
- **MongoDBDataDirSize** - Maximum size of your cloud drive where mongod data files are stored. Currently the maximum size can be 1TB.
- **MongoDBLogVerbosity** - Verbosity for mongod logging. Default is `-v`
- **RecycleOnExit** - This dictates whether the worker role is recycled when MongoDB exits. Default is true. Set to false initially when debugging.
- **Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString** - Ensure that the connection mode is `https`

![Configuration settings](image)

**Note** - If deploying multiple Azure instances make sure you use different storage accounts for each of the deployments or different replica set names if using the same storage account.

Local Storage

Configure the amount of local storage required depending on the VM size chosen. Ensure Clean on recycle role is unchecked. The following are recommendations of Local Storage **Note** All sizes are in MB.

<table>
<thead>
<tr>
<th>VM Size</th>
<th>MongoDBLocalDataDir</th>
<th>MongodLogDir</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium</td>
<td>256000 (250 GB)</td>
<td>51200 (50GB)</td>
</tr>
<tr>
<td>Large</td>
<td>768000 (750 GB)</td>
<td>51200 (50GB)</td>
</tr>
<tr>
<td>Extra Large</td>
<td>1024000 (1000GB)</td>
<td>51200 (50GB)</td>
</tr>
</tbody>
</table>
Package and Publish

Create the Azure package by right clicking on the cloud project and choosing Package. Choose the Service Configuration as Cloud and Build Configuration as Release.


When deploying to the Azure cloud make sure to check deployment warnings/errors to see for any breaking issues. Some common errors are

- Remote desktop is enabled but the remote desktop certificate has not been uploaded to the portal
- https is not chosen for diagnostics connection string
- If deploying the sample MvcMovie application, you can safely ignore the warning that indicates you are only running 1 instance of it.

Troubleshooting

- Excessive Disk Space
- The Linux Out of Memory OOM Killer
- Too Many Open Files
- mongod process "disappeared"
- Socket errors in sharded clusters and replica sets
- 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:
Socket errors in sharded clusters and replica sets

If you're experiencing unexplainable socket errors between nodes in a sharded cluster or replica set, you may want to check your TCP keepalive time. The default Linux keepalive time is 7200 seconds (2 hours); we recommend that you change this to 300 seconds (five minutes).

To check your current setting:

cat /proc/sys/net/ipv4/tcp_keepalive_time

You'll usually see this:

7200

To change this setting:

echo 300 > /proc/sys/net/ipv4/tcp_keepalive_time

Note that you must alter the `tcp_keepalive_time` value on all machines hosting MongoDB processes. This include machines hosting `mongos` routers, config servers, and `mongod` servers. You should not need to restart the `mongo` processes themselves, the setting will take effect immediately.

See Also

- Diagnostic Tools

Excessive Disk Space

- Understanding Disk Usage
  - `local.*` files and replication
  - Datafile Preallocation
  - Recovering Deleted Space
  - Running out of disk space
  - Checking Size of a Collection
  - Helpful scripts

Understanding Disk Usage

You may notice that for a given set of data the MongoDB datafiles in `/data/db` are larger than the data set inserted into the database. There are several reasons for this.

`local.*` files and replication

The replication oplog is preallocated as a capped collection in the local database.

The default allocation is approximately 5% of disk space (64 bit installations).

If you would like a smaller oplog size use the `--oplogSize` command line parameter.

Datafile Preallocation

Each datafile is preallocated to a particular size. (This is done to prevent file system fragmentation, among other reasons.) The first filename for a database is `<dbname>.0`, then `< dbname >.1`, etc. `<dbname>.0` will be 64MB, `<dbname>.1` 128MB, et cetera, up to 2GB. Once the files reach 2GB in size, each successive file is also 2GB.

Thus, if the last datafile present is, say, 1GB, that file might be 90% empty if it was recently created.
Additionally, on Unix, mongod will preallocate an additional datafile in the background and do background initialization of this file. These files are prefilled with zero bytes. This initialization can take up to a minute (less on a fast disk subsystem) for larger datafiles. Pre-filling in the background prevents significant delays when a new database file is next allocated.

On Windows prior to v2.1.2, additional datafiles are not preallocated. NTFS can allocate large files filled with zeroes relatively quickly, rendering preallocation unnecessary.

As soon as a datafile starts to be used, the next one will be preallocated.

You can disable preallocation with the `--noprealloc` command line parameter. This flag is nice for tests with small datasets where you drop the database after each test. It should not be used on production servers.

For large databases (hundreds of GB or more), this is of no significant consequence as the unallocated space is relatively small.

On Linux systems you can use `hdparm` to allocate files to get an idea of how costly allocation might be:

```
  time hdparm --fallocate $((1024*1024)) testfile
```

Recovering Deleted Space

MongoDB maintains lists of deleted blocks within the datafiles when objects or collections are deleted. This space is reused by MongoDB but never freed to the operating system.

To shrink the amount of physical space used by the datafiles themselves, by reclaiming deleted blocks, you must rebuild the database by using `db.repairDatabase()`.

```
  repairDatabase copies all the database records to new file(s). You will need enough free disk space to hold both the old and new database files while the repair is running. Be aware that repairDatabase will block and will take a long time to complete.
```

Rather than compacting an entire database, you can compact just a single collection by using `db.runCommand({compact:'collectionname'})`. This does not shrink any datafiles, however; it only defragments deleted space so that larger objects might reuse it. The compact command will never delete or shrink database files, and in general requires extra space to do its work. Thus, it is not a good option when you are running critically low on disk space.

```
  When testing and investigating the size of datafiles, if your data is just test data, use db.dropDatabase() to clear all datafiles and start fresh.
```

Running out of disk space

If your server runs out of disk space for data files you will see something like this in the log:

```
Thu Aug 11 13:06:09 [FileAllocator] allocating new datafile dbms/test.13, filling with zeroes...
Thu Aug 11 13:06:09 [FileAllocator] will try again in 10 seconds
Thu Aug 11 13:06:19 [FileAllocator] allocating new datafile dbms/test.13, filling with zeroes...
Thu Aug 11 13:06:19 [FileAllocator] will try again in 10 seconds
```

The server remains in this state forever blocking all writes including deletes. However, reads still work. To delete some data and compact (see above), you must restart the server first.

If your server runs out of disk space for journal files, the server process will abort (exit). By default, journal files are created in a subdirectory called "journal" in the data directory, but you may elect to put the journal files on another disk by using a symlink.

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.$_id_
# extents:1
datasize:8192 nrecords:1 lastExtentSize:131072
padding:1
first extent:
loc:0:8100 nxnext:null xprev:null
ns:test.foo.$_id_
size:131072 firstRecord:0:81b0 lastRecord:0:81b0
1 objects found, nob:1
8208 bytes data w/headers
8192 bytes data w/out/headers
deletedList: 0000000000001000000
deleted: n: 1 size: 122688
indexes:0
", "ok" : 1, "valid" : true, "lastExtentSize" : 131072}

**Helpful scripts**

These one-line scripts will print the stats for each db/collection:

```
db._adminCommand("listDatabases").databases.forEach(function (d) {mdb = db.getSiblingDB(d.name);
printjson(mdb.stats())})
```

```
db._adminCommand("listDatabases").databases.forEach(function (d) {mdb = db.getSiblingDB(d.name);
mdb.getCollectionNames().forEach(function(c) {s = mdb[c].stats(); printjson(s)}))
```

**The Linux Out of Memory OOM Killer**

The Linux out of memory killer kills processes using too much memory. On a kill event you will see a line such as the following in the system log file:

```
Feb 13 04:33:23 hostm1 kernel: [279318.262555] mongod invoked oom-killer: gfp_mask=0x1201d2, order=0, oomkilladj=0
```

On such an event, check in future runs that memory is not leaking. This can be checked by verifying that virtualbytes - mappedbytes for mongod is bounded. Check this in `serverStatus` and/or `mongostat`.

> db.serverStatus()
ulimit

ulimit -m with mongod may cause kills that are false positives. This setting is not recommended. MongoDB uses memory mapped files. The entire data is mapped. Over time, if there is no memory pressure, the mongod resident bytes may approach total memory, as the resident bytes includes file system cache bytes for the file pages open and touched by mongod.

swap

Please see the production notes about swap.

Too Many Open Files

If you receive the error “too many open files” or “too many open connections” in the mongod log, there are a couple of possible reasons for this.

First, to check what file descriptors are in use, run lsof (some variations shown below):

```
ls/of | grep mongod
ls/of | grep mongod | grep TCP
ls/of | grep mongod | grep data | wc
```

If most lines include “TCP”, there are many open connections from client sockets. If most lines include the name of your data directory, the open files are mostly datafiles.

ulimit

If the numbers from lsof look reasonable, check your ulimit settings. The default for file handles (often 1024) might be too low for production usage. Run ulimit -a (or limit -a depending on shell) to check.

Use ulimit -n X to change the max number of file handles to X. If your OS is configured to not allow modifications to that setting you might need to reconfigure first. On ubuntu you'll need to edit /etc/security/limits.conf and add a line something like the following (where user is the username and X is the desired limit):

```
user hard nofile X
```

Upstart uses a different mechanism for setting file descriptor limits - add something like this to your job file:

```
limit nofile X
```

High TCP Connection Count

If lsof shows a large number of open TCP sockets, it could be that one or more clients is opening too many connections to the database. Check that your client apps are using connection pooling.

Mongod (hard) connection limit

Currently, there is a limit of 20,000 connections per process; this will be changed when better connection management code (maybe async io) will be implemented.

Data files count with very large databases

If your database is many terabytes, there will be a fairly large number of datafiles – be sure your ulimit setting is appropriate. You will need approximately 500 file handles per terabyte. (Also note if you run a repairdatabase operation, you will need double as two copies will be open at the same time.)

Estimating ulimit setting

You can always set the ulimit to some very large number like 64,000 or you can estimate the number. In general you will need to take your database sizes in GB and divide by 2GB plus the number of database, the number of possible connections and some overhead to get the number of files needed.
Production Deployments

MongoDB can be used for a lot of different projects; the goal is for it to be a fairly general purpose tool. As you can see, many of the use cases are online or operational databases in which someone is writing an application in front of the database. Business intelligence / analytics use is not unusual but usually more specialized. For example, real time dashboarding is highly popular, and when the data set gets really big, things can make a lot of sense. See the use cases page for more on this and for more on when you wouldn't use it.

If you're using MongoDB in production, we'd love to list you here! Please complete this web form and we will add you.

- Archiving
- Cloud Infrastructure Providers
- Content Management
- Ecommerce
- Education
- Email
- Finance
- Gaming
- Government
- Healthcare
- Infrastructure
- Intranet
- Metadata Storage
- Mobile
- News & Media
- Online Advertising and Marketing
- Online Collaboration
- Real-time stats/analytics
- Scientific
- Social Networks
- Telecommunications
- More MongoDB Users
- See also

Archiving

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>craigslist</td>
<td>Craigslist uses MongoDB to archive billions of records.</td>
</tr>
<tr>
<td></td>
<td>- Lessons Learned from Migrating 2+ Billion Documents at Craigslist - MongoSF (May 2011)</td>
</tr>
<tr>
<td></td>
<td>- MongoDB Live at Craigslist - MongoDB blog (May 2011)</td>
</tr>
<tr>
<td></td>
<td>- MongoDB and Craigslist's Data Storage Evolution - MongoSV (December 2010)</td>
</tr>
</tbody>
</table>

Cloud Infrastructure Providers

<table>
<thead>
<tr>
<th>Firebase</th>
<th>Firebase is a cloud service that automatically synchronizes data between clients and cloud servers. Apps built with Firebase can be written entirely with client-side JavaScript, update in real-time out-of-the-box, are inherently scalable, and interoperate well with existing services. Firebase offers developers a massively scaleable real-time backend for web applications. Firebase stores all its data in MongoDB which offers the built-in capability for apps to scale automatically and gives each piece of data its own unique URL stored in JSON documents.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoHQ</td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAP</td>
<td>SAP uses MongoDB as a core component of SAP’s platform-as-a-service (PaaS) offering. MongoDB was selected for the enterprise content management (ECM) section of the platform as its flexibility and scalability will enable SAP to scale its content management service on its PaaS offering to meet customers’ requirements while managing data from different applications. More information available on the 10gen blog.</td>
</tr>
</tbody>
</table>

- Introduction SAP’s Java Platform as Service - Di Guendisch’s Presentation at MongoUK (September 2011)

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTV</td>
<td>MongoDB is the repository that powers MTV Networks’ next-generation CMS, which will eventually be used to manage and distribute content for all of MTV Networks’ major websites. Current deployments include SpikeTV.com and Comedy Central Indecision, and MTV Networks will be rolling out MongoDB on many other major sites within the next year, most likely including gametrailers.com, thedailyshow.com, comedycentral.com, nick.com, and numerous international properties. Read more on the MongoDB blog.</td>
</tr>
</tbody>
</table>

- How MTV Leverages MongoDB for CMS - Jeff Yemin’s Presentation at MongoBoston 2011 (October 2011)
MongoDB is used for back-end storage on SourceForge front pages, project pages, and download pages for all projects.

- **Realtime Analytics using MongoDB, Python, Gevent, and ZeroMQ**
  - Rick Copeland
  - Presentation at MongoSV (December 2011)

- **Achieving 2 Years of Mongo Stability and Performance on SF.net**
  - Mark Ramm's Presentation at MongoDallas (November 2011)

- **Rapid and Scalable Development with MongoDB, PyMongo, and Ming**
  - Mark Ramm's Presentation at MongoDallas (November 2011)

- **Allura - An Open-Source MongoDB-based Document Oriented SourceForge**
  - Rick Copeland
  - Presentation at MongoSF (May 2011)

- **How SourceForge Uses MongoDB**
  - Rick Copeland's Presentation at MongoAtlanta (February 2011)

- **Scaling SourceForge with MongoDB**
  - OSCON Presentation (July 2010)

- **MongoDB at SourceForge**
  - QCon London Presentation (March 2011)

- **How Python, TurboGears, and MongoDB are Transforming SourceForge**
  - PyCon (February 2011)
Wordnik stores its entire text corpus in MongoDB - 3.5T of data in 20 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.
From the Cloud and Back - Presentation at MongoSV (December 2011)
What drove Wordnik non-relational (August 2011)
Building a Directed Graph with Mongol - Tony Tam's Presentation at MongoSF (May 2011)
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)
Keeping the Lights on with MongoDB - Tony Tam's Presentation at MongoSV 2010 (December 2010)
12 Months with MongoDB - Wordnik Blog (October 2010)
B is for Billion - Wordnik Blog (July 2010)
MongoDB: Migration from Mysql at Wordnik - Scalable Web Architecture (May 2010)
Tony Tam's Presentation at MongoSF (April 2010)
What has technology done for words lately? - Wordnik Blog (February 2010)
Harmony is a powerful web-based platform for creating and managing websites. It helps developers and 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.

- Real World Modeling with MongoDB at Harmony - Stephen Smith's Presentation at MongoBoston (September 2010)
- Steve Smith presentation about Harmony at MongoSF (April 2010)

eHow, a web property of Demand Media, receives 100MM+ unique users on a large content database. MongoDB perfectly suited for eHow's content management, as the document model exactly matches the delivery model of documents and pages. The Demand Media team rewrote the legacy .NET/SQL system to PHP/Python/MongoDB in a cachelss, tierless design, all on one machine with replica

MeteKamil uses MongoDB for browsing hundreds of thousands of videos on the websites:
- YouUnbox.com
- YouGamePlay.com
- YouOverClock.com
- DroidFather.com
- YouTouchDown.com
Ecommerce

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
</table>
| CustomInk        | At CustomInk, we believe custom t-shirts have the power to turn your group into a team, elevate gatherings to an event, and make your special moments more... well, momentous. That's why we make it easy to create awesome shirts. CustomInk uses MongoDB for a number of different applications, including:  
  - Supply Chain Manager: MongoDB stores information about t-shirt suppliers and local screen printers.  
  - Content Manager: MongoDB is used to store the product catalog. CustomInk sells hundreds of products, not just t-shirts.  
  - Logging: MongoDB is used for logging interactions on the website and as a central store for application logs.  
Learn more about how CustomInk is using MongoDB from their presentation given at MongoSV 2011.
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.

- Blending RDBMS with NoSQL for e-commerce
  Steve Francia's presentation at MongoDB NYC (June 2011)
- Augmenting RDBMS with NoSQL for e-commerce
  Steve Francia's presentation at Pg East (March 2011)
- MongoDB & Ecommerce: A Perfect Combination
  Video from Steve Francia's presentation to the New York MongoDB User Group (October 2010)

Gilt Groupe is an inv only luxury shopping site. Gilt uses MongoDB for real-time ecommerce analytics.

- Gilt CTO Mike Bryzek's presentation at MongoSF in 2010.
- Hummingbird: a real-time web traffic visualization tool developed by Gilt and powered by MongoDB
edelight is a social shopping portal for product recommendations.

- MongoDB: edelight: A Tale of Pain and Success in 10 Slides Fabian Schladitz Presentation at MongoBerlin (October 2010)
- MongoDB: Edelight statt MySQL auf MongoDB FW Exciting Ecommerce (September)

Totsy offers moms on-the-go and moms-to-be access to brand-specific sales, up to 70% off. 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)

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/consultants to PHP, jQuery, and MongoDB who utilize Agile efficiency of their core frameworks to bring PlumWillow from concept-to-launch in a few months.
Shopwiki uses MongoDB as a data store for its shopping search engine, where they commit all the data generated, such as analytics. MongoDB's performance is such that ShopWiki uses it in cases where MySQL would not be practical. ShopWiki also uses 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 MongoNYC ShopWiki D (June 2010)

Yabblr uses MongoDB for everything, including inventory management and orders. Yabblr is an E-Commerce platform that flips 'deals' on its head. Instead of pushing random offers at you, Yabblr empowers you to group together with friends deals on products you select. Yabblr uses MongoDB exclusively as a data store for the entire platform, including:
- User data
- Real time metrics that are used for analyzing user behavior and system health
- Inventory management
- Customer orders
- Event queues and message bus between different systems.

Ordoro uses MongoDB to store orders, inventory and suppliers. Ordoro deals with everything that happens after a shopper clicks checkout on the retailer's website: print packing lists and shipping labels, manage inventory levels, manage dropshipping, manage customers, manage suppliers etc. Ordoro is the backoffice control panel for SMB retailers.

SavingStar uses MongoDB to store our coupon content and service metadata, as well as all of our coupon activation data. The company is the first national, fully digital, eCoupon service, available for free. There's nothing to clip, nothing to print.
**Wedding Paper divas™**

*Wedding Paper Diva*

MongoDB to store our categories and products. The company allows users to create custom wedding invitations online using their personalization tools to make unique and trendy wedding invitations quickly and easily.

**Storeden**

*Storeden*

is a cloud solution totally based on MongoDB. The easiness of the popular shopping websites meets the performance of an advanced online solution. Storeden is revolutionary e-Commerce platform that allows users and web-agencies to create amazing online stores with just a few clicks. Everyone has the chance to create applications, graphic themes and “suites” and sell them to the other users in the Cloud. Storeden can be perfectly integrated with the most common ERP and CRM platforms, allowing users to effectively leverage on Facebook, Ebay, Shopping and Kelkoo as further sales channels. It is natively optimized for SEO.

**Gittigidiyor.com**

*Gittigidiyor.com*

is the biggest e-commerce site in Turkey with more than 7 million registered users. Gittigidiyor.com have a replicated and shard MongoDB cluster for storing and caching a wide range of data.

**iPara**

*iPara*

uses MongoDB at iPara payment pages to log all kinds of user activities. Logging, they analyze logs to determine new designs and processes. iPara offers safe and convenient payment and discount campaigns, providing many advantages for both users and businesses.

**Coinbase**

*Coinbase*

uses MongoDB for their primary data for their web app, API requests, etc. Coinbase is a decentralized digital currency that is changing the world of payments.
Zeel Networks, Inc. uses MongoDB for their primary datastore for their web app, API requests, etc. Coinbase is a decentralized digital currency that is changing the world of payments.

See also: http://www.slideshare.net/mongodb/mongodb-at-ebay

**Education**

**Codecademy**

Codecademy is the easiest way to learn to code online. Driven by the belief that programming is one of the most important skills of the twenty-first century, Codecademy has educated more than one million people in over 100 countries.

**LearnBoost**

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)

**Courseoff**

Courseoff uses MongoDB stores student created schedules as well as the schedule listings themselves.

**Boundless Learning**

Boundless Learning uses MongoDB as a rapid-prototyping technology to develop a robust content management system.

**Foothill-De Anza Community College District**

Foothill-De Anza Community College District migrated a problematic document storage system into a web application backed by MongoDB, and using the Mongo GORM plug-in for Grails. They have indexed (and stored with GridFS) over 300,000 documents (56GB) into a single Mongo database. Mongo is much easier to get online than a comparable RDBMS, and the rich feature set helped significantly reduce the amount of required application coding. All said and done, switching to Mongo for this project has been a big win.
Email

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, geolocate, 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)

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.

- MongoDB at fiesta.cc - Mike Dirolf's Presentation for NYC Meetup Group (November 2011)
- Behind the Scenes at fiesta.cc - Mike Dirolf's Presentation at PyGotham (September 2011)

Finance

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecondMarket</td>
<td>SecondMarket is the online marketplace for alternative assets such as</td>
</tr>
<tr>
<td></td>
<td>private company stock, structured products &amp; bankruptcy claims. SecondMarket uses mongoDB for storing its diverse asset classes in a single collection utilizing the power of the schema free nature of mongoDB. We also store organization news and activity in mongoDB.</td>
</tr>
<tr>
<td></td>
<td>- Mongo &amp; Mongeez on the SecondMarket Engineering Blog - Feb 2012</td>
</tr>
<tr>
<td></td>
<td>- Mongeez the open source, change management tool developed for MongoDB. ]</td>
</tr>
<tr>
<td></td>
<td>- Second Market, MongoDB and Mongeez for Change Management - Michael Lysaght's Presentation at MongoNYC (June 2011)</td>
</tr>
<tr>
<td></td>
<td>- Mongeez at SecondMarket -Michael Lysaght's presentation at the MongoDB User Group (January 2012)</td>
</tr>
<tr>
<td>Athena Capital Research</td>
<td>Athena Capital Research is a quantitative investment manager, specializing in automated trading strategies.</td>
</tr>
<tr>
<td></td>
<td>- How a Hedge Fund Uses MongoDB - Roman ShlyIman's Presentation at MongoNYC (June 2011)</td>
</tr>
<tr>
<td></td>
<td>- Low Latency Event Logging with BSON - Andrew Morrow's Presentation at MongoSV (December 2010)</td>
</tr>
<tr>
<td>Equilar</td>
<td>Equilar uncovers the direct pathways to the most influential high</td>
</tr>
<tr>
<td></td>
<td>net worth individuals, and delivers immediate and actionable insight into</td>
</tr>
<tr>
<td></td>
<td>the equity wealth events that drive business development opportunities. Equilar uses MongoDB to map and analyze the connections between over 300,000 executives and board members in companies worldwide.</td>
</tr>
</tbody>
</table>
**Gaming**

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disney</td>
<td>Disney built a common set of tools and APIs for all games within the Interactive Media Group, using MongoDB as a common object repository to persist state information.</td>
</tr>
<tr>
<td>- A Year with MongoDB: Running Operations to Keep the Game Magic Alive - Curt Steven’s Presentation at MongoSV (December 2011)</td>
<td></td>
</tr>
<tr>
<td>- Disney Central Services Storage: Leveraging Knowledge and skillsets - MongoSF (May 2011)</td>
<td></td>
</tr>
</tbody>
</table>

| IGN Entertainment | 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) |

<p>| WordSquared (formerly Scrabbly) | WordSquared (formerly Scrabbly) is a massively multiplayer online scrabble crossword. Uses MongoDB geospatial indexing. |
| - Mapping Flatland: Using MongoDB for an MMO Crossword Game - Presentation at Mongo Seattle 2011 |
| - Building a Scrabble MMO in 48 Hours - Startup Monkeys Blog (September 2010) |</p>
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EA download manager</td>
<td>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.</td>
</tr>
<tr>
<td>Kano Games</td>
<td>Kano Games uses MongoDB to build an online gaming web app that integrates social elements. It is a gaming web app that offers a large collection of html5 and flash games with new ones added daily. MongoDB is used to store all persistent data at Kano Games. Reliability, scalability, performance, redundancy and flexibility were all considered when selecting a persistent storage solution and MongoDB beat out the competition as it best fit Kano Games needs.</td>
</tr>
<tr>
<td>Shadelight</td>
<td>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.</td>
</tr>
<tr>
<td>Freerice</td>
<td>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.</td>
</tr>
<tr>
<td>Lichess.org</td>
<td>Lichess.org uses MongoDB to store more than 3 million chess games. It also contains the analysis, users, forum posts and chat messages. Lichess.org needs blazing fast updates and efficient indexing; mongodb gives it both. Lichess.org is OpenSource! Check the code on <a href="http://github.com/ornicar/lila">http://github.com/ornicar/lila</a>. MongodDB is the database behind Moshen Limited's Facebook game Gambino Poker. MongoDB is used to store all game events and player profiles. We use a 3 node replica-set and RAID 10 EBS Volumes.</td>
</tr>
</tbody>
</table>
MongodDB is the database behind Moshen Limited's Facebook game Gambino Poker. MongoDB is used to store all game events and player profiles. Moshen uses a 3 node replica-set and RAID 10 EBS Volumes.

Hitcents uses MongodDB for primary data storage for their iOS/Android app PictureTHIS. The app is a cooperative social word game.

Gamera Labs uses MongoDB for the Real-Time game High5Poker. The app is a cooperative social word game. MongoDB is used in production for data storage like players information and statistics.

**Government**

The National Archives (UK) made details of 11m records available through an application interface it published as part of an ongoing programme to get more official records online. It is consolidating numerous existing electronic archives, either by porting them directly or putting them in a service wrapper that can communicate with its unified system. The unified system uses MongoDB and is built on a Microsoft software stack.

- National Archives releases public application programming interface for 11m records - ComputerWorld (Sept 2011)
- From SQL Server to MongoDB - Presentation at MongoDB UK (Sept 2011)
The British Government launched a beta of its GOV.UK platform, testing a single domain for that could be used throughout government. According to James Stewart, the Tech Lead on the beta of GOV.UK, “We started out building everything using MySQL but moved to MongoDB as we realised how much of our content fitted its document-centric approach,” said Stewart. “Over time we’ve been more and more impressed with it and expect to increase our usage of it in the future.”

- Colophon for the GOV.UK beta - Feb 2012
- With GOV.UK, British government redefines the online government platform - O’Reilly Radar (Feb 2012)

Sunlight Labs is a community of open source developers and designers dedicated to opening up our government to make it more transparent, accountable and responsible. MongoDB powers the National Data Catalog and the Drumbone API, which is an aggregator of data about members of Congress.

- MongoDB at Sunlight - Luis Martinez’s Presentation Slides
- Civic Hacking - Video from Luigi Montanez’s presentation at MongoNYC (May 2010)
- How We Use MongoDB at Sunlight blog post (May 2010)

In addition, MongoDB has been used for various projects at the federal, state, and city levels in the U.S.

Note: If you are in Washington D.C. area, be sure to check out the annual Mongo conference there ("MongoDC"), as well as local mongo user groups.

Healthcare

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Santosoft</strong></td>
<td>Santosoft uses MongoDB for virtually 100% of its data related stack. They run a fully blown shard cluster for transactional data, as well as analytics. They are building apps and tools for MongoDB and also hope to build the largest common care database globally.</td>
</tr>
<tr>
<td><strong>HolaDoctor</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>vitals</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>TalkAboutHealth</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Earlydoc</strong></td>
<td>Earlydoc models medical data, which means things like master-identity management and data normalization occupy a lot of development time: this, specifically, is where the schemaless design of MongoDB is a huge win.</td>
</tr>
</tbody>
</table>
D Sharp uses MongoDB to store detailed medical observations and perform complex analysis to help people with diabetes better manage their blood sugars. Built using the Express web framework for Node.js and JQuery Mobile, D Sharp is the only diabetes application that supports all types of diabetes on any modern smartphone or browser. This includes iPhone, iPad, Android, Windows Mobile, Blackberry and desktop browsers.

Vigilant Medical uses MongoDB to manage the large, complex, and often sparse metadata surrounding medical images. MongoDB also provides the entire administrative backbone of Vigilant's enterprise web application, ImageShare.

Mini Medical Record is designed to improve medical care. While designed to help everyone, it is especially useful for travelers, and others who may receive care through multiple medical systems. Whether paper or electronic, medical records in cutting edge hospitals are often geared more for billing and medicolegal protection. Critical up-to-date information may be very challenging to discover. In addition, even simple information like emergency contact phone numbers and latest medication lists, may be inaccessible for patients who seek care in settings where they do not routinely receive their care. Mini Medical Record alleviates those issues.

### Infrastructure

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>LexisNexis Risk Solutions</td>
<td>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.</td>
</tr>
</tbody>
</table>

### Intranet

<table>
<thead>
<tr>
<th>SPARC</th>
<th>SPARC uses MongoDB as its primary store for our employee engagement platform. SPARC has been very happy with how simple data migrations have been since leaving SQL behind.</th>
</tr>
</thead>
</table>

A Fortune 10 company uses MongoDB for their internal intranet app for their 100,000+ employees.

### Metadata Storage

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Shutterfly is an Internet-based social expression and personal publishing service. MongoDB is used for various persistent data storage requirements within Shutterfly. MongoDB helps Shutterfly build an unrivaled service that enables deeper, more personal relationships between customers and those who matter most in their lives.

- **Performance Tuning and Scalability** - Kenny Gorman's Presentation at MongoSV (December 2011)
- **The Shutterfly Data Layer and Schema Definitions** - Luciano Resende's Presentation at MongoSV (December 2011)
- **MongoDB Profiling and Tuning** - MongoSF (May 2011)
- **Q & A with Shutterfly Data Architect Kenny Gorman** (January 2011)
- **Sharing Life’s Joy using MongoDB: A Shutterfly Case Study** - Kenny Gorman's Presentation at MongoSV (December 2010)
- **Implementing MongoDB at Shutterfly** from MongoSF (April 2010) Slides and Video
Lulu’s open publishing platform empowers more creators to sell more content to more readers more profitably than ever before. Lulu uses MongoDB to store the bibliographic, product, and file metadata for its creators’ works. MongoDB’s document-based data model makes it easy to manage the differences between print books, eBooks, photo books, and calendars and to keep up with the rapidly evolving world of self-publishing.

- Why we decided NoSQL was right for us,
- How we came to choose MongoDB - Sept 2011 Presentation

3DRepo.org uses MongoDB as a domain specific repository to store scene graph components as well as associated non-linear revision history of 3D assets. The main reasons for choosing MongoDB over alternative NoSQL solutions is the use of BSON documents for storage and retrieval, automated sharding, MapReduce functionality and built-in support for geospatial indexing.

---

**Mobile**

(See also “Social” section.)

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
</table>

---
<table>
<thead>
<tr>
<th>Company</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MoPub</td>
<td>MoPub uses MongoDB for realtime stats, budgeting systems, and user storage.</td>
</tr>
<tr>
<td>Countly</td>
<td>Countly is an extensible, open source mobile analytics solution for mobile application developers. It tracks applications, customer behaviour or game mechanics - so you can focus on increasing user loyalty and engagement. With Countly, collected data is converted into meaningful information in true real-time, thanks to underlying infrastructure with MongoDB, Node.js and Nginx.</td>
</tr>
<tr>
<td>Localytics</td>
<td>Localytics uses MongoDB to process over 100M datapoints every day for their mobile analytics service.</td>
</tr>
<tr>
<td>Mobile Helix</td>
<td>Mobile Helix uses MongoDB to provide fast data read/write for the Mobile Helix Gateway, which analyzes data that needs to be delivered from an enterprise’s network to a mobile device and pre-fetches, batches and schedules its delivery to be as quick and as efficient as possible. MongoDB enables the gateway to seamlessly scale to millions of devices accessing the Mobile Helix system from all over the globe.</td>
</tr>
<tr>
<td>Nearley</td>
<td>Nearley is a mobile geo location application to list all facebook users located within your distance. Nearley uses MongoDB to store user interaction and activities in real-time.</td>
</tr>
</tbody>
</table>
O2 uses MongoDB for several applications, including their Priority Moments location based offer service.

### News & Media

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guardian.co.uk</td>
<td>Evolving from relational to document store - MongoSF (May 2011)</td>
</tr>
<tr>
<td></td>
<td>Why I Chose MongoDB for guardian.co.uk - QCon London (August 2011)</td>
</tr>
<tr>
<td></td>
<td>MongoDB at the Guardian - MongoDB U (Sept 2011)</td>
</tr>
<tr>
<td>Examiner.com</td>
<td>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.</td>
</tr>
</tbody>
</table>
Forbes has been around for nearly 100 years and on the web for more than a decade; recently, the media space has been a changing landscape. One of the changes that Forbes has undertaken starting in 2010 was the “opening up” of its digital and print platforms to a global collection of content creators, marketers and audience members. This necessitated the changing of how the site worked, allowing content to flow easily through the site.

Forbes began to evaluate how they store and serve their content and decided to use MongoDB. They are currently using MongoDB for storing articles and companies data and are working to move more of their core assets onto it.

- Supporting Distributed Global Workforce of Contributors with MongoDB
- An Inside Look At The Guts Of Our Tech Infrastructure (June 2011 Article)
- How Business Insider Uses MongoDB (May 2010 Presentation)
- How This Website Uses MongoDB (November 2009 Article)

Business Insider has been using MongoDB since the beginning of 2008. All of the site’s data, including posts, comments, and even the images, are stored on MongoDB. For more information:

- An Inside Look At The Guts Of Our Tech Infrastructure (June 2011 Article)
- How Business Insider Uses MongoDB (May 2010 Presentation)
- How This Website Uses MongoDB (November 2009 Article)
The New York Times

using MongoDB in a form-building application for photo submissions. Mongo dynamic schema gives producers the ability to define any combination of custom form fields. For more information:

- The New Yr Times R&D Lab and MongoDB
- Building a Better Submission Form - NYTimes Open Blog (May 25, 2010)

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.

CNN Turk uses MongoDB for its infrastructure and content management system, including the tv.cnnturk.com, www.cnnturk.com and video.cnnturk.com portals.

Online Advertising and Marketing

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MongoDB will be the central data store in Magnetic's system, providing low-latency, high-volume reads in a distributed cluster behind its pixel servers, and near-real-time aggregate counts to its clients for forecasting. Magnetic is excited by the speed, replication features, flexible data model, and map-reduce style query access, which will let it dramatically simplify its system relative to the current architecture using a key-value store.

- Tracking & Analytics with MongoDB at Signpost; MongoDB for Online Advertising - Matt Insler's Presentation for NYC Meetup Group
- MongoDB Analytics for Online Advertising at Magnetic - Webinar hosted by Mark Weiss

Aggregate Knowledge is an IaaS company that combines data management and multi-touch attribution to make media accountable helping advertisers, agencies, ad networks, DSPs, SSPs, and publishers manage and exploit all of their data around a single view of the user. With more data to process, and less time at hand, Aggregate Knowledge pairs MongoDB with Fusion-io flash memory solutions for a winning combination.

MediaMath is the leader in the new and rapidly growing world of digital media trading.

Konverta.ru is the first Real Time Bidding (RTB) ad exchange on the Russian and CIS online advertising market. MongoDB is used to store all ads, ad impressions, clicks, and other data, as well as for real-time reporting and optimization.

G5 is the largest and fastest growing provider of vertical-specific local marketing solutions that help mid-market companies get found online, generate more qualified leads, convert more leads into new customers, track marketing performance - including offline, and optimize to the marketing sources with the best return on investment. G5 migrated its analytics platform from MySQL to MongoDB due to the heavy demands of storing & processing analytical data. MongoDB has proven to be fast, scalable, flexible, & maintainable. Best of all, MongoDB is supported by a fantastic community!

Yodle uses MongoDB to persist queues of items to be synchronized with several partner APIs. Mongo is ideally suited for this update-heavy performance sensitive workload.

The Localstars platform makes it easy for local advertisers to build and manage locally targeted advertising campaigns. The new Localstars advert server cluster uses sharded MongoDB to provide super high performance real time ad serving decision making and campaign statistics.
EveryScreen is built on a scalable, high-performance, real-time mobile advertising bidding platform in Amazon EC2.

Rafflecopter is a web app that helps online publishers run sweepstakes on their own blogs & other sites. Rafflecopter logs and stores every entry to every contest that’s ever been run, all contest details, user profiles, various logs/metrics, etc.

LocalEver is a location based online market place to buy, sell & rent residential/commercial properties, automobiles, and a host of other products. MongoDB is the backbone of the system, which powers the whole range of features from geo spatial search to caching.

### Online Collaboration

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trello</td>
<td>Trello is a collaboration tool that organizes your projects into boards. In one glance, Trello tells you what's being worked on, who's working on what, and where something is in a process. Trello stores all historical and non-ephemeral data in MongoDB. The Trello team was attracted to MongoDB for its speed. MongoDB offers very fast writes, faster reads, and better denormalization support — it allows them to store the data of each individual card on the Trello board in a single document in the database and still have the ability to query into (and index) subfields of the document. They were also attracted to MongoDB’s reliability: it is really easy to replicate, back up, and restore. Other benefits for the Trello team: using a loose document store makes it easy to run different versions of the Trello code against the same database without fooling around with DB schema migrations. This has a lot of benefits when they push a new version of Trello; there is seldom (if ever) a need to stop access to the app while we do a DB update or backfill.</td>
</tr>
</tbody>
</table>
|         | - The Trello Stack January 2012 from the Fog Creek Blog  
|         | - Trello Architecture |
| Flowdock | Flowdock is a modern web-based team messenger, that helps your team to become more organized simply by chatting. Flowdock backend uses MongoDB to store all messages. |
|         | - Why Flowdock migrated from Cassandra to MongoDB - Flowdock Blog (July 2010) |
**Moxie Software™** is an innovative software company that provides integrated social enterprise software for employee and customer engagement through its Spaces™ by Moxie platform. Designed in the social media era and using MongoDB, Employee Spaces™ is an enterprise social software built for “The Way People Work.” It enables employees to collaborate on projects, innovate by sharing and co-creating knowledge and accelerate innovation.

**Sherl.tv** is a service for sharing plain text terminal screencasts from a Unix terminal that uses MongoDB as the main data storage for records.

### Real-time stats/Analytics

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuit</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>- Deriving deep customer insights using MongoDB - Presentation at MongoSV (December 2010)</td>
<td></td>
</tr>
<tr>
<td><strong>Buddy Media</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>- Social Analytics on MongoDB (Video and Slides) - Presentation from February 2011 New York MongoDB User Group</td>
<td></td>
</tr>
<tr>
<td>- The New Buddy Media Publisher: What You Need To Know - Buddy Media blog (November 2010)</td>
<td></td>
</tr>
<tr>
<td><strong>bit.ly</strong> allows users to shorten, share, and track links. bit.ly uses MongoDB to store user history. For more information:</td>
<td></td>
</tr>
<tr>
<td>- bit.ly user history, auto-sharded presentation at MongoNYC (May 2010)</td>
<td></td>
</tr>
</tbody>
</table>
**10gen** is the initiator, contributor and continual sponsor of the MongoDB project. 10gen built **MongoDB Monitoring Service (MMS)**, a scalable software as a service monitoring tool built using MongoDB. MMS displays data in charts that track performance, resource utilization, availability, and response times. 10gen built MMS to provide operational insight into thousands of MongoDB deployments, and uses it to better diagnose and resolve customer issues.

**Chartbeat** is a revolutionary real-time analytics service that enables people to understand emergent behaviour in real-time and exploit or mitigate it. Chartbeat stores all historical analytics data in MongoDB.

- **MongoDB & EC2: A love story?** - NY MongoDB User Group (Oct 2011)
- **The Secret Weapons Behind Chartbeat** - Kushal's coding blog (April 2010)
- **Kushal Dave's Presentation at MongoNYC** (May 2010)

**Server Density** is a server monitoring tool from Boxed Ice. They have used MongoDB since June 2009 and are now processing billions of documents every month. Server Density also includes an addon for **MongoDB monitoring** and they have written extensively about MongoDB itself.

**Blog posts:**
- Why we migrated from mysql to mongodb
- Automating partitioning, sharding and failover with MongoDB
- Notes from a production MongoDB deployment
- Many more

**Presentations:**
- **MongoDB Monitoring and Queueing** - David Mytton's Presentation at MongoSF (May 2011)
- **Monitoring MongoDB** - David Mytton's Presentation at MongoSV (December 2010)
- **Humongous Data at Server Density** - MongoUK Presentation (June 2010)
- **MongoDB in Production at Boxed Ice** - Webinar (May 2010)

**Zuberance** started using MongoDB as a reporting engine in their enterprise product. In Q3 of 2011, Zuberance decided to build their self serve product to scale it to thousands of brand marketers to have them identify, energize their advocates while giving them real-time analytics.

**Blog posts:**
- Why We Like MongoDB at Zuberance - Zuberance Blog (January 2012)
<table>
<thead>
<tr>
<th>Company</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShareThis</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>-MongoDB is Powering ShareThis Count System - Lenin Gali’s Presentation at MongoSV (December 2010)</td>
</tr>
<tr>
<td>GitHub</td>
<td>GitHub, the social coding site, is using MongoDB for an internal reporting application.</td>
</tr>
<tr>
<td>Eventbrite</td>
<td>Eventbrite gives you all the online tools you need to bring people together for an event and sell tickets.</td>
</tr>
<tr>
<td></td>
<td>- Building a Social Graph with MongoDB at Eventbrite - Brian Zambrano’s presentation at MongoSV (December 2010)</td>
</tr>
<tr>
<td></td>
<td>- Tech Corner: Auto recovery with MongoDB replica sets - Eventbrite Blog (October 2010)</td>
</tr>
<tr>
<td></td>
<td>- Why you should track page views with MongoDB - Eventbrite Blog (June 2010)</td>
</tr>
<tr>
<td>BLiNQ Media</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>- MongoDB Delivers Results for Facebook Advertisers - Presentation at Mongo Atlanta (February 2011)</td>
</tr>
<tr>
<td>Yottaa</td>
<td>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 its scalable analytics engine.</td>
</tr>
<tr>
<td></td>
<td>- How Yottaa Uses MongoDB - Jared Rosoff’s presentation at MongoBoston (September 2010)</td>
</tr>
<tr>
<td></td>
<td>- Scalable Event Analytics with MongoDB and Ruby - Jared Rosoff’s presentation at RubyConfChina (June 2010)</td>
</tr>
<tr>
<td>BuzzFeed</td>
<td>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.</td>
</tr>
<tr>
<td>Company</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UberVu</td>
<td>UberVu is an intuitive social media management and analytics service used by companies of all sizes in 20+ countries. UberVu uses MongoDB as its primary data store because of its fast writes, its schemaless approach, and reliability.</td>
</tr>
<tr>
<td>CopperEgg</td>
<td>CopperEgg RevealCloud – Awesome Cloud Monitoring – CopperEgg is able to achieve its super real-time updates and alerts of the RevealCloud product with the help of MongoDB. Mongo was the only DB that fit the requirements of extremely rapid storage and retrieval of customer monitoring data to provide on-screen updates every few seconds (NOT minutes). CopperEgg monitors thousands of endpoints world-wide in real-time. RevealCloud is available to application development organizations to give the instant visibility of system performance in any stage of the development lifecycle, from initial design through production, helping ensure SLAs...and giving your customers the response times they expect – right now.</td>
</tr>
<tr>
<td>Loggly</td>
<td>Loggly uses MongoDB as a repository of statistics and summary information for time series log data. Loggly collects and stores metrics into MongoDB including the size and count of log events, and the results of saved searches run against our customized full-text search engine based on SOLR. These metrics are used to drive graphs, reports, dashboards, and exciting visualizations for users.</td>
</tr>
<tr>
<td>Pattern Builders</td>
<td>Pattern Builders built a .NET based streaming analytics engine using MongoDB. Relational databases and MSMQ were integrated with MongoDB for high performance and a great user experience.</td>
</tr>
<tr>
<td>GoGraph</td>
<td>GoGraph Stock Photography offers over 7 million stock images and graphics. We use MongoDB to track impression data on our search results pages. It helps us to track which results are served when a user does a search query. Using that data we are able to get a better idea of what our customers are looking for, and we are able to serve up better results which will translate into a better overall user experience.</td>
</tr>
<tr>
<td>VinTank</td>
<td>VinTank uses MongoDB to power the analytics and reporting for their social media listening and social CRM features. The company is the only digital think tank for the wine industry.</td>
</tr>
</tbody>
</table>
Trunk Club uses MongoDB for a variety of applications, including real-time analytics, business intelligence and machine learning. The company allows guys discover awesome clothes that are perfect for them without ever having to go shopping by combining top brands, expert service, and unparalleled convenience to deliver a highly personalized experience that helps guys look their best and saves them time.

MinFil.se uses MongoDB for distributed & redundant file storage capability. Using MongoDB, GridFS' service basically scales infinitely---they just add more servers as their storage requirement grows, Real-Time Analytics for all our users, all individual files, and for the service in general, tracking pageviews, downloads, unique downloads, referrers, etc.

Appboy uses MongoDB for distributed & redundant file storage capability. Appboy is a customer relationship management, marketing, and user analytics platform for mobile applications. They use MongoDB as their primary datastore and analytics backend.

Ukrainian Navigation Systems uses MongoDB for its main project database. The company manufactures advanced systems for GPS-monitoring.

ChannelMeter harvests hourly and daily analytics from YouTube for millions of videos and tens of thousands of channels and stores them using MongoDB. This information powers ChannelMeter's website, which provides reports, analytics, and recommendations for individual vloggers and corporate giants alike.

Scientific

- CERN uses MongoDB for Large Hadron Collider data.
  - MongoDB at the Energy Frontier - MongoNYC presentation (May 2012)
  - Holy Large Hadron Collider, Batman! - MongoDB Blog (June 2010)
- The Mount Sinai Institute for Genomics and Multiscale Biology uses MongoDB to help with various computational genomics tasks
- 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.

Social Networks
### Foursquare

Foursquare is a location-based social network that incorporates gaming elements. Foursquare uses MongoDB to store venues and user "check-ins" into venues, sharding the data over more than 25 machines on Amazon EC2.

- **Experiences Deploying MongoDB on AWS** - Cooper Bethea's Presentation at MongoSV (December 2011)
- **Practical Data Storage: MongoDB at foursquare** - MongoNYC (June 2011)
- **MongoDB at foursquare** - MongoSF (May 2011)
- **Scaling foursquare with MongoDB** - New York MongoDB User Group (December 2010)
- **MongoDB Q&A** - New York MongoDB User Group (December 2010)
- **MongoDB presentation** - Slides and Video (May 2010)

### Musweet

Musweet keeps track of what artists and bands publish on the social web.

- **Handling Humongous Data Sets from Social Net** - Nader Cseryn and Grischa Andreev's Presentation at MongoBerlin (October 2010)

### Behance

Behance uses MongoDB to power its Behance's Activity Feed, which displays what's new and what's buzzing in the Behance community from the people you follow. The Activity Feed provides a way of exploring the creative world through everyone you know as a curator in real time.

### Guildwork

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.
<table>
<thead>
<tr>
<th><strong>Silentale</strong> 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 <a href="#">TechCrunch article</a>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>• One Year with MongoDB presentation from MongoUK (June 2010): <a href="#">Slides</a> and <a href="#">Video</a></td>
</tr>
<tr>
<td><strong>Squarespace</strong> 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 MongoDB 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.</td>
</tr>
<tr>
<td><strong>Shelby.tv</strong> Shelby uses a few physically distinct MongoDB replica sets as the primary data stores in development and production. That data is accessed by web and iOS through our internal Ruby API.</td>
</tr>
<tr>
<td><strong>Jorjun Technical Services</strong> uses MongoDB for caching JSON-based REST API calls, and increasingly for production data. The company offers solid, reliable software applications by offering focus, experience, lateral thinking and experienced evaluation of technical options.</td>
</tr>
<tr>
<td><strong>Tripl</strong> stores all checkin data, trips and notifications as well as queues in MongoDB. It is built from social geo-data on Facebook, Foursquare and Instagram to highlight trips and present travel stories from your friends.</td>
</tr>
</tbody>
</table>
Dev, Sandbox, and myList's Production all use MongoDB for their backend. myList is a Facebook application that allows people to discover and organize the things in their lives. All back end data is in MongoDB with Solr for indexing and Hadoop for Map/Reduce jobs against logs.

Bawaal Labs uses MongoDB for user management and storing all data related to user posts. Bawaal contains views on technology, social networking and current happenings.

eZuce openUC features an advanced SIP Services Oriented Architecture (SSOA) which uses MongoDB to persistently store all user profile and transaction data in a distributed way. While MongoDB's stellar performance is critical to SSOA, its main benefit for SSOA is the horizontal sharding that’s part of its architecture, which allows horizontal scaling across several nodes. MongoDB also holds all client registration data and makes it available to all session manager nodes. The effect to the user is completely seamless failover at a regional or global level. Many session manager nodes can participate in such a globally redundant cluster, which allows enterprises to easily build a robust communications backbone that spans the entire enterprise.

BranchOut uses MongoDB to store profile statistics, back their feed data, endorsements and other key features of the application. BranchOut is a Facebook application designed for finding jobs, networking professionally, and recruiting employees.

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>myList</td>
<td>Dev, Sandbox, and Production all use MongoDB for their backend. myList is a Facebook application that allows people to discover and organize the things in their lives. All back end data is in MongoDB with Solr for indexing and Hadoop for Map/Reduce jobs against logs.</td>
</tr>
<tr>
<td>Bawaal Labs</td>
<td>uses MongoDB for user management and storing all data related to user posts. Bawaal contains views on technology, social networking and current happenings.</td>
</tr>
<tr>
<td>eZuce</td>
<td>openUC features an advanced SIP Services Oriented Architecture (SSOA) which uses MongoDB to persistently store all user profile and transaction data in a distributed way. While MongoDB's stellar performance is critical to SSOA, its main benefit for SSOA is the horizontal sharding that’s part of its architecture, which allows horizontal scaling across several nodes. MongoDB also holds all client registration data and makes it available to all session manager nodes. The effect to the user is completely seamless failover at a regional or global level. Many session manager nodes can participate in such a globally redundant cluster, which allows enterprises to easily build a robust communications backbone that spans the entire enterprise.</td>
</tr>
<tr>
<td>BranchOut</td>
<td>uses MongoDB to store profile statistics, back their feed data, endorsements and other key features of the application. BranchOut is a Facebook application designed for finding jobs, networking professionally, and recruiting employees.</td>
</tr>
</tbody>
</table>
Viber Media is using MongoDB as the core cloud infrastructure for its popular iPhone and Android applications that offer free phone calls and text messaging between users over 3G and wireless networks. Viber selected MongoDB for its ability to scale as more users join the Viber community and to manage peak and unpredictable data loads from its 30 million plus registered mobile users.

- MongoDB at Viber Media: The Platform Enabling Free Phone Calls and Text Messaging for Over 18 Million Active Users - MyNoSQL (Jan 2012)
- Viber Media Selects MongoDB as Database in the Cloud for Popular Mobile Apps - 10gen blog (Nov 2011)
- Where NoSQL, cloud computing and free texting converge - GigaOm (Nov 2011)

O2 uses MongoDB for several applications, including their Priority Moments location based offer service.

- Ericsson
- Telefonica

More MongoDB Users

<table>
<thead>
<tr>
<th>Company</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sugar CRM</td>
<td>Sugar CRM uses MongoDB to store comments and whether a user likes or dislikes portions of the application.</td>
</tr>
<tr>
<td>WuuhuuOnline.com</td>
<td>WuuhuuOnline.com uses MongoDB to store partial basic data and selected data. Because of MongoDB's high performance, it uses MongoDB as a cache layer, and it works very well.</td>
</tr>
<tr>
<td>catch.com</td>
<td>catch.com is the easiest way to capture and find your notable thoughts, ideas, images, places, lists &amp; anything else you'd rather not forget.</td>
</tr>
<tr>
<td>CollegeHumor</td>
<td>CollegeHumor uses MongoDB to accelerate its user-specific search, content browsing, and geolocated search. Currently, it's employed mostly as a cache and not as the primary store with the notable exception of click-track data. It's also moving to push its log info into MongoDB to give it better analytics as part of its continuous integration initiative.</td>
</tr>
<tr>
<td>Thrillist</td>
<td>Thrillist is using MongoDB to support our click-track data. Currently, it's employed mostly as a cache and not as the primary store with the notable exception of our click-track data. It's also moving to push its log info into MongoDB to give it better analytics as part of our continuous integration initiative.</td>
</tr>
<tr>
<td>Company</td>
<td>Uses MongoDB for analytics and quick reporting.</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>Evite</td>
<td></td>
</tr>
<tr>
<td>Justin.tv</td>
<td>is the easy, analytics tools for vir provide.</td>
</tr>
<tr>
<td>WHERE®</td>
<td>is a local : mobile coupons in th restaurant reviews, t from local merchants Where, Inc. uses Mc Ads™ - a hyper-loca</td>
</tr>
<tr>
<td>PhoneTag</td>
<td>is a service and SMS. PhoneTag</td>
</tr>
<tr>
<td>PiCloud</td>
<td>enables scientific power in public and computing applications sets in a highly distri</td>
</tr>
<tr>
<td>Hashrocket</td>
<td>is an expert Medication Manager resolving drug-relate</td>
</tr>
<tr>
<td>PharmMD</td>
<td>Medication Management application in Ruby on Rails. The system contains functionality for identifying and resolving drug-related problems for millions of patients.</td>
</tr>
<tr>
<td>The Mozilla open-source project uses MongoDB for back-end storage. Source code is available on bitbucket.</td>
<td></td>
</tr>
<tr>
<td>Sedue</td>
<td>is the enterprise search/recommendation engine, provided by Preferred Infrastructure, Inc. Currently uses MongoDB to store the document in a flexible and highly-available way.</td>
</tr>
<tr>
<td>Codaset</td>
<td>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)</td>
</tr>
<tr>
<td>Handling Dates in MongoDB</td>
<td>- Codaset Blog (July 2010)</td>
</tr>
<tr>
<td>PiCloud</td>
<td>enables scientific power in public and computing applications sets in a highly distri</td>
</tr>
<tr>
<td>Hashrocket</td>
<td>is an expert Medication Manager resolving drug-relate</td>
</tr>
<tr>
<td>PharmMD</td>
<td>Medication Management application in Ruby on Rails. The system contains functionality for identifying and resolving drug-related problems for millions of patients.</td>
</tr>
<tr>
<td>The Mozilla open-source project uses MongoDB for back-end storage. Source code is available on bitbucket.</td>
<td></td>
</tr>
<tr>
<td>Sedue</td>
<td>is the enterprise search/recommendation engine, provided by Preferred Infrastructure, Inc. Currently uses MongoDB to store the document in a flexible and highly-available way.</td>
</tr>
<tr>
<td>Codaset</td>
<td>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)</td>
</tr>
<tr>
<td>Handling Dates in MongoDB</td>
<td>- Codaset Blog (July 2010)</td>
</tr>
<tr>
<td>Company</td>
<td>Use Case</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Punchbowl.com</td>
<td>Immortalis uses MongoDB for tracking user behavior and datamining.</td>
</tr>
<tr>
<td>photostre.am</td>
<td>photostre.am streams image data from flickr and uses MongoDB as its only data storage.</td>
</tr>
<tr>
<td>fotopedia.com</td>
<td>Fotopedia uses MongoDB for storing timelines, a feature that stores tiny HTML fragments in its Varnish cache.</td>
</tr>
<tr>
<td>Grooveshark</td>
<td>Grooveshark is currently using MongoDB.</td>
</tr>
<tr>
<td>Struq</td>
<td>Struq develops technology that personalizes online display advertising in real time.</td>
</tr>
<tr>
<td>Pitchfork</td>
<td>Pitchfork is using MongoDB for its year-end readers survey and internal analytics.</td>
</tr>
<tr>
<td>Floxee</td>
<td>Floxee, a web toolkit, is award-winning.</td>
</tr>
<tr>
<td>TeachStreet</td>
<td>TeachStreet helps power their teaching businesses with MongoDB.</td>
</tr>
<tr>
<td>Visibiz</td>
<td>Visibiz is a socially and email-infused relationship management solution designed to improve the productivity of business professionals.</td>
</tr>
<tr>
<td>Defensio</td>
<td>Defensio is a comment-spam blocker that uses MongoDB for back-end storage.</td>
</tr>
<tr>
<td>TweetSaver</td>
<td>TweetSaver is a web service for backing up, searching, and tagging tweets.</td>
</tr>
<tr>
<td>Bloom</td>
<td>Bloom Digital's AdGear platform is a next-generation ad platform. MongoDB is used for back-end reporting and storage for AdGear.</td>
</tr>
<tr>
<td>KLATU Networks</td>
<td>KLATU Networks designs, develops and markets asset monitoring solutions which help 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.</td>
</tr>
<tr>
<td>Application</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>This or That!</strong></td>
<td>A user-driven comparison site that encourages people to create, share, vote</td>
</tr>
<tr>
<td>and comment.</td>
<td></td>
</tr>
<tr>
<td><strong>songkick</strong></td>
<td>Lets you track your favorite artists so you never miss a gig again.</td>
</tr>
<tr>
<td><strong>CrowdTap</strong></td>
<td>Uses MongoDB to drive its leaderboard.</td>
</tr>
<tr>
<td><strong>Detexify</strong></td>
<td>A cool app to find LaTeX symbols easily.</td>
</tr>
<tr>
<td><strong>Sluggy Freelancer</strong></td>
<td>Uses MongoDB for back-end storage.</td>
</tr>
<tr>
<td><strong>EDITD</strong></td>
<td>Using MongoDB to store opinions from social media, blogs, forums and other</td>
</tr>
<tr>
<td></td>
<td>sources to use in its sentiment analysis system, Zeitgeist.</td>
</tr>
<tr>
<td><strong>@trackmeet</strong></td>
<td>Helps you take notes with Twitter and is built on MongoDB.</td>
</tr>
<tr>
<td><strong>eFlyover</strong></td>
<td>Leverages the Google Earth Browser Plugin and MongoDB to provide interactive</td>
</tr>
<tr>
<td></td>
<td>flyover tours of over two thousand golf courses worldwide.</td>
</tr>
<tr>
<td><strong>Shapado</strong></td>
<td>A multi-topic question and answer site in the style of Stack Overflow.</td>
</tr>
<tr>
<td><strong>Sifino</strong></td>
<td>Enables students to help each other with their studies.</td>
</tr>
<tr>
<td></td>
<td>Students can share notes, course summaries, and old exams.</td>
</tr>
<tr>
<td><strong>GameChanger</strong></td>
<td>Provides mobile apps that replace pencil-and-paper scorekeeping and online</td>
</tr>
<tr>
<td></td>
<td>tools that distribute real-time game updates for amateur sports.</td>
</tr>
<tr>
<td><strong>soliMAP</strong></td>
<td>Uses MongoDB for storage.</td>
</tr>
<tr>
<td><strong>MyBankTracker iPhone App</strong></td>
<td>Uses MongoDB to store all user data, including large amounts of billing</td>
</tr>
<tr>
<td></td>
<td>information. This is used by the live site and also by BillMonitor's internal</td>
</tr>
<tr>
<td></td>
<td>data analysis tools.</td>
</tr>
<tr>
<td><strong>Tubricator</strong></td>
<td>Allows you to create easy to remember links to YouTube videos.</td>
</tr>
<tr>
<td></td>
<td>It's built on MongoDB and Django.</td>
</tr>
</tbody>
</table>
Mu.ly uses MongoDB 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.

Topsy is a search engine powered by Tweets that uses MongoDB for real-time log processing and analysis.

Cold Hard Code makes featureful, beautiful web applications, including Codepeek and Jarvis, which use MongoDB.

Similaria.pl is an online search engine for OpenCourseWare. It stores all the course materials in MongoDB and uses Sphinx to index these courses.

ToTuTam uses MongoDB to store information about users' preferences.

themoviedb.org is a free, user-driven movie database that uses MongoDB as its primary database.

One Year with MongoDB - Presentation from PyCon Ukraine (October 2010)

Mixero is the new generation Twitter client for people who value their time and are tired of information noise. Mixero uses MongoDB to store users' preferences and data.

Biggo is an advanced site constructor with e-commerce modules. Biggo 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 MongoDB backend in August 2009. See Bruno Morency's presentation for more information.

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.

Full Text Search with Sphinx - Presentation from MongoUK (June 2010)
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.

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.

Joomla Ads is a software development firm that has been using MongoDB for various projects since January 2010.

Eiwa System Management, Inc. 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.

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.

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 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).
<p>| <strong>Kehalim</strong> | 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. |
| <strong>Givemebeats.net</strong> | 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. |
| <strong>Cheméo</strong> | 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 chemo.com/doc/technology. |
| <strong>Planetaki</strong> | is a 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) |
| <strong>ChinaVisual.com</strong> | 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. |
| <strong>RowFeeder</strong> | 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) |
| <strong>Open Dining Network</strong> | 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. |
| <strong>URLi.st</strong> | 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. |
| <strong>Kidiso</strong> | 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). |
| <strong>Carbon Calculated</strong> | 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: <a href="http://vow.ch/2ij">http://vow.ch/2ij</a> |</p>
<table>
<thead>
<tr>
<th>Logo</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Ros Spending" /></td>
<td>Ros Spending 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.</td>
</tr>
<tr>
<td><img src="image" alt="BlueSpark" /></td>
<td>BlueSpark designs a development, we have a passion for creating great user experiences and products that feel simple to use.</td>
</tr>
<tr>
<td><img src="image" alt="Aghora" /></td>
<td>Aghora is a time all governmental requirement information.</td>
</tr>
<tr>
<td><img src="image" alt="Man of the House" /></td>
<td>Man of the House is and at home, as a father and as a husband. PeerIndex is an algo firehose of social me</td>
</tr>
<tr>
<td><img src="image" alt="PeerIndex" /></td>
<td>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.</td>
</tr>
<tr>
<td><img src="image" alt="sahibinden.com" /></td>
<td>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.</td>
</tr>
<tr>
<td><img src="image" alt="*ylastic" /></td>
<td>Ylastic is using MongoDB capability.</td>
</tr>
<tr>
<td><img src="image" alt="BRAINREPUBLIC" /></td>
<td>BRAINREPUBLIC is a tool that allows users to view all of their social networks on a single map.</td>
</tr>
<tr>
<td><img src="image" alt="Friendmaps" /></td>
<td>Friendmaps is a tool for more information.</td>
</tr>
<tr>
<td><img src="image" alt="jounce" /></td>
<td>jounce 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.</td>
</tr>
<tr>
<td><img src="image" alt="Jounce" /></td>
<td>Jounce 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.</td>
</tr>
<tr>
<td><img src="image" alt="Virb" /></td>
<td>Virb is using MongoDB extensively in production. For example, MongoDB powers Ylastic’s monitors capability.</td>
</tr>
<tr>
<td><img src="image" alt="dealmachine" /></td>
<td>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.</td>
</tr>
<tr>
<td><img src="image" alt="ArrivalGuides.com" /></td>
<td>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.</td>
</tr>
</tbody>
</table>
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!

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 where you can 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.jp and techcrunch.com. 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 millions 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.

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 data storage for all application data.


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.

vsChart.com

tyap.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 lets us represent rich conference schedule data in a way that's complementary to its 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. Workbreeze 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.
<table>
<thead>
<tr>
<th>Company</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Storm</td>
<td>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 an 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 store the ID of the property 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.</td>
</tr>
<tr>
<td>Mashape</td>
<td>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. How We Swapped Presentation</td>
</tr>
<tr>
<td>The UK Jobsite</td>
<td>is an easy to use job board allowing users to quickly search and apply for job vacancies. The UK Jobsite uses MongoDB for all aspects of the site—in its 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 why we chose MongoDB and continue to use it in a full production site—this can be found at <a href="http://www.theukjobsite.co.uk/tech.php">http://www.theukjobsite.co.uk/tech.php</a>.</td>
</tr>
<tr>
<td>Tastebuds</td>
<td>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 favorite 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.</td>
</tr>
<tr>
<td>Skimlinks</td>
<td>enables publishers to easily monetize 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 MongoDB to log and report on the almost 2 billion impressions we get every month.</td>
</tr>
<tr>
<td>Vanilla Desk</td>
<td>is an 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 Vanilla Desk to process records fast and keep them safe and available.</td>
</tr>
<tr>
<td>Summify</td>
<td>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.).</td>
</tr>
<tr>
<td>Dakwak</td>
<td>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.</td>
</tr>
<tr>
<td>Kapost</td>
<td>uses MongoDB and does various things. 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.</td>
</tr>
<tr>
<td>ContactMe</td>
<td>is a &quot;light&quot; 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 user's leads and contacts into one place, making for easy and efficient contact management.</td>
</tr>
<tr>
<td>Application</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Moontoast Impulse</td>
<td>A free Facebook application that lets fans play, share, and purchase music right from a Facebook fan page. Their tools power co-fan-building and revenue opportunities.</td>
</tr>
<tr>
<td>Moontoast</td>
<td>A social commerce company based in Boston and Nashville. Their tools power communities, stores, and private sales clubs. They leverage MongoDB, PHP 5.3, Zend Framework, Doctrine 2, jQuery, Gearman, Ruby, MySQL, and more.</td>
</tr>
<tr>
<td>Moontoast Impulse</td>
<td></td>
</tr>
<tr>
<td>Shopperhive</td>
<td>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.</td>
</tr>
<tr>
<td>Tracknose</td>
<td>Develops location-based applications for mobile phones that detect your GPS position and track you as 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.</td>
</tr>
<tr>
<td>Wusoup</td>
<td>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.</td>
</tr>
<tr>
<td>Fave</td>
<td>A local search engine designed to help people find and communicate with their favorite businesses, and to help businesses expand their presence on major search engines. Fave is 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.</td>
</tr>
<tr>
<td>Doodle</td>
<td>The world's leading consensus-scheduling service with millions of users worldwide. Doodle uses MongoDB as an addition to MySQL and plans to slowly but steadily persist more data via MongoDB and less data via MySQL.</td>
</tr>
<tr>
<td>FindTheBest</td>
<td>An objective, socially curated comparison engine that allows you to find a topic, compare your options and decide what's best for you. FindTheBest uses MongoDB for traffic monitoring and analysis as well as storage of user reviews.</td>
</tr>
<tr>
<td>Fuseware</td>
<td>Provides real-time 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 MongoDB clusters to provide scalable data storage for the myriad different web objects that we store. We have hundreds of gigabytes of stored data, and MongoDB allows us full horizontal scalability and data access.</td>
</tr>
<tr>
<td>Wherevent</td>
<td>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.</td>
</tr>
</tbody>
</table>
Skyline Innovations
We deploy custom MongoDB for data warehousing and fast, flexible multidimensional analyses of our real-world data.

- Time Series
  2011

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.

Fundastic.info 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.

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.
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.

A Beautiful Marriage: MongoDB and node.js - Chris Ricca's Presentation at MongoNYC (June 2011)

Dayload provides a hassle-free AWS Usage Monitoring

Avivo is creating mobile, web and businessThrough design and interaction. Ultimate that build brand loyalty is leading us to dive best for our clients. A primary database for

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.

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.
<table>
<thead>
<tr>
<th>Image</th>
<th>Website/Service Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Republika" /></td>
<td><strong>Republika Online</strong> is the biggest Islamic news portal in Indonesia. It's an online version of “Harian Republika Online”. We use sharded MongoDB as storage of all news and comments.</td>
</tr>
<tr>
<td><img src="image2.png" alt="Grepler" /></td>
<td><strong>Grepler.com</strong> is a distributed open system that merges several newsgroups, groups, forums and mailing lists to enable single user accounts will be able to join all discussions. Everything is now running from MongoDB except the user accounts which will be mapped in the near future.</td>
</tr>
<tr>
<td><img src="image3.png" alt="Thuisvergelijken" /></td>
<td><strong>Thuisvergelijken.nl</strong> is one of the biggest Dutch product price comparison and shopping guides. Offering a wide range of products we serve a broad audience of online shoppers. We use MongoDB for pretty much all our data storage: webshops, product data feeds, reviews and our custom written analytics system. MongoDB’s flexibility allows us to develop rapidly and still we have the benefits of great performance.</td>
</tr>
<tr>
<td><img src="image4.png" alt="salsadb" /></td>
<td><strong>salsadb.com</strong> uses MongoDB’s capabilities. They use MongoDB for importing and merging data sets from spreadsheets. MongoDB’s query capabilities are used extensively to allow users to filter and summarize their data.</td>
</tr>
<tr>
<td><img src="image5.png" alt="CHECK24" /></td>
<td><strong>CHECK24</strong> is one of Germany`s leading price comparison websites. On CHECK24.de customers can compare a wide range of products from the fields of insurance, money, energy, telecommunications, and travel and thus save hundreds of Euros per year. In contrast to most other comparison websites which link-out the customer when they want to buy a product, CHECK24 offers integrated ways of effecting a contract. The comparison services for prices and product features are free of charge for the customer. Furthermore, customers get free independent advice by phone from qualified product experts.</td>
</tr>
<tr>
<td><img src="image6.png" alt="Wheelhouse CMS" /></td>
<td><strong>Wheelhouse CMS</strong> is a Ruby on Rails Content Management System (CMS) designed to be incredibly flexible to develop sites with, and pain-free for you and your clients to use. MongoDB (and MongoModel) is used for all data storage within Wheelhouse CMS.</td>
</tr>
<tr>
<td><img src="image7.png" alt="Qwk.io" /></td>
<td><strong>Qwk.io</strong> is the quickest, easiest way to conduct surveys. It’s built on Ruby on Rails and MongoDB for the backend.</td>
</tr>
<tr>
<td><img src="image8.png" alt="CyberAgent" /></td>
<td><strong>CyberAgent, Inc.</strong> has strongly expanded its business into a wide range of internet services. Currently, CyberAgent’s major business categories are Internet Media, Internet Advertising Agency and the Investment Development. CyberAgent, Inc. is using MongoDB in “Ameba” services. We also have an American office: <strong>CyberAgent America</strong>. Social Game with node.js + MongoDB.</td>
</tr>
<tr>
<td><img src="image9.png" alt="HeiaHeia" /></td>
<td><strong>HeiaHeia.com</strong> provides web and mobile services that motivate people to move more, share activities with friends, and cheer others – bringing a fun and social dimension into physical activity and wellbeing. Users can choose from over 350 activities and the workouts can be shared also on Facebook and Twitter. HeiaHeia.com’s technology offers a cost-efficient, commercially attractive way for various communities to provide engaging, customized online services for wellbeing.</td>
</tr>
<tr>
<td><img src="image10.png" alt="NewsCurve" /></td>
<td><strong>NewsCurve</strong> is an analytics tool providing effectiveness metrics of online content in real-time. We use MongoDB as primary database storage to store analytics data.</td>
</tr>
<tr>
<td>Company</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>A-SaaS</td>
<td>Accounting SaaS Japan provides Accounting Services via SaaS targeted towards Japanese certified accountants and small and mid-size businesses. By using MongoDB in our tax return service, we are able to achieve the flexibility necessary to meet the rapidly changing demands and specifications vital for such a system—something that is difficult to accomplish with the relational or file-based approach.</td>
</tr>
<tr>
<td>Fabric Structures Co</td>
<td>It is a project to ensure the enlargement of the textile structures market share in the overall construction sector by providing more comprehensive and up-to-date information to potential customers about the textile structures range and opportunities, as well as to companies operating in this field. MongoDB is used for storing and indexing our member profiles and company directory.</td>
</tr>
<tr>
<td>Brenden Digital</td>
<td>Uses MongoDB for everything except transactional features, and even those are being migrated. We appreciate MDB for its smaller size, much faster performance and ease of implementation. It repeatedly exhibits an elegance, a sublime aura that is the mark of excellence and thorough comprehension.</td>
</tr>
<tr>
<td>Attachments.me</td>
<td>It is a search engine for your email attachments. MongoDB stores all the meta-information surrounding attachments, messages, and users.</td>
</tr>
<tr>
<td>Thumbtack</td>
<td>It is an online marketplace for local services. We use MongoDB as the datastore for all of our analytics, due to its excellent performance, flexible document format and powerful query language. More details can be found in our recent blog post: <a href="http://johnpwood.net/2011/05/31/fast-queries-on-large-datasets-using-mongodb-and-summary-documents/">Building our own tracking engine with MongoDB</a>.</td>
</tr>
<tr>
<td>ChirpAt.Me</td>
<td>It enables Twitter users to build their own real-time discussion board to share their passions and expertise with their friends and followers. ChirpAt.Me uses MongoDB for all of their data storage from user sessions to messages and preferences.</td>
</tr>
<tr>
<td>DianPing.com</td>
<td>It is a leading city life guide site and local businesses promotion platform in China, it provides objective and rich local information, in the areas of restaurants, entertainment, shopping, beauty, weddings and various other categories. MongoDB is used for: user-behavior analysis system, the counter on businesses, users, groups, simple message queue service.</td>
</tr>
<tr>
<td>Venmo</td>
<td>It is a free and easy to use mobile app friends can use to pay each other back for lunch, dinner, drinks, rent, groceries, tickets, and trips. Venmo uses MongoDB for internal reporting and for storing activities published by users for their friends to browse.</td>
</tr>
<tr>
<td>Intelie</td>
<td>It is a software that receives data from several sources and, using Complex Event Processing (CEP) and machine learning techniques, makes it possible to detect anomalies and known problems on data centers in real time. We use MongoDB to store all the data that Intelie analyzes over time, about 5000 documents per minute for one company. Storing these events allow us to 'replay' past scenarios to test new rules, in addition to providing dashboards and charts of historical data.</td>
</tr>
<tr>
<td>First, we are using</td>
<td>MongoDB’s geospatial queries to power location-based marketing campaigns. Our users can create a SMS marketing campaign to ask users for their zip code in reply to an SMS message, perform a lookup of known locations (stores) based on that zip code, and return the nearest location to the user. Second, we are using MongoDB to hold a set of summary documents to help us calculate a series of statistics for our Email and SMS subscription lists. Instead of gathering these statistics by running SQL queries in tables with excess of 10 million rows, we’re able to get the same numbers by looking at anywhere from 30 - 365 summary documents. As you can imagine, this dramatically speeds up these queries. This use case is documented at <a href="http://johnpwood.net/2011/05/31/fast-queries-on-large-datasets-using-mongodb-and-summary-documents/">http://johnpwood.net/2011/05/31/fast-queries-on-large-datasets-using-mongodb-and-summary-documents/</a>.</td>
</tr>
<tr>
<td>Directdialogs.com</td>
<td>It is a cloud-based direct marketing application with email/mobile campaign management capability and a flexible no-card loyalty program. MongoDB works alongside SQL Azure and serves as a powerful cache with analytical capabilities. Segmentation of customer/transaction data for marketing purposes, OLAP reporting/analytics based on customer purchase data and data mining models run on MongoDB.</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Company</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC Storm</td>
<td>Helps marketers increase sales, deliver higher ROI and target digital marketing spend.</td>
</tr>
<tr>
<td>Bouncely</td>
<td>Uses MongoDB to save details of every bounced message from Amazon SES.</td>
</tr>
<tr>
<td>PCClicks</td>
<td>Uses MongoDB for real-time analytics.</td>
</tr>
<tr>
<td>Magnetic</td>
<td>Uses MongoDB on servers processing billions of events a month.</td>
</tr>
<tr>
<td>Okezone.com</td>
<td>Uses MongoDB for inline content voting system and rating, polling, and commenting system.</td>
</tr>
<tr>
<td>OpenChime</td>
<td>Helps customers connect with local service providers.</td>
</tr>
<tr>
<td>Monoloop</td>
<td>A behavior platform that delivers real-time personalization to websites.</td>
</tr>
<tr>
<td>Yeay.me</td>
<td>Uses MongoDB for all its stored data.</td>
</tr>
<tr>
<td>Social City Deals</td>
<td>Uses MongoDB to store rapidly changing daily deals related data.</td>
</tr>
<tr>
<td>Activesphere</td>
<td>Provides services using Rails and MongoDB.</td>
</tr>
<tr>
<td>Thin PHP Framework</td>
<td>A lightweight, flexible open source PHP5 MVC framework.</td>
</tr>
<tr>
<td>Company</td>
<td>Uses MongoDB for</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>Newsman App</td>
<td>Email service provider focusing on speed and high deliverability. Uses MongoDB for now to build a powerful subscribers segmentation tool. Uses multiple variables for each subscriber, making MongoDB schema-free the perfect choice. Plans to switch to MongoDB for all other database operations in the future.</td>
</tr>
<tr>
<td>Sizzix</td>
<td>An e-commerce website selling thousands of products online. Uses MongoDB as the backend for its business-to-business side, storing tens of millions of raw products across a range of companies.</td>
</tr>
<tr>
<td>Gazaro</td>
<td>Uses MongoDB to store content, banners, taglines, and other information. Has also introduced OAuth 2.0 and implemented it using Python/Twisted/MongoDB tandem.</td>
</tr>
<tr>
<td>Dropzone.by</td>
<td>Uses MongoDB to queue incoming text messages before their application processes them and outgoing text messages after their application processes them for delivery.</td>
</tr>
<tr>
<td>Cairenhui</td>
<td>A Financial Community. The web application is coded in Java using Morphia ORM framework and uses MongoDB 1.8.1 (based on auto-sharding) to store all user-generated content.</td>
</tr>
<tr>
<td>Ez Texting</td>
<td>Uses MongoDB to queue incoming text messages before their application processes them and outgoing text messages after their application processes them for delivery.</td>
</tr>
<tr>
<td>iKeepm</td>
<td>A cloud-based home inventory application. Uses MongoDB to store photos and file uploads in GridFS.</td>
</tr>
<tr>
<td>Maansu.com</td>
<td>A retail book store. Uses MongoDB to store the details of the 16 million books they sell.</td>
</tr>
<tr>
<td>Motores24H</td>
<td>A cloud-based home inventory application. Uses MongoDB to store photos and file uploads in GridFS.</td>
</tr>
<tr>
<td>Techunits</td>
<td>A cloud-based home inventory application. Uses MongoDB to store photos and file uploads in GridFS.</td>
</tr>
<tr>
<td>Buzzers.co.za</td>
<td>A social tool for the Web of Things. Uses MongoDB for all persistent data storage requirements: performance, horizontal scalability, sharding, heterogeneous data, and NoSQL.</td>
</tr>
<tr>
<td>Paraimpu</td>
<td>A social tool for the Web of Things. Uses MongoDB for all persistent data storage requirements: performance, horizontal scalability, sharding, heterogeneous data, and NoSQL.</td>
</tr>
</tbody>
</table>
**Trivian** is a location-based social network that incorporates quiz gaming elements. Trivian uses MongoDB to store quiz questions and places.

**Business.com** is the premier online destination for businesses of all sizes to research, find, and compare the products and services they need to run their businesses. We use MongoDB to power our CMS and taxonomy systems. We also use Mongo's GridFS to store advertiser logos and other static assets.

**General Flows** is building an app where people can create databases of their stuff, then plug their stuff into some pre-baked 'flows' to let them do useful things with their stuff. So for example you could quickly build a booking system for presentation equipment by building a database of laptop and projectors and plugging them into the 'booking' flow. So we’re trying to build an app that deals with totally unknown data structures, because the data structure is created by the end user as they go. We messed around with code-generation techniques (define a conventional model layer from user input) and SQL techniques (EAV) but found the all frustrating to swapped to no-sql style DBs. We evaluated CouchDB, Mongo and Google App Engine and Mongo made it possible to do exactly what we needed. The database is now up and running; we’re just rolling out sets and soon we should have some flows up and running, too.

**Art.sy** is a new way to discover art you’ll love, featuring work from leading galleries, museums, and private collections around the world. Art.sy is powered by RoR with MongoDB.

**Spoondate** is a social community that connects people through shared “cravings” and unique dining experiences. The site enables members to post what they’re “craving” and other Spoondates can engage in a conversation, or suggest meeting for a meal in real time. MongoDB powers our geolocation services, data and member search, analytics, and logging. Thanks to MongoDB, we have been able to add features rapidly and consistently improve our products over time.

**uQuery Inc.** uses a search engine to process about 30GB of data per day that we receive from Apple with all the AppStore content, apps, prices, etc. We have migrated our entire software stack from MySQL to MongoDB.

**46elks** uses MongoDB for the backend of its cloud communication platform. By using MongoDB we can make sure the data of our SMS and phone calling customers is kept safe and always accessible.

**Chalkboard** is a daily marketing solution for small businesses to communicate instantly, locally and socially over mobile and web. Chalkboard works with thousands of local businesses and reaches millions of consumers across third party mobile applications, websites and navigation systems. We utilize MongoDB to archive the 100 millions of analytics data records where we can easily analyze the data internally. With the growing usage of Chalkboard, MongoDB allows to scale our infrastructure in the near future.

**tisProperty** is a Realestate Vertical Search Engine that allows users to search for ALL for sale and for rent realestates listed on the Internet. We crawl the Internet daily and MongoDB is being used as a huge cache to store our crawling and indexing results.

**Digital Media Exchange** uses MongoDB for several things actually, and we’ve been using it more and more instead of our main postgres db. The following are some of our use cases: 1. Search engine index storage 2. We run a lot of promo’s for our products so we store promo data in mongo. Love the schema-less design. 3. We use it as replacement for our table views to store denormalized data These are just some of the use cases, and I’m very confident that we will find more uses for MongoDB.

**Daily Gourmet** provides regular offers on artisanal and gourmet food products. “We’re using MongoDB for our extensive logging/analytics. MongoDB’s performance has enabled us to capture information that just wasn’t possible with MySQL.”
<table>
<thead>
<tr>
<th><strong>THE LOCKER PROJECT</strong></th>
<th>Sponsored by <em>Singly</em> over their personal data as a personal data s</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Loc-cit</strong></th>
<th><strong>Loc. cit.</strong> is a small ne can be explosive. Tr. database for everyth</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>NAVER</strong></th>
<th><strong>Naver Japan</strong> utilizes recently released ph <a href="http://itunes.apple.com/jp/app/id449576650">http://itunes.apple.com/jp/app/id449576650</a> with MongoDB (Jap: <a href="http://www.slideshare.net/naverjapan/mongodb-9422893">http://www.slideshare.net/naverjapan/mongodb-9422893</a></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>My eStore App</strong></th>
<th><strong>My eStore App</strong> uses content via Mongo.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>FOTOSEARCH® photos and images</strong></th>
<th><strong>Fotosearch Stock Pr images are served w customer is looking f results.</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>cloudamp</strong></th>
<th><strong>CloudAmp</strong> is a data many data sources across different priva</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Piyavate International</strong></th>
<th><strong>Piyavate International</strong> new website need search with Solr. Mu</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Mindvalley</strong></th>
<th><strong>Mindvalley</strong> is using Mongoid. The ability speed, while its docu</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Cookfollower</strong></th>
<th><strong>Cookfollower</strong> is a Tw meet new people. M</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Weconext</strong></th>
<th><strong>Weconext</strong> develops secured web platform allows simple collab</th>
</tr>
</thead>
<tbody>
<tr>
<td>Company</td>
<td>Uses MongoDB for:</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>Visual.ly</td>
<td>Building Data Visualization platform for Big Data.</td>
</tr>
<tr>
<td>Trendrr</td>
<td>Provides the key to managing, understanding, engaging, and activating real-time user experiences.</td>
</tr>
<tr>
<td>Wireclub</td>
<td>Storage of terabytes of statistical data and content used in curation, charting, and analytics products.</td>
</tr>
<tr>
<td>KANO/APPS</td>
<td>Create addictive social games.</td>
</tr>
<tr>
<td>OnePageCRM</td>
<td>Used MySQL some time ago, now fully migrated to MongoDB.</td>
</tr>
<tr>
<td>Blabbermouth Social</td>
<td>Collect and store large amounts of data for social media campaigns.</td>
</tr>
<tr>
<td>Persik.me</td>
<td>Power CMS. Please note that this website is NSFW.</td>
</tr>
<tr>
<td>Kungfuters LLC</td>
<td>Uses MongoDB as a backend for their socially driven contest site, awkwardturtle.com. Users must sign in using an existing social network, such as Google+, Facebook, or Twitter and users can submit stories of awkward moments in their lives. Visitors can read the stories and vote on their favorites. Each week, the story with the most votes wins a sponsored prize. MongoDB drives the entire backend. This site was formerly connecting to a MySQL database via JPA, but it was too slow. All Contests, Stories, Users, Votes, etc. are stored in Mongo. The end result is a site reborn and blazing fast.</td>
</tr>
<tr>
<td>Company</td>
<td>Use of MongoDB</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>PunchTab</td>
<td>Store all loyalty activities</td>
</tr>
<tr>
<td>Nexon Corp.</td>
<td>Uses MongoDB to archive game records.</td>
</tr>
<tr>
<td>Heyzap</td>
<td>Uses MongoDB to power real-time activity streams for millions of users on Android and iOS gaming apps.</td>
</tr>
<tr>
<td>Fishidy.com</td>
<td>Uses MongoDB for storing social data entered by end users.</td>
</tr>
<tr>
<td>Wokeey.com</td>
<td>Uses MongoDB for storing data with big records, approximately in one day can store hundreds of records.</td>
</tr>
<tr>
<td>ThinktankSocial</td>
<td>Uses MongoDB as one of the core technologies behind their platform to enable mass storage with a smaller overhead than MySQL or other SQL-based databases.</td>
</tr>
<tr>
<td>Data Publica</td>
<td>Uses MongoDB as the primary data store for users and all user-created content, as well as logging and analytics.</td>
</tr>
<tr>
<td>Noktacom Medya</td>
<td>Uses MongoDB in <a href="http://www.izlesene.com">www.izlesene.com</a> (Turkey's #1 video site) to log analytics. Tracks everything from video impressions to page clicks, videos watched, etc. Also built a custom video scoring system using MongoDB depending on video popularity, which is used on video suggestion system. Uses 3 machines running MongoDB and has around 20 to 50 insert calls per second. Will expand using MongoDB and switch from MySQL to MongoDB fully because of no usage of caching systems required with MongoDB.</td>
</tr>
<tr>
<td>Searce</td>
<td>Uses MongoDB for a very large set of data. Almost 5-10 collections having millions of documents.</td>
</tr>
<tr>
<td>Deegr</td>
<td>Is using MongoDB to store millions of ratings and provide an ultra-quick and innovative recommendation engine. Aims to build a cloud-based architecture and shard MongoDB in the cloud in the near future.</td>
</tr>
</tbody>
</table>
AppHarbor is a .NET platform as a service. It uses MongoDB to log, track and retain error messages generated by thousands of platform applications. Applications hosted on AppHarbor can provision MongoDB databases as on-demand services.

Pictomist is a simple and attractive way to browse & share photos. It was written in two basic parts: the first being a service layer that parses Twitter pushes into a queue-style mongo collection (not sharded yet, but will be as we grow it) where a separate thread will then pop off the top elements and parse these results/resize images and push them into our media collection. The second part is the web layer which grabs media from a Mongo collection indexed by hashtag, re-aggregates them in memory, and sends them to the view. The speed and flexibility of MongoDB was great in helping us spin out the bulk of this idea over a single weekend coding binge while still allowing us to grow our data layer without mucking around tedious database code and/or bulky ORMs.

Xperantum Xemantics uses MongoDB to enable users and organizations to define their own data structures, capture the right content from files and analyze that seamlessly.

4Apps uses MongoDB as the main database to save lists and also uses it as a user session store after memcached.

GLO AB uses MongoDB in their 'Home-grown' inventory control and manufacturing execution system (MES) software product for Nanowire LED production as the main database for storing unit/wafer/chip data gathered during the manufacturing process. They are generating an average of 10,000 documents per day, but expecting to grow to several millions per day. Currently they have a cluster consisting of 3 servers running within company network.

CMI Soft uses MongoDB to run the backend of all their custom (Ruby on Rails based) CMS driven websites & e-commerce stores including their company website www.cmisoft.in. They have also developed intranets, social networking apps and have 2 community-oriented e-commerce concepts under development which use Rails with MongoDB.

Fyndlr implements MongoDB from the backend service that powers fyndlr. All data about lost & found posts and users is organized in MongoDB collections. They rely heavily on MongoDB for their geolocation functionality. Since their service was written in Python, they implemented PyMongo to interface with their MongoDB instance.
Exceptiontail uses MongoDB in order to store user and exception information. Exceptiontail is a .Net exception management service that allows developers to know when their applications crashed and lets them see the entire system/process/exception information for that crash.

AdHui.com uses MongoDB for real-time cookie analytics. Leading provider for DSP and SEM optimization.

Travelmap uses MongoDB to store hotels, tickets and holiday-packages aggregated live from a bunch of Travelmap sources. Travelmap is one of the biggest travel websites in the Netherlands. MongoDB proved to be up to 5 times faster and much more scalable then our previous MySQL installation, on the same hardware! On average our database contains ~20 million objects.

Global Advertisers uses MongoDB to help with blogging. Global Advertisers is India's most progressive outdoor advertising agency. Armed with Mumbai's largest hoarding @ Bandra to widest hoarding @ Airport. Global is one stop for anything and everything under OOH.

Melt DSP is using MongoDB as its backend database for customer info.

Startup Threads Monthly uses MongoDB to hold all the data that drives the web site. Songza is a Music Concierge - a free streaming service that finds you the right playlist at the right time. Users, playlists, interaction history, etc. The schema-free design is a big win for us. When we used MySQL, schema migrations were a major pain, and occasionally a cause of extended downtime.

Ciris is a new application where you can find contact information about the people in Costa Rica. We use MongoDB to store the documents for every person and registered business in the country.

Socialbakers Inc. uses MongoDB to store profiles, and associate creative skills with our users.

Neon Grid develops second screen applications for large TV shows around the world. This typically involves tens or hundreds of thousands of people interacting with our clusters at exactly the same time and as a result we needed a storage solution that offers very good performance, is easy to scale up fast and remains cost-efficient regardless of scale. MongoDB offers all of the above along with a helpful, active community. We never looked back.

Ex Machina develops social networks and as a result we needed a storage solution that offers very good performance, is easy to scale up fast and remains cost-efficient regardless of scale. MongoDB offers all of the above along with a helpful, active community. We never looked back.
<table>
<thead>
<tr>
<th>Company</th>
<th>Uses MongoDB for...</th>
</tr>
</thead>
<tbody>
<tr>
<td>BellyBallot</td>
<td>a new site for expecting parents that mixes social media with baby names. Uses MongoDB for its primary content serving which includes the thousands of names around the world, and their associated meanings, pronunciations, etc. BellyBallot's custom content management system (built on Rails) is also on MongoDB.</td>
</tr>
<tr>
<td>AdMaster</td>
<td>social networking analytics product, WeiboMaster, to store the information of over 40,000,000 followers and 700,000 tweets with 35,000,000 relevant comments and retweets.</td>
</tr>
<tr>
<td>GetVega</td>
<td>is a second memory relevant data from the list on GetVega where GetVega.com uses it. MySQL. MongoDB allows versatile data storage in a blazing fast scalable way!</td>
</tr>
<tr>
<td>GetVega.com</td>
<td>uses MongoDB for the whole back-end (except search with Solr), after a first prototype with MySQL. MongoDB allows us to store versatile data in a blazing fast scalable way!</td>
</tr>
<tr>
<td>5Searches</td>
<td>uses MongoDB to manage all of my social media users and profiles!</td>
</tr>
<tr>
<td>barbull.co.uk</td>
<td>uses MongoDB to store a list of searchable attributes for personal trainers, which is fast and reliable and the ability to add custom functions and map reduce means that our complex distance calculations are a breeze.</td>
</tr>
<tr>
<td>Power-lan</td>
<td>uses MongoDB to store real-time data from IP sensors connected to the internet (over 3G). Main applications are: - Non-destructive testing (Highway, Aeolian, ...) - Automation and smart grid applications.</td>
</tr>
<tr>
<td>Sidebuy</td>
<td>is a product/deal aggregator that crawls the web and uses an intelligent system to automatically find products and deals. The challenge with traditional DBMS was to store entities with different properties. With MongoDB, entities are stored as documents. Also, with MongoDB expansion problems across our servers have been solved. Sidebuy crawls thousands of webpages per day and stores everything in BSON documents now. Geospatial querying is also a big plus for Sidebuy.com.</td>
</tr>
<tr>
<td>OviPets</td>
<td>uses MongoDB for everything in our new Facebook game OviPets (<a href="http://ovipets.com">http://ovipets.com</a>) and are very happy with the performance and simplicity of MongoDB. We're growing rapidly and MongoDB is scaling beautifully with us. Apart from various game-related collections, we also use MongoDB for session storage and image caching.</td>
</tr>
</tbody>
</table>
Tip or Skip uses MongoDB for all of our database needs; user and product storage, interaction, analytics, and job queues.

Nodegrid provides a hosting platform for MongoDB and also uses MongoDB as the back-end for its service.

Courseoff uses MongoDB to store student created schedules as well as the schedule listings themselves.

Courseoff uses MongoDB to store student created schedules as well as the schedule listings themselves.

Courseoff uses MongoDB to store student created schedules as well as the schedule listings themselves.

ClaimAble operates a web app for insurance claims management. MongoDB is our primary datastore. We use it exclusively to power our platform and API.

Ponder Design uses MongoDB to power all custom webapps we write for our customers. From a ticket portal for a large student event to CMS systems to custom surveys. Mongo lets us be very flexible in implementing new features or adjusting existing ones as we discover our customers' real requirements and wishes.
Stamped is a recommendation-based social network application that allows users to advocate for their favorite places and things. MongoDB is used to store and receive user data and are responsible for coordinating the creation and enrichment of data about stampable entities.

Hopper uses MongoDB for storage of all of our user data and content. Hopper uses MongoDB for 5 cluster servers with 3 shards.

Recommendly INC is a shopping and price comparison engine. We use MongoDB for many things, including inventory management and transaction logging.

ITERNOVA SL proudly uses MongoDB in our Intelligent Transport Systems (ITS) management system, to store real-time data obtained from devices installed in roads (weather stations, mesh and sensorial networks...), and vehicles.

Ob1b.com uses MongoDB to store storage meta-data of programme in mytv with different data-type. Since MongoDB is schema free it fits our needs. We also use MongoDB in real-time analytics with map-reduce processing. We create statistics for site traffic and a real-time programme chart.

Ob1b.com is one of China's price compare websites of 3C procuts. On ob1b.com customers can compare price and find products information from E-Business websites in China, and customers can find the right products which they want to buy. We use Mongdb as Data store engine. We have chosen MongoDB most because of its simplicity, easy to study and faster than traditional database.

Talis uses MongoDB to store and serve the graph-based data model in its Community Edition product (http://community.talisaspire.com). By using MongoDB as a replacement for the existing database, we achieved an order of magnitude better performance. At the same time converting to MongoDB allowed us to simplify our stack by removing complex caching infrastructure external to our old database.

Digital Flow uses MongoDB as the database of choice for our clients' deployments - the sheer scalability, ease of use and deployment are a no brainer for us.

Transmachin uses MongoDB to store and serve the graph-based data model in its Community Edition product (http://community.talisaspire.com). By using MongoDB as a replacement for the existing database, we achieved an order of magnitude better performance. At the same time converting to MongoDB allowed us to simplify our stack by removing complex caching infrastructure external to our old database.
About.me is currently processing millions of messages per day. After outgrowing an existing MQ solution, About.me turned to MongoDB. An old MQ cluster was replaced with a sharded Mongo deployment. In addition to the sharding ability, Mongo provided options that allows About.me to tailor durability to the level required.

Dealer.com deployed a site configuration management system based on MongoDB. The current system supports configuration files for over 15,000 customer sites with thousands of revisions per site and an on-disk size of almost 200GB.

Famigo stores all valuable data in MongoDB and serves all requests via Amazon EC2 instances.

By scaling materials computations over supercomputing clusters, The Materials Project has computed the properties of over 80,000 materials and screened 25,000 of these for Li-ion batteries. The Materials Project is making these materials and their properties available to scientists around the world through a sophisticated web interface. MongoDB is at the core of the Materials Project architecture. It is used to schedule and track quantum mechanical calculations of materials properties on supercomputers, to store and search the results of these computations, and to perform advanced analytics on the computed materials properties.
The MapMyFitness user base more than doubled in 2011, beginning an era of rapid data growth, and the traditional MySQL solution for the MapMyFitness web applications hit its ceiling. MongoDB was chosen as the candidate for exploration into NoSQL implementations, and now serves as their data store for rapid application deployment. MongoDB has assisted in serving 2TB+ of geolocation data, to time-series data for live tracking, to user sessions, app logging, and more.

SimpleReach powers the Slide, a recommendation powered content discovery technology for websites. SimpleReach builds schemas in MongoDB and Node.js for powerful, real-time data delivery.

MongoDB is used for Stripe’s in-house fraud detection, or a cron job that aggregates totals, fees, and more.

Tuttur.com is a social betting platform built and maintained by Tart New Media. Users are encouraged to share their betting experience and win together. MongoDB holds all the social experience data over 6 Linux machines.

WireLoad uses MongoDB in-house for event logging. Whenever a user logs in, a payment is received, or a cron job runs, an event is logged into our MongoDB event store. These events update aggregate totals, feed into fraud algorithms, and can be analyzed as a change log of the system.

Illicotravel is a social travel platform that compares hundreds of travel sites. We use MongoDB to store tickets, holiday-packages, hotels and more. We love MongoDB.

MKN’s (MKN Web Solutions) Facebook game “Drawabble” utilizes MongoDB 100% for all database operations. The database stores user interaction, games, game drawings, and everything else.

Spideo is using MongoDB to store customer information, in order to recommend movies related to their own tastes.

radi8.co.uk is using MongoDB within Ruby on Rails Development Projects.

PRIJSZOEKEN.nl is a Dutch price and product comparison website. We are offering more than 1.5 million products from 500 connected shops. We use MongoDB to handle pretty much everything: all our data-feeds, our custom made analytics system and product specifications. MongoDB is perfect for us, it allows us to develop rapidly and the performance is great.
<table>
<thead>
<tr>
<th>Company</th>
<th>Uses MongoDB for their primary database or data store</th>
</tr>
</thead>
<tbody>
<tr>
<td>JustReco</td>
<td>uses MongoDB as its primary database for their social networking site.</td>
</tr>
<tr>
<td>Gimme Bar</td>
<td>uses MongoDB for the entire backend datastore (everything except media assets, which are stored directly in S3). We currently have a multi-node replica set. Not (yet) sharded. Current VSIZE is around 70g.</td>
</tr>
<tr>
<td>Nodex</td>
<td>use MongoDB as our primary data store for our CMS which is soon to be open sourced. We have many websites running on MongoDB and are expanding at a rate of around 1 per week (avg). Using MongoDB enables rapid development of fast and secure websites. If you would like some more information I am happy to write a blog post more outlining how we use it.</td>
</tr>
<tr>
<td>FramtidsPost AB</td>
<td>uses MongoDB for all our storage needs in the HeyFuture, FuturePing and BackFuture services. The FuturePing messages are very small and count to millions. The HeyFuture and BackFuture messages are big and can carry significant payload. For very different storage characteristics we have been able to use MongoDB in a multi-region setup with replication and sharding.</td>
</tr>
<tr>
<td>Beanstalk Data</td>
<td>uses MongoDB for its aggregation abilities and its amazing insert capacity.</td>
</tr>
<tr>
<td>wline</td>
<td>uses MongoDB to store level system data, experience points, and levels.</td>
</tr>
<tr>
<td>TalkOver</td>
<td>uses MongoDB as its primary database for online communication service. MongoDB is very good for our &quot;feeds&quot; functionality. It's easy to create channels and subscribers to show interesting and friendly content to users. Sharding is solving our problem with high load.</td>
</tr>
<tr>
<td>StoreHR</td>
<td>uses MongoDB as a super fast and scalable platform for our Proactive Web Based HR software which keeps employee data and documents compliant, secure and always available - Perfect when it comes to due diligence at exit time! Mongo on AWS let us get up and running quickly during development and lets us scale horizontally as needed to meet customer demand.</td>
</tr>
</tbody>
</table>
Trillsy uses MongoDB for everything, user data, multi site data, stats and content.

Bottomless Endless Online Gaming Engine 64, born from Zynga's original Mafia Wars game in 2008.

Review19 uses MongoDB as a complete solution for its scalable distributed Massive Multiplayer Online Gaming Engine 64.

Trillsy uses MongoDB as a complete solution for its scalable distributed Massive Multiplayer Online Gaming Engine 64.

iQmetrix uses MongoDB for our product catalog and for our configuration tool for XQ, our interactive, customer facing, solution for the wireless retail industry. We are running a MongoDB replica set in Microsoft's cloud platform Windows Azure.

UseMongoDB to hold all tracking information in a highly available and scalable sharded replications cluster. MongoDB provides us with the speed and scalability that we need to store and process all the document reading information we gather from document recipients, producing insight on readers behavior and real-time notifications to our users.

JackThreads, an ecommerce company, uses MongoDB for inventory tracking and reporting, allowing us to track products down to the individual unit.

Shelf9 combines the power of a shopping search engine with social shopping features. We uses MongoDB as storage backend for our real-time data-mining code.

Luminis Technologies is using MongoDB to build the Leren-op-Maat system which is an e-learning platform focussed on personalized learning for high schools. We are currently working with several high schools in the Netherlands where the system is used in the classroom. Most of our data is stored in a replica set running on Amazon AWS. It fits our RESTful architecture very well (our domain model is designed to work well as JSON structures) and the flexible scaling is very important to easily handle the extra load of new schools.

Betterez is a SaaS reservations, marketing and planning platform for the transportation industry designed to improve both passenger and operator experiences. MongoDB is used extensively in a replica set configuration spanning multiple data centers. MongoDB drives the Node.js web application and houses both the main data store and analytics data store. The flexibility and resiliency of the platform, combined with it's natural interface with JavaScript, enables the team to build technology with confidence.

DUVRI 81.08 is a governative web application to help Italian corporates so that can manage workspace safety risks to prevent accidents and protect workers' life. DUVRI 81.08 is compliant to safety workspace's law (d.lgs 81/2008). The application is built in PHP with Zend Framework and use MongoDB, configured in replica sets, as documental database wrapped by Doctrine ODM.
<table>
<thead>
<tr>
<th>Company</th>
<th>Database Usage Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Zomby</strong></td>
<td>Zomby is a massively multiplayer, location-based zombie apocalypse game. It was developed by Kudzu Creative Group. We chose MongoDB for our server primarily because of the ease of scalability. As our user-base expands globally, we will need to have multiple servers around the world, and MongoDB (and Erlang) will allow us to do that with minimal headache. Furthermore, due to the location-based nature of our game, we make extensive use of MongoDB’s geospatial queries, which are both convenient and efficient.</td>
</tr>
<tr>
<td><strong>Gusanito.com</strong></td>
<td>Gusanito.com uses MongoDB.</td>
</tr>
<tr>
<td><strong>Aion Innovations</strong></td>
<td>Aion Innovations uses MongoDB as the database for the whole application.</td>
</tr>
<tr>
<td><strong>Support.com, Inc</strong></td>
<td>Support.com, Inc uses MongoDB to store massive data sets related to automated snapshots of state and error information for computers, devices, and networks. We analyze this data in real-time to identify potential root causes and outcomes of home technology issues, make relevant product recommendations and predict patterns that may lead to subscriber churn.</td>
</tr>
<tr>
<td><strong>Footballrrating</strong></td>
<td>Footballrrating uses MongoDB to store all user data, such as login information, connecting to a social network, match votes, and player tracking configurations. Schema-less design means great flexibility in pushing new features without having to migrate clunky SQL tables around multiple servers.</td>
</tr>
<tr>
<td><strong>Factile</strong></td>
<td>Factile is a online free survey platform based on the flexible structure that MongoDB provides. Each survey is different in structure. Factile uses MongoDB for its fast performance a survey tool need and we run analytics on the survey responses easily.</td>
</tr>
<tr>
<td><strong>Dir</strong></td>
<td>Dir uses MongoDB to store all of its data models and finds it very useful for creating embed documents.</td>
</tr>
<tr>
<td><strong>Socialbakers</strong></td>
<td>Socialbakers uses MongoDB for its API. The company has developed an application that makes it easy for to get sophisticated metrics on performance through Facebook.</td>
</tr>
</tbody>
</table>
YMIRLINK is a Tokyo based e-mail service provider specializing in mass e-mail sending service for email marketing. In March 2012 we sent a record 1.6 billion e-mail messages per month via our SaaS-based mail system. In July 2012, we further improved the performances of our e-mail sending service by using MongoDB as our new mail transfer agent.

jScore lets recruiters users score candidates through the Job Des descriptions and also keep their environment waiting.

Mailgun uses MongoDB as their primary datastore and also for storing their big data stream which they use for analytics. Blazing fast reads help keep their environment snappy, and no-wait heavy writes help them keep our users working without waiting.

The National Register of Haunted Places uses MongoDB as its primary database, which they use for all non-relational data: accounts, Mailgun stats, logs, DNS cache—everything. Every replica set they have runs with an average daily load of 5,000 queries per second using just a couple of servers without breaking a sweat. Mailgun is a set of powerful APIs that allow you to send, receive, track and store email effortlessly.

4thex Solutions offer a security module/realm called Camofish for Glassfish, which uses MongoDB to persist users and passwords.

See also
- MongoDB Apps
- Use Cases
- User Feedback

Contributors
- JS Benchmarking Harness
- MongoDB kernel code development rules
- Project Ideas
- Roadmap
- UI
- Source Code
- Building
- Database Internals
- Contributing to the Documentation
- 10gen Contributor Agreement

JS Benchmarking Harness

This benchRun command is designed as a QA baseline perf measurement tool, not designed to be a "benchmark".

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 )
}

Dynamic values

res = benchRun( { ops : [ { ns : t.getFullName() ,
                           op : "update" ,
                           query : { _id : { "#RAND_INT" : [ 0 , 100 ] } } ,
                           update : { $inc : { x : 1 } } ] ,
                           parallel : 2 ,
                           seconds : 1 ,
                           totals : true } } )

Options

- host - the hostname of the machine mongod is running on (defaults to localhost)
- username - the username to use when authenticating to mongod (only use if running with auth)
- password - the password to use when authenticating to mongod (only use if running with auth)
- db - the database to authenticate to (only necessary if running with auth)
- ops - a list of objects describing the operations to run (documented below)
- parallel - the number of threads to run (defaults to single thread)
- seconds - the amount of time to run the tests for (defaults to one second)

Operation Options

- ns - the namespace of the collection you are running the operation on, should be of the form "db.collection"
- op - the type of operation, can be "findOne", "insert", or "update"
- query - the query object to use when querying or updating documents
- update - the update object (same as 2nd argument of update() function)
- doc - the document to insert into the database (only for insert)
- safe - boolean specifying whether to use safe writes (only for update and insert)

Dynamic operators

- RAND_INT [ min , max , <multiplier > ]
  - [ 0 , 10 , 4 ] would produce random numbers between 0 and 10 and then multiply by 4

More info:
http://github.com/mongodb/mongo/commit/3db3cb13dc1c522db8b597545d6c74b0967f1611c

MongoDB kernel code development rules

- C++ Style Guide
- JS Style Guide
- Git Committing and Pushing
- User Facing Conventions
  - Use camelCase for about everything
  - Include units in fields

C++ Style Guide

Coding conventions for the MongoDB C++ code...
Kernel class rules
Kernel code style
Kernel concurrency rules
Kernel exception architecture
Kernel logging
Kernel string manipulation
Memory management
Writing tests

For anything not mentioned here, default to google c++ style guide

JS Style Guide

10gen follows the google javascript style guide

Git Committing and Pushing

- commit messages should have the case in the message SERVER-XXX
- commit messages should be descriptive enough that a glance can tell the basics
- commits should only include 1 thought.
- do NOT push until running the test suite

User Facing Conventions

⚠️ These are very important as we can't change them easily – Much more than code conventions!

Anything users see – command line options, command names, command output, we need to think hard and carefully about the name to be used, and the exact format and consistency of the items output. For example, serverStatus output is a bit of a mismash of lowercase and camelCase. Let's fix that over time, starting with new things.

Anything user facing must be ran by several team members first.

- Do NOT add a new $operator without signoff by the entire team.
- Do NOT add a new command without signoff by the entire team.

Use camelCase for about everything

- --commandLineOptions
- { commandNames : 1, commandOption : “abc” }
- Names of fields in objects we generate - such as command responses, profiler fields.

Include units in fields

In things like serverStatus, include the units in the stat name if there is any chance of ambiguity. For example:

- writtenMB
- timeMs

We should have standards for these – i.e. megabytes should always be "MB" and not "Mb" and "Megabytes" in different places. So the standards are:

- for bytes : use "MB" and show in megabytes unless you know it will be tiny. Note you can use a float so 0.1MB is fine to show.
- for time: use millis ("Ms") for time by default. you can also use Secs and a float for times you know will be very long.
- for microseconds, use "Micros" as the suffix, e.g., timeMicros.

Kernel class rules

Design guidelines

- Never use multiple inheritance. If you need the service of several classes, use delegation. The only possible but highly unlikely exception to this is if your class inherits from other pure abstract classes.
- Have a comment before a class that explains its purpose. Perhaps the class name is so clear that this is obvious. Then some commentary on what you are up to.
- Only add members and methods to a class if they make sense w.r.t the bullet above. If you find yourself unsure to where to hook a piece of logic, rethink the class and surrounding classes purposes.
• Class names and methods names are to be descriptive of what they do. Avoid generic overloaded names (e.g., write, add, ...) to make grep easier (and maybe reading too).

• Don’t put implementation details in the header unless the user of the class needs to know them. Sometimes single line inline implementations are good “documentation”. If something needs to be inline for performance, put it at the bottom of the file using the inline keyword instead of in the middle of the class definition (if the implementation is more than a line or two long).

• Assume all methods can throw a DBException. If a class should never throw (e.g can be called in a destructor), that should be clear.

• Write a unit test for each class you create. If you can’t easily write a unit test for the class, that is a strong hint it has way too many external dependencies.

• Do not create early hierarchies. An early hierarchy is a one where there is only one type of derived class. If you need to separate functionality, use delegation instead. In that case, make sure to test separately.

• Avoid friend.

• Default to making classes non-assignable and non-copyable. (Use boost::noncopyable.)

Layout guidelines

• For classes where layout matters (anything with #pragma pack), put data members together at the top of the class. You must also have BOOST_STATIC_ASSERT(sizeof(ClassName) == EXPECTED_SIZE) either directly under the class or in the associated .cpp file

Kernel code style

• basics
• case
• comments
• inlines
• strings
• brackets
• class members
• functions
• templates
• namespaces
• start of file
• assertions
• return early
• numeric constants
• explicit constructors
• #includes
  • For cpp files:
  • For h files:
• file names
• casting
• RAII and bare vs. smart pointers

basics

• Use spaces, no literal tabs.
• 4 spaces per indentation.
• Limit lines to 100 columns.
• Use LF (Unix-style) line endings, not CR-LF (DOS)
  • git has a config option in Windows to convert line endings automatically (core.autocrlf)
• For anything that isn’t explicitly covered here, default to the Google C++ Style Guide:
  http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

case

Use camelCase for most varNames

See important notes on case on the parent page for user facing names!

comments

We follow http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Comments for placement of comments

As for style, we use javadoc’s in classes and methods (public or private) and simple comments for variables and inside code
/**
 * My class has X as a goal in life
 * Note: my class is fully synchronized
 */
class DoesX {
...
/**
 * This methods prints something and turns off the lights.
 * @param y the something to be printed
 */
void printAndGo(const std::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
std::map<std::string, BSONObj> _chunkMap;
/**
 * Helper that finds the light switch
 */
Pos findSwitch() const;
/** @return the light switch state. */
return switchState() const;
};

void DoX(bool y) {
    // if y is false, we do not need to do a certain action and explaining
    // why that is takes multiple lines.
    if (!y) {
        printf("y is true!\n");
    }
}

Don't forget – even if a class's purpose is obvious, you can put a
comment on it as to why it exists!

inlines

- Put long inline functions in a -inl.h file.
- If your inline function is a single line long, put it and its decl on the same line e.g.:
  ```
  int length() const { return _length; }
  ```
- If a function is not performance sensitive, and it isn't one (or 2) lines long, put it in the cpp file. Keep code out of headers.

strings

See

- util/mongoutils/str.h
- bson/stringdata.h

Use str::startsWith(), str::endsWith(), not strstr().

Use << 'c' not << "c".

Use str[0] == '\0' not strlen(str) == 0.

See Kernel string manipulation.

brackets
if (0) {
}
else if (0) {
}
else {
}
do {
} while (0);

class members

class Foo {
    int _bar;
};

functions

Declaration:

void foo(int v, MyType myItem);

Avoid declarations of extern functions in source files! Instead, include a proper .h file. Be sure to match the header filename to the source filename where the function definition appears.

Definition:

void foo(int v, MyType myItem) {
}

Invocation:

foo(1, MyType());

templates

set<int> s;

namespaces

namespace foo {
    int foo;
    namespace bar {
        int bar;
    }
}

start of file

license (AGPL or Apache, depending on C++ driverness)

assertions

See Kernel exception architecture.

return early
BAD

```cpp
int foo() {
    if (x) {
        ...
    }
}
```

GOOD

```cpp
int foo() {
    if (!x)
        return x;
    ...
}
```

Keeps indentation levels down and makes more readable.

**numeric constants**

Large, round numeric constants should be written in multiplied form so that you never need to count digits.

```cpp
const int tenMillion = 10*1000*1000;
const int megabyte = 1024*1024;
```

**explicit constructors**

To avoid implicit type conversion, use the "explicit" keyword before constructors that take a single parameter.

**#includes**

- Use "double quotes" for 10gen code, <angle brackets> for 3rd party or library headers.

```cpp
#include "mongo/pch.h"
#include <boost/thread.h>
#include <vector>
```

- Always use forward relative path from mongo/src/; do not use "."*

```cpp
correct:
#include "mongo/db/namespace_details.h"
incorrect:
#include "../db/namespace_details.h"
```

*For cpp files:*

- Include mongo/pch.h first, blank line.
- Include your .h file next, if applicable, blank line.
- Include third party headers next, sorted, blank line.
- Include 10gen headers last, sorted
example for classy.cpp:
#include "mongopch.h"

#include "mongo/db/classy.h"
#include <boost/thread.h>
#include <stdio.h>
#include <string>
#include "mongo/db/db.h"
#include "mongo/db/namespace_details.h"
#include "mongo/util/concurrency/qlock.h"

For h files:
- #pragma once at the top
- Forward declare, if possible, in lieu of including 10gen headers in headers. Only include things that are directly used in the header itself.
- Include third party headers first, sorted. blank line.
- Include 10gen headers last, sorted.
- Be extremely careful about including pch.h in a header.

file names
- Class definitions should go in a header file with the same name as the class. Insert _ in place of a capital letter. Do not use capital letters in filenames! example: ClassyClass's definition goes in "classy_class.h". ClassyClass's member function implementations go in "classy_class.cpp".
- Do not be afraid to make another file, even if it is really small. This is preferable to inserting your new class into a preexisting file.

casting
- Do not use C-style casts, ever.
- Use static_cast<> or const_cast<>.
- Be aware that dynamic_cast<>., unlike other casts, is done at run-time and calls a function.
- You should always check the return status of dynamic_cast<> for null.
- reinterpret_cast<> should be used sparingly and is typically done for converting structures to raw bytes for use with I/O drivers.

RAII and bare vs. smart pointers
- Aspire to embrace RAII
- When writing functions that take or return bare pointers, document the ownership semantics in the header comment. Is the caller responsible for managing returned pointer's memory? Does the callee take ownership of the pointed-to parameter, or does the callee retain ownership. Prefer caller-retains ownership of parameters and takes ownership of returned pointers, but use the appropriate policy for each situation.
- Generally, bare calls to delete and free() are red flags
  - except in destructors
- Use smart pointers such as boost::scoped_ptr and std::auto_ptr (know the difference between them!) to avoid memory leaks and ensure all new's and malloc's are paired with delete's and free's
- Use ON_BLOCK_EXIT or ScopeGuard to protect other resources that must be released
  - e.g. fopen/fclose pairs
  - Or, write an object to do this for you via constructor and destructor

Kernel concurrency rules

All concurrency classes must be placed under utils/concurrency. You will find several helper libraries there.

- Do not add mutexes without discussion with others. Concurrency and correctness is very hard in the large. Great care is required. For example the concurrency model in replica sets is hard to understand and error prone (at least it was initially and probably still is).

If you think there is a real need for an exception to the list below let's have the group weigh in and get a consenssus on the exception:

- Do not use/add recursive locks.
- Do not use rwlocks.
- Always acquire locks in a consistent order. In fact, the MutexDebugger can assist with verification of this. MutexDebugger is on for _DEBUG builds and will alert if locks are taken in opposing orders during the run.

Kernel exception architecture
There are several different types of assertions used in the MongoDB code. In brief:

- `uassert` checks for per-operation user errors. Operation-fatal.
- `massert` checks per-operation invariants. Operation-fatal.
- `verify` is a synonym for `massert`, that doesn’t require an error code.
- `fassert` checks fatal process invariants. Process-fatal.
- `wassert` warn (log) and continue.
- Calling `assert` is not allowed. Use one of the above instead.
- `dassert` just calls `verify` but only in debug mode. Do not use!

When per-operation invariant checks fail, the current operation fails, but the process and connection persist. This means that `massert`, `uassert`, and `verify` only terminate the current operation, not the whole process. Be careful not to corrupt process state by mistakenly using these assertions midway through mutating process state. Examples of this include `uassert` and `massert` inside of constructors and destructors. `fassert` failures will terminate the entire process; this is used for low-level checks where continuing might lead to corrupt data or loss of data on disk.

Both `massert` and `uassert` take error codes, so that all errors have codes associated with them. These error codes are assigned incrementally; the numbers have no meaning other than a way to associate a log message with a line of code. `scons` checks for duplicates, but if you want the next available code you can run:

```python
python buildscripts/errorcodes.py
```

A failed operation-fatal assertion throws an `AssertionException` or a child of that. The inheritance hierarchy is something like:

- `std::exception`
  - `mongo::DBException`
    - `mongo::AssertionException`
      - `mongo::UserException`
      - `mongo::MsgAssertionException`

See `util/assert_util.h`.

Generally, code in the server should be prepared to catch a `DBException`. `UserAssertionException`’s are particularly common as errors and should be expected. We use `resource acquisition is initialization` heavily.

Gotchas to watch out for:

- generally, don’t throw a `AssertionException` directly. Functions like `uasserted()` do work beyond just that. In particular, it makes sure that the `getLastError` structures are set up properly.
- think about where your asserts are in constructors, as the destructor wouldn’t be called. (But at a minimum, use `wassert` a lot therein, we want to know if something is wrong.)
- don’t throw in destructors of course.

**Kernel logging**

- **Basic Rules**
  - `cout/cerr` should never be used

- **Normal Logging**
  - debugging with levels of verbosity, default=0 (use just `log()` for that). See `-v` command line option.
    ```
    LOG( int x ) << ...
    ```
  - informational
    ```
    log() << ...
    ```
  - rate limited
    ```
    LOGSOME() << ...
    ```
  - warnings
    - recoverable
    - e.g. replica set node down
Kernel string manipulation

For string manipulation, use the util/mongoutils/str.h library.

`str.h`

`util/mongoutils/str.h` provides string helper functions for each manipulation. Add new functions here rather than lines and lines of code to your app that are not generic.

Typically these functions return a string and take two as parameters: string f(string,string). Thus we wrap them all in a namespace called `str`

`str::stream()` is quite useful:

```cpp
uassert(12345, str::stream() << "bad ns:" << ns, isok);
```

`StringData`

```cpp
/** A StringData object wraps a 'const string&' or a 'const char*' without
 * copying its contents. The most common usage is as a function argument that
 * takes any of the two forms of strings above. Fundamentally, this class tries
 * go around the fact that string literals in C++ are char[N]'s.
 * 
 * Note that the object StringData wraps around must be alive while the StringData
 * is.
 */
class StringData {
```

See also `bson/stringdata.h`.

`mongoutils`

MongoUtils has its own namespace. Its code has these basic properties:

1. are not database specific, rather, true utilities
2. are cross platform
3. may require boost headers, but not libs (header-only works with mongoutils)
4. are clean and easy to use in any c++ project without pulling in lots of other stuff
5. apache license

Memory management

Overall guidelines

- avoid using bare pointers for dynamically allocated objects. Prefer scoped_ptr, shared_ptr, or another RAII class such as BSONObj.
• do not use auto_ptr's and refactor legacy ones out whenever possible. (Careful with c++ driver and backward compatibility though.)

• If you assign the output of new/malloc() directly to a bare pointer you should document where it gets deleted/freed, who owns it along the way, and how exception safety is ensured. If you cannot answer all three questions then you probably have a leak.

Writing tests

We have three general flavors of tests currently.

General guidelines

It is important that tests can be run in parallel and still succeed.

For example, make sure that:

• try to use a unique collection name, for example named after the test

```
t = db.jstests_currentop
```

• if checking on current operations, make sure to add an NS filter

```
db.currentOp({ns: "test.mytest"})
```

• if possible, try to avoid things that are global to mongod or the database (oplog, profiling, fsync)

If the test cannot be run in parallel, add it to the blacklist in "skipTests" in shell/utils.js.

dbtests

Tests can be written in C++. These are in the dbtests/ subfolder. See the code therein. To run:

```scons test
./test --help```

jstests

Many tests are written as .js scripts that are executed via the mongo shell. See the Smoke Tests link at the bottom for how to run comprehensive sets of tests. To run a particular test:

```
# start mongod first then run a few simple tests:
mongo jstests/basic*.js
```

Note there are several subdirectories for different test suites. slowNightly is run by the buildbots only once a night; slowWeekly only once a week. Most other tests are ran every CI cycle (all the time).

Also note that that the js tests rely on functions defined in the "shell" directory (see servers.js and utils.js in particular).

Lightweight startup test.

You can inherit from class mongo::UnitTest and make a test that runs at program startup. These tests run EVERY TIME the program starts. Thus, they should be minimal: the test should ideally take 1ms or less to run. Why run the tests in the general program? This gives some validation at program run time that the build is reasonable. For example, we test that pcre supports UTF8 regex in one of these tests at startup. If someone had built the server with other settings, this would be flagged upon execution, even if the test suite has not been invoked. Most tests are not of this sort.

See Also

• Smoke Tests
  • http://buildbot.mongodb.org/

Project Ideas

If you're interested in getting involved in the MongoDB community (or the open source community in general) a great way to do so is by starting or contributing to a MongoDB related project. Here we've listed some project ideas for you to get started on. For some of these ideas projects are
already underway, and for others nothing (that we know of) has been started yet.

**A GUI**

One feature that is often requested for MongoDB is a GUI, much like CouchDB’s futon or phpMyAdmin. There are a couple of projects working on this sort of thing that are worth checking out:

http://github.com/sbellity/futon4mongo
http://www.mongodb.org/display/DOCS/Http+Interface
http://www.mongohq.com

We’ve also started to spec out the features that a tool like this should provide.

**Try Mongo!**

It would be neat to have a web version of the MongoDB Shell that allowed users to interact with a real MongoDB instance (for doing the tutorial, etc). A project that does something similar (using a basic MongoDB emulator) is here:

http://github.com/banker/mongulator

**Real-time Full Text Search Integration**

It would be interesting to try to nicely integrate a search backend like Xapian, Lucene or Sphinx with MongoDB. One idea would be to use MongoDB’s oplog (which is used for master-slave replication) to keep the search engine up to date.

**GridFS FUSE**

There is a project working towards creating a FUSE filesystem on top of GridFS - something like this would create a bunch of interesting potential uses for MongoDB and GridFS:

http://github.com/mikejs/gridfs-fuse

**Framework Adaptors**

Working towards adding MongoDB support to major web frameworks is a great project, and work has been started on this for a variety of different frameworks (please use google to find out if work has already been started for your favorite framework).

**Logging and Session Adaptors**

MongoDB works great for storing logs and session information. There are a couple of projects working on supporting this use case directly.

Logging:
- 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)
Write a killer app that uses MongoDB as the persistence layer!

**Roadmap**

Please see [jira](jira).

**UI**

Spec/requirements for a future MongoDB admin UI.

- list databases
  - repair, drop, clone?
- collections
  - validate(), datasize, indexsize, clone/copy
- indexes
- queries - explain() output
- security: view users, adjust
- see replication status of slave and master
- sharding
- system.profile viewer ; enable disable profiling
- curop / killop support

**Source Code**

Source for MongoDB and mongodb.org supported drivers is open source and hosted at [Github](https://github).

- [Mongo Database](https://github) (includes C++ driver)
- [Python Driver](https://github)
- [PHP Driver](https://github)
- [Ruby Driver](https://github)
- [Java Driver](https://github)
- [Perl Driver](https://github)
- [C# Driver](https://github)
- [Scala Driver](https://github)
- [Erlang Driver](https://github)
- [Haskell Driver](https://github)

(Additionally, community drivers and tools also exist and will be found in other places.)

**See Also**

- Building
- License

**Building**

Note: see the [Downloads](https://github) page for prebuilt binaries, it's recommended to use those as all full QA occurs after those are built.

This section provides instructions on setting up your environment to write Mongo drivers or other infrastructure code. For specific instructions, go to the document that corresponds to your setup.

Sub-sections of this section:

- Building Boost
- Building for FreeBSD
- Building for Linux
- Building for OS X
- Building for Solaris
- Building for Windows
- Building Spider Monkey
- Building with V8
- scons
Building Boost

NOTE
This is not necessary when building mongo versions 2.1.1 or later.

MongoDB uses the [www.boost.org] C++ libraries.

Windows

See also the prebuilt libraries page.

By default c:\boost is checked for the boost files. Include files should be under 'boost\boost, and libraries in 'boost\lib'.

First download the boost source. Then use the 7 Zip utility to extract 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.

```bash
> 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.nts6\bjam --help
> rem see also mongo/buildscripts/buildboost*.bat
> rem build DEBUG libraries:
> tools\jam\src\bin.nts6\bjam variant=debug threading=multi --with-program_options --with-filesystem --with-date_time --with-thread
> mkdir lib
> move stage\lib\* lib\n```

Linux

It's common with linux to install boost via a package manager – see the Building for Linux page for specifics.

However one might also want to build from boost.org sources in some cases, for example to use a newer version of boost, to generate static libraries, etc.

The following – far from comprehensive, rather an example – shows manually building of boost libraries on Linux.

```bash
$ sudo ./bootstrap.sh
$ ./b2 --help
$ # now build it
$ ./b2
$ # or
$ ./b2 --with-program_options --with-filesystem --with-date_time --with-thread
$ sudo ./b2 install
```

Troubleshooting

Unresolved external get_system_category() when linking MongoDB
Try defining BOOST_SYSTEM_NO_DEPRECATED in the MongoDB SConstruct file.
Building for FreeBSD

On FreeBSD 8.0 and later, there is a mongodb port you can use.

For FreeBSD <= 7.2:

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 libunwind:
   
   ```
   $ cd /usr/ports/devel/libunwind && make && make install
   ```

7. Change to the database source directory:

8. `scons` .

See Also

- Building for Linux - many of the details there including how to clone from git apply here too.

Building for Linux

Note: Binaries are available for most platforms. Most users won't need to compile mongo themselves; in addition every prebuilt binary has been regression tested. See the Downloads page for these prebuilt binaries.

- Build Prerequisites
  - Fedora
  - Ubuntu
- Building
  - Building Older Versions
    - Fedora
    - Ubuntu
      - Ubuntu 8.04 and 8.10
      - Ubuntu 9.04 and Newer
    - Building
    - Troubleshooting

These instructions apply to the git master branch, and versions 2.1.1 and onward. At the end of the document are instructions for older versions.

Build Prerequisites

You'll need SCons, the gnu C++ toolchain, and glibc-devel. To get the code from github, you'll probably also want git.

**Fedora**

```
sudo yum -y install git-core scons gcc-c++ glibc-devel
```
**Ubuntu**

```
sudo apt-get install git-core build-essential scons
```

**Building**

1. Install any needed prerequisites (see above).
2. Get the source code

```
  git clone git://github.com/mongodb/mongo.git
cd mongo
```
3. Pick a version to build (only use "master" if you're doing development work).
   a. List all tagged versions

```
  git tag -l
```
   b. Check out a tagged release, e.g. 2.0.4

```
  git checkout r2.0.4
```
4. Compile

```
  scons all
```
5. Install. Use `--prefix` to specify where you want to install your binaries. Defaults to /usr/local.

```
  scons --prefix=/opt/mongo  install
```

**Building Older Versions**

**Fedora**

The following steps have been reported to work for Fedora versions from 11 to 16. (If they don't work on newer versions, please report this to mongodb-user@googlegroups.com.) Fedora versions 10 and earlier ship with a version of SCons that is too old to build MongoDB, but may work if you manually install SCons.

```
  sudo yum -y install git-core scons gcc-c++ glibc-devel
  sudo yum -y install boost-devel boost-devel-static
```

**Ubuntu**

Note: See SpiderMonkey note above.

Use `cat /etc/lsb-release` to see your version.

**Ubuntu 8.04 and 8.10**

```
  apt-get -y install git-core build-essential
  apt-get -y install libboost-dev libboost-program-options-dev libboost-thread-dev
  libboost-filesystem-dev
```

Ubuntu 8.04 and 8.10 ship with a version of SCons that is too old to build MongoDB, but may work if you manually install SCons.

**Ubuntu 9.04 and Newer**
aptitude install -y git-core build-essential scons
aptitude install -y libboost-dev libboost-program-options-dev libboost-thread-dev
libboost-filesystem-dev

Building

1. Install prerequisites
2. Get source
   
   ```
   git clone git://github.com/mongodb/mongo.git
cd mongo
   # pick a stable version unless doing true dev
   git tag -l
   # Switch to a stable branch (unless doing development) --
   # an even second number indicates "stable". For example:
   git checkout r2.0.4
   ```
3. Build
   
   ```
   scons all
   ```
4. Install --prefix can be anywhere you want to contain your binaries, e.g., /usr/local or /opt/mongo.
   
   ```
   scons --prefix=/opt/mongo install
   ```

Troubleshooting

- Link errors. If link errors occur, the -t gcc option is useful for troubleshooting. Try adding -t to the SConstruct file’s LINKFLAGS.
- Static libraries. The --release scons option will build a binary using static libraries. You may need to install static boost libraries when using this option.

Building for OS X

- Prerequisites
  - Xcode
  - SCons
  - git
- Building
- Older Versions of Mongo
- 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:

Prerequisites

Xcode

Available in the App Store. You only need to get the command line tools, if you don’t want to install the whole IDE.

SCons

SCons is the build tool used to compile mongo binaries. It is available from http://www.scons.org.
If you have easy_install or pip already installed, you can use them to install scons.

```
easy_install scons
```

```
pip install scons
```

**git**

An installer package is available from [http://git-scm.com/](http://git-scm.com/)

**Building**

1. Install any needed prerequisites (see above).
2. Get the source code

```
git clone git://github.com/mongodb/mongo.git
cd mongo
```

3. Pick a version to build (only use “master” if you’re doing development work).
   a. List all tagged versions

```
git tag -l
```

   b. Check out a tagged release, e.g. 2.0.4

```
git checkout r2.0.4
```

4. Compile

```
scons all
```

5. Install. Use --prefix to specify where you want to install your binaries. Defaults to /usr/local.

```
scons --prefix=/opt/mongo install
```

**Older Versions of Mongo**

**Upgrading to Snow Leopard**

If you have installed Snow Leopard, the builds will be 64-bit -- so if moving from a previous OS release, a bit more setup may be required than one might first expect.

**Setup**

**Sources**

The mongodb source is on github. To get sources first download the git client and install it.

- Then git clone git://github.com/mongodb/mongo.git (more info)
  
  **Note** If you do not wish to install git you can instead get the source code from the [Downloads](http://git-scm.com/) page.

**Prerequisites**

- Install gcc. Install XCode tools for Snow Leopard. gcc version 4.0.1 (from older XCode Tools install) works, but you will receive compiler warnings. One way to get a newer gcc is to install Command Line Tools for XCode from developer.apple.com.

**Package Manager Setup**
1. **Install Homebrew**

2. Update/install dependencies:

   ```bash
   brew install boost
   ```

3. **Install SCons**:

   ```bash
   brew install scons
   ```

**Manual Setup**

**Install Apple developer tools**

**Install libraries (32-bit option)**

1. Download boost 1.37.0. Apply the following patch:

   ```diff
   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
   @@ -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=\$=) $\$(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=\$=) $\$(OPTIONS) $\$(USER_OPTIONS)
   }
   ```

   then,

   ```bash
   ./configure; make; sudo make install
   ```

2. **Install pcre (must enable UTF8)**

   ```bash
   ./configure --enable-utf8 --enable-unicode-properties --with-match-limit=200000
   --with-match-limit-recursion=4000; make; sudo make install
   ```


   ```bash
   ./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:

```bash
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) $<
+    $(CC) -c -MD $*.d $(CFLAGS) $<
-CPU_ARCH = $(shell uname -m)
+CPU_ARCH = "X86_64"
 ifdef (86,${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 -FPI
+endif
```

compile and install

```bash
cd src
make -f Makefile.ref
sudo JS_DIST=/usr/64 make -f Makefile.ref export
```

remove the dynamic library

```bash
sudo rm /usr/64/lib64/libjs.dylib
```

# Download boost 1.37.0 Apply the following patch:
diff --u -r a/configure b/configure
--- a/configure 2009-01-26 14:10:42.000000000 -0500
++ b/configure 2009-01-26 10:21:29.000000000 -0500
@@ -9,9 +9,9 @@
BJAM=""
TOOLSET=""
-BJAM_CONFIG=""
+BJAM_CONFIG=\ "architecture=x86 address-model=64 --layout=system"
BUILD=""
-PREFIX=/usr/local
+PREFIX=/usr/64
EPREFIX=
LIBDIR=
INCLUDEDIR=

diff --u -r a/tools/build/v2/tools/darwin.jam b/tools/build/v2/tools/darwin.jam
@@ -367,5 +367,5 @@
 actions link.dll bind LIBRARIES
 |
 - "$\{\(CONFIG_COMMAND\)\}" -dynamiclib -Wl,-single_module -install_name "$\{<(B):(S):\}" -L"$(LINKPATH)"
-o "$\{<(C):""$(<):""$(LIBRARIES):""-l$(FINDLIBS-SA) -l$(FINDLIBS-ST) $(FRAMEWORK_PATH)
-\(FRAMEWORK\)=<D>:<S>= $\{OPTIONS\} $\{USER_OPTIONS\}
+ "$\{\(CONFIG_COMMAND\)\}" -dynamiclib -Wl,-single_module -install_name "=/usr/64/lib/$\{<(B):(S):\}" -L
 "$\{LINKPATH\}" -o "$\{<(C):""$(<):""$(LIBRARIES):""-l$(FINDLIBS-SA) -l$(FINDLIBS-ST) $(FRAMEWORK_PATH)
-\(FRAMEWORK\)=<D>:<S>= $\{OPTIONS\} $\{USER_OPTIONS\}
 |
 |
 |
then,

./configure; make; sudo make install

# Install pcre (must enable UTF8)

CFLAGS="-m64" CXXFLAGS="-m64" LDFLAGS="-m64" ./configure --enable-utf8 --with-match-limit=200000
--with-match-limit-recursion=4000 --enable-unicode-properties --prefix /usr/64; make; sudo make install

# Install unit test framework http://unittest.red-bean.com/ (optional)

CFLAGS="-m64" CXXFLAGS="-m64" LDFLAGS="-m64" ./configure --prefix /usr/64; make; sudo make install

Compiling

To compile 32-bit, just run:

scons

To compile 64-bit on 10.5 (64 is default on 10.6), run:

scons --64

See the, MongoDB scons page for more details/compile options.

Troubleshooting

- Undefined symbols: "_PR_NewLock", referenced from: _JS_Init in libjs.a.
- Try not using the scons --release option (if you are using it). That option attempts to use static libraries.
Building for Solaris

MongoDB server currently supports little endian Solaris operation. (Although most drivers – not the database server – work on both.)

Community: Help us make this rough page better please! (And help us add support for big endian please...)

Prerequisites:

- g++ 4.x (SUNWgcc)
- scons (need to install from source)
- spider monkey Building Spider Monkey
- pcre (SUNWpcre)
- boost (need to install from source)

See Also

- Joyent
- Building for Linux - many of the details there including how to clone from git apply here too

Building for Windows

Binaries are available for most platforms. Most users won't need to compile mongo themselves; in addition every prebuilt binary has been regression tested. See the Downloads page for these prebuilt binaries.

MongoDB can be compiled for Windows (32 and 64 bit); You will need to have the platform sdk installed. The platform sdk can be installed manually, and it comes with Visual (Studio) C++ as well. SCons is the make mechanism, although several .vcxprojs and a .sln solution file are also included in the project for convenience when using the Visual Studio 2010 IDE.

There are several dependencies exist which are listed below; you may find it easier to simply download a pre-built binary.

- Building with Visual Studio 2008
- Building with Visual Studio 2010
- Building the Shell

See Also

- Prebuilt Boost Libraries
- Prebuilt SpiderMonkey for VS2010
- Building Boost
- Building SpiderMonkey
- Windows Quick Links
- scons

Boost 1.41.0 Visual Studio 2010 Binary

This is OLD and was for the VS2010 BETA. See the new Boost and Windows page instead.

The following is a prebuilt boost binary (libraries) for Visual Studio 2010 beta 2.

The MongoDB vcxproj files assume this package is unzipped under c:\Program Files\boost\boost_1_41_0\.

- http://downloads.mongodb.org/misc/boost_1_41_0_binary_vs10beta2.zipx

Note: we're not boost build gurus please let us know if there are things wrong with the build.

See also the prebuilt boost binaries at http://www.boostpro.com/download.

Boost and Windows

- Visual Studio 2010
  - Prebuilt from mongodb.org
  - Building Yourself
- Visual Studio 2008
Visual Studio 2010

Prebuilt from mongodb.org

Click here for a prebuilt boost library for Visual Studio 2010. 7zip format.

Building Yourself

- Download the boost source from boost.org. Move it to C:\boost/.
- We have successfully compiled version 1.42 – you might want to try that version or higher, but not 1.45 or later. 1.45 changed the interface to the boost::filesystem library and we’ve yet to catch up. See additional notes section at end of this page too.
- Run C:\Program Files (x86)\Microsoft Visual Studio 10.0\vc\vcvarsall.bat.
- From the MongoDB source project, run buildscripts\buildboost.bat. Or, buildboost64.bat for the 64 bit version.

Visual Studio 2008

Prebuilt from mongodb.org

Click here for a prebuilt boost library for Visual Studio 2008. 7zip format. This file has what you need to build MongoDB, but not some other boost libs, so it's partial.

Prebuilt from boostpro.com

Or, you can download a complete prebuilt boost library for 32 bit VS2008 at http://www.boostpro.com/products/free. Install the prebuilt libraries for Boost version 1.35.0 (or higher - generally newer is better). During installation, for release builds choose static multithread libraries for installation. The Debug version of the project uses the DLL libraries; choose all multithread libraries if you plan to do development. From the BoostPro installer, be sure to select all relevant libraries that mongodb uses -- for example, you need Filesystem, Regex, Threads, and ProgramOptions (and perhaps others).

Building Yourself

- Download the boost source from boost.org. Move it to C:\boost/.
- From the Visual Studio 2008 IDE, choose Tools. Visual Studio Command Prompt to get a command prompt with all PATH variables set nicely for the C++ compiler.
- From the MongoDB source project, run buildscripts\buildboost.bat. Or, buildboost64.bat for the 64 bit version.

Additional Notes

When using bjam, MongoDB expects

- variant=debug for debug builds, and variant=release for release builds
- threading=multi
- link=static runtime-link=static for release builds
- address-model=64 for 64 bit

Building the Mongo Shell on Windows

You can build the mongo shell with either scons or a Visual Studio 2010 project file.

Scons

scons mongo.exe

Visual Studio 2010 Project File

A VS2010 vcxproj file is available for building the shell. From the mongo directory open shell/msvc/mongo.vcxproj.

For versions of the shell prior to version 1.9, the project file assumes that GNU readline is installed in ..\readline\ relative to the mongo project. If you would prefer to build without having to install readline, remove the definition of USE_READLINE in the preprocessor definitions section of the project file, and exclude readline.lib from the project.

The project file currently only supports 32 bit builds of the shell (scons can do 32 and 64 bit). However this seems sufficient given there is no real need for a 64 bit version of the shell.
Readline Library

Versions of the shell prior to 1.9 used the GNU readline library to facilitate command line editing and history. For these older versions, you can build the shell without readline but would then lose that functionality. USE_READLINE is defined when building with readline. SCons will look for readline and if not found build without it.

See Also

- Prebuilt readline for Windows 32 bit at SourceForge (DLL version)

Building with Visual Studio 2008

NOTE

These instructions are for versions of mongo prior to 2.1.1. For version 2.1.1 and newer, the instructions for Visual Studio 2010 and Visual Studio 2008 are the same.

- Get the MongoDB Source Code
- Get Boost Libraries
- Get SpiderMonkey
- Install SCons
- Building MongoDB with SCons
- Troubleshooting

MongoDB can be compiled for Windows (32 and 64 bit) using Visual C++. SCons is the make mechanism we use with VS2008. (Although it is possible to build from a sln file with vs2010.)

There are several dependencies exist which are listed below; you may find it easier to simply download a pre-built binary.

Get the MongoDB Source Code

Download the source code from Downloads.

Or install Git. Then:

- `git clone git://github.com/mongodb/mongo.git` (more info)
- `git tag` 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:

```
n junction "c:\Program Files\boost\latest" "c:\Program Files\boost\boost_1_42"
```

This should return the following output:
Get SpiderMonkey

Build a SpiderMonkey js engine library (js.lib) – details here.

Install SCons

If building with scons, install SCons:

- First install Python: [link](http://www.python.org/download/releases/2.6.4/).
- Then SCons itself: [link](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

- 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.

Building with Visual Studio 2010

- Binaries are available for most platforms. Most users won't need to compile mongo themselves; in addition every prebuilt binary has been regression tested. See the Downloads page for these prebuilt binaries.

Older versions

- Get the MongoDB Source Code
- Building MongoDB from the IDE
- Building with SCons
- Troubleshooting

v2.1.1+

- Get the MongoDB Source Code
- Get Boost Libraries
- Get SpiderMonkey
- Building MongoDB from the IDE
- Install 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.)

These instructions are for mongo versions 2.1.1 and later.

**Get the MongoDB Source Code**

Download the source code from Downloads.

Or install Git. Then:

- `git clone git://github.com/mongodb/mongo.git` (more info)
- `git tag -l` to see tagged version numbers
- Switch to a stable branch (unless doing development) -- an even second number indicates "stable". For example:
  - `git checkout r1.4.1`

**Building MongoDB from the IDE**

Open the db\db_10.sln solution file.

Note: a separate project file exists for the mongo shell. Currently the C++ client libraries must be built from scons (this obviously needs to be fixed...)

**Building with SCons**

1. Install SCons:
   a. First install Python: http://www.python.org/download/releases/2.7.2/.
      - **Note** Make sure to install the 32 bit version of python and not the 64 bit as the scons binaries below are 32 bit.
   d. Add the python scripts directory (e.g., C:\Python27\Scripts) to your PATH.

1. Build:

```bash
scons // build mongod
scons --release mongoclient.lib // build C++ client driver library
scons --release core // build all end user components
```

Add --64 or --32 to get the 64 and 32-bit versions, respectively. Replace --release with --dd to build a debug build.

**Troubleshooting**

If you are using scons, check the file config.log which is generated.

If scons does not automatically find Visual Studio, try using the Visual Studio Command Prompt, which will set your path for you. Alternatively, set your path manually by running the VS2010 vcvars*.bat files. Location may vary with install but usually it is something like:

- C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common\Tools\vsvars32.bat
- C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\amd64\vcvars64.bat

**Older versions**

The following instructions are for versions of mongo prior to 2.1.1.

There are several dependencies exist which are listed below; you may find it easier to simply download a pre-built binary.

**Get the MongoDB Source Code**

Download the source code from Downloads.

Or install Git. Then:

- `git clone git://github.com/mongodb/mongo.git` (more info)
- `git tag -l` to see tagged version numbers
Switch to a stable branch (unless doing development) -- an even second number indicates "stable". (Although with sharding you will want the latest if the latest is less than 1.6.0.) For example:

- `git checkout r1.4.1`

**Get Boost Libraries**

- Click here for a prebuilt boost library for Visual Studio. 7zip format. This file has what you need to build MongoDB, but not some other boost libs, so it's partial. Uncompress this to the c:\boost directory. Your actual files are in c:\boost\boost
- See the Boost and Windows page for other options. Use v1.42 or higher with VS2010.

**Get SpiderMonkey**

- Download prebuilt libraries and headers here for VS2010. Place these files in ..\js\ relative to your mongo project directory.
- Or (more work) build SpiderMonkey js.lib yourself -- details here.

**Building MongoDB from the IDE**

Open the db\db_10.sln solution file.

Note: a separate project file exists for the mongo shell. Currently the C++ client libraries must be built from scons (this obviously needs to be fixed...)

**Install SCons**

If building with scons, install SCons:

- First install Python: [http://www.python.org/download/releases/2.7.2/](http://www.python.org/download/releases/2.7.2/).
  - **Note** Make sure to install the 32 bit version of python and not the 64 bit as the scons binaries below are 32 bit.
  - It's recommended you install pywin32 if you want to do parallel builds (scons -j).
  - [http://sourceforge.net/projects/pywin32/files/pywin32/PyWin32-1.5.7.win32-py2.7.exe/download](http://sourceforge.net/projects/pywin32/files/pywin32/PyWin32-1.5.7.win32-py2.7.exe/download)
  - Add the python scripts directory (e.g., C:\Python27\Scripts) to your PATH.

The SConstruct file from the MongoDB project is the preferred way to perform production builds. Run scons in the mongo project directory to build.

If scons does not automatically find Visual Studio, preset your path for it by running the VS2010 vcvars*.bat files. Location may vary with install but usually it is something like:

- C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\Tools\vsvars32.bat
- C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\amd64\vcvars64.bat

To build:

```
scons // build mongod
scons mongoclient.lib // build C++ client driver library
scons all // build all end user components
scons . // build all including unit tests and C++ driver zip file
scons --64 // build the 64 bit version
scons --dd // build with debug symbols
scons -jX // perform X steps in parallel (e.g. -j16 will compile 16 files at once)
```

**Troubleshooting**

- Can't find jstypes.h when compiling.
  - This file is generated when building SpiderMonkey. See the Building SpiderMonkey page for more info.
- Can't find / run cl.exe when building with scons.
  - Be sure to use Visual Studio Command Prompt so that your path is set correctly.
- LINK : fatal error LNK1104: cannot open file js64d.lib js64r.lib js32d.lib js32r.lib
  - Get the prebuilt spidermonkey libraries -- or copy your self-built js.lib to the above name.
  - You can also see this if you're using the wrong compiler; this is the result if you try to use Visual Studio Express instead of Visual Studio Professional/Ultimate, which is a different product.

**Building Spider Monkey**
MongoDB uses SpiderMonkey for server-side Javascript execution.

Pre v2.0: MongoDB requires a js.lib file when linking. This page details how to build js.lib.

v2.0+: this is handled automatically by the Mongo build scripts via files under third_party/ in the MongoDB project directory.

Note: V8 Javascript support is under development.

**Building js.lib - Unix**

Remove any existing xulrunner

First find out what has been installed

dpkg -l | grep xulrunner

e.g.

```
ii  xulrunner-1.9.1                   1.9.1.13+build1+nobinonly-0ubuntu0.9.10.1 XUL + XPCOM
ii  xulrunner-1.9.1-dev               1.9.1.13+build1+nobinonly-0ubuntu0.9.10.1 XUL + XPCOM
```

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"
m preval Makefile.ref
make -f Makefile.ref
```

SpiderMonkey does not use UTF-8 by default, so we enable before building.

An experimental SConstruct build file is available [here].

Install

```
JS_DIST=/usr make -f Makefile.ref export
```

By default, the mongo scons project expects spidermonkey to be located at ../js/.
Building js.lib - Windows

Prebuilt

- VS2008: a prebuilt SpiderMonkey library and headers for Win32 is attached to this document (this file may or may not work depending on your compile settings and compiler version).
- VS2010 prebuilt libraries (js64d.lib, etc.)

Alternatively, follow the steps below to build yourself.

Download

From an msysgit or cygwin shell, run:

```
curl -O ftp://ftp.mozilla.org/pub/mozilla.org/js/js-1.7.0.tar.gz
tar zxvf js-1.7.0.tar.gz
```

Build

```
cd js/src
export CFLAGS="-DJS_C_STRINGS_ARE_UTF8"
make -f Makefile.ref
```

If cl.exe is not found, launch Tools...Visual Studio Command Prompt from inside Visual Studio -- your path should then be correct for make.

If you do not have a suitable make utility installed, you may prefer to build using scons. An experimental SConstruct file is available in the mongdb/snippets project. For example:

```
cd js/src
git clone git://github.com/mongodb/mongo-snippets.git
cp mongo-snippets/jslib-sconstruct js/src/SConstruct
scons
```

Troubleshooting scons

Note that scons does not use your PATH to find Visual Studio. If you get an error running cl.exe, try changing the following line in the msvc.py scons source file from:

```
MVSdir = os.getenv('ProgramFiles') + r'\Microsoft Visual Studio 8'
```

to

```
MVSdir = os.getenv('ProgramFiles') + r'\Microsoft Visual Studio ' + version
```

See Also

- Building MongoDB

Building with V8

Linux or OSX
$ pwd
~/mongo
$ cd ..
$ svn checkout http://v8.googlecode.com/svn/trunk/ v8
$ cd v8
$ scons arch=x64 debuggersupport=off snapshot=off profilingsupport=off
$ cd ../mongo
$ scons --usev8

Windows

$ pwd
~/mongo
$ cd ..
$ svn checkout http://v8.googlecode.com/svn/trunk/ v8
$ cd v8
$ scons env="PATH:%PATH%,INCLUDE:%INCLUDE%,LIB:%LIB%" arch=x64
$ cd ../mongo
$ scons --usev8 --release --64

scons

Use **scons** to build MongoDB and related utilities and libraries. See the SConstruct file for details.

Run **scons --help** to see all options.

**Targets**

Run **scons <target>**.

- scons .
- scons all
- scons mongod build mongod
- scons mongo build the shell
- scons shell generate (just) the shell .cpp files (from .js files)
- scons mongoclient build just the client library (builds libmongoclient.a on unix)
- scons test build the unit test binary test

**Options**

- **--d** debug build; all this does is turns optimization off
- **--dd** debug build with _DEBUG defined (extra asserts, checks, etc.)
- **--release**
- **--32** force 32 bit
- **--64** force 64 bit
- **--clean**
- **--mute** suppress compile and link command lines

**Troubleshooting**

scons generates a **config.log** file. See this file when there are problems building.

**See Also**

Smoke Tests

**Database Internals**

This section provides information for developers who want to write drivers or tools for MongoDB, contribute code to the MongoDB codebase itself, and for those who are just curious how it works internally.

Sub-sections of this section:
Caching

Memory Mapped Storage Engine

This is the current storage engine for MongoDB, and it uses memory-mapped files for all disk I/O. Using this strategy, the operating system's virtual memory manager is in charge of caching. This has several implications:

- There is no redundancy between file system cache and database cache: they are one and the same.
- MongoDB can use all free memory on the server for cache space automatically without any configuration of a cache size.
- Virtual memory size and resident size will appear to be very large for the mongod process. This is benign: virtual memory space will be just larger than the size of the datafiles open and mapped; resident size will vary depending on the amount of memory not used by other processes on the machine.
- Caching behavior such as Least Recently Used (LRU) discarding of pages, and laziness of page writes is controlled by the operating system (the quality of the VMM implementation will vary by OS.)

To monitor or check memory usage see: Checking Server Memory Usage

Memory per connection

A thread is associated with each connection from clients to the database. Each thread has a stack that has a size of a few MB. The unused portions of these stacks can be swapped out by the OS as long as the connections live for a while (would not if connect / operation / disconnect in 2 seconds).

Binary footprint

You can get a feel for the “inherent” memory footprint of Mongo by starting it fresh, with no connections, with an empty /data/db directory and looking at the resident bytes.

See Also
- Checking Server Memory Usage
- The Linux Out of Memory OOM Killer

Durability Internals

The main durability page (not the internals page) is the Journaling page.

Files

The data file format is unchanged.

Journal files are placed in /data/db/journal/.

Running

Run with --journal to enable journaling/durable storage. Both mongod and test support this option.

Declaring Write Intent
When writing mongod kernel code, one must now declare an intention to write. Declaration of the intent occurs before the actual write. See db/dur.h. The actual write must occur before releasing the write lock.

When you do your actual writing, use the pointer that dur::writing() returns, rather than the original pointer.

```c
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

```c
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.

```sh
mongo --nodb jstests/dur/<testname>.js
```

**Administrative**

```sh
# dump journal entries during any recover, and then start normally
mongod --journal --durOptions 1

# recover and terminate
mongod --journal --durOptions 4

# dump journal entries (doing nothing else) and then terminate
mongod --journal --durOptions 7

# extra checks that everything is correct (slow but good for qa)
mongod --journal --durOptions 8
```

**Diagrams**

- diagram 1 - process steps
- diagram 2 - journal file structure

**Parsing Stack Traces**

`addr2line`
c++filt

You can use `c++filt` to demangle function names by pasting the whole stack trace to stdin.

Finding the right binary

To find the binary you need:

- Get the commit at the header of any of our logs.
- Use git to locate that commit and check for the surrounding "version bump" commit

Download and open the binary:

```
curl -O http://s3.amazonaws.com/downloads.mongodb.org/linux/mongodb-linux-x86_64-debugsymbols-1.x.x.tgz
```

You can also find debugsymbols for official builds by clicking "list" on the Downloads page.

Example 1

Then, the log has lines like this:

```
/home/abc/mongod(_ZN5mongo15printStackTraceERSo+0x27) [0x689280]
```

You want the address in between the brackets [0x689280]

Note you will get more than one stack frame for the address if the code is inlined.

Example 2

Actual example from a v1.8.1 64 bit linux build:

```
$ curl http://downloads.mongodb.org/linux/mongodb-linux-x86_64-debugsymbols-1.8.1.tgz > out.tgz
$ tar -xzf out.tgz
$ cd mongodb-linux-x86_64-debugsymbols-1.8.1/
$ cd bin
$ addr2line --help
$ addr2line -i -e mongod 0x6d6a74
/mnt/home/buildbot/slave/Linux_64bit_V1.8/mongo/db/repl/health.cpp:394
$ addr2line -i -e mongod 0x6d0694
/mnt/home/buildbot/slave/Linux_64bit_V1.8/mongo/db/repl/rs.h:385
/mnt/home/buildbot/slave/Linux_64bit_V1.8/mongo/db/repl/replset_commands.cpp:111
```

Cursors

Redirection Notice
This page should redirect to Internals.

Error Codes
If you have an error event and it isn’t obvious what the error is, query for that error code on Jira. If still nothing please post to support forums.

This list is HIGHLY incomplete. This page is a stub.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>10003</td>
<td>objects in a capped ns cannot grow</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>duplicate key error</td>
<td>_id values must be unique in a collection</td>
</tr>
<tr>
<td>11001</td>
<td>duplicate key on update</td>
<td></td>
</tr>
<tr>
<td>12000</td>
<td>idxNo fails</td>
<td>an internal error</td>
</tr>
<tr>
<td>12001</td>
<td>can’t sort with $snapshot</td>
<td>the $snapshot feature does not support sorting yet</td>
</tr>
<tr>
<td>12010, 12011, 12012</td>
<td>can’t $inc/$set an indexed field</td>
<td></td>
</tr>
<tr>
<td>13312</td>
<td>replSet error : logOp() but not primary?</td>
<td>Fixed in v2.0. Report if seen v2.0+</td>
</tr>
<tr>
<td>13440</td>
<td>bad offset accessing a datafile</td>
<td>Run a database --repair. If journaling is on this shouldn’t happen.</td>
</tr>
</tbody>
</table>

Internal Commands

Most commands have helper functions and do not require the $cmd.findOne() syntax. These are primarily internal and administrative.

```javascript
> 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:

```javascript
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:

```javascript
t = now();
cloneDatabase();
end = now();
applyOperations(t..end);
```

cCloneDatabase effectively exports/imports all the data in the database. Note the actual "image" we will get may or may not include data modifications in the time range (t..end). Thus, we apply all logged operations from that range when the cloning is complete. Because of our repeatability property, this is safe.

See class Cloner for more information.

Smoke Tests

- Test Organization
- Running all the tests
- smoke.py
- Running a jstest manually
- Running the C++ unit tests
- See Also

Test Organization

1. dbtests/*.cpp has C++ unit tests
2. jstests/*.js has core tests
3. jstests/repl/*.js has replication tests
4. jstests/sharding/*.js has sharding tests
5. slowNightly/*.js has tests that take longer and run only at night
6. slowWeekly/*.js has tests that take even longer and run only once a week

### Running all the tests

```bash
scons smoke smokeDisk smokeTool smokeAuth startMongod smokeClient smokeJs
scons startMongodSmallOplog smokeJs
scons startMongod smokeJsSlowNightly
scons smokeTool
scons smokeReplSets
scons smokeDur
scons mongosTest smokeSharding
scons smokeRepl smokeClone
scons startMongod smokeParallel
```

### smoke.py

`smoke.py` lets you run a subsets of the tests in `jstests/`. When it is running tests, it starts up an instance of mongod, runs the tests, and then shuts it down again. You can run it while running other instances of MongoDB on the same machine: it uses ports in the 30000 range and its own data directories.

For the moment, `smoke.py` must be run from the top-level directory of a MongoDB source repository. This directory must contain at least the `mongo` and `mongod` binaries. To run certain tests, you'll also need to build the tools and `mongodb`. 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:

```bash
$ python buildscripts/smoke.py --help
```

**Usage:**

```
Usage: smoke.py [OPTIONS] ARGS*
```

**Options:**

- `-h, --help` show this help message and exit
- `--mode=MODE` If "files", ARGS are filenames; if "suite", ARGS are sets of tests (suite)
- `--test-path=TEST_PATH` Path to the test executables to run, currently only used for 'client' (none)
- `--mongod=MONGOD_EXECUTABLE` Path to mongod to run (/Users/mike/10gen/mongo/mongod)
- `--port=MONGOD_PORT` Port the mongod will bind to (32000)
- `--mongo=SHELL_EXECUTABLE` Path to mongo, for .js test files (/Users/mike/10gen/mongo/mongo)
- `--continue-on-failure` If supplied, continue testing even after a test fails
- `--from-file=FILE` Run tests/suites named in FILE, one test per line, '-' means stdin
- `--smoke-db-prefix=SMOKE_DB_PREFIX` Prefix to use for the mongods' dbpaths ('')
- `--small-oplog` Run tests with master/slave replication & use a small oplog

By default, `smoke.py` will run tests that create data in `/data/db`, which may interfere with other MongoDB instances you are running. To change the directory in which the smoke tests create databases, use `--smoke-db-prefix=/some/other/path`

To run specific tests, use the `--mode=files` option:

```bash
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
```

Build the unit tests (in src/mongo/unittest/ by running:

```
$ scons build/unittests.txt
/* build output */
Generating build/unittests.txt
   build/linux2/dd/durableDefaultOff/mongo/platform/atomic_word_test
   ...
   build/linux2/dd/durableDefaultOff/mongo/bson_template_evaluator_test
   build/linux2/dd/durableDefaultOff/mongo/balancer_policy_test
scons: done building targets.
```

Then use the line above to run the binary generated:

```
$ ./build/linux2/dd/durableDefaultOff/mongo/platform/atomic_word_test
```

### See Also

- [scons](#)

### Pairing Internals
Policy for reconciling divergent oplogs

pairing is deprecated

In a paired environment, a situation may arise in which each member of a pair has logged operations as master that have not been applied to the other server. In such a situation, the following procedure will be used to ensure consistency between the two servers:

1. The new master will scan through its own oplog from the point at which it last applied an operation from it's peer's oplog to the end. It will create a set C of object ids for which changes were made. It will create a set M of object ids for which only modifier changes were made. The values of C and M will be updated as client operations are applied to the new master.
2. The new master will iterate through its peer's oplog, applying only operations that will not affect an object having an id in C.
3. For any operation in the peer's oplog that may not be applied due to the constraint in the previous step, if the id of the 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=iasn(SKtm --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 haven’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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Appearance</th>
<th>Markup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object name (the type of &quot;object&quot; that &quot;object-oriented programming&quot; deals with)</td>
<td>monospace</td>
<td>{{term}}</td>
</tr>
<tr>
<td>short code fragment inline</td>
<td>monospace</td>
<td>{{term}}</td>
</tr>
<tr>
<td>file path/name, extension</td>
<td>italic</td>
<td><em>term</em></td>
</tr>
<tr>
<td>programming command, statement or expression</td>
<td>monospace</td>
<td>{{term}}</td>
</tr>
<tr>
<td>variable or &quot;replaceable item&quot;</td>
<td>monospace italic</td>
<td><em>term</em></td>
</tr>
<tr>
<td>Placeholders in paths, directories, or other text that would be italic anyway</td>
<td>angle brackets around &lt;item&gt;</td>
<td>&lt;item&gt;</td>
</tr>
<tr>
<td>GUI element (menus menu items, buttons)</td>
<td>bold</td>
<td><em>term</em></td>
</tr>
<tr>
<td>First instance of a technical term</td>
<td>italic</td>
<td><em>term</em></td>
</tr>
<tr>
<td>tag (in HTML or XML, for example)</td>
<td>monospace</td>
<td>{{term}}</td>
</tr>
<tr>
<td>Extended code sample</td>
<td>code block</td>
<td>(code) program code</td>
</tr>
</tbody>
</table>

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 [MyPage#GoHere], and a link to that anchor from a different page would look like [GoHere]. (See the sidebar for information about adding other text to the body of the link.)

**Special Characters**

- You will often need to insert code samples that contain curly braces. As Dwight has pointed out, Confluence gets confused by this unless you "escape" them by preceding them with a backslash, thusly:

```
\{  
\}
```

You must do the same for "\[", "\]", "\_" and some others. Within a `{code}` block you don't have to worry about this. If you are inserting code fragments inline using `{{` and `}}`, however, you still need to escape these characters. Further notes about this:

- If you are enclosing a complex code expression with `{{` and `}}`, do NOT leave a space between the last character of the expression and the `}}`. This confuses Confluence.
- Confluence also gets confused (at least sometimes) if you use `{{` and `}}`, to enclose a code sample that includes escaped curly brackets.

**About MongoDB’s Confluence wiki**

Confluence has this idea of "spaces". Each person has a private space, and there are also group spaces as well.

The MongoDB Confluence wiki has three group spaces defined currently:

- **Mongo Documentation** - The publicly accessible area for most Mongo documentation
- **Contributor** - Looks like, the publicly accessible space for information for "Contributors"
- **Private** - a space open to MongoDB developers, but not to the public at large.

As I said in my email on Friday, all of the (relevant) info from the old wiki now lives in the "Mongo Documentation".

**Standard elements of Wiki pages**

You shouldn't have to spend a lot of time worrying about this kind of thing, but I do have just a few suggestions:

- Since these wiki pages are (or can be) arranged hierarchically, you may have "landing pages" that do little more than list their child pages. I think Confluence actually adds a list of children automatically, but it only goes down to the next hierarchical level. To insert a hierarchical list of a page’s children, all you have to do is insert the following Confluence "macro":

```
{children:all=true}
```

See the Confluence documentation for more options and switches for this macro.

- For pages with actual text, I tried to follow these guidelines:
  - For top-level headings, I used "h2" not "h1"
  - I never began a page with a heading. I figured the title of the page served as one.
  - I always tried to include a "See Also" section that listed links to other Mongo docs.
  - I usually tried to include a link to the “Talk to us about Mongo” page.

**Community**

- Technical Support
- Bug/Feature Tracker (Jira)
- Blog
- Mailing List
- Events
- Job Board
- Twitter etc.
- Store
- Resources for Driver and Database Developers
  - Source
  - Developer List
  - Project Ideas
- Contribute!
  - Write a book
  - Write a driver, framework, and other tools
  - Help with Free Support
• Work on the DB

Technical Support
See the Support page, free support is available and its usage encouraged!

Bug/Feature Tracker (Jira)
File, track, and vote on bugs and feature requests. There is issue tracking for MongoDB and all supported drivers.

Blog
http://blog.mongodb.org/

Mailing List
http://groups.google.com/group/mongodb-announce - for release announcement and important bug fixes.

Events
The events page includes information about MongoDB conferences, webcasts, users groups, local meetups, and open office hours.

Job Board
• Click Here to access the Job Board. The Board is a community resource for all employers to post MongoDB-related jobs. Please feel free to post/investigate positions!
• See also the Indeed MongoDB jobs list

Twitter etc.
• @mongodb
• facebook
• linkedin

Store
• Visit the MongoDB store on Cafepress.

Resources for Driver and Database Developers

Source
The source code for the database and drivers is available at the http://github.com/mongodb.

Developer List
This mongodb-dev mailing list is for people developing drivers and tools, or who are contributing to the MongoDB codebase itself.

Project Ideas
Start or contribute to a MongoDB-related project.

Contribute!

Write a book
If interested contact info@10gen.com we'll try to get you in touch with publishers.

Write a driver, framework, and other tools
Writing Drivers and Tools

**Help with Free Support**

Jump in with answers on [http://groups.google.com/group/mongodb-user](http://groups.google.com/group/mongodb-user) and IRC (freenode.net#mongodb).

**Work on the DB**

[http://groups.google.com/group/mongodb-dev](http://groups.google.com/group/mongodb-dev)

**MongoDB Masters**

- Roman Alexis Anastasini
- Peter Bell
- Rick Copeland
- Justin Dearing
- Mike Dirolf
- Michael Fiedler
- Kenny Gorman
- Jonas Haag
- Nathen Harvey
- Jon Hoffman
- Takahiro Inoue
- Lennart Koopmann
- Lynn Langit
- Nat Luengnaruemitchai
- David Makogon
- Harry Marr
- David Mytton
- Gustavo Niemeyer
- John Nunemaker
- Niall O'Higgins
- Flavio Percoco
- Mitch Pirtle
- Karl Seguin
- Mark Smalley
- James Summerfield
- Tony Tam
- Rose Toomey
- Jonathan Wage
- Ian White
- Aristarkh Zagorodnikov

*Sorted alphabetically by last name*

**Roman Alexis Anastasini**

Roman is a Scala and C# Developer based in Beijing. Originally from Germany, he was formerly a Technical Manager at Gameforge in Kahlsrue Germany. He's now excited to encounter new adventures in Beijing.

- [@foliba](https://twitter.com/foliba) on Twitter

**Peter Bell**

Peter is Senior VP Engineering and Senior Fellow at General Assembly, a campus for technology, design, and entrepreneurship. He's a regular presenter at national and international conferences on ruby, nodejs, NoSQL (especially MongoDB and neo4j), cloud computing, software craftsmanship, java, groovy, javascript, and requirements and estimating. He is on the program committee for Code Generation in Cambridge, England and the Domain Specific Modeling workshop at SPLASH (was ooPSLA) and reviews and shepherds proposals for the BCS SPA conference.


**Github**
Rick Copeland

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

GitHub
@rick446 on Twitter
Blog
Presentations

MongoDB contributions

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

Justin Dearing

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

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

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

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

@Zippy1981 on Twitter
GitHub
Blog

Mike Dirolf

Mike was the original author of the PyMongo project and a maintainer of the mongo-ruby-driver. He co-authored O’Reilly’s MongoDB: The Definitive Guide. He maintains several MongoDB-related open source projects, and is the founder of a web service, Fiesta (https://fiesta.cc), that uses MongoDB as its primary data store.

@mrdirolf on Twitter
GitHub
Blog

MongoDB Contributions

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

Michael Fiedler

Mike is a long-time systems engineer, building a variety of platforms with all technologies. He has spoken at MongoDB conferences, worked with 10gen on systems-oriented features, and manages sharded clusters. He contributes to many open source projects and writes a few of his own. On the side he’s a roller derby referee and licensed skydiver.

MongoDB Contributions

- MongoDB-Chef Cookbook
- @mikefiedler on Twitter
- Github
- Blog

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

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

**Jonas Haag**

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

**MongoDB Contributions**
- Django MongoDB Engine
- PyMongo

**Nathen Harvey**

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

**MongoDB contributions**
- MongoDBDC User Group

**Jon Hoffman**

Jon is a software engineer and currently runs the infrastructure engineering team at foursquare. His team is tasked with designing the backends of new features, building robust systems to keep the platform running smoothly and helping other engineers get things done. In the course of his tenure at foursquare Jon has made contributions to the mongoDB’s core server, mongo-java-driver, Lift Framework, and scalaj-http projects. Jon recently presented work the foursquare infrastructure team built to add robustness to mongo on top of unreliable disk. Before foursquare Jon spent a brief time working at a three person startup, built distributed systems at Goldman Sachs, worked on VoIP apps for the telephone company, created a medical record app for palm pilot, and graduated from Carnegie Mellon.

**MongoDB Contributions**
- Core Server
- MongoDB Java Driver
- MongoDB in Production: Data and Availability, March NYC MongoDB User Group

**Takahiro Inoue**

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

**MongoDB Contributions**
Lennart Koopmann

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

@_Lennart on Twitter
GitHub
Blog

MongoDB Contributions

Graylog
Mongo Analyzer

Lynn Langit

Lynn Langit was a developer evangelist for the Microsoft MSDN team for the previous 4 years. Prior to working at Microsoft, she founded and served as lead architect of a development firm that created BI solutions. She holds a number of Microsoft certifications, including MCITP, MCSD, MCDBA, and MCT. Lynn left Microsoft to do consulting and training in October 2011. Lately she's authored and taught for DevelopMentor (SQL Server 2012 and Google App Engine) and she's created a series of talks around 'NoSQL for the RDBMS professional' and 'MongoDB for the .NET Developer'.

- @lynnlangit On Twitter
- [blog ]
- MongoDB YouTube playlist [http://www.youtube.com/playlist?list=PLCE5902FB1D284880&feature=view_all]

Nat Luengnaruemitchai

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

Github

MongoDB Contributions

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

David Makogon

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

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

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

Twitter
Blog
Presentations

Harry Marr

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

Github
@harrymarr
Blog

MongoDB Contributions

MongoEngine
MongoEngine source

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

Gustavo Niemeyer

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

John Nunemaker

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

Niall O’Higgins

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

Flavio Percoco

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

**Mitch Pirtle**

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

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

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

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

**Karl Seguin**

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

@KarlSeguin on Twitter
GitHub
Blog

**MongoDB Contributions**

Mongoly.com
The Little MongoDB Book

**Mark Smalley**

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

Twitter
GitHub
Blog

**MongoDB Contributions**

MongoBase
Geoply
MongoPress

**James Summerfield**

@jsummerfield on twitter
Github

James is a development manager at Rangespan where he divides his time between architecting, building and supporting large scale distributed systems using MongoDB and Hadoop. He founded Interview Zen to help find and hire great developers. He previously led software development projects at HM Treasury, where he designed systems to analyse economic recovery, and at UBS Investment Bank, where he developed real-time trade analytics. He has an MA in Computer Science from Oxford University. James enjoys presenting his work with MongoDB, most recently at the MongoDB UK conference and a number of London MongoDB User Groups.

**Tony Tam**

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

Rose Toomey

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

Jonathan Wage

Software engineer from Nashville, TN currently working for OpenSky.com

@Jwage on Twitter

GitHub

Blog

Ian White

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

@EonWhite on Twitter

GitHub

Blog

Aristarkh Zagorodnikov

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

GitHub

BitBucket

Blog

Technical Support

- Free Support Forum - http://groups.google.com/group/mongodb-user
**MongoDB Commercial Services Providers**

**10gen**

10gen is the initiator, contributor and continual sponsor of the MongoDB project. 10gen offers subscriptions that include production support, commercial licensing, and MongoDB Monitoring Service. In addition, 10gen offers advisory consulting and training.

---

<table>
<thead>
<tr>
<th>Company</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIGNEX Datamatics</td>
<td>(a subsidiary of Datamatics Global Services Ltd.) is the global leader in Commercial Open Source enterprise solutions and a global partner of 10gen (MongoDB) offering advisory consulting, implementation, and other services around the MongoDB application.</td>
</tr>
<tr>
<td>comSysto</td>
<td>is a Munich based software company specialized in lean business and technology development. While supporting all three steps of a well known Build-Measure-Learn lean feedback loop, comSysto focuses on open source frameworks and software as major enablers of short, agile Build-Measure-Learn iterations and fast gains in validated learning. Powerful MongoDB technology provides the needed flexibility and agility for turning ideas into products as well as performance for handling Big Data while turning data into knowledge. We also enjoy developing with Spring framework and its sub-projects, Apache Wicket, Gradle, Git, Oracle DB and Oracle BI. comSysto is dedicated to eliminating waste in both business and technology since 2005.</td>
</tr>
<tr>
<td>codecentric AG</td>
<td>specialises in developing customised IT-solutions. The company is one of the leading German providers in the areas of agility, architecture, Java performance, Java and Enterprise Content Management. codecentric offers development, IT-consulting and services throughout the complete life cycle of applications and infrastructures: from individual software solutions to performance optimisation of Java-applications, to support of organisational processes within the company. The more than 120 codecentric employees work at various locations in Germany and Europe.</td>
</tr>
<tr>
<td>EPAM’s</td>
<td>core competencies include complex software product engineering for leading global software and technology vendors as well as development, testing, maintenance, and support of mission critical business applications and vertically oriented IT consulting services for global Fortune 2000 corporations.</td>
</tr>
</tbody>
</table>
Since its inception in 2005, Equal Experts has been providing bespoke application development services to blue chip clients, including major investment banks and leading Telecom companies. Equal Experts has defined a unique approach to Agile software development that is both pragmatic and Enterprise-friendly. By harnessing the potential of modern application development techniques, our goal is to radically improve our customer’s efficiency.

 iTechArt Group Since 2003 iTechArt Group, a leading consulting firm, has been providing R&D Services for emerging technology companies with heavy focus on web and mobile development. iTechArt offers a traditional IT outsourcing model complete with project-based services giving our clients the ability to build and manage their own dedicated team in one of our development centers in Eastern Europe. iTechArt specializes in product research and conceptualization, helping our clients select the correct architecture design & platform, implementation & customization, testing & QA, product installation, migration & porting, as well as modernization & product extension.

NewWave Telecom & Technologies Inc. is SEI CMMI® Maturity Level 2 appraised; a GSA Schedule 70 IT Service Provider and 8(a) STARS II holder, service Information Technology and Business Services firm who is a leading implementer of business solutions using service-oriented technologies. We specialize in providing end-to-end, mission critical information technology solutions for healthcare, finance, telecoms, government agencies and private sector clients.

OpenCredo is a team of highly experienced consultants who are amongst the best in the business. Dedicated to maximizing the ability to deliver value from any enterprise software development and delivery capability. They help their clients use leading-edge technologies and practices to drive waste from their software delivery projects.

PalominoDB is a boutique consultancy, known for being top MySQL experts, and specializing in open source DB technologies including MongoDB, MySQL, Postgres, Cassandra and HBase.

Thermopylae Sciences + Technology (TST) is a Service-Disabled Veteran-Owned Small Business (SDVOSB) that focuses on excellence in all we do for our government and private sector customer base. We particularly excel in software development, cyber security, geospatial engineering, cloud/grid computing in cleared environments, and mobile application development.

Thumbtack is a full service application development partner specializing in highly scalable and cloud-based solutions. We provide strategic advice, technical architecture, software development, as well as quality assurance and deployment support to companies in Media, Publishing, and Financial Services. With a core competency in NoSQL, Thumbtack has brought robust MongoDB solutions to production for a variety of clients and industries.

Rothbury Software is a very experienced and successful boutique software consulting company. Our impressive customer list says more about us than we can state here.
<table>
<thead>
<tr>
<th>Company</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xebia</td>
<td>Xebia is an international IT consultancy and project organization focused on Enterprise Java technology, Agile development methods and outsourcing services. Xebia consists of over 250 professionals, all committed to be the best in their field of expertise. Passion for in depth technology, in combination with Lean, Agile and Scrum practices are Xebia’s driving factors and competitive edge.</td>
</tr>
<tr>
<td>FastConnect</td>
<td>FastConnect is a french consulting and engineering company specialised in distributed architectures. Our expertise spans the domains of SOA and EDA, Process and Decision Management, Cloud, Big Data and Analytics for business-critical applications. We bring to our clients our significant and strategic investments in specific technologies and development methodologies. Our consultants are helping our clients in designing and delivering flexible and linearly performing applications at the best price/performance ratio using state of the art architecture and enterprise-ready tools. For more information visit <a href="http://www.fastconnect.fr/">http://www.fastconnect.fr/</a>.</td>
</tr>
<tr>
<td>Silpion IT-Solutions GmbH</td>
<td>Silpion IT-Solutions GmbH is an expanding consultancy in Germany. Founded in 2000, today 97 highly qualified employees work for us. Of these, 57 are employed directly and 40 as freelancers. Our focus is safety and functionality, while we focus on innovative concepts and advanced technologies.</td>
</tr>
<tr>
<td>Orange11</td>
<td>Orange11 is a leading full service supplier of high-quality custom-built applications. Our specialist teams provide end-to-end project delivery services to support the full project life-cycle: from the first design stages to ongoing maintenance. In addition, Orange11 can provide high-end consulting and training services.</td>
</tr>
<tr>
<td>KMMX</td>
<td>KMMX Located in Mexico City, KMMX provides innovative training on web and mobile technologies. KMMX’s mission is to bring the best tools and learning environment to meet the needs of Mexico’s IT professionals.</td>
</tr>
<tr>
<td>Rivet Logic</td>
<td>Rivet Logic is an award-winning consulting and systems integration firm that helps organizations better engage with customers, improve collaboration and streamline business operations. Through a full suite of solutions for content management, collaboration and community, Rivet Logic enables organizations to fully leverage the power of industry-leading open source software. Rivet Logic-Artisans of Open Source. Visit <a href="http://www.rivetlogic.com">www.rivetlogic.com</a>.</td>
</tr>
<tr>
<td>RadiantBlue Technologies</td>
<td>RadiantBlue Technologies is a small business offering specialized information technology solutions, consulting, and program support services for national security, counter-terrorism, and other federal government customers. RadiantBlue has offices in Chantilly, VA, Colorado Springs, CO, and Melbourne Beach, FL. Areas of expertise include: open source software support and design, geospatial software, modeling and simulation and signal processing.</td>
</tr>
<tr>
<td>El Taller Web</td>
<td>El Taller Web is a knowledge center focused on laying the necessary foundations to increase innovation, awareness and the use of web related technologies, leveraging that into IT solutions for both our clients and our community. For our customers, our certified staff works on helping them obtain solutions that deliver the most cost-effective value for their needs, reducing TCO and time to market, and increasing ROI. For our community, we strive to give back by sharing what we’ve learned in order to advance individual and collective knowledge in our midst.</td>
</tr>
</tbody>
</table>
Circar Consulting is a global leader in Open Source product development, web application development, SaaS product development and mobile apps development. We are big on Big data and a global partner of 10gen (MongoDB) offering advisory consulting, implementation, and other services around the MongoDB application. Headquartered in the heart of Silicon Valley, United States, Circar Consulting has two development centers in USA and India. We provide services to companies of all sizes, from Fortune 500 to Silicon Valley start ups.

Other Resources

- Hashrocket is a full-service design and development firm that builds successful web businesses. Hashrocket continually creates and follows best practices and surround themselves with passionate and talented craftsmen to ensure the best results for you and your business.
- LightCube Solutions provides PHP development and consulting services, as well as a lightweight PHP framework designed for MongoDB called 'photon'.
- Squeejee builds web applications on top of MongoDB with multiple sites already in production.

Analytics and BI Tools

- Business Intelligence

Visit the 10gen Offices

Bay Area

10gen’s West Coast office is located in downtown Palo Alto.

555 University Ave.
Palo Alto, CA 94301

Directions:

From 101:

- Take 101 to the University Ave Exit.
- Drive West to downtown Palo Alto. The 10gen office is at the northeast corner of the intersection of Tasso St. and University Ave.

From 280:

- Driving southbound, take 280 to the Sand Hill Road exit. Driving northbound, take 280 to the Alpine Road exit; make a right onto Sand Hill Road from Alpine Road.
- Take Sand Hill Road east to the end at El Camino Real
- Make a right onto El Camino Real, then veer right towards University Ave.
- Make a left onto University Ave. The 10gen office is at the northeast corner of the intersection of Tasso St. and University Ave.

New York City

10gen’s East Coast office is located in the SOHO neighborhood of NYC.

578 Broadway
7th Floor
New York, NY 10012

User Feedback
"I just have to get my head around that mongodb is really _this_ good"
  - muckster, #mongodb

"Guys at Redmond should get a long course from you about what is the software development and support 😞*
  - kunthar@gmail.com, mongodb-user list

"#mongoDB keep me up all night. I think I have found the 'perfect' storage for my app 😊".
  - elpargo, Twitter

"Dude, you guys are legends!"
  - Stii, mongodb-user list

"Times I've been wowed using MongoDB this week: 7."
  - tpitale, Twitter

**Community Blog Posts**

B is for Billion
  - Wordnik (July 9, 2010)

[Reflections on MongoDB]
  - Brandon Keepers, Collective Idea (June 15, 2010)

Building a Better Submission Form

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
Philosophy

Design Philosophy

- New database technologies are needed to facilitate horizontal scaling of the data layer, easier development, and the ability to store order(s) of magnitude more data than was used in the past.

- A non-relational approach is the best path to database solutions which scale horizontally to many machines.

- It is unacceptable if these new technologies make writing applications harder. Writing code should be faster, easier, and more agile.

- The document data model (JSON/BSON) is easy to code to, easy to manage(schemaless), and yields excellent performance by grouping relevant data together internally.

- It is important to keep deep functionality to keep programming fast and simple. While some things must be left out, keep as much as possible – for example secondaries indexes, unique key constraints, atomic operations, multi-document updates.

- Database technology should run anywhere, being available both for running on your own servers or VMs, and also as a cloud pay-for-what-you-use service.
Focus

MongoDB focuses on four main things: flexibility, power, speed, and ease of use. To that end, it sometimes sacrifices things like fine grained control and tuning, overly powerful functionality like MVCC that require a lot of complicated code and logic in the application layer, and certain ACID features like multi-document transactions.

Flexibility

MongoDB stores data in JSON documents (which we serialize to BSON). JSON provides us a rich data model that seamlessly maps to native programming language types, and since its schema-less, makes it much easier to evolve your data model than with a system with enforced schemas such as a RDBMS.

Power

MongoDB provides a lot of the features of a traditional RDBMS such as secondary indexes, dynamic queries, sorting, rich updates, upserts (update if document exists, insert if it doesn't), and easy aggregation. This gives you the breadth of functionality that you are used to from an RDBMS, with the flexibility and scaling capability that the non-relational model allows.

Speed/Scaling

By keeping related data together in documents, queries can be much faster than in a relational database where related data is separated into multiple tables and then needs to be joined later. MongoDB also makes it easy to scale out your database. Autosharding allows you to scale your cluster linearly by adding more machines. It is possible to increase capacity without any downtime, which is very important on the web when load can increase suddenly and bringing down the website for extended maintenance can cost your business large amounts of revenue.

Ease of use

MongoDB works hard to be very easy to install, configure, maintain, and use. To this end, MongoDB provides few configuration options, and instead tries to automatically do the "right thing" whenever possible. This means that MongoDB works right out of the box, and you can dive right into developing your application, instead of spending a lot of time fine-tuning obscure database configurations.

See also:

- Introduction

Use Cases

- Why do people use MongoDB?
- When should you consider using MongoDB?
- When should you use something else?
  - If you had to pick one thing you wouldn't use it for, what's the first thing that comes to mind?
- Do big companies use MongoDB?
- How may databases should my organization standardize on? Is one-size-fits-all over?
- Common Use Cases: Articles and Videos

The goal of MongoDB is to be a fairly general purpose tool that can be used for a variety of applications. Common uses include online/operational tasks and applications, as well as select business intelligence / analytics use cases.

The Production Deployments page provides hundreds of examples of real world use cases; see also the use case docs in the MongoDB Manual.
Why do people use MongoDB?

Two reasons: easy scale-out, and coding velocity ("agility").

The original catalyst for the NoSQL space is the growth of "big data" and the need for scalable databases to store that data. However, a secondary but often equally important factor in MongoDB adoption is increased software development "agility."

MongoDB developers report big productivity gains writing their apps, and often use Mongo regardless of whether or not scaling out is required. While there are MongoDB systems that exceed one million operations per second, MongoDB is also used for many apps that easily run on a single server.

MongoDB makes app development faster because it

1. eliminates object-relational-mapping work and the so-called "impedance mismatch". (Mongo uses JSON, which maps well to object-style data.)
2. allows dynamic schemas ("schemaless" operation), which is synergistic with agile software development methodologies, which themselves hugely speed project completion.
3. makes it much much easier for the developer to store and manipulate complex data and polymorphic data.
4. reduces the amount of work required to scale out the application and increase system speed.

When should you consider using MongoDB?

- You find yourself coding around database performance issues – for example adding lots of caching.
- You are storing data in flat files.
- You are batch processing yet you need real-time.
- You are doing agile development, for example, Scrum.
- Your data is complex to model in a relational db. For example a complex derivative security might be hard to store in a traditional format. Electronic health records is another example. If you were considering using an XML store, that's a strong sign to consider MongoDB and its use of JSON/BSON.
- Your project is very late, and it is database intensive.
- You have been forced to use expensive SANs, proprietary servers, or proprietary networks for your existing database solution.
- You are deploying to a public or private cloud.

When should you use something else?

The following problem domains are not a good match for MongoDB.

- Problems requiring SQL. MongoDB supports ad hoc queries and has its own query language, but does not support SQL. This is likely a show stopper for many legacy apps until they are due for a refresh.
- 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 – for example many users have built e-commerce systems using MongoDB.
- Traditional Non-Realtime Data Warehousing (sometimes). Traditional relational data warehouses and variants (columnar relational) are well suited for certain business intelligence problems – especially if you need SQL for your client tool (e.g. MicroStrategy). Exceptions where MongoDB is good are:
  - cases where the analytics are realtime
  - cases where the data very complicated to model in relational
• when the data volume is huge
• when the source data is already in a mongo database

The new aggregation framework (v2.2+) improves MongoDB's reporting capabilities, as well as availability of a Hadoop adapter. MongoDB support is being added to more client analytic tools as time passes (e.g., Jaspersoft and Pentaho now have MongoDB support).

If you had to pick one thing you wouldn't use it for, what's the first thing that comes to mind?

A double-entry bookkeeping accounting system is probably the perfect anti-example.

• The intrinsically tabular application data maps well to relational data models
• The application requires very complex transactions at times
• The application requires a good amount of reporting. Relational and SQL are quite good at reporting
• The volume of data is probably relatively small

That said, one Fortune 500 company uses MongoDB as a base for a complex application that reconciles billions of dollars in revenues – so don't project too far off the above example.

Do big companies use MongoDB?

Yes. Quite a bit actually, including some companies with over $100B in annual revenues.

How may databases should my organization standardize on? Is one-size-fits-all over?

Historically there is already some diversity: most large enterprises use an RDBMS for OLTP (e.g., Oracle), a data warehouse technology (e.g., Netezza), and some niche tools for special problems (e.g. a time series db).

Today we are seeing organizations add one more db tool to their toolbox: a NoSQL database.

We are also seeing some organizations (for example, one very large UK media company) adopt a Mongo First policy. That is to say, MongoDB is their default for building new applications. They use other tools, but by default, MongoDB. This is a good example that MongoDB is fairly general purpose. Note the word "application": they might at times use other things for pure reporting. Other companies are beginning to do this too; obviously these organizations have already done a few projects with MongoDB and are quite comfortable with it at this point.

Oh, and if you are a startup, just use MongoDB. :-)
analytics to MongoDB. Specifically, Pentaho’s tools provide a complete business analytics suite, visual tools to explore data in MongoDB, quick insights and reporting from MongoDB data and integration into enterprise-level analytics solutions.

To get started,

- Visit the Pentaho for MongoDB resource center.
- Once you’ve got MongoDB setup and running, download and install either the Pentaho Business Analytics Enterprise Edition or the Pentaho Kettle Community Edition.
- Next, review the Pentaho Big Data toolset and then learn about how to integrate MongoDB and Pentaho’s tools.

Jaspersoft

Jaspersoft develops business intelligence tools and integration tools for data from several sources. Jaspersoft’s intelligent connector integrates MongoDB with the full Jaspersoft BI Suite, providing flexible and affordable reporting, ad hoc analysis, and dashboarding of MongoDB data. As a result, both business and technical users can visualize and analyze MongoDB data more rapidly for competitive advantage. JasperReports Server supports native connections to MongoDB, both on-premises and in the cloud.

- First, install and setup MongoDB
- Next you can download and install JasperReports Server (Enterprise or Community) to create a centralized reporting infrastructure.
- To design reports download the iReport Report Designer.
- To get started, visit Simple Reporting on MongoDB for steps on using iReport to design reports with data from MongoDB.

Hadoop Quick Start

- Prerequisites
  - Hadoop
  - MongoDB
  - Miscellaneous
- Building MongoDB Adapter
- Examples
  - Load Sample Data
  - Treasury Yield
  - UFO Sightings

MongoDB and Hadoop are a powerful combination and can be used together to deliver complex analytics and data processing for data stored in MongoDB. The following guide shows how you can start working with the MongoDB-Hadoop adapter. Once you become familiar with the adapter, you can use it to pull your MongoDB data into Hadoop Map-Reduce jobs, process the data and return results back to a MongoDB collection.

Prerequisites

Hadoop

In order to use the following guide, you should already have Hadoop up and running. This can range from a deployed cluster containing multiple nodes or a single node pseudo-distributed Hadoop installation running locally. As long as you are able to run any of the examples on your Hadoop installation, you should be all set. The following versions of Hadoop are currently supported:

- 0.20/0.20.x
- 1.0/1.0.x
- 0.21/0.21.x
- CDH3
- CDH4

MongoDB

The latest version of MongoDB should be installed and running. In addition, the MongoDB commands should be in your $PATH.

Miscellaneous

In addition to Hadoop, you should also have git and JDK 1.6 installed.

Building MongoDB Adapter

The MongoDB-Hadoop adapter source is available on github. First, clone the repository and get the release-1.0 branch:

```
$ git clone https://github.com/mongodb/mongo-hadoop.git
$ git checkout release-1.0
```

Now, edit build.sbt and update the build target in hadoopRelease in ThisBuild. In this example, we’re using the CDH3 Hadoop
distribution from Cloudera so I’ll set it as follows:

```scala
hadoopRelease in ThisBuild := "cdh3"
```

To build the adapter, use the self-bootstrapping version of `sbt` that ships with the MongoDB-Hadoop adapter:

```
$ ./sbt
```

Once the adapter is built, you will need to copy it and the latest stable version of the MongoDB Java driver to your `$HADOOP_HOME/lib` directory. For example, if you have Hadoop installed in `/usr/lib/hadoop`:

```
$ wget --no-check-certificate https://github.com/downloads/mongodb/mongo-java-driver/mongo-2.7.3.jar
$ cp mongo-2.7.3.jar /usr/lib/hadoop/lib/
$ cp core/target/mongo-hadoop-core_cdh3u3-1.0.0.jar /usr/lib/hadoop/lib/
```

**Examples**

**Load Sample Data**

The MongoDB-Hadoop adapter ships with a few examples of how to use the adapter in your own setup. In this guide, we’ll focus on the UFO Sightings and Treasury Yield examples. To get started, first load the sample data for these examples:

```
$ ./sbt load-sample-data
```

To confirm that the sample data was loaded, start the mongo client and look for the `mongo_hadoop` database and be sure that it contains the `ufo_sightings.in` and `yield_historical.in` collections:

```
$ mongo
MongoDB shell version: 2.0.5
connecting to: test
> show dbs
mongo_hadoop 0.453125GB
> use mongo_hadoop
switched to db mongo_hadoop
> show collections
system.indexes
ufo_sightings.in
yield_historical.in
```

**Treasury Yield**

To build the Treasury Yield example, we’ll need to first edit one of the configuration files used by the example code and set the MongoDB location for the input (`mongo.input.uri`) and output (`mongo.output.uri`) collections (in this example, Hadoop is running on a single node alongside MongoDB):

```
```

```
Next, edit the main class that we’ll use for our MapReduce job (TreasuryYieldXMLConfig.java) and update the class definition as follows:

```java
public class TreasuryYieldXMLConfig extends MongoTool {

    static {
        // Load the XML config defined in hadoop-local.xml
        Configuration.addDefaultResource( "hadoop-local.xml" );
        Configuration.addDefaultResource( "mongo-defaults.xml" );
        Configuration.addDefaultResource( "mongo-treasury_yield.xml" );
    }

    public static void main( final String[] pArgs ) throws Exception{
        System.exit( ToolRunner.run( new TreasuryYieldXMLConfig(), pArgs ) );
    }
}
```

Now let's build the Treasury Yield example:

```
$ ./sbt treasury-example/package
```

Once the example is done building we can submit our MapReduce job:

```
$ hadoop jar examples/treasury_yield/target/treasury-example_cdh3u3-1.0.0.jar com.mongodb.hadoop.examples.treasury.TreasuryYieldXMLConfig
```

This job should only take a few moments as it's a relatively small amount of data. Now check the output collection data in MongoDB to confirm that the MapReduce job was successful:
UFO Sightings

This will follow much of the same process as with the Treasury Yield example with one extra step; we'll need to add an entry into the build file to compile this example. First, open the file for editing:

```shell
$ emacs project/MongoHadoopBuild.scala
```

Next, add the following lines starting at line 72 in the build file:

```scala
lazy val ufoExample = Project( id = "ufo-sightings",
    base = file("examples/ufo_sightings"),
    settings = exampleSettings ) dependsOn ( core )
```

Now edit the UFO Sightings config file and update the `mongo.input.uri` and `mongo.output.uri` properties:

```xml
$ emacs examples/ufo_sightings/src/main/resources/mongo-ufo_sightings.xml
```

```xml
...<property>
  <!-- If you are reading from mongo, the URI -->
  <name>mongo.input.uri</name>
  <value>mongodb://127.0.0.1/mongo_hadoop.ufo_sightings.in</value>
</property>
<!-- If you are writing to mongo, the URI -->
<name>mongo.output.uri</name>
  <value>mongodb://127.0.0.1/mongo_hadoop.ufo_sightings.out</value>
</property>
...
Next edit the main class for the MapReduce job in `UfoSightingsXMLConfig.java` to use the configuration file:

```java
public class UfoSightingsXMLConfig extends MongoTool {
    static {
        // Load the XML config defined in hadoop-local.xml
        Configuration.addDefaultResource( "hadoop-local.xml" );
        Configuration.addDefaultResource( "mongo-defaults.xml" );
        Configuration.addDefaultResource( "mongo-ufo_sightings.xml" );
    }

    public static void main( final String[] pArgs ) throws Exception{
        System.exit( ToolRunner.run( new UfoSightingsXMLConfig(), pArgs ) );
    }
}
```

Now build the UFO Sightings example:

```bash
$ ./sbt ufo-sightings/package
```

Once the example is built, execute the MapReduce job:

```bash
$ hadoop jar examples/ufo_sightings/target/ufo-sightings_cdh3u3-1.0.0.jar com.mongodb.hadoop.examples.UfoSightingsXMLConfig
```

This MapReduce job will take just a bit longer than the Treasury Yield example. Once it's complete, check the output collection in MongoDB to see that the job was successful:

```bash
$ mongo
MongoDB shell version: 2.0.5
connecting to: test
> use mongo_hadoop
switched to db mongo_hadoop
> db.ufo_sightings.out.find().count()
21850
```

### Hadoop Scenarios

- **Batch Aggregation**
- **Data Warehouse**
- **ETL Data**

The following are some example deployments with MongoDB and Hadoop. The goal is to provide a high-level description of how MongoDB and Hadoop can fit together in a typical Big Data stack. In each of the following examples MongoDB is used as the "operational" real-time data store and Hadoop is used for offline batch data processing and analysis.

#### Batch Aggregation

In several scenarios the built-in aggregation functionality provided by MongoDB is sufficient for analyzing your data. However in certain cases, significantly more complex data aggregation may be necessary. This is where Hadoop can provide a powerful framework for complex analytics.

In this scenario data is pulled from MongoDB and processed within Hadoop via one or more Map-Reduce jobs. Data may also be brought in from additional sources within these Map-Reduce jobs to develop a multi-datasource solution. Output from these Map-Reduce jobs can then be written back to MongoDB for later querying and ad-hoc analysis. Applications built on top of MongoDB can now use the information from the batch analytics to present to the end user or to drive other downstream features.
Data Warehouse

In a typical production scenario, your application's data may live in multiple datastores, each with their own query language and functionality. To reduce complexity in these scenarios, Hadoop can be used as a data warehouse and act as a centralized repository for data from the various sources.

In this situation, you could have periodic Map-Reduce jobs that load data from MongoDB into Hadoop. This could be in the form of "daily" or "weekly" data loads pulled from MongoDB via Map-Reduce. Once the data from MongoDB is available from within Hadoop, and data from other sources are also available, the larger dataset data can be queried against. Data analysts now have the option of using either Map-Reduce or Pig to create jobs that query the larger datasets that incorporate data from MongoDB.

ETL Data

MongoDB may be the operational datastore for your application but there may also be other datastores that are holding your organization's data. In this scenario it is useful to be able to move data from one datastore to another, either from your application's data to another database or vice versa. Moving the data is much more complex than simply piping it from one mechanism to another, which is where Hadoop can be used.

In this scenario, Map-Reduce jobs are used to extract, transform and load data from one store to another. Hadoop can act as a complex ETL mechanism to migrate data in various forms via one or more Map-Reduce jobs that pull the data from one store, apply multiple transformations (applying new data layouts or other aggregation) and loading the data to another store. This approach can be used to move data from or to MongoDB, depending on the desired result.
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:
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,
In this case the friendIDs field is an array with a list of IDs corresponding to profiles of users that are my friends. This data can then be used to generate personalized feeds based on content that my friends have viewed or liked. IGN’s social network, MY IGN, uses MongoDB to store profiles of users and generate personalized fields. Users have the ability to import their friends from facebook, ‘follow’ IGN authors, or follow specific game titles they are interested in. When they log in, they are presented with a personalized feed composed from this data.

**How-to guides:**

- Storing user profiles with MongoDB (coming soon)
- Importing social graphs into MongoDB (coming soon)
- Generating personalized feeds with MongoDB (coming soon)

**MongoDB for Mobile/Tablet Apps**

Serving similar content across desktop browsers, mobile browsers, as well as mobile apps is driving developers to build standardized API layers that can be accessed by traditional web application servers, mobile client applications, as well as 3rd party applications. Typically these are RESTful APIs that serve JSON data. With MongoDB’s JSON-like data format, building these APIs on top of MongoDB is simplified as minimal code is necessary to translate MongoDB documents and query to JSON representation. Additionally, features such as in-built two-dimensional geospatial indexing allow developers to easily incorporate location-based functionality into their applications.

**MongoDB for Data-driven journalism**

One of the strengths of MongoDB is dealing with semi-structured data. Data sources such as those produced by governments and other organizations are often denormalized and distributed in formats like CSV files. MongoDB, with its schema-free JSON-like documents is an ideal store for processing and storing these sparse datasets.

The Chicago Tribune uses MongoDB in its Illinois School Report Cards application, which is generated from a nearly 9,000 column denormalized database dump produced annually by the State Board of Education. The application allows readers to search by school name, city, county, or district and to view demographic, economic, and performance data for both schools and districts.

**How-to guides:**

- Importing and Exporting data from MongoDB (coming soon)
- Reporting and Visualization with MongoDB (coming soon)

**Presentations**

- How MTV Networks leverages MongoDB for CMS - MongoNYC Presentation (June 2011)
- Schema Design for Content Management: eHow on MongoDB - MongoSF Presentation (May 2011)
- More Presentations

**Use Case - Session Objects**

MongoDB is a good tool for storing HTTP session objects.

One implementation model is to have a sessions collection, and store the session object’s _id value in a browser cookie.

With its update-in-place design and general optimization to make updates fast, the database is efficient at receiving an update to the session object on every single app server page view.

**Aging Out Old Sessions**

The best way to age out old sessions is to use the auto-LRU facility of capped collections. The one complication is that objects in capped collections may not grow beyond their initial allocation size. To handle this, we can "pre-pad" the objects to some maximum size on initial addition, and then on further updates we are fine if we do not go above the limit. The following mongo shell javascript example demonstrates padding.

(Note: a clean padding mechanism should be added to the db so the steps below are not necessary.)
```javascript
> 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" : "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" }
> db.sessions.save(s1)
> s
{"info" : "example" , "padding" : "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" , "_id" : ObjectId("4aaf74a5761d147677233b0") }
> // when updating later
> s1 = db.sessions.find( { _id : ObjectId("4aaf74a5761d147677233b0") } )
{"_id" : ObjectId("4aaf74a5761d147677233b0") , "info" : "example", "padding" : "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" }
> delete s._padding;
true
> s.x = 3; // add a new field
3
> db.sessions.save(s);
> s
{"_id" : ObjectId("4aaf5a25761d147677233af") , "info" : "example", "x" : 3}
```

---

**MongoDB-Based Applications**

Please list applications that leverage MongoDB here. If you're using MongoDB for your application, we'd love to list you here! Email meghan@10gen.com.

**See Also**
- Production Deployments - Companies and Sites using MongoDB
- Hosting Center

**Applications Using MongoDB**

**c5t**

Content-management using TurboGears and Mongo

**Calipso**

Content management system built using NodeJS and MongoDB

**Cube**

Cube is an open-source system for visualizing time series data, built on MongoDB, Node and D3.

**ErrorApp**

ErrorApp tracks errors from your apps. It reports them to you and gathers all information and make reports available to you.

**Forward**

A full-featured, developer centric open source e-commerce platform that makes custom code easy, with powerful templates & expressive syntax.

**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.

Locomotive
Locomotive is an open source CMS for Rails. It's flexible and integrates with Heroku and Amazon S3.

Mogade
Mogade offers a free and simple to use leaderboard and achievement services for mobile game developers.

MongoLantern
MongoLantern is an open source full text search server using MongoDB as index storage, which allows MongoLantern to migrate any changes very easily into account using MongoDB API. It's written originally written in PHP can be migrated to any desired language as required using it's future APIs.

MongoPress
A flexible CMS that uses MongoDB and PHP.

Mongs
A simple, web-based data browser for MongoDB.

Mongeez
Mongeez is an opensource solution allowing you to manage your mongo document changes in a manner that is easy to synchronize with your code changes. Check out mongeez.org.

NewsBlur
NewsBlur is an open source visual feed reader that powers http://newsblur.com. NewsBlur is built with Django, MongoDB, Postgres and RabbitMQ.

phpMyEngine
A free, open source CMS licensed under the GPL v.3. For storage, the default database is MongoDB.

Quantum GIS
Plugin for Quantum GIS that lets you plot geographical data stored in MongoDB.

Scribe
Open source image transcription tool.

Shapado
Free and open source Q&A software, open source stackoverflow style app written in ruby, rails, mongomapper and mongodb.

Thundergrid
Thundergrid is a simple framework written in PHP that allows you to store large files in your Mongo database in seconds.

Websko
Websko is a content management system designed for individual Web developers and cooperative teams.
Events

- Upcoming Meetups and Conferences
- Webinars
- MongoDB User Groups
  - Training
    - "Office Hours"
- See Also

There are MongoDB User Groups (MUGs) all over the world. Please check out the full listing.

### Upcoming Meetups and Conferences

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
<th>What/Where</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 18, 2012</td>
<td><strong>MongoDB for the Relational DBA</strong>&lt;br&gt;Justin Dearing&lt;br&gt;NJ SQL Server User Group</td>
<td>New Jersey, USA</td>
</tr>
<tr>
<td>September 19, 2012</td>
<td><strong>Orlando PHP, Derick Rethans on MongoDB</strong>&lt;br&gt;Derick Rethans, 10gen&lt;br&gt;Orlando PHP</td>
<td>Orlando, FL</td>
</tr>
<tr>
<td>September 20, 2012</td>
<td><strong>MongoDB and PHP</strong>&lt;br&gt;Derick Rethans, 10gen&lt;br&gt;Jacksonville PHP</td>
<td>Jacksonville, FL, USA</td>
</tr>
<tr>
<td>September 21, 2012</td>
<td><strong>MongoDB and PHP</strong>&lt;br&gt;Derick Rethans&lt;br&gt;10gen&lt;br&gt;Jacksonville PHP</td>
<td>Atlanta, GA</td>
</tr>
<tr>
<td>Sept 24, 2012</td>
<td><strong>MongoDB and PHP</strong>&lt;br&gt;Derick Rethans, 10gen&lt;br&gt;Nashville PHP</td>
<td>Nashville, TN, USA</td>
</tr>
<tr>
<td>Sept 24, 2012</td>
<td><strong>MongoDB Labs at DataWeek</strong></td>
<td>San Francisco, CA</td>
</tr>
<tr>
<td>Oct 1, 2012</td>
<td>Choosing a fault-tolerant database to support cloud infrastructure&lt;br&gt;Alexander Hardy, Nimbula&lt;br&gt;YAC</td>
<td>Russia</td>
</tr>
<tr>
<td>October 14-17, 2012</td>
<td><strong>WebTech Conference</strong>&lt;br&gt;Store a Country in Your Database&lt;br&gt;Derick Rethans&lt;br&gt;10gen</td>
<td>Rheingoldhalle Mainz, DE</td>
</tr>
<tr>
<td>November 5-9, 2012</td>
<td><strong>Building Hybrid Applications with MongoDB and RDBMS</strong>&lt;br&gt;Chris Harris&lt;br&gt;10gen</td>
<td>Malmö, Sweden</td>
</tr>
<tr>
<td>November 12-17, 2012</td>
<td><strong>PyCon Argentina</strong>&lt;br&gt;Event of Python Community in Argentina. They are expected 500 attendances. We will send to talk, one of MongoDB evangelist in Latin America, Thiago Avelino&lt;br&gt;10gen</td>
<td>Buenos Aires, Argentina</td>
</tr>
</tbody>
</table>

### Webinars

<table>
<thead>
<tr>
<th>Webinar Date</th>
<th>Sessions</th>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 20, 2012</td>
<td>1:00PM EDT/10:00AM PDT and 5:00PM UTC</td>
<td><strong>Online Conference - What’s new in MongoDB Version 2.2</strong></td>
<td>MongoDB Version 2.2 includes a wide array of feature enhancements and performance improvements. This includes major improvements to concurrency as well as several new features such as the aggregation framework, TTL collections, and tag aware sharding. In this free, online conference, we will introduce many of the key updates to the core server and talk about updating your deployment.</td>
</tr>
<tr>
<td>Date</td>
<td>Time</td>
<td>Title</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>September 25, 2012</td>
<td>11:00AM EDT/8:00AM PDT/3:00PM UTC and 1:00PM EDT/10:00AM PDT/5:00PM UTC</td>
<td><strong>Introduction to MongoDB and Ruby</strong></td>
<td>This webinar explores MongoDB in the Ruby ecosystem. If you’re using MongoDB with Ruby, you should first know a few things about the Ruby driver. So in the first part of the webinar, we’ll provide an overview of the driver, show how to use it with replica sets, and cover some common gotchas. We’ll talk about some of the advantages of using JRuby with MongoDB and describe the Ruby driver support for that platform. And finally, we’ll look at the most popular object mappers in Ruby, adumbrate their respective features, and describe just what to consider before you start using one.</td>
</tr>
<tr>
<td>September 26, 2012</td>
<td>12:00PM EDT/9:00AM PDT</td>
<td><strong>Asegurando Datos en MongoDB con Gazzang - Webinar in Spanish</strong></td>
<td>Si la seguridad de datos es importante para ti, no te pierdas este seminario web. 10gen está trabajando con Gazzang para asegurar que tus datos sensibles seancriptados cuando son almacenados en la base de datos MongoDB y tus llaves criptográficas se mantengan seguras y en conformidad con HIPAA, PCI-DSS, FISMA y otras regulaciones de seguridad de datos. En este seminario web, compartiremos algunos casos de uso de la vida real de clientes que aseguran sus datos en MongoDB, y mostraremos como instalar de forma rápida la protección de datos “enterprise-class” en tu ambiente.</td>
</tr>
<tr>
<td>September 27, 2012</td>
<td>11:00AM EDT/8:00AM PDT/3:00PM UTC and 2:00PM EDT/11:00AM PDT/6:00PM UTC</td>
<td><strong>Common MongoDB Use Cases</strong></td>
<td>MongoDB works differently than other databases. It’s document oriented data model, range based partitioning, and strong consistency model are well suited to some problems, and less well suited to others. In this webinar, we’ll go through real world use cases of MongoDB that take advantage of these unique features. We’ll cover specific customers using MongoDB, how they implemented their solution and how you can build similar solutions for your own organization.</td>
</tr>
<tr>
<td>October 4, 2012</td>
<td>1:00PM EDT/10:00AM PDT</td>
<td><strong>What’s new in the .NET Driver</strong></td>
<td>Recent releases of the .NET driver have added lots of cool new features. In this webinar we will highlight some of the most important ones. We will begin by discussing serialization. We will describe how serialization is normally handled, and how you can customize the process when you need to, including some tips on migration strategies when your class definitions change. We will continue with a discussion of the new Query builder, which now includes support for typed queries. A major new feature of recent releases is support for LINQ queries. We will show you how the .NET driver supports LINQ and discuss what kinds of LINQ queries are supported. Finally, we will discuss what you need to do differently in your application when authentication is enabled at the server.</td>
</tr>
<tr>
<td>October 11, 2012</td>
<td>11:00AM EDT/8:00AM PDT and 2:00PM EDT/11:00AM PDT</td>
<td><strong>Operational Intelligence with MongoDB</strong></td>
<td>One of the most common use cases for MongoDB is operational intelligence. In this presentation we will go through schema design, indexing and sharding for real time analytics workloads. We’ll talk in detail about the ways you can best model your data for analytics, how to make use of MongoDB’s unique features to construct your system, and talk about customers employing these techniques to scale their own systems.</td>
</tr>
<tr>
<td>October 18, 2012</td>
<td>2:00PM EDT/11:00AM PDT</td>
<td><strong>Building your first Java Application with MongoDB</strong></td>
<td>This webinar will introduce how to build your first Java application with MongoDB by walking you through how one can build a simple location based application. The talk will cover the basics of MongoDB’s document model, query language, aggregation framework and deployment architecture. New features, fixes and improvements in the latest release will also be covered.</td>
</tr>
<tr>
<td>October 25, 2012</td>
<td>4:00PM GMT/12:00PM EDT/12:00PM EDT/9:00AM PDT</td>
<td><strong>Ensuring High Availability for Real-time Analytics featuring Boxed Ice / Server Density</strong></td>
<td>This will cover what to consider for high write throughput performance from hardware configuration through to the use of replica sets, multi-data centre deployments, monitoring and sharding to ensure your database is fast and stays online.</td>
</tr>
<tr>
<td>Date</td>
<td>Time</td>
<td>Title</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>November 1, 2012</td>
<td>3:00PM GMT/11:00AM EDT/8:00AM PDT</td>
<td>Building Web Applications with MongoDB and Spring</td>
<td>Spring Data MongoDB provides a feature rich library for writing MongoDB applications in Java. It builds upon the Spring Framework and as such promotes a POJO programming model with a strong emphasis on productivity. In this webinar we will looking to how the The Spring MongoDB Project provides a rich object mapper, a MongoTemplate helper class to simplify common document access operations using query, criteria, and update DSLs, automatic repository interface implementations, QueryDSL integration, and cross-store persistence. We will also look into real-world use cases for simplifying application development with MongoDB and Spring framework.</td>
</tr>
<tr>
<td>November 8, 2012</td>
<td>11:00AM EDT/8:00AM PDT/3:00PM UTC and 2:00PM EDT/11:00AM PDT/6:00PM UTC</td>
<td>Replication and Replica Sets</td>
<td>MongoDB supports replication for failover and redundancy. In this session we will introduce the basic concepts around replica sets, which provide automated failover and recovery of nodes. We'll show you how to set up, configure, and initiate a replica set, and methods for using replication to scale reads. We'll also discuss proper architecture for durability.</td>
</tr>
<tr>
<td>November 15, 2012</td>
<td>11:00AM EDT/8:00AM PDT/3:00PM UTC and 2:00PM EDT/11:00AM PDT/6:00PM UTC</td>
<td>What's New in the PHP Driver</td>
<td>In this session, we'll provide an overview of recent changes to the PHP driver for MongoDB. In particular, we'll discuss the types of changes that we had to make in order to improve high availability connection handling. We'll cover best practices for configuring replica sets, and demonstrate how the PHP driver manages failover and elections within a replica set. In addition, we'll show off some new command wrappers in the latest version of the PHP driver.</td>
</tr>
<tr>
<td>November 21, 2012</td>
<td>1:00PM GMT/8:00AM ET</td>
<td>Building your first Java Application with MongoDB (EMEA)</td>
<td>This talk will introduce how to build your first Java application with MongoDB by walking you through how one can build a simple location based application. The talk will cover the basics of MongoDB's document model, query language, aggregation framework and deployment architecture. New features, fixes and improvements in the latest release will also be covered.</td>
</tr>
<tr>
<td>November 29, 2012</td>
<td>1:00PM EDT/10:00AM PDT/6:00PM UTC</td>
<td>MongoDB on the JVM</td>
<td>Brendan McAdams explores the deeper relationship between the MongoDB database and various languages on the Java Virtual Machine such as Java, Scala, Clojure, JRuby and Python as well as the challenges posted getting MongoDB to play nice with these tools and their syntax. Also examined will be frameworks and integration points popular between MongoDB and the JVM such as Spring Data, Morphia and Lift's MongoDB-Record component.</td>
</tr>
<tr>
<td>December 6, 2012</td>
<td>11:00AM EDT/8:00AM PDT/3:00PM UTC and 2:00PM EDT/11:00AM PDT/6:00PM UTC</td>
<td>MongoDB for Content Management</td>
<td>MongoDB's flexible schema makes it a great fit for your next content management application. MongoDB’s data model makes it easy to catalog multiple content types with diverse meta data. In this session, we'll review schema design for content management, using GridFS for storing binary files, and how you can leverage MongoDB's auto-sharding to partition your content across multiple servers.</td>
</tr>
<tr>
<td>December 13, 2012</td>
<td>1:00PM EDT/10:00AM PDT/6:00PM UTC</td>
<td>Simplifying Persistence for Java and MongoDB</td>
<td>Jeff Yemin will host a webinar covering the design and major features of Morphia, an Object Document Mapper (ODM) for Java and MongoDB. The webinar will start with a short introduction to MongoDB and the various options for building MongoDB applications on the JVM before taking a deep dive into Morphia, which will be presented as an extended example format that demonstrates, for each feature, the domain model, a test driver, and the results as they appear in MongoDB.</td>
</tr>
</tbody>
</table>
Sharding Best Practices

Sharding allows you to distribute load across multiple servers and keep your data balanced across those servers. This session will review MongoDB’s sharding support, including an architectural overview, design principles, and automation. Also will be presented varied scenarios of handling large amount of data, along with their best sharding strategies.

MongoDB User Groups

Your Go-To Guide to Running A MongoDB User Group

<table>
<thead>
<tr>
<th>North America</th>
<th>MEETUPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
<td>July 25</td>
</tr>
<tr>
<td></td>
<td>July Atlanta MongoDB User Group</td>
</tr>
<tr>
<td>Boston</td>
<td></td>
</tr>
<tr>
<td>Chicago</td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td></td>
</tr>
<tr>
<td>Hawaii</td>
<td></td>
</tr>
<tr>
<td>Huntsville</td>
<td></td>
</tr>
<tr>
<td>Houston</td>
<td></td>
</tr>
<tr>
<td>Los Angeles</td>
<td></td>
</tr>
<tr>
<td>Madison, WI</td>
<td>None currently scheduled. Join us to be notified!</td>
</tr>
<tr>
<td>Mexico</td>
<td></td>
</tr>
<tr>
<td>New York</td>
<td></td>
</tr>
<tr>
<td>New Jersey</td>
<td></td>
</tr>
<tr>
<td>Philadelphia</td>
<td></td>
</tr>
<tr>
<td>Phoenix, AZ</td>
<td></td>
</tr>
<tr>
<td>Raleigh</td>
<td></td>
</tr>
<tr>
<td>San Francisco Bay Area</td>
<td></td>
</tr>
<tr>
<td>Seattle</td>
<td>Date TBD</td>
</tr>
<tr>
<td></td>
<td>Mapping Flatland: Using MongoDB for an MMO Crossword Game</td>
</tr>
<tr>
<td>St. Louis</td>
<td></td>
</tr>
<tr>
<td>Toronto</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>South America</th>
<th>MEETUPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sao Palo</td>
<td></td>
</tr>
<tr>
<td>*Santa Caterina</td>
<td></td>
</tr>
<tr>
<td>Belo Horizonte</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EUROPE</th>
<th>MEETUPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barcelona</td>
<td></td>
</tr>
<tr>
<td>Bristol</td>
<td></td>
</tr>
<tr>
<td>Finland</td>
<td></td>
</tr>
</tbody>
</table>
Learn about setting-up MongoDB on Linode from Gareth and searching MongoDB with richer context from Tristan | 

If you're interested in having someone present MongoDB at your conference or meetup, or if you would like to list your MongoDB event on this page, contact meghan at 10gen dot com. Want some MongoDB stickers to give out at your talk? Complete the Swag Request Form.

**Training**

MongoDB training from 10gen is the best way to get you and your team up to speed quickly. Our MongoDB Developer Training and MongoDB Administrator Training are comprehensive, two day courses.

Contact us at training@10gen.com if you would like to buy training credits for multiple students or classes, schedule an on-site training, or discuss a custom training option.

<table>
<thead>
<tr>
<th>City</th>
<th>Date</th>
<th>Time</th>
<th>Location</th>
<th>Look For</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Hanoi, Vietnam</em></td>
<td>None currently scheduled. Like the facebook page to receive updates!</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Melbourne</td>
<td>None currently scheduled. Please check back</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

"Office Hours"
<table>
<thead>
<tr>
<th>Location</th>
<th>Day</th>
<th>Time</th>
<th>Details</th>
<th>See Also</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta, GA</td>
<td>4-6pm</td>
<td></td>
<td>Please check the <a href="http://www.10gen.com/events">Atlanta MongoDB User Group</a> page for upcoming office hours.</td>
<td><a href="http://www.10gen.com/presentations">See Also</a></td>
</tr>
<tr>
<td>New York, NY</td>
<td>Wednesdays</td>
<td>4-6:30pm</td>
<td>10gen Headquarters, 578 Broadway, 7th Floor. 10gen holds weekly open &quot;office hours&quot; with whiteboarding and hack sessions at 10gen headquarters.</td>
<td><a href="http://www.10gen.com/events">See Also</a></td>
</tr>
<tr>
<td>Palo Alto, CA</td>
<td>Thursdays</td>
<td>4-6pm</td>
<td>10gen CA office, 555 University Avenue, Palo Alto, CA 94301. Have questions about MongoDB? Visit the 10gen office in Palo Alto to speak directly with the MongoDB engineers (or just come say hi!).</td>
<td><a href="http://www.10gen.com/events">See Also</a></td>
</tr>
<tr>
<td>San Francisco, CA</td>
<td>Every other Monday</td>
<td>5-7pm</td>
<td>Epicenter Cafe, 764 Harrison St, Between 4th St &amp; Lapu St, San Francisco, CA. Stop by the Epicenter Cafe in San Francisco to meet 10gen Software Engineers Sridhar Nanjundeswaran. Ask questions, hack, have some coffee. Look for a laptop with a &quot;Powered by MongoDB&quot; sticker.</td>
<td><a href="http://www.10gen.com/events">See Also</a></td>
</tr>
<tr>
<td>London, UK</td>
<td>Every other Wednesday</td>
<td>5-7pm</td>
<td>Look Mum No Hands!, Unit 2G 5-25 Scrutton Street, London, EC2A 4HJ. Stop by Look Mum No Hands! Meet 10gen Engineers Ross Lawley, Chris Harris and Dan Roberts. Ask questions, hack, have some coffee. Look for a laptop with a MongoDB leaf sticker.</td>
<td><a href="http://www.10gen.com/events">See Also</a></td>
</tr>
<tr>
<td>München, DE</td>
<td>Scheduled through Meetup.com</td>
<td></td>
<td>comSysto GmbH Lindwurmstr. 97, 80337 Munich. Look for the MongoDB Logo!</td>
<td><a href="http://www.10gen.com/events">See Also</a></td>
</tr>
</tbody>
</table>

**MongoDB User Groups (MUGs)**

MongoDB User Groups (MUGs) are a great way for the community to learn from one another about MongoDB best practices, to network, and to have fun. Interested in starting a user group in your city? Submit a proposal and check out our [guide to starting a MongoDB User Group](http://www.10gen.com/events) to get started! Need some ideas? Check out our **go-to-guide for running a MongoDB user group**.

**MongoDB User Groups**

**North America**

- Atlanta, GA
- Boston, MA
- Chicago, IL
- Cleveland, OH
- Cincinnati, OH
- Charlotte, NC
- Denver, CO
- Los Angeles, CA
- Huntsville, AL
- Hawaii
- Milwaukee, WI
  - [Google+](https://plus.google.com)
  - [@BrewMUG on Twitter](https://twitter.com)
- New Jersey
- New York, NY
- Philadelphia, PA
- Phoenix, AZ
  - [@PHXMUG on Twitter](https://twitter.com)
- Raleigh, NC
- Richmond, VA
- San Diego
- San Francisco, CA
- Seattle, WA
- Silicon Valley, CA
- St. Louis, MO
- Toronto, Ontario

---

**See Also**

- [http://www.10gen.com/presentations](http://www.10gen.com/presentations)
- [http://www.10gen.com/events](http://www.10gen.com/events)
- [http://lanyrd.com/topics/mongodb/](http://lanyrd.com/topics/mongodb/)
- Washington, DC

**South America**
- Belo Horizonte, Brazil
- Sao Paulo, Brazil
- Lima, Peru
- Santa Caterina

**Europe**
- Barcelona, ESP
- Berlin, DE
  - @MUGBerlin on Twitter
- Bristol, UK
- Brussels
- Sofia, Bulgaria
- Dublin
- Düsseldorf
  - @MongoDUS on Twitter
- Frankfurt
- Helsinki, Finland
- Paris, France
- London, UK
  - @MongoDBLondon on Twitter
- Mallorca, ESP
- Milan, Italy
- Minsk, Belarus
- München, Germany
- Omsk, Russia
  - Google+
  - Vkontakte
- Roma, Italia
  - @MongoDBRoma on Twitter
- Sophia-Antipolis, France
- St. Petersburg
- Stockholm, Sweden
- Thames Valley, UK
- Zurich, Switzerland

***German MongoDB User Group on Xing

**Middle East**
- Israel

**Africa**
- Cape Town

**Asia**
- Bangalore
- Colombo, Sri Lanka
- China
- Delhi
- Hanoi
- Indonesia
- Tokyo, Japan
- Thailand
- Turkey
- Kuala Lumpur, Malaysia
- Pune
- Singapore
- South Korea

**Australia**
- Melbourne
- Sydney
- Canberra
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 Diroll, 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 Diroll 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 Diroll, Software Engineer, 10gen
Gluecon
Denver, CO
Wednesday May 26 & Thursday May 27

Schema Design with MongoDB
Kyle Banker, Software Engineer, 10gen
Webinar
Tuesday April 27
Recording and Slides

Dropping ACID with MongoDB
Kristina Chodorow, Software Engineer, 10gen
San Francisco MySQL Meetup
San Francisco, CA
Monday, April 12
Video

Introduction to MongoDB
Mike Dirolf, Software Engineer, 10gen
Emerging Technologies for the Enterprise Conference
Philadelphia, PA
Friday, April 9
Slides

Indexing with MongoDB
Aaron Staple, Software Engineer, 10gen
Webinar
Tuesday April 6, 2010
Video | Slides

TechZing Interview with Mike Dirolf, Software Engineer, 10gen
Monday, April 5
Podcast

Hot Potato and MongoDB
New York Tech Talks Meetup
Justin Shaffer and Lincoln Hochberg
New York, NY
Tuesday March 30
Video

MongoDB Day
Geek Austin Data Series
Austin, TX
Saturday March 27
Photo

Mongo Scale!
Kristina Chodorow, Software Engineer, 10gen
Webcast
Friday March 26
Webcast

NoSQL Live Boston
Boston, MA
Thursday March 11
Recap with slides and MP3

MongoDB: How it Works
Mike Dirolf, Software Engineer, 10gen
Monday March 8, 12:30 PM Eastern Time
Slides

Intro to MongoDB
Alex Sharp, Founder / Lead Software Architect, FrothLogic
LA WebDev Meetup
February 23, 2010
Slides

Introduction to MongoDB
Kristina Chodorow, Software Engineer, 10gen
FOSDEM - Brussels, Belgium
February 7, 2010
Video | Slides | Photos

If you're interested in having someone present MongoDB at your conference or meetup, or if you would like to list your MongoDB event on this page, contact meghan at 10gen dot com.

Your Go-to Resource for Running a MongoDB User Group

- Tips for Running a Successful User Group
- Interested in Starting a User Group?
- Logistics
- Once you Meet Up...
- Stay Engaged
- More Tips and Tricks
Tips for Running a Successful User Group

Interested in Starting a User Group?

Organizing a user group is a fantastic way to meet and learn from other MongoDB fans in your local community. Interested in starting up a user group in your city? Submit a proposal!

Logistics

Use Meetup.com: This one may be obvious but important to emphasize. Not only does meetup have a lot of great tools for event organizers, but they do a really good job of making it easy for meetup members to find relevant groups. Make sure to tag your group and include a description with keywords so that your group appears in meetup searches!

Consistency is important: It's important to establish a routine early on. If you consistently meet on, the second Tuesday of every month, your members will come to expect the meetup. The first few meetings of any user group will be small, but at every meetup, new members will join your group. So meeting at least on a monthly basis is very important. We have all the NY MUG meetups listed far in advance, even if we don't have a speaker lined up. This also makes your life easier when you are approaching speakers and hosts. It's much easier to ask a speaker "Can you present at the May 19 NY MUG?" than going back and forth coordinating dates. And hosts will appreciate having the events reserved far in advance.

Cross promote: Consider partnering with other technology meetups. This is a great way for communities to learn from one another and gain exposure to new technologies. It could be as simple as occasionally posting on other meetup lists. For example, when we had a presentation on Scalable Event Analytics with Ruby on Rails and MongoDB, I cross-posted the meetup on the NYC Ruby mailing list and we soon had a dozen new members in the group. I also typically list our events in Startup Digest, LinkedInNYC, Gary's Guide, Charlie O'Donnell's newsletter, Mashable, and more.

Once you Meet Up...

Get great speakers: Make a wish list of speakers, and then just start asking people! After organizing dozens of MongoDB events, I've been amazed at how willing people are to present. Most of the time, it's just a matter of asking. And if the person says no, ask them to refer someone else.

Get Clever About Getting Great Speakers: Are the speakers on your wish list 3,000 miles away from you? That's okay. We live in a wired world. Use Skype or WebX to bring your speakers to you. You can do screen shares to see demos, and provide your meetup members with a great learning experience.

Host Lightning Talks: Sometimes your user group members will not have the chance to pull together a full 30 minute presentation on a product or feature, but some of them may be interested in giving a 5-10 minute lightning talk on something they've designed, an issue they're having with MongoDB, or their dream project. Offer the opportunity when you send out the Meetup invite. If you don't get any submissions, don't worry. Deliver the first lightning talk at the event. Someone might step up to the plate.

Start a Study Group: Does your meetup have a lot of inexperienced MongoDB Users? Start a study group and learn together. Learning in a group is a great experience, and study groups are very common in different developer communities--particularly Ruby on Rails and Node.js

Host a Helpathon: The NYC On Rails Meetup is famous for innovating on the Hackathon to produce the Helpathon, a four hour session for helping one another get through programming hurdles.

Start a Book Club: If your group is at a loss for topics to discuss you could start a reading group. This is similar to a study group and can be a great way to create a more tightly knit User Group for the future.

Raffle off prizes: Prizes are an excellent way to get people to come to your meetup. An easy and free way to get great prizes is to join the O'Reilly User Group program. They will send you free books to give out at your group, as well as discounts on upcoming conferences.

Host a Coding Contest: Google is planning a coding competition for tickets to Google I/O, their largest developer contest. Why not do the same before your first user group? Present a small problem to your members and ask them to bring their files the day of to show off their solution. You can then have judges or the audience choose the winning solution(s).

Stay Engaged

Social media: Consider creating a twitter handle or hashtag for the group. Ask the presenter to tweet or blog about the event, and ask the key members of the group to do the same. Post the hashtag or twitter handle at the event so that members know to use it.

Continue the discussion after the meetup: You can easily record videos and take photos at sessions to share information after the meetup. Encourage the presenter to send a message the group with the slides to start a conversation among the entire meetup.

Send out Group Polls: Get a better understanding of what your user group wants to hear by sending out polls. If you use Meetup.com, you can try out the Polls feature. Otherwise, use SurveyMonkey.

More Tips and Tricks
Presentation Tips for MongoDB User Groups

Presenting at a MongoDB User Group (MUG) is a great way to share your experiences working with MongoDB, drive adoption of your product and create a closer connection between your company’s services and MongoDB. If you're new to public speaking, it's a great place to get presentation experience or test out a talk for a future conference. Here are some tips for giving presentations at a user group.

Have a suggestion that’s not listed here? Email our community team with your ideas.

Content: What Should I Talk About At a User Group?

- Teach them something: MongoDB User Group members have different levels of experience with MongoDB but everyone has the same goal: they want to learn more. Offering your insights, tips and tricks often turns into a great presentation with a lot of discussion.
  - Many people in our user groups are new to MongoDB, while others are veterans and have years of experience under their belt but are interested in expanding their knowledge base. It's sometimes a good idea to ask the user group organizer about the experience level of their members so you can cater your presentation their interests.
- Some great topics for a MongoDB User Group (with some examples)
  - MongoDB in the Cloud
    - Growing MongoDB on Amazon Web Services
    - MongoDB on AWS
  - Monitoring Your MongoDB Deployment
  - How I use MongoDB in Production
    - Krossover + MongoDB
    - Fiesta.cc + MongoDB
    - Inside Wordnik's Architecture
  - MongoDB in a Specific Language Library
    - MongoDB and PHP
    - Getting Started with MongoDB and Scala
    - MongoDB on the JVM
- Want to get your feet wet? Offer to give a 5-10 minute lightening talk or demo of a MongoDB tool to build up your presentation skills.

Slides

- Be precise in your slides - Most user group organizers make slides available online after the user group, so make sure they are correct and to the point.
- Be visual - a clever picture, easy to read chart, or even simple bullet points convey clear information.
- Use Big Font! Some meeting rooms are large and it is always hard to see from the back of the room
- Avoid including full text such as sentences and paragraphs in your slides. This makes it difficult for people to pay attention to your thoughts and expertise. You should be the center of the presentation, the slides should be there to guide you.
- Know your audience - The audience is very technical, and generally knowledgeable about MongoDB and related technologies. They are interested in hearing your insights and experience so they can become better MongoDB developers.
- Some great resources to help you build well-designed slides: Duarte have great insight into presentation design and storytelling. Check out their blog for weekly tips and their Tools for Revolutionaries with free video guides to reworking your presentations.

Logistics

- Ask the organizer what time you should arrive (also be courteous and arrive on time)
- Make sure your computer or presentation will be compatible with the presentation setup available at the meetup group. Ask the organizer to make sure you have the correct connecting equipment. If you need a different adaptor for your laptop, please try to bring it with you.
- Send you presentations to the organizer so they can distribute it with the group. It's great to share this information
- Internet connectivity: wifi can be tedious in some places. To avoid any painful situations with wifi, try to make any demos in your presentations available offline.
- Invite your friends! It's always fun to bring along friends, family and coworkers to a user group—but be sure to check with the organizer to ensure the meeting location has enough capacity.

MongoDB Archives and 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
mongo, the MongoDB Shell
Other great posts
MongoDB 2.2 Released

Best Blogs from the MongoDB Community

- Using MongoDB as a queue on OpenShift with a Twilio application
- Working with MongoDB and Poweshell
- Stability in the Midst of Chaos, Jon Hoffman, foursquare
- How this Website Uses MongoDB, Ian White, former CTO, Business Insider.
- Lessons Learnt from using MongoDB with High Volume Data by David Henderson of Triggered Messaging.
- Forward Intel Uses MongoDB for “casual” analytics
- Mobile Geolocation App in 30 Minutes on OpenShift with MongoDB
- Creating a REST API in Python using Bottle and MongoDB

Articles / Key Doc Pages

- On Atomic Operations
- Reaching into Objects - how to do sophisticated query operations on nested JSON-style objects
- Schema Design
- Full Text Search in Mongo
- MongoDB Production Deployments

Presentations

- Presentations from MongoDB Conferences
- MongoDB for Rubyists (February 2010 Chicago Ruby Meetup)
- Introduction to MongoDB (FOSDEM February 2010)
- NY MySql Meetup - NoSQL, Scaling, MongoDB
- Teach Me To Code - Introduction to MongoDB
- DCVIE

Benchmarks

MongoDB does not publish any official benchmarks.

We recommend running application performance tests on your application’s work-load to find bottleneck and for performance tuning.

See Also

- JS Benchmarking Harness

FAQ

Redirection Notice
This page should redirect to http://docs.mongodb.org/manual/faq/fundamentals/.

Misc

Demo App in Python

From an Interop 2009 presentation

Code: http://github.com/mdirolf/simple-messaging-service/tree/master
### MongoDB, CouchDB, MySQL Compare Grid

<table>
<thead>
<tr>
<th></th>
<th>CouchDB</th>
<th>MongoDB</th>
<th>MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Model</strong></td>
<td>Document-Oriented (JSON)</td>
<td>Document-Oriented (BSON)</td>
<td>Relational</td>
</tr>
<tr>
<td><strong>Data Types</strong></td>
<td>string, number, boolean, array, object</td>
<td>string, int, double, boolean, date, bytearray, object, array, others</td>
<td>link</td>
</tr>
<tr>
<td><strong>Large Objects (Files)</strong></td>
<td>Yes (attachments)</td>
<td>Yes (GridFS)</td>
<td>Blobs</td>
</tr>
<tr>
<td><strong>Horizontal partitioning scheme</strong></td>
<td>CouchDB Lounge</td>
<td>Auto-sharding</td>
<td>Partitioning</td>
</tr>
<tr>
<td><strong>Replication</strong></td>
<td>Master-master (with developer supplied conflict resolution)</td>
<td>Master-slave and replica sets</td>
<td>Master-slave, multi-master, and circular replication</td>
</tr>
<tr>
<td><strong>Object(row) Storage</strong></td>
<td>One large repository</td>
<td>Collection-based</td>
<td>Table-based</td>
</tr>
<tr>
<td><strong>Query Method</strong></td>
<td>Map/reduce of javascript functions to lazily build an index per query</td>
<td>Dynamic; object-based query language</td>
<td>Dynamic; SQL</td>
</tr>
<tr>
<td><strong>Secondary Indexes</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Atomicity</strong></td>
<td>Single document</td>
<td>Single document</td>
<td>Yes - advanced</td>
</tr>
<tr>
<td><strong>Interface</strong></td>
<td>REST</td>
<td>Native drivers ; REST add-on</td>
<td>Native drivers</td>
</tr>
<tr>
<td><strong>Server-side batch data manipulation</strong></td>
<td>?</td>
<td>Map/Reduce, server-side javascript</td>
<td>Yes (SQL)</td>
</tr>
<tr>
<td><strong>Written in</strong></td>
<td>Erlang</td>
<td>C++</td>
<td>C++</td>
</tr>
<tr>
<td><strong>Concurrency Control</strong></td>
<td>MVCC</td>
<td>Update in Place</td>
<td></td>
</tr>
<tr>
<td><strong>Geospatial Indexes</strong></td>
<td>GeoCouch</td>
<td>Yes</td>
<td>Spatial extensions</td>
</tr>
<tr>
<td><strong>Distributed Consistency Model</strong></td>
<td>Eventually consistent (master-master replication with versioning and version reconciliation)</td>
<td>Strong consistency. Eventually consistent reads from secondaries are available.</td>
<td>Strong consistency. Eventually consistent reads from secondaries are available.</td>
</tr>
</tbody>
</table>

### See Also
- Comparing Mongo DB and Couch DB

### Comparing Mongo DB and Couch DB

We are getting a lot of questions "how are mongo db and couch different?" It's a good question: both are document-oriented databases with schemaless JSON-style object data storage. Both products have their place -- we are big believers that databases are specializing and "one size fits all" no longer applies.

We are not CouchDB gurus so please let us know in the forums if we have something wrong.

### MVCC

One big difference is that CouchDB is MVCC based, and MongoDB is more of a traditional update-in-place store. MVCC is very good for certain classes of problems: problems which need intense versioning; problems with offline databases that resync later; problems where you want a large amount of master-master replication happening. Along with MVCC comes some work too: first, the database must be compacted periodically, if there are many updates. Second, when conflicts occur on transactions, they must be handled by the programmer manually (unless the db also does conventional locking -- although then master-master replication is likely lost).

MongoDB updates an object in-place when possible. Problems requiring high update rates of objects are a great fit; compaction is not necessary.
Mongo's replication works great but, without the MVCC model, it is more oriented towards master/slave and auto failover configurations than to complex master-master setups. With MongoDB you should see high write performance, especially for updates.

**Horizontal Scalability**

One fundamental difference is that a number of Couch users use replication as a way to scale. With Mongo, we tend to think of replication as a way to gain reliability/failover rather than scalability. Mongo uses (auto) sharding as our path to 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.

Update Aug2011: Couch is adding a new query language "UNQL".

**Atomicity**

Both MongoDB and CouchDB support concurrent modifications of single documents. Both forego complex transactions involving large numbers of objects.

**Durability**

CouchDB is a "crash-only" design where the db can terminate at any time and remain consistent.

Previous versions of MongoDB used a storage engine that would require a repairDatabase() operation when starting up after a hard crash (similar to MySQL's MyISAM). Version 1.7.5 and higher offer durability via journaling; specify the --journal command line option

**Map Reduce**

Both CouchDB and MongoDB support map/reduce operations. For CouchDB map/reduce is inherent to the building of all views. With MongoDB, map/reduce is only for data processing jobs but not for traditional queries.

**Javascript**

Both CouchDB and MongoDB make use of Javascript. CouchDB uses Javascript extensively including in the building of views.

MongoDB supports the use of Javascript but more as an adjunct. In MongoDB, query expressions are typically expressed as JSON-style query objects; however one may also specify a javascript expression as part of the query. MongoDB also supports running arbitrary javascript functions server-side and uses javascript for map/reduce operations.

**REST**

Couch uses REST as its interface to the database. With its focus on performance, MongoDB relies on language-specific database drivers for access to the database over a custom binary protocol. Of course, one could add a REST interface atop an existing MongoDB driver at any time -- that would be a very nice community project. Some early stage REST implementations exist for MongoDB.

**Performance**

Philosophically, Mongo is very oriented toward performance, at the expense of features that would impede performance. We seeMongo DB being useful for many problems where databases have not been used in the past because databases are too "heavy". Features that give MongoDB good performance are:

- client driver per language: native socket protocol for client/server interface (not REST)
- use of memory mapped files for data storage
- collection-oriented storage (objects from the same collection are stored contiguously)
- update-in-place (not MVCC)
- written in C++

**Use Cases**

It may be helpful to look at some particular problems and consider how we could solve them.

- if we were building Lotus Notes, we would use Couch as its programmer versioning reconciliation/MVCC model fits perfectly. Any
problem where data is offline for hours then back online would fit this. In general, if we need several eventually consistent master-master replica databases, geographically distributed, often offline, we would use Couch.

- mobile
  - Couch is better as a mobile embedded database on phones, primarily because of its online/offline replication/sync capabilities.
  - we like Mongo server-side; one reason is its geospatial indexes.
- if we had very high performance requirements we would use Mongo. For example, web site user profile object storage and caching of data from other sources.
- for a problem with very high update rates, we would use Mongo as it is good at that because of its "update-in-place" design. For example see updating real time analytics counters
- in contrast to the above, couch is better when lots of snapshotting is a requirement because of its MVCC design.

Generally, we find MongoDB to be a very good fit for building web infrastructure.

Licensing

- Database:
  - Free Software Foundation’s GNU AGPL v3.0.
  - Commercial licenses are also available from 10gen, including free evaluation licenses.
- Drivers:
  - mongodb.org supported drivers: Apache License v2.0.
  - Third parties have created drivers too; licenses will vary there.
- Documentation: Creative Commons.

The goal of the server license is to require that enhancements to MongoDB be released to the community. Traditional GPL often does not achieve this anymore as a huge amount of software runs in the cloud. For example, Google has no obligation to release their improvements to the MySQL kernel – if they do they are being nice.

To make the above practical, we promise that your client application which uses the database is a separate work. To facilitate this, the mongodb.org supported drivers (the part you link with your application) are released under Apache license, which is copyleft free. Note: if you would like a signed letter asserting the above promise please contact us.

If the above isn’t enough to satisfy your organization’s vast legal department (some will not approve GPL in any form), please contact us – commercial licenses are available including free evaluation licenses. We will try hard to make the situation work for everyone.

International Docs

Most documentation for MongoDB is currently written in English. We are looking for volunteers to contribute documentation in other languages. If you’re interested in contributing to documentation in another language please email docs at 10gen dot com

Language Homepages

- Deutsch
- Español
- Français
- Italiano
- Magyar
- Português
- Svenska

Books
You can download samples at 10gen.com/books.

**MongoDB: The Definitive Guide**
Kristina Chodorow and Mike Diroll

**The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing**
Peter Membrey

**Scaling MongoDB**
Kristina Chodorow

**MongoDB in Action**
Kyle Banker

**MongoDB and Python: Patterns and processes for the popular document-oriented database**
Niall O'Higgins

**MongoDB and PHP**
Steve Francia

**MongoDB for Web Development**
Mitch Pirtle
MongoDB: Sag Ja zu NoSQL
Marc Boeker

The Little MongoDB Book
Karl Seguin
Free ebook

50 Tips and Tricks for MongoDB Developers
Kristina Chodorow

MongoDB with Python and Ming
Rick Copeland

PHP and MongoDB Web Development Beginner’s Guide book and eBook
Rubayeet Islam

Ruby and MongoDB Web Development Beginner’s Guide
Gautam Rege

Slides and Video

Upcoming Events

- http://www.10gen.com/events

Presentations from past conferences

- http://www.10gen.com/presentations

Some past webinars

- Analytics
  MongoDB Analytics for Online Advertising at Magnetic
- DevOps
  Data Center Awareness with MongoDB
- DevOps
  Deployment Best Practices
- DevOps
  Diagnostics and Performance Tuning
- DevOps
  Keeping your data safe: replication, backup/restore, mongodump
- DevOps
  MongoDB and Amazon Web Services
- DevOps
  MongoDB on Amazon EC2
- DevOps
  Scaling with MongoDB
- DevOps
  Tips for a Successful, Scalable MongoDB Deployment
- Geo
  Geospatial Indexing with MongoDB
- Hadoop
  Hadoop plugin for MongoDB: The Elephant in the Room
- Indexing
  Indexing Matters: A MongoDB Optimization Primer
- Indexing
  Indexing With MongoDB
- Intro
  Developing Applications with MongoDB: A Quickstart Tutorial
- Intro
  Introduction to MongoDB
- Java
  What’s New in Java Driver Version 2.5
- Java
  Zero to MongoDB - Introduction for MongoDB with Java
- Journaling
  Journaling in MongoDB version 1.8
- Node.js
  Node.js and MongoDB, a Panel Discussion
- Ruby
  MongoDB and Ruby
- Scala
  Scala MongoDB and Scala
- Schema
  Schema Design with MongoDB
- Use Cases
  How MongoDB helps Visibiz tackle social CRM
- Use Cases
  Humongous Data at Server Density: Approaching 1 Billion Documents in MongoDB
- Hybrid
  Applications with MongoDB and RDBMS
- v1.8
  What’s new in MongoDB v1.8.0
- v2.0
  What You Need to Know About MongoDB 2.0

Alerts

This page lists critical alerts and advisories for MongoDB. This page is a work in progress and will be enhanced over time.

See http://jira.mongodb.org/ for a comprehensive list of bugs and feature requests.
Data Integrity Related

- Documents may be missing on a replication secondary after initial sync if a high number of updates occur during the sync that move the document (i.e., the documents are growing).
  - [https://jira.mongodb.org/browse/SERVER-3956](https://jira.mongodb.org/browse/SERVER-3956) Replica set case. Fixed: 1.8.4, 2.0.1
  - [https://jira.mongodb.org/browse/SERVER-4270](https://jira.mongodb.org/browse/SERVER-4270) Master/slave case. Fixed: 1.8.5, 2.0.2

- When stepping down a replica set primary, the primary’s connection to clients is not automatically closed in 2.0.1. This means that if the primary steps down while an application is issuing fire-and-forget write, the application won’t necessarily fail over until it issues a query. Applications that only issue safe writes are not affected by this. The issue is fixed in 2.0.2. See [https://jira.mongodb.org/browse/SERVER-4405](https://jira.mongodb.org/browse/SERVER-4405) for details.

Security Related

- Limit access of __system when running --auth without replica sets.
  - [https://jira.mongodb.org/browse/SERVER-3666](https://jira.mongodb.org/browse/SERVER-3666) Fixed: 1.8.4

Make sure to subscribe to [http://groups.google.com/group/mongodb-announce](http://groups.google.com/group/mongodb-announce) for important announcements.