
MongoDB Essentials Training

Release 3.4

MongoDB, Inc.

Mar 24, 2017

Contents

1	Introduction	4
1.1	Warm Up	4
1.2	MongoDB Overview	5
1.3	MongoDB Stores Documents	9
1.4	Lab: Installing and Configuring MongoDB	13
2	CRUD	19
2.1	Creating and Deleting Documents	19
2.2	Reading Documents	25
2.3	Query Operators	33
2.4	Lab: Finding Documents	38
2.5	Updating Documents	39
2.6	Lab: Updating Documents	49
3	Indexes	52
3.1	Index Fundamentals	52
3.2	Compound Indexes	60
3.3	Lab: Optimizing an Index	67
3.4	Multikey Indexes	68
3.5	Hashed Indexes	73
3.6	Geospatial Indexes	74
3.7	TTL Indexes	82
3.8	Text Indexes	83
3.9	Lab: Finding and Addressing Slow Operations	86
3.10	Lab: Using <code>explain()</code>	87
4	Storage	88
4.1	Introduction to Storage Engines	88
5	Drivers	94
5.1	Introduction to MongoDB Drivers	94
5.2	Lab: Driver Tutorial (Optional)	97
6	Aggregation	98
6.1	Aggregation Tutorial	98
6.2	Optimizing Aggregation	113
6.3	Lab: Aggregation Framework	115
6.4	Lab: Using <code>\$graphLookup</code>	118

7	Introduction to Schema Design	119
7.1	Schema Design Core Concepts	119
7.2	Schema Evolution	126
7.3	Common Schema Design Patterns	130
8	Replica Sets	135
8.1	Introduction to Replica Sets	135
8.2	Elections in Replica Sets	139
8.3	Replica Set Roles and Configuration	145
8.4	The Oplog: Statement Based Replication	147
8.5	Lab: Working with the Oplog	150
8.6	Write Concern	152
8.7	Read Preference	156
8.8	Lab: Setting up a Replica Set	157
9	Sharding	162
9.1	Introduction to Sharding	162
9.2	Balancing Shards	170
9.3	Shard Tags	173
9.4	Lab: Setting Up a Sharded Cluster	175
10	Application Engineering	182
10.1	MongoMart Introduction	182
10.2	Java Driver Labs (MongoMart)	184
10.3	Python Driver Labs (MongoMart)	185
11	Security	187
11.1	Authorization	187
11.2	Lab: Administration Users	195
11.3	Lab: Create User-Defined Role (Optional)	197
11.4	Authentication	199
11.5	Lab: Secure mongod	201
11.6	Auditing	202
11.7	Encryption	205
11.8	Lab: Secured Replica Set - KeyFile (Optional)	207
11.9	Lab: LDAP Authentication & Authorization (Optional)	211
12	Views	214
12.1	Views Tutorial	214
12.2	Lab: Vertical Views	216
12.3	Lab: Horizontal Views	218
12.4	Lab: Reshaped Views	219
13	Reporting Tools and Diagnostics	221
13.1	Performance Troubleshooting	221
14	Backup and Recovery	229
14.1	Backup and Recovery	229
15	MongoDB Cloud & Ops Manager Fundamentals	234
15.1	MongoDB Cloud & Ops Manager	234
15.2	Automation	236
15.3	Lab: Cluster Automation	239
15.4	Monitoring	240
15.5	Lab: Create an Alert	242

15.6 Backups	242
16 MongoDB Cloud & Ops Manager Under the Hood	
16.1 API	246
16.2 Lab: Cloud Manager API	247
16.3 Architecture (Ops Manager)	249
16.4 Security (Ops Manager)	251
16.5 Lab: Install Ops Manager	252

1 Introduction

Warm Up (page 4) Activities to get the class started

MongoDB Overview (page 5) MongoDB philosophy and features

MongoDB Stores Documents (page 9) The structure of data in MongoDB

Lab: Installing and Configuring MongoDB (page 13) Install MongoDB and experiment with a few operations.

1.1 Warm Up

Introductions

- Who am I?
- My role at MongoDB
- My background and prior experience

Note:

- Tell the students about yourself:
 - Your role
 - Prior experience

Getting to Know You

- Who are you?
- What role do you play in your organization?
- What is your background?
- Do you have prior experience with MongoDB?

Note:

- Ask students to go around the room and introduce themselves.
- Make sure the names match the roster of attendees.
- Ask about what roles the students play in their organization and note on attendance sheet.
- Ask what software stacks students are using.
 - With MongoDB and in general.
 - Note this information as well.

MongoDB Experience

- Who has never used MongoDB?
- Who has some experience?
- Who has worked with production MongoDB deployments?
- Who is more of a developer?
- Who is more of operations person?

10gen

- MongoDB was initially created in 2008 as part of a hosted application stack.
- The company was originally called 10gen.
- As part of their overarching plan to create the 10gen platform, the company built a database.
- Suddenly everybody said: “I like that! Give me that database!”

Origin of MongoDB

- 10gen became a database company.
- In 2013, the company rebranded as MongoDB, Inc.
- The founders have other startups to their credit: DoubleClick, ShopWiki, Gilt.
- The motivation for the database came from observing the following pattern with application development.
 - The user base grows.
 - The associated body of data grows.
 - Eventually the application outgrows the database.
 - Meeting performance requirements becomes difficult.

1.2 MongoDB Overview

Learning Objectives

Upon completing this module students should understand:

- MongoDB vs. relational databases and key/value stores
- Vertical vs. horizontal scaling
- The role of MongoDB in the development stack
- The structure of documents in MongoDB
- Array fields
- Embedded documents
- Fundamentals of BSON

MongoDB is a Document Database

Documents are associative arrays like:

- Python dictionaries
- Ruby hashes
- PHP arrays
- JSON objects

An Example MongoDB Document

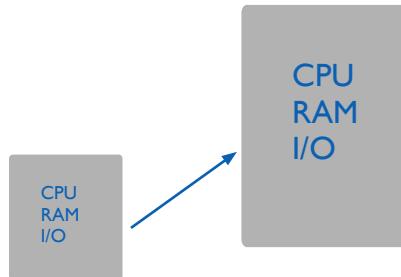
A MongoDB document expressed using JSON syntax.

```
{  
  "_id" : "/apple-reports-second-quarter-revenue",  
  "headline" : "Apple Reported Second Quarter Revenue Today",  
  "date" : ISODate("2015-03-24T22:35:21.908Z"),  
  "author" : {  
    "name" : "Bob Walker",  
    "title" : "Lead Business Editor"  
  },  
  "copy" : "Apple beat Wall St expectations by reporting ...",  
  "tags" : [  
    "AAPL", "Earnings", "Cupertino"  
  ],  
  "comments" : [  
    { "name" : "Frank", "comment" : "Great Story" },  
    { "name" : "Wendy", "comment" : "When can I buy an Apple Watch?" }  
  ]  
}
```

Note:

- How would you represent this document in a relational database? How many tables, how many queries per page load?
 - What are the pros/cons to this design? (hint: 1 million comments)
 - Where relational databases store rows, MongoDB stores documents.
 - Documents are hierarchical data structures.
 - This is a fundamental departure from relational databases where rows are flat.
-

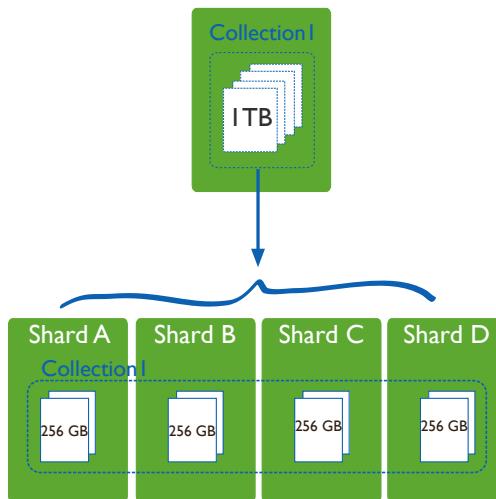
Vertical Scaling



Note: Another difference is in terms of scalability. With an RDBMS:

- If you need to support a larger workload, you buy a bigger machine.
 - The problem is, machines are not priced linearly.
 - The largest machines cost much more than commodity hardware.
 - If your application is successful, you may find you simply cannot buy a large enough a machine to support your workload.
-

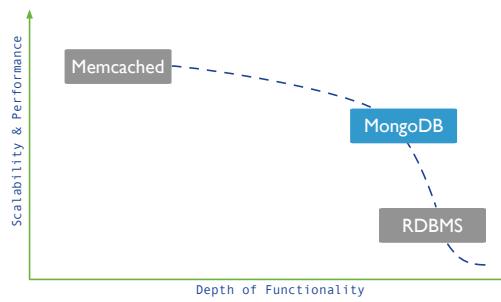
Scaling with MongoDB



Note:

- MongoDB is designed to be horizontally scalable (linear).
 - MongoDB scales by enabling you to shard your data.
 - When you need more performance, you just buy another machine and add it to your cluster.
 - MongoDB is highly performant on commodity hardware.
-

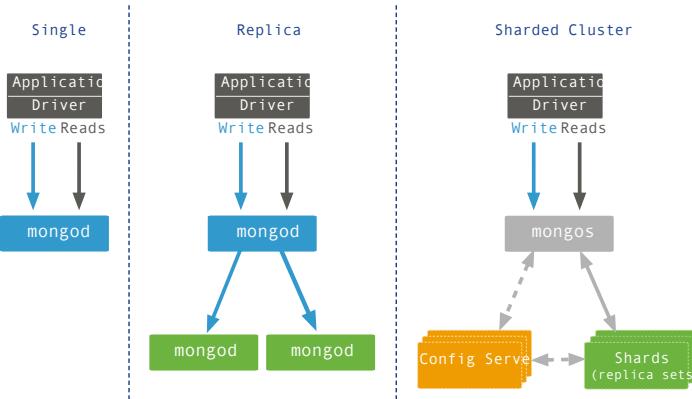
Database Landscape



Note:

- We've plotted each technology by scalability and functionality.
 - At the top left, are key/value stores like memcached.
 - These scale well, but lack features that make developers productive.
 - At the far right we have traditional RDBMS technologies.
 - These are full featured, but will not scale easily.
 - Joins and transactions are difficult to run in parallel.
 - MongoDB has nearly as much scalability as key-value stores.
 - Gives up only the features that prevent scaling.
 - We have compensating features that mitigate the impact of that design decision.
-

MongoDB Deployment Models



Note:

- MongoDB supports high availability through automated failover.
 - Do not use a single-server deployment in production.
 - Typical deployments use replica sets of 3 or more nodes.
 - The primary node will accept all writes, and possibly all reads.
 - Each secondary will replicate from another node.
 - If the primary fails, a secondary will automatically step up.
 - Replica sets provide redundancy and high availability.
 - In production, you typically build a fully sharded cluster:
 - Your data is distributed across several shards.
 - The shards are themselves replica sets.
 - This provides high availability and redundancy at scale.
-

1.3 MongoDB Stores Documents

Learning Objectives

Upon completing this module, students should understand:

- JSON
- BSON basics
- That documents are organized into collections
- ObjectIds

JSON

- JavaScript Object Notation
- Objects are associative arrays.
- They are composed of key-value pairs.

A Simple JSON Object

```
{  
  "firstname" : "Thomas",  
  "lastname" : "Smith",  
  "age" : 29  
}
```

JSON Keys and Values

- Keys must be strings.
- Values may be any of the following:
 - string (e.g., “Thomas”)
 - number (e.g., 29, 3.7)
 - true / false
 - null
 - array (e.g., [88.5, 91.3, 67.1])
 - object
- More detail at json.org¹.

Example Field Values

```
{  
  "headline" : "Apple Reported Second Quarter Revenue Today",  
  "date" : ISODate("2015-03-24T22:35:21.908Z"),  
  "views" : 1234,  
  "author" : {  
    "name" : "Bob Walker",  
    "title" : "Lead Business Editor"  
  },  
  "tags" : [  
    "AAPL",  
    23,  
    { "name" : "city", "value" : "Cupertino" },  
    [ "Electronics", "Computers" ]  
  ]  
}
```

¹ <http://json.org/>

BSON

- MongoDB stores data as Binary JSON (BSON).
 - MongoDB drivers send and receive data in this format.
 - They map BSON to native data structures.
 - BSON provides support for all JSON data types and several others.
 - BSON was designed to be lightweight, traversable and efficient.
 - See bsonspec.org².

Note: E.g., a BSON object will be mapped to a dictionary in Python.

BSON Hello World

```
// JSON
{ "hello" : "world" }

// BSON
"\x16\x00\x00\x00\x02hello\x00
\x06\x00\x00\x00world\x00\x00"
```

Note:

- \x16\x00\x00\x00 (document size)
 - \x02 = string (data type of field value)
 - hello\x00 (key/field name, \x00 is null and delimits the end of the name)
 - \x06\x00\x00\x00 (size of field value including end null)
 - world\x00 (field value)
 - \x00 (end of the document)

A More Complex BSON Example

² <http://bsonspec.org/#/specification>

Documents, Collections, and Databases

- Documents are stored in collections.
- Collections are contained in a database.
- Example:
 - Database: products
 - Collections: books, movies, music
- Each database-collection combination defines a namespace.
 - products.books
 - products.movies
 - products.music

The `_id` Field

- All documents must have an `_id` field.
- The `_id` is immutable.
- If no `_id` is specified when a document is inserted, MongoDB will add the `_id` field.
- MongoDB assigns a unique ObjectId as the value of `_id`.
- Most drivers will actually create the ObjectId if no `_id` is specified.
- The `_id` field is unique to a collection (namespace).

ObjectIds



Note:

- An ObjectId is a 12-byte value.
 - The first 4 bytes are a datetime reflecting when the ObjectId was created.
 - The next 3 bytes are the MAC address of the server.
 - Then a 2-byte process ID
 - Finally, 3 bytes that are monotonically increasing for each new ObjectId created within a collection.
-

1.4 Lab: Installing and Configuring MongoDB

Learning Objectives

Upon completing this exercise students should understand:

- How MongoDB is distributed
- How to install MongoDB
- Configuration steps for setting up a simple MongoDB deployment
- How to run MongoDB
- How to run the Mongo shell

Production Releases

64-bit production releases of MongoDB are available for the following platforms.

- Windows
- OSX
- Linux
- Solaris

Installing MongoDB

- Visit <https://docs.mongodb.com/manual/installation/>.
- Please install the Enterprise version of MongoDB.
- Click on the appropriate link, such as “Install on Windows” or “Install on OS X” and follow the instructions.
- Versions:
 - Even-numbered builds are production releases, e.g., 2.4.x, 2.6.x.
 - Odd-numbers indicate development releases, e.g., 2.5.x, 2.7.x.

Linux Setup

```
PATH=$PATH:<path to mongodb>/bin  
  
sudo mkdir -p /data/db  
  
sudo chmod -R 744 /data/db  
  
sudo chown -R `whoami` /data/db
```

Note:

- You might want to add the MongoDB bin directory to your path, e.g.
- Once installed, create the MongoDB data directory.
- Make sure you have write permission on this directory.

If you are using Koding these are a few instructions you can follow:

- Download MongoDB tarball and setup the environment

```
wget http://downloads.mongodb.org/linux/mongodb-linux-x86_64-ubuntu1204-3.2.  
˓→1.tgz  
tar xzvf mongodb-linux-x86_64-ubuntu1204-3.2.1.tgz  
cd mongodb-linux-x86_64-ubuntu1204-3.2.1/bin  
export PATH=`pwd`:$PATH
```

Install on Windows

- Download and run the .msi Windows installer from mongodb.org/downloads.
- By default, binaries will be placed in the following directory.

```
C:\Program Files\MongoDB\Server\<VERSION>\bin
```

- It is helpful to add the location of the MongoDB binaries to your path.
- To do this, from “System Properties” select “Advanced” then “Environment Variables”

Note: Can also install Windows as a service, but not recommended since we need multiple mongod processes for future exercises

Create a Data Directory on Windows

- Ensure there is a directory for your MongoDB data files.
- The default location is \data\db.
- Create a data directory with a command such as the following.

```
md \data\db
```

Note: Optionally, talk about the –dbpath variable and specifying a different location for the data files

Launch a mongod

Explore the mongod command.

```
<path to mongodb>/bin/mongod --help
```

Launch a mongod with the MMAPv1 storage engine:

```
<path to mongodb>/bin/mongod --storageEngine mmapv1
```

Alternatively, launch with the WiredTiger storage engine (default).

```
<path to mongodb>/bin/mongod
```

Specify an alternate path for data files using the --dbpath option. (Make sure the directory already exists.) E.g.,

```
<path to mongodb>/bin/mongod --dbpath /test/mongodb/data/wt
```

Note:

- Please verify that all students have successfully installed MongoDB.
 - Please verify that all can successfully launch a mongod.
-

The MMAPv1 Data Directory

```
ls /data/db
```

- The mongod.lock file
 - This prevents multiple mongods from using the same data directory simultaneously.
 - Each MongoDB database directory has one .lock.
 - The lock file contains the process id of the mongod that is using the directory.
- Data files
 - The names of the files correspond to available databases.
 - A single database may have multiple files.

Note: Files for a single database increase in size as follows:

- sample.0 is 64 MB
 - sample.1 is 128 MB
 - sample.2 is 256 MB, etc.
 - This continues until sample.5, which is 2 GB
 - All subsequent data files are also 2 GB.
-

The WiredTiger Data Directory

```
ls /data/db
```

- The mongod.lock file
 - Used in the same way as MMAPv1.
- Data files
 - Each collection and index stored in its own file.
 - Will fail to start if MMAPv1 files found

Import Exercise Data

```
unzip usb_drive.zip

cd usb_drive

mongoimport -d sample -c tweets twitter.json

mongoimport -d sample -c zips zips.json

mongoimport -d sample -c grades grades.json

cd dump

mongorestore -d sample city

mongorestore -d sample digg
```

Note: If there is an error importing data directly from a USB drive, please copy the sampledata.zip file to your local computer first.

Note: For local installs

- Import the data provided on the USB drive into the *sample* database.

For Koding environment

- Download *sample* data from:

```
wget https://www.dropbox.com/s/54xsjwq59zoqlfe/sample.tgz
```

Launch a Mongo Shell

Open another command shell. Then type the following to start the Mongo shell.

```
mongo
```

Display available commands.

```
help
```

Note: On Koding environment do the following:

- Create a new *Terminal* and rename it to **Client**
-

Explore Databases

Display available databases.

```
show dbs
```

To use a particular database we can type the following.

```
use <database_name>
db
```

Note:

- This assigns the variable `db` to a connection object for the selected database.
- We can display the name of the database we are currently using by evaluating `db` in the mongo shell.
- Highlight the power of the Mongo shell here.
- It is a fully programmable JavaScript environment.
 - To demonstrate this you can use the following code block

```
for(i=0;i<10;i++){ print("this is line "+i) }
```

Exploring Collections

```
show collections  
db.<COLLECTION>.help()  
db.<COLLECTION>.find()
```

Note:

- Show the collections available in this database.
 - Show methods on the collection with parameters and a brief explanation.
 - Finally, we can query for the documents in a collection.
-

Admin Commands

- There are also a number of admin commands at our disposal.
- The following will shut down the mongod we are connected to through the Mongo shell.
- You can also just kill with Ctrl-C in the shell window from which you launched the mongod.

```
db.adminCommand( { shutdown : 1 } )
```

- Confirm that the mongod process has indeed stopped.
- Once you have, please restart it.

2 CRUD

Creating and Deleting Documents (page 19) Inserting documents into collections, deleting documents, and dropping collections

Reading Documents (page 25) The find() command, query documents, dot notation, and cursors

Query Operators (page 33) MongoDB query operators including: comparison, logical, element, and array operators

Lab: Finding Documents (page 38) Exercises for querying documents in MongoDB

Updating Documents (page 39) Using update methods and associated operators to mutate existing documents

Lab: Updating Documents (page 49) Exercises for updating documents in MongoDB

2.1 Creating and Deleting Documents

Learning Objectives

Upon completing this module students should understand:

- How to insert documents into MongoDB collections.
- `_id` fields:
- How to delete documents from a collection
- How to remove a collection from a database
- How to remove a database from a MongoDB deployment

Creating New Documents

- Create documents using `insertOne()` and `insertMany()`.
- For example:

```
// Specify the collection name
db.<COLLECTION>.insertOne( { "name" : "Mongo" } )

// For example
db.people.insertOne( { "name" : "Mongo" } )
```

Example: Inserting a Document

Experiment with the following commands.

```
use sample  
  
db.movies.insertOne( { "title" : "Jaws" } )  
  
db.movies.find()
```

Note:

- Make sure the students are performing the operations along with you.
 - Some students will have trouble starting things up, so be helpful at this stage.
-

Implicit `_id` Assignment

- We did not specify an `_id` in the document we inserted.
- If you do not assign one, MongoDB will create one automatically.
- The value will be of type ObjectId.

Example: Assigning `_ids`

Experiment with the following commands.

```
db.movies.insertOne( { "_id" : "Jaws", "year" : 1975 } )  
db.movies.find()
```

Note:

- Note that you can assign an `_id` to be of almost any type.
 - It does not need to be an ObjectId.
-

Inserts will fail if...

- There is already a document in the collection with that `_id`.
- You try to assign an array to the `_id`.
- The argument is not a well-formed document.

Example: Inserts will fail if...

```
// fails because _id can't have an array value
db.movies.insertOne( { "_id" : [ "Star Wars",
                               "The Empire Strikes Back",
                               "Return of the Jedi" ] } )

// succeeds
db.movies.insertOne( { "_id" : "Star Wars" } )

// fails because of duplicate id
db.movies.insertOne( { "_id" : "Star Wars" } )

// malformed document
db.movies.insertOne( { "Star Wars" } )
```

Note:

- The following will fail because it attempts to use an array as an `_id`.

```
db.movies.insertOne( { "_id" : [ "Star Wars",
                               "The Empire Strikes Back",
                               "Return of the Jedi" ] } )
```

- The second insert with `_id : "Star Wars"` will fail because there is already a document with `_id` of "Star Wars" in the collection.
- The following will fail because it is a malformed document (i.e. no field name, just a value).

```
db.movies.insertOne( { "Star Wars" } )
```

`insertMany()`

- You may bulk insert using an array of documents.
- Use `insertMany()` instead of `insertOne()`

Note:

- In the case of an ordered bulk operation, every operation will be executed in the order they are added to the bulk operation.
- In the case of an unordered bulk operation however there is no guarantee what order the operations are executed.
- With an unordered bulk operation, the operations in the list may be reordered to increase performance.

Ordered insertMany()

- For ordered inserts MongoDB will stop processing inserts upon encountering an error.
- Meaning that only inserts occurring before an error will complete.
- The default setting for db.<COLLECTION>.insertMany is an ordered insert.
- See the next exercise for an example.

Example: Ordered insertMany()

Experiment with the following operation.

```
db.movies.insertMany( [ { "_id" : "Batman", "year" : 1989 },
                      { "_id" : "Home Alone", "year" : 1990 },
                      { "_id" : "Ghostbusters", "year" : 1984 },
                      { "_id" : "Ghostbusters", "year" : 1984 } ] )
db.movies.find()
```

Note:

- This example has a duplicate key error.
 - Only the first 3 documents will be inserted.
-

Unordered insertMany()

- Pass { ordered : false } to insertMany() to perform unordered inserts.
- If any given insert fails, MongoDB will still attempt all of the others.
- The inserts may be executed in a different order than you specified.
- The next exercise is very similar to the previous one.
- However, we are using { ordered : false }.
- One insert will fail, but all the rest will succeed.

Example: Unordered insertMany()

Experiment with the following insert.

```
db.movies.insertMany( [ { "_id" : "Jaws", "year" : 1975 },
                      { "_id" : "Titanic", "year" : 1997 },
                      { "_id" : "The Lion King", "year" : 1994 } ],
                      { ordered : false } )
db.movies.find()
```

The Shell is a JavaScript Interpreter

- Sometimes it is convenient to create test data using a little JavaScript.
- The mongo shell is a fully-functional JavaScript interpreter. You may:
 - Define functions
 - Use loops
 - Assign variables
 - Perform inserts

Exercise: Creating Data in the Shell

Experiment with the following commands.

```
for (i=1; i<=10000; i++) {  
    db.stuff.insert( { "a" : i } )  
}  
  
db.stuff.find()
```

Deleting Documents

You may delete documents from a MongoDB deployment in several ways.

- Use `deleteOne()` and `deleteMany()` to delete documents matching a specific set of conditions.
- Drop an entire collection.
- Drop a database.

Using `deleteOne()`

- Delete a document from a collection using `deleteOne()`
- This command has one required parameter, a query document.
- The first document in the collection matching the query document will be deleted.

Using `deleteMany()`

- Delete multiple documents from a collection using `deleteMany()`.
- This command has one required parameter, a query document.
- All documents in the collection matching the query document will be deleted.
- Pass an empty document to delete all documents.

Example: Deleting Documents

Experiment with removing documents. Do a `find()` after each `deleteMany()` command below.

```
for (i=1; i<=20; i++) { db.testcol.insertOne( { _id : i, a : i } ) }

db.testcol.deleteMany( { a : 1 } ) // Delete the first document

// $lt is a query operator that enables us to select documents that
// are less than some value. More on operators soon.
db.testcol.deleteMany( { a : { $lt : 5 } } ) // Remove three more

db.testcol.deleteOne( { a : { $lt : 10 } } ) // Remove one more

db.testcol.deleteMany() // Error: requires a query document.

db.testcol.deleteMany( { } ) // All documents removed
```

Dropping a Collection

- You can drop an entire collection with `db.<COLLECTION>.drop()`
- The collection and all documents will be deleted.
- It will also remove any metadata associated with that collection.
- Indexes are one type of metadata removed.
- **All collection and indexes files are removed and space allocated reclaimed.**
 - Wired Tiger only!
- More on meta data later.

Note: Mention that `drop()` is more performant than `deleteMany()`.

Example: Dropping a Collection

```
db.colToBeDropped.insertOne( { a : 1 } )
show collections // Shows the colToBeDropped collection

db.colToBeDropped.drop()
show collections // collection is gone
```

Dropping a Database

- You can drop an entire database with `db.dropDatabase()`
- This drops the database on which the method is called.
- It also deletes the associated data files from disk, freeing disk space.
- Beware that in the mongo shell, this does not change database context.

Example: Dropping a Database

```
use tempDB
db.testcoll.insertOne( { a : 1 } )
db.testcol2.insertOne( { a : 1 } )

show dbs // Here they are
show collections // Shows the two collections

db.dropDatabase()
show collections // No collections
show dbs // The db is gone

use sample // take us back to the sample db
```

2.2 Reading Documents

Learning Objectives

Upon completing this module students should understand:

- The query-by-example paradigm of MongoDB
- How to query on array elements
- How to query embedded documents using dot notation
- How the mongo shell and drivers use cursors
- Projections
- Cursor methods: `.count()`, `.sort()`, `.skip()`, `.limit()`

The `find()` Method

- This is the fundamental method by which we read data from MongoDB.
- We have already used it in its basic form.
- `find()` returns a cursor that enables us to iterate through all documents matching a query.
- We will discuss cursors later.

Query by Example

- To query MongoDB, specify a document containing the key / value pairs you want to match
- You need only specify values for fields you care about.
- Other fields will not be used to exclude documents.
- The result set will include all documents in a collection that match.

Example: Querying by Example

Experiment with the following sequence of commands.

```
db.movies.drop()
db.movies.insertMany( [
    { "title" : "Jaws", "year" : 1975, "imdb_rating" : 8.1 },
    { "title" : "Batman", "year" : 1989, "imdb_rating" : 7.6 }
] )
db.movies.find()

db.movies.find( { "year" : 1975 } )

// Multiple Batman movies from different years, find the correct one
db.movies.find( { "year" : 1989, "title" : "Batman" } )
```

Note: Matching Rules:

- Any field specified in the query must be in each document returned.
 - Values for returned documents must match the conditions specified in the query document.
 - If multiple fields are specified, all must be present in each document returned.
 - Think of it as a logical AND for all fields.
-

Querying Arrays

- In MongoDB you may query array fields.
- Specify a single value you expect to find in that array in desired documents.
- Alternatively, you may specify an entire array in the query document.
- As we will see later, there are also several operators that enhance our ability to query array fields.

Note: Students might find it helpful to think of an array field as having multiple values – one for each of its elements.

Example: Querying Arrays

```
db.movies.drop()
db.movies.insertMany([
  { "title" : "Batman", "category" : [ "action", "adventure" ] },
  { "title" : "Godzilla", "category" : [ "action", "adventure", "sci-fi" ] },
  { "title" : "Home Alone", "category" : [ "family", "comedy" ] }
])

// Match documents where "category" contains the value specified
db.movies.find( { "category" : "action" } )

// Match documents where "category" equals the value specified
db.movies.find( { "category" : [ "action", "sci-fi" ] } ) // no documents

// only the second document
db.movies.find( { "category" : [ "action", "adventure", "sci-fi" ] } )
```

Note: Later, we'll see operators that will allow us to do things like match all documents where an array field contains any of a set of values.

Querying with Dot Notation

- Dot notation is used to query on fields in embedded documents.
- The syntax is:

```
"field1.field2" : value
```

- Put quotes around the field name when using dot notation.

Example: Querying with Dot Notation

```
db.movies.insertMany([
  {
    "title" : "Avatar",
    "box_office" : { "gross" : 760,
                    "budget" : 237,
                    "opening_weekend" : 77
                  }
  },
  {
    "title" : "E.T.",
    "box_office" : { "gross" : 349,
                    "budget" : 10.5,
                    "opening_weekend" : 14
                  }
  }
])

db.movies.find( { "box_office" : { "gross" : 760 } } ) // no values

// dot notation
db.movies.find( { "box_office.gross" : 760 } ) // expected value
```

Example: Arrays and Dot Notation

```
db.movies.insertMany([
  { "title" : "E.T.",
    "filming_locations" :
      [ { "city" : "Culver City", "state" : "CA", "country" : "USA" },
        { "city" : "Los Angeles", "state" : "CA", "country" : "USA" },
        { "city" : "Cresecent City", "state" : "CA", "country" : "USA" }
      ],
  { "title": "Star Wars",
    "filming_locations" :
      [ { "city" : "Ajim", "state" : "Jerba", "country" : "Tunisia" },
        { "city" : "Yuma", "state" : "AZ", "country" : "USA" }
      ]
    }
])

db.movies.find( { "filming_locations.country" : "USA" } ) // two documents
```

Note:

- This query finds documents where:
 - There is a `filming_locations` field.
 - The `filming_locations` field contains one or more embedded documents.
 - At least one embedded document has a field `country`.
 - The field `country` has the specified value (“USA”).
 - In this collection, `filming_locations` is actually an array field.
 - The embedded documents we are matching are held within these arrays.
-

Projections

- You may choose to have only certain fields appear in result documents.
- This is called projection.
- You specify a projection by passing a second parameter to `find()`.

Projection: Example (Setup)

```
db.movies.insertOne(  
{  
    "title" : "Forrest Gump",  
    "category" : [ "drama", "romance" ],  
    "imdb_rating" : 8.8,  
    "filming_locations" : [  
        { "city" : "Savannah", "state" : "GA", "country" : "USA" },  
        { "city" : "Monument Valley", "state" : "UT", "country" : "USA" },  
        { "city" : "Los Angeles", "state" : "CA", "country" : "USA" }  
    ],  
    "box_office" : {  
        "gross" : 557,  
        "opening_weekend" : 24,  
        "budget" : 55  
    }  
})
```

Projection: Example

```
db.movies.findOne( { "title" : "Forrest Gump" },  
                  { "title" : 1, "imdb_rating" : 1 } )  
{  
    "_id" : ObjectId("5515942d31117f52a5122353"),  
    "title" : "Forrest Gump",  
    "imdb_rating" : 8.8  
}
```

Projection Documents

- Include fields with `fieldName: 1`.
 - Any field not named will be excluded
 - except `_id`, which must be explicitly excluded.
- Exclude fields with `fieldName: 0`.
 - Any field not named will be included.

Example: Projections

```
for (i=1; i<=20; i++) {
    db.movies.insertOne(
        { "_id" : i, "title" : i,
          "imdb_rating" : i, "box_office" : i } )
}
db.movies.find()
// no "box_office"
db.movies.find( { "_id" : 3 }, { "title" : 1, "imdb_rating" : 1 } )
// no "imdb_rating"
db.movies.find( { "_id" : { $gte : 10 } }, { "imdb_rating" : 0 } )
// just "title"
db.movies.find( { "_id" : 4 }, { "_id" : 0, "title" : 1 } )
// just "imdb_rating", "box_office"
db.movies.find( { "_id" : 5 }, { _id : 0, "title" : 0 } )
// Can't mix inclusion/exclusion except _id
db.movies.find( { "_id" : 6 }, { "title" : 1, "imdb_rating" : 0 } )
```

The last `find()` fails because MongoDB cannot determine how to handle unnamed fields such as `box_office`.

Cursors

- When you use `find()`, MongoDB returns a cursor.
- A cursor is a pointer to the result set
- You can get iterate through documents in the result using `next()`.
- By default, the mongo shell will iterate through 20 documents at a time.

Example: Introducing Cursors

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    db.testcol.insertOne( { a : Math.floor( Math.random() * 100 + 1 ),
                           b : Math.floor( Math.random() * 100 + 1 ) } )
}
db.testcol.find()

it
it
```

Note:

- With the `find()` above, the shell iterates over the first 20 documents.
 - `it` causes the shell to iterate over the next 20 documents.
 - Can continue issuing `it` commands until all documents are seen.
-

Example: Cursor Objects in the Mongo Shell

```
// Assigns the cursor returned by find() to a variable x
var x = db.testcol.find()

// Displays the first document in the result set.
x.next()

// True because there are more documents in the result set.
x.hasNext()

// Assigns the next document in the result set to the variable y.
y = x.next()

// Return value is the value of the a field of this document.
y.a

// Displaying a cursor prints the next 20 documents in the result set.
x
```

Cursor Methods

- `count()`: Returns the number of documents in the result set.
- `limit()`: Limits the result set to the number of documents specified.
- `skip()`: Skips the number of documents specified.

Example: Using `count()`

```
db.testcol.drop()
for (i=1; i<=100; i++) { db.testcol.insertOne( { a : i } ) }

// all 100
db.testcol.count()

// just 41 docs
db.testcol.count( { a : { $lt : 42 } } )

// Another way of writing the same query
db.testcol.find( { a : { $lt : 42 } } ).count()
```

Note:

- You may pass a query document like you would to `find()`.
 - `count()` will count only the documents matching the query.
 - Will return the number of documents in the collection if you do not specify a query document.
 - The last query in the above achieves the same result because it operates on the cursor returned by `find()`.
-

Example: Using sort ()

```
db.testcol.drop()
for (i=1; i<=20; i++) {
    db.testcol.insertOne( { a : Math.floor( Math.random() * 10 + 1 ),
                           b : Math.floor( Math.random() * 10 + 1 ) } )
}

db.testcol.find()

// sort descending; use 1 for ascending
db.testcol.find().sort( { a : -1 } )

// sort by b, then a
db.testcol.find().sort( { b : 1, a : 1 } )

// $natural order is just the order on disk.
db.testcol.find().sort( { $natural : 1 } )
```

Note:

- Sort can be executed on a cursor until the point where the first document is actually read.
 - If you never delete any documents or change their size, this will be the same order in which you inserted them.
 - Sorting two or more fields breaks the convention of javascript objects that key / value pairs are unordered.
 - In some drivers you may need to take special care with this.
 - For example, in Python, you would usually query with a dictionary.
 - But dictionaries are unordered in Python, so you would use an array of tuples instead.
-

The skip () Method

- Skips the specified number of documents in the result set.
- The returned cursor will begin at the first document beyond the number specified.
- Regardless of the order in which you specify skip () and sort () on a cursor, sort () happens first.

The limit () Method

- Limits the number of documents in a result set to the first k.
- Specify k as the argument to limit ()
- Regardless of the order in which you specify limit (), skip (), and sort () on a cursor, sort () happens first.
- Helps reduce resources consumed by queries.

The `distinct()` Method

- Returns all values for a field found in a collection.
- Only works on one field at a time.
- Input is a string (not a document)

Example: Using `distinct()`

```
db.movie_reviews.drop()
db.movie_reviews.insertMany( [
    { "title" : "Jaws", "rating" : 5 },
    { "title" : "Home Alone", "rating" : 1 },
    { "title" : "Jaws", "rating" : 7 },
    { "title" : "Jaws", "rating" : 4 },
    { "title" : "Jaws", "rating" : 8 } ] )
db.movie_reviews.distinct( "title" )
```

Note: Returns

```
{  
    "values" : [ "Jaws", "Home Alone" ],  
    "stats" : { ... },  
    "ok" : 1  
}
```

2.3 Query Operators

Learning Objectives

Upon completing this module students should understand the following types of MongoDB query operators:

- Comparison operators
- Logical operators
- Element query operators
- Operators on arrays

Comparison Query Operators

- \$lt: Exists and is less than
- \$lte: Exists and is less than or equal to
- \$gt: Exists and is greater than
- \$gte: Exists and is greater than or equal to
- \$ne: Does not exist or does but is not equal to
- \$in: Exists and is in a set
- \$nin: Does not exist or is not in a set

Example (Setup)

```
// insert sample data
db.movies.insertMany([
  {
    "title" : "Batman",
    "category" : [ "action", "adventure" ],
    "imdb_rating" : 7.6,
    "budget" : 35
  },
  {
    "title" : "Godzilla",
    "category" : [ "action",
      "adventure", "sci-fi" ],
    "imdb_rating" : 6.6
  },
  {
    "title" : "Home Alone",
    "category" : [ "family", "comedy" ],
    "imdb_rating" : 7.4
  }
])
```

Example: Comparison Operators

```
db.movies.find()

db.movies.find( { "imdb_rating" : { $gte : 7 } } )

db.movies.find( { "category" : { $ne : "family" } } )

db.movies.find( { "title" : { $in : [ "Batman", "Godzilla" ] } } )

db.movies.find( { "title" : { $nin : [ "Batman", "Godzilla" ] } } )
```

Logical Query Operators

- `$or`: Match either of two or more values
- `$not`: Used with other operators
- `$nor`: Match neither of two or more values
- `$and`: Match both of two or more values
 - This is the default behavior for queries specifying more than one condition.
 - Use `$and` if you need to include the same operator more than once in a query.

Example: Logical Operators

```
db.movies.find( { $or : [  
    { "category" : "sci-fi" }, { "imdb_rating" : { $gte : 7 } }  
] } )  
  
// more complex $or, really good sci-fi movie or mediocre family movie  
db.movies.find( { $or : [  
    { "category" : "sci-fi", "imdb_rating" : { $gte : 8 } },  
    { "category" : "family", "imdb_rating" : { $gte : 7 } }  
] } )  
  
// find bad movies  
db.movies.find( { "imdb_rating" : { $not : { $gt : 7 } } } )
```

Note:

- `db.movies.find({ "imdb_rating" : { $not : { $gt : 7 } } })` also returns everything that doesn't have an "imdb_rating"
-

Example: Logical Operators

```
// find movies within an imdb_rating range  
db.movies.find( { "imdb_rating" : { $gt : 5 , $lte : 7 } } ) // and is implicit  
  
// queries can be nested, why are there no results?  
db.movies.find( { $and : [  
    { $or : [  
        { "category" : "sci-fi", "imdb_rating" : { $gte : 8 } },  
        { "category" : "family", "imdb_rating" : { $gte : 7 } }  
    ] },  
    { $or : [  
        { "category" : "action", "imdb_rating" : { $gte : 6 } }  
    ] }  
] } )
```

Element Query Operators

- `$exists`: Select documents based on the existence of a particular field.
- `$type`: Select documents based on their type.
- See [BSON types³](#) for reference on types.

Example: Element Operators

```
db.movies.find( { "budget" : { $exists : true } } )  
  
// type 1 is Double  
db.movies.find( { "budget" : { $type : 1 } } )  
  
// type 3 is Object (embedded document)  
db.movies.find( { "budget" : { $type : 3 } } )
```

Array Query Operators

- `$all`: Array field must contain all values listed.
- `$size`: Array must have a particular size. E.g., `$size : 2` means 2 elements in the array
- `$elemMatch`: All conditions must be matched by at least one element in the array

Example: Array Operators

```
db.movies.find( { "category" : { $all : [ "sci-fi", "action" ] } } )  
  
db.movies.find( { "category" : { $size : 3 } } )
```

Example: \$elemMatch

```
db.movies.insertOne( {  
    "title" : "Raiders of the Lost Ark",  
    "filming_locations" : [  
        { "city" : "Los Angeles", "state" : "CA", "country" : "USA" },  
        { "city" : "Rome", "state" : "Lazio", "country" : "Italy" },  
        { "city" : "Florence", "state" : "SC", "country" : "USA" }  
    ] } )  
  
// This query is incorrect, it won't return what we want  
db.movies.find( {  
    "filming_locations.city" : "Florence",  
    "filming_locations.country" : "Italy"  
} )  
  
// $elemMatch is needed, now there are no results, this is expected  
db.movies.find( {  
    "filming_locations" : {  
        $elemMatch : {
```

³ <http://docs.mongodb.org/manual/reference/bson-types>

```
"city" : "Florence",
"country" : "Italy"
} } } )
```

Note:

- Comparing the last two queries demonstrates \$elemMatch.
-

2.4 Lab: Finding Documents

Exercise: student_id < 65

In the sample database, how many documents in the grades collection have a student_id less than 65?

Note:

- 650

```
db.grades.find( { student_id: { $lt: 65 } } ).count()
```

Exercise: Inspection Result “Fail” & “Pass”

In the sample database, how many documents in the inspections collection have result “Pass” or “Fail”?

Note:

- 16808

```
db.inspections.find({ "result": { $in: [ "Pass", "Fail" ] } }).count()
```

Exercise: View Count > 1000

In the stories collection, write a query to find all stories where the view count is greater than 1000.

Note:

- Requires querying into subdocuments

```
db.stories.find( { "shorturl.view_count": { $gt: 1000 } } )
```

Exercise: Most comments

Find the news article that has the most comments in the stories collection

Note:

- You can .limit() with .sort()

```
db.stories.find({media:"news"}).sort({comments:-1}).limit(1)[0].comments
```

Exercise: Television or Videos

Find all digg stories where the topic name is “Television” or the media type is “videos”. Skip the first 5 results and limit the result set to 10.

Note:

```
db.stories.find( { "$or": [ { "topic.name": "Television" },
                           { media: "videos" } ] } ).skip(5).limit(10)
```

Exercise: News or Images

Query for all digg stories whose media type is either “news” or “images” and where the topic name is “Comedy”. (For extra practice, construct two queries using different sets of operators to do this.)

Note:

```
db.stories.find( { media: { $in: [ "news", "images" ] },
                   "topic.name": "Comedy" } )
```

2.5 Updating Documents

Learning Objectives

Upon completing this module students should understand

- The `replaceOne()` method
- The `updateOne()` method
- The `updateMany()` method
- The required parameters for these methods
- Field update operators
- Array update operators
- The concept of an upsert and use cases.
- The `findOneAndReplace()` and `findOneAndUpdate()` methods

The `replaceOne()` Method

- Takes one document and replaces it with another
 - But leaves the `_id` unchanged
- Takes two parameters:
 - A matching document
 - A replacement document
- This is, in some sense, the simplest form of update

Note:

- By “simplest,” we mean that it’s simple conceptually – that replacing a document is a sort of basic idea of how an update happens.
 - We will later see update methods that will involve only changing some fields.
-

First Parameter to `replaceOne()`

- Required parameters for `replaceOne()`
 - The query parameter:
 - * Use the same syntax as with `find()`
 - * Only the first document found is replaced
- `replaceOne()` cannot delete a document

Second Parameter to `replaceOne()`

- The second parameter is the replacement parameter:
 - The document to replace the original document
- The `_id` must stay the same
- You must replace the entire document
 - You cannot modify just one field
 - Except for the `_id`

Note:

- If they try to modify the `_id`, it will throw an error
-

Example: `replaceOne()`

```
db.movies.insertOne( { title: "Batman" } )
db.movies.find()
db.movies.replaceOne( { title : "Batman" }, { imdb_rating : 7.7 } )
db.movies.find()
db.movies.replaceOne( { imdb_rating: 7.7 },
                      { title: "Batman", imdb_rating: 7.7 } )
db.movies.find()
db.movies.replaceOne( { }, { title: "Batman" } )
db.movies.find() // back in original state
db.movies.replaceOne( { }, { _id : ObjectId() } )
```

Note:

- Ask the students why the first replace killed the `title` field
 - Ask why the final replace failed
-

The `updateOne()` Method

- Mutate one document in MongoDB using `updateOne()`
 - Affects only the `_first_` document found
- Two parameters:
 - A query document
 - * same syntax as with `find()`
 - Change document
 - * Operators specify the fields and changes

`$set` and `$unset`

- Use to specify fields to update for `UpdateOne()`
- If the field already exists, using `$set` will change its value
 - If not, `$set` will create it, set to the new value
- Only specified fields will change
- Alternatively, remove a field using `$unset`

Example (Setup)

```
db.movies.insertMany( [
  {
    "title" : "Batman",
    "category" : [ "action", "adventure" ],
    "imdb_rating" : 7.6,
    "budget" : 35
  },
  {
    "title" : "Godzilla",
    "category" : [ "action", "adventure", "sci-fi" ],
    "imdb_rating" : 6.6
  },
  {
    "title" : "Home Alone",
    "category" : [ "family", "comedy" ],
    "imdb_rating" : 7.4
  }
])
```

Example: \$set and \$unset

```
db.movies.updateOne( { "title" : "Batman" },
                      { $set : { "imdb_rating" : 7.7 } } )
db.movies.updateOne( { "title" : "Godzilla" },
                      { $set : { "budget" : 1 } } )
db.movies.updateOne( { "title" : "Home Alone" },
                      { $set : { "budget" : 15,
                                 "imdb_rating" : 5.5 } } )
db.movies.updateOne( { "title" : "Home Alone" },
                      { $unset : { "budget" : 1 } } )
db.movies.find()
```

Update Operators

- **\$inc:** Increment a field's value by the specified amount.
- **\$mul:** Multiply a field's value by the specified amount.
- **\$rename:** Rename a field.
- **\$set:** Update one or more fields (already discussed).
- **\$unset:** Delete a field (already discussed).
- **\$min:** Updates the field value to a specified value if the specified value is less than the current value of the field
- **\$max:** Updates the field value to a specified value if the specified value is greater than the current value of the field
- **\$currentDate:** Set the value of a field to the current date or timestamp.

Example: Update Operators

```
db.movies.updateOne( { title: "Batman" }, { $inc: { "imdb_rating" : 2 } } )
db.movies.updateOne( { title: "Home Alone" }, { $inc: { "budget" : 5 } } )
db.movies.updateOne( { title: "Batman" }, { $mul: { "imdb_rating" : 4 } } )
db.movies.updateOne( { title: "Batman" },
                     { $rename: { budget: "estimated_budget" } } )
db.movies.updateOne( { title: "Home Alone" }, { $min: { budget: 5 } } )
db.movies.updateOne( { title: "Home Alone" },
                     { $currentDate : { last_updated: { $type: "timestamp" } } } )
// increment movie mentions by 10
db.movie_mentions.updateOne( { title: "E.T." },
                             { $inc: { "mentions_per_hour.5" : 10 } } )
```

The updateMany() Method

- Takes the same arguments as updateOne
- Updates all documents that match
 - updateOne stops after the first match
 - updateMany continues until it has matched all

Warning: Without an appropriate index, you may scan every document in the collection.

Example: updateMany()

```
// let's start tracking the number of sequels for each movie
db.movies.updateOne( { }, { $set : { "sequels" : 0 } } )
db.movies.find()
// we need updateMany to change all documents
db.movies.updateMany( { }, { $set : { "sequels" : 0 } } )
db.movies.find()
```

Array Element Updates by Index

- You can use dot notation to specify an array index
- You will update only that element
 - Other elements will not be affected

Example: Update Array Elements by Index

```
// add a sample document to track mentions per hour
db.movie_mentions.insertOne(
  { "title" : "E.T.",
    "day" : ISODate("2015-03-27T00:00:00.000Z"),
    "mentions_per_hour" : [ 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0 ]
  }
)

// update all mentions for the fifth hour of the day
db.movie_mentions.updateOne(
  { "title" : "E.T." },
  { $set : { "mentions_per_hour.5" : 2300 } } )
```

Note:

- Pattern for time series data
 - Displaying charts is easy
 - Can change granularity to by the minute, hour, day, etc.
-

Array Operators

- `$push`: Appends an element to the end of the array.
 - `$pushAll`: Appends multiple elements to the end of the array.
 - `$pop`: Removes one element from the end of the array.
 - `$pull`: Removes all elements in the array that match a specified value.
 - `$pullAll`: Removes all elements in the array that match any of the specified values.
 - `$addToSet`: Appends an element to the array if not already present.
-

Note:

- These operators may be applied to array fields.
-

Example: Array Operators

```
db.movies.updateOne(  
  { "title" : "Batman" },  
  { $push : { "category" : "superhero" } } )  
db.movies.updateOne(  
  { "title" : "Batman" },  
  { $pushAll : { "category" : [ "villain", "comic-based" ] } } )  
db.movies.updateOne(  
  { "title" : "Batman" },  
  { $pop : { "category" : 1 } } )  
db.movies.updateOne(  
  { "title" : "Batman" },  
  { $pull : { "category" : "action" } } )  
db.movies.updateOne(  
  { "title" : "Batman" },  
  { $pullAll : { "category" : [ "villain", "comic-based" ] } } )
```

Note:

- Pass \$pop a value of -1 to remove the first element of an array and 1 to remove the last element in an array.
-

The Positional \$ Operator

- $\4 is a positional operator that specifies an element in an array to update.
- It acts as a placeholder for the first element that matches the query document.
- \$ replaces the element in the specified position with the value given.
- Example:

```
db.<COLLECTION>.updateOne(  
  { <array> : value ... },  
  { <update operator> : { "<array>.$" : value } }  
)
```

Example: The Positional \$ Operator

```
// the "action" category needs to be changed to "action-adventure"  
db.movies.updateMany( { "category": "action", },  
  { $set: { "category.$" : "action-adventure" } } )
```

⁴ <http://docs.mongodb.org/manual/reference/operator/update/postional>

Upserts

- If no document matches a write query:
 - By default, nothing happens
 - With `upsert: true`, inserts one new document
- Works for `updateOne()`, `updateMany()`, `replaceOne()`
- Syntax:

```
db.<COLLECTION>.updateOne( <query document>,
                            <update document>,
                            { upsert: true } )
```

Upsert Mechanics

- Will update if documents matching the query exist
- Will insert if no documents match
 - Creates a new document using equality conditions in the query document
 - Adds an `_id` if the query did not specify one
 - Performs the write on the new document
- `updateMany()` will only create one document
 - If none match, of course

Example: Upserts

```
db.movies.updateOne( { "title" : "Jaws" },
                      { $inc: { "budget" : 5 } },
                      { upsert: true } )

db.movies.updateMany( { "title" : "Jaws II" },
                      { $inc: { "budget" : 5 } },
                      { upsert: true } )

db.movies.replaceOne( { "title" : "E.T.", "category" : [ "scifi" ] },
                      { "title" : "E.T.", "category" : [ "scifi" ], "budget" : 1 },
                      { upsert: true } )
```

Note:

- Note that an `updateMany` works just like `updateOne` when no matching documents are found.
 - First query updates the document with “`title`” = “`Jaws`” by incrementing “`budget`”
 - Second query: 1) creates a new document, 2) assigns an `_id`, 3) sets “`title`” to “`Jaws II`” 4) performs the update
 - Third query: 1) creates a new document, 2) sets “`title`” : “`Jaws III`”, 3) Set budget to 1
-

save()

- The `db.<COLLECTION>.save()` method is syntactic sugar
 - Similar to `replaceOne()`, querying the `_id` field
 - Upsert if `_id` is not in the collection
- Syntax:

```
db.<COLLECTION>.save( <document> )
```

Example: save()

- If the document in the argument does not contain an `_id` field, then the `save()` method acts like `insertOne()` method
 - An ObjectId will be assigned to the `_id` field.
- If the document in the argument contains an `_id` field: then the `save()` method is equivalent to a `replaceOne()` with the query argument on `_id` and the upsert option set to true

```
// insert
db.movies.save( { "title" : "Beverly Hills Cops", "imdb_rating" : 7.3 } )

// update with { upsert: true }
db.movies.save( { "_id" : 1234, "title" : "Spider Man", "imdb_rating" : 7.3 } )
```

Note:

- A lot of users prefer to use update/insert, to have more explicit control over the operation
-

Be careful with save()

Careful not to modify stale data when using `save()`. Example:

```
db.movies.drop()
db.movies.insertOne( { "title" : "Jaws", "imdb_rating" : 7.3 } )

db.movies.find( { "title" : "Jaws" } )

// store the complete document in the application
doc = db.movies.findOne( { "title" : "Jaws" } )

db.movies.updateOne( { "title" : "Jaws" }, { $inc: { "imdb_rating" : 2 } } )
db.movies.find()

doc.imdb_rating = 7.4

db.movies.save(doc) // just lost our incrementing of "imdb_rating"
db.movies.find()
```

findOneAndUpdate () and findOneAndReplace ()

- Update (or replace) one document and return it
 - By default, the document is returned pre-write
- Can return the state before or after the update
- Makes a read plus a write atomic
- Can be used with upsert to insert a document

findOneAndUpdate () and findOneAndReplace () Options

- The following are optional fields for the options document
- `projection: <document>` - select the fields to see
- `sort: <document>` - sort to select the first document
- `maxTimeoutMS: <number>` - how long to wait
 - Returns an error, kills operation if exceeded
- `upsert: <boolean>` if true, performs an upsert

Example: findOneAndUpdate ()

```
db.worker_queue.findOneAndUpdate(
  { state : "unprocessed" },
  { $set: { "worker_id" : 123, "state" : "processing" } },
  { upsert: true } )
```

findOneAndDelete ()

- Not an update operation, but fits in with findOneAnd ...
- Returns the document and deletes it.
- Example:

```
db.foo.drop();
db.foo.insertMany( [ { a : 1 }, { a : 2 }, { a : 3 } ] );
db.foo.find(); // shows the documents.
db.foo.findOneAndDelete( { a : { $lte : 3 } } );
db.foo.find();
```

2.6 Lab: Updating Documents

Exercise: Pass Inspections

In the sample.inspections namespace, let's imagine that we want to do a little data cleaning. We've decided to eliminate the "Completed" inspection result and use only "No Violation Issued" for such inspection cases. Please update all inspections accordingly.

Note:

```
db.inspections.updateMany({result: "Completed"},  
                           {$set: {result: "No Violation Issued"}})  
{  
  "acknowledged": true,  
  "matchedCount": 20,  
  "modifiedCount": 20  
}
```

Exercise: Set fine value

For all inspections that failed, set a `fine` value of 100.

Note:

```
db.inspections.updateMany({result: "Fail"},  
                           {$set: {fine: 100}})  
{  
  "acknowledged": true,  
  "matchedCount": 1120,  
  "modifiedCount": 1120  
}
```

Exercise: Increase fine in ROSEDALE

- Update all inspections done in the city of "ROSEDALE".
- For failed inspections, raise the "fine" value by 150.

Note:

```
db.inspections.updateMany({"address.city": "ROSEDALE", result: "Fail"},  
                           {$inc: {fine: 150}})  
{  
  "acknowledged": true,  
  "matchedCount": 1120,  
  "modifiedCount": 1120  
}
```

Exercise: Give a pass to “MONGODB”

- Today MongoDB got a visit from the inspectors.
- We passed, of course.
- So go ahead and update “MongoDB” and set the result to “AWESOME”
- MongoDB’s address is

```
{city: 'New York', zip: 10036, street: '43', number: 229}
```

Note:

```
db.inspections.updateOne({business_name: "MongoDB"},  
    {$set: {  
        address: {  
            city: "New York",  
            zip: 10036,  
            street: "43",  
            number: 229 },  
        result: "AWESOME",  
        id: "XXXXXXX",  
        certificate_number: 140021221},  
    $currentDate: {date: {$type: "date"}},  
    {upsert: true})  
{  
    "acknowledged" : true,  
    "matchedCount" : 0,  
    "modifiedCount" : 0,  
    "upsertedId" : ObjectId("573f29d8dc8e6b0ba6e8f594")  
}
```

We can also add a variation to see if students can determine how to sort results so they can look at certificate numbers granted in sequence. Kudos to students that recognize the need to filter for certificate_number values that are integers and also do some form of projection.

```
db.inspections.find(  
    {certificate_number: {$type:16}},  
    {certificate_number: 1,  
     id:1}).sort({certificate_number:-1}).limit(1)
```

Exercise: Updating Array Elements

Insert a document representing product metrics for a backpack:

```
db.product_metrics.insertOne(  
    { name: "backpack",  
      purchasesPast7Days: [ 0, 0, 0, 0, 0, 0, 0 ] })
```

Each 0 within the “purchasesPast7Days” field corresponds to a day of the week. The first element is Monday, the second element is Tuesday, etc.).

Write an update statement to increment the number of backpacks sold on Friday by 200.

Note:

- Talk about how this can be used for time series data, real-time graphs/charts

```
db.product_metrics.updateOne(  
    {name: "backpack"},  
    {$inc: { "purchases_past_7_days.4" : 200 } } )
```

3 Indexes

Index Fundamentals (page 52) An introduction to MongoDB indexes

Compound Indexes (page 60) Indexes on two or more fields

Lab: Optimizing an Index (page 67) Lab on optimizing a compound index

Multikey Indexes (page 68) Indexes on array fields

Hashed Indexes (page 73) Hashed indexes

Geospatial Indexes (page 74) Geospatial indexes: both those on legacy coordinate pairs and those supporting queries that calculate geometries on an earth-like sphere.

TTL Indexes (page 82) Time-To-Live indexes

Text Indexes (page 83) Free text indexes on string fields

Lab: Finding and Addressing Slow Operations (page 86) Lab on finding and addressing slow queries

Lab: Using explain() (page 87) Lab on using the explain operation to review execution stats

3.1 Index Fundamentals

Learning Objectives

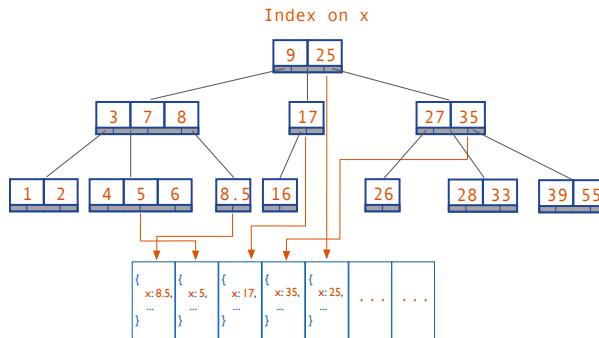
Upon completing this module students should understand:

- The impact of indexing on read performance
- The impact of indexing on write performance
- How to choose effective indexes
- The utility of specific indexes for particular query patterns

Note:

- Ask how many people in the room are familiar with indexes in a relational database.
 - If the class is already familiar with indexes, just explain that they work the same way in MongoDB.
-

Why Indexes?



Note:

- Without an index, in order to find all documents matching a query, MongoDB must scan every document in the collection.
 - This is murder for read performance, and often write performance, too.
 - If all your documents do not fit into memory, the system will page data in and out in order to scan the entire collection.
 - An index enables MongoDB to locate exactly which documents match the query and where they are located on disk.
 - MongoDB indexes are based on B-trees.
-

Types of Indexes

- Single-field indexes
- Compound indexes
- Multikey indexes
- Geospatial indexes
- Text indexes

Note:

- There are also hashed indexes and TTL indexes.
 - We will discuss those elsewhere.
-

Exercise: Using `explain()`

Let's explore what MongoDB does for the following query by using `explain()`.

We are projecting only `user.name` so that the results are easy to read.

```
db.tweets.find( { "user.followers_count" : 1000 },
                 { "_id" : 0, "user.name": 1 } )

db.tweets.find( { "user.followers_count" : 1000 } ).explain()
```

Note:

- Make sure the students are using the sample database.
 - Review the structure of documents in the tweets collection by doing a `find()`.
 - We'll be looking at the `user` subdocument for documents in this collection.
-

Results of `explain()`

With the default `explain()` verbosity, you will see results similar to the following:

```
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "twitter.tweets",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "user.followers_count" : {
        "$eq" : 1000
      }
    },
  },
```

Results of `explain()` - Continued

```
"winningPlan" : {
  "stage" : "COLLSCAN",
  "filter" : {
    "user.followers_count" : {
      "$eq" : 1000
    }
  },
  "direction" : "forward"
},
"rejectedPlans" : [ ]
},
```

...

`explain()` Verbosity Can Be Adjusted

- **default:** determines the winning query plan but does not execute query
- **executionStats:** executes query and gathers statistics
- **allPlansExecution:** runs all candidate plans to completion and gathers statistics

Note:

- Default will be helpful if you're worried running the query could cause sever performance problems
 - executionStats will be the most common verbosity level used
 - allPlansExecution is for trying to determine WHY it is choosing the index it is (out of other candidates)
-

```
explain("executionStats")
```

```
> db.tweets.find( { "user.followers_count" : 1000 } )  
.explain("executionStats")
```

Now we have query statistics:

```
..  
"executionStats" : {  
    "executionSuccess" : true,  
    "nReturned" : 8,  
    "executionTimeMillis" : 107,  
    "totalKeysExamined" : 0,  
    "totalDocsExamined" : 51428,  
    "executionStages" : {  
        "stage" : "COLLSCAN",  
        "filter" : {  
            "user.followers_count" : {  
                "$eq" : 1000  
            }  
        },  
    },
```

`explain("executionStats")` - Continued

```
"nReturned" : 8,  
"executionTimeMillisEstimate" : 100,  
"works" : 51430,  
"advanced" : 8,  
"needTime" : 51421,  
"needFetch" : 0,  
"saveState" : 401,  
"restoreState" : 401,  
"isEOF" : 1,  
"invalidates" : 0,  
"direction" : "forward",  
"docsExamined" : 51428  
}  
...  
}
```

`explain("executionStats")` Output

- `nReturned` : number of documents returned by the query
- `totalDocsExamined` : number of documents touched during the query
- `totalKeysExamined` : number of index keys scanned
- A `totalKeysExamined` or `totalDocsExamined` value much higher than `nReturned` indicates we need a better index
- Based `.explain()` output, this query would benefit from a better index

Note:

- By documents “touched”, we mean that they had to be in memory (either already there, or else loaded during the query)
 - By “better” index, we mean one that matches the query more closely.
-

Other Operations

In addition to `find()`, we often want to use `explain()` to understand how other operations will be handled.

- `aggregate()`
- `count()`
- `group()`
- `update()`
- `remove()`
- `findAndModify()`
- `insert()`

Note:

- Has not yet been implemented for the new CRUD API.
 - No `updateOne()`, `replaceOne()`, `updateMany()`, `deleteOne()`, `deleteMany()`, `findOneAndUpdate()`, `findOneAndDelete()`, `findOneAndReplace()`, `insertMany()`
-

```
db.<COLLECTION>.explain()
```

db.<COLLECTION>.explain() returns an ExplainableCollection.

```
> var explainable = db.tweets.explain()  
> explainable.find( { "user.followers_count" : 1000 } )
```

equivalent to

```
> db.tweets.explain().find( { "user.followers_count" : 1000 } )
```

also equivalent to

```
> db.tweets.find( { "user.followers_count" : 1000 } ).explain()
```

Note:

- This will get confusing for students, may want to spend extra time here with more examples
-

Using `explain()` for Write Operations

Simulate the number of writes that would have occurred and determine the index(es) used:

```
> db.tweets.explain("executionStats").remove( { "user.followers_count" : 1000 } )
```

```
> db.tweets.explain("executionStats").update( { "user.followers_count" : 1000 },  
  { $set : { "large_following" : true } }, { multi: true } )
```

Note:

- Walk through the “nWouldModify” field in the output to show how many documents would have been updated
-

Single-Field Indexes

- Single-field indexes are based on a single field of the documents in a collection.
- The field may be a top-level field.
- You may also create an index on fields in embedded documents.

Creating an Index

The following creates a single-field index on `user.followers_count`.

```
db.tweets.createIndex( { "user.followers_count" : 1 } )
db.tweets.find( { "user.followers_count" : 1000 } ).explain()
```

`explain()` indicated there will be a substantial performance improvement in handling this type of query.

Note:

- `nscannedObjects` should now be a much smaller number, e.g., 8.
 - Operations teams are accustomed to thinking about indexes.
 - With MongoDB, developers need to be more involved in the creation and use of indexes.
-

Listing Indexes

List indexes for a collection:

```
db.tweets.getIndexes()
```

List index keys:

```
db.tweets.getIndexKeys()
```

Indexes and Read/Write Performance

- Indexes improve read performance for queries that are supported by the index.
- Inserts will be slower when there are indexes that MongoDB must also update.
- The speed of updates may be improved because MongoDB will not need to do a collection scan to find target documents.
- An index is modified any time a document:
 - Is inserted (applies to *all* indexes)
 - Is deleted (applies to *all* indexes)
 - Is updated in such a way that its indexed field changes

Note:

- For `mmapv1`, all indexes will be modified whenever the document moves on disk
 - i.e., When it outgrows its record space
-

Index Limitations

- You can have up to 64 indexes per collection.
- You should NEVER be anywhere close to that upper bound.
- Write performance will degrade to unusable at somewhere between 20-30.

Use Indexes with Care

- Every query should use an index.
- Every index should be used by a query.
- Any write that touches an indexed field will update every index that touches that field.
- Indexes require RAM.
- Be mindful about the choice of key.

Note:

- If your system has limited RAM, then using the index will force other data out of memory.
 - When you need to access those documents, they will need to be paged in again.
-

Additional Index Options

- Sparse
- Unique
- Background

Sparse Indexes in MongoDB

- Sparse indexes only contain entries for documents that have the indexed field.

```
db.<COLLECTION>.createIndex ( 
  { field_name : 1 },
  { sparse : true } )
```

Note:

- Partial indexes are now preferred to sparse.
 - You can create the functional equivalent of a sparse index with { field : { \$exists : true } } for your partialFilterExpression.
-

Defining Unique Indexes

- Enforce a unique constraint on the index
 - On a per-collection basis
- Can't insert documents with a duplicate value for the field
 - Or update to a duplicate value
- No duplicate values may exist prior to defining the index

```
db.<COLLECTION>.createIndex(  
  { field_name : 1 },  
  { unique : true } )
```

Building Indexes in the Background

- Building indexes in foreground is a blocking operation.
- Background index creation is non-blocking, however, takes longer to build.
- Initially larger, or less compact, than an index built in the foreground.

```
db.<COLLECTION>.createIndex(  
  { field_name : 1 },  
  { background : true } )
```

3.2 Compound Indexes

Learning Objectives

Upon completing this module students should understand:

- What a compound index is.
- How compound indexes are created.
- The importance of considering field order when creating compound indexes.
- How to efficiently handle queries involving some combination of equality matches, ranges, and sorting.
- Some limitations on compound indexes.

Introduction to Compound Indexes

- It is common to create indexes based on more than one field.
- These are called compound indexes.
- You may use up to 31 fields in a compound index.
- You may not use hashed index fields.

The Order of Fields Matters

Specifically we want to consider how the index will be used for:

- Equality tests, e.g.,

```
db.movies.find( { "budget" : 7, "imdb_rating" : 8 } )
```

- Range queries, e.g.,

```
db.movies.find( { "budget" : 10, "imdb_rating" : { $lt : 9 } } )
```

- Sorting, e.g.,

```
db.movies.find( { "budget" : 10, "imdb_rating" : 6 } )  
    .sort( { "imdb_rating" : -1 } )
```

Note:

- The order in which the fields are specified is of critical importance.
 - It is especially important to consider query patterns that require two or more of these operations.
-

Designing Compound Indexes

- Let's look at some guiding principles for building compound indexes.
- These will generally produce a good if not optimal index.
- You can optimize after a little experimentation.
- We will explore this in the context of a running example.

Example: A Simple Message Board

Requirements:

- Find all messages in a specified timestamp range.
- Select for whether the messages are anonymous or not.
- Sort by rating from highest to lowest.

Load the Data

```
a = [ { "timestamp" : 1, "username" : "anonymous", "rating" : 3 },
      { "timestamp" : 2, "username" : "anonymous", "rating" : 5 },
      { "timestamp" : 3, "username" : "sam", "rating" : 1 },
      { "timestamp" : 4, "username" : "anonymous", "rating" : 2 },
      { "timestamp" : 5, "username" : "martha", "rating" : 5 } ]
db.messages.insertMany(a)
```

Start with a Simple Index

Start by building an index on { timestamp : 1 }

```
db.messages.createIndex( { timestamp : 1 }, { name : "myindex" } )
```

Now let's query for messages with timestamp in the range 2 through 4 inclusive.

```
db.messages.find( { timestamp : { $gte : 2, $lte : 4 } } ).explain("executionStats")
```

Analysis:

- Explain plan shows good performance, i.e. totalKeysExamined = n.
- However, this does not satisfy our query.
- Need to query again with {username: "anonymous"} as part of the query.

Query Adding username

Let's add the user field to our query.

```
db.messages.find( { timestamp : { $gte : 2, $lte : 4 },
                    username : "anonymous" } ).explain("executionStats")
```

totalKeysExamined > n.

Include `username` in Our Index

```
db.messages.dropIndex( "myindex" );
db.messages.createIndex( { timestamp : 1, username : 1 },
                        { name : "myindex" } )
db.messages.find( { timestamp : { $gte : 2, $lte : 4 },
                    username : "anonymous" } ).explain("executionStats")
```

`totalKeysExamined` is still > n. Why?

totalKeysExamined > n

timestamp	username
1	"anonymous"
2	"anonymous"
3	"sam"
4	"anonymous"
5	"martha"

Note:

- The index we have created stores the range values before the equality values.
 - The documents with timestamp values 2, 3, and 4 were found first.
 - Then the associated anonymous values had to be evaluated.
-

A Different Compound Index

Drop the index and build a new one with user.

```
db.messages.dropIndex( "myindex" );
db.messages.createIndex( { username : 1 , timestamp : 1 },
                        { name : "myindex" } )

db.messages.find( { timestamp : { $gte : 2, $lte : 4 },
                    username : "anonymous" } ).explain("executionStats")
```

`totalKeysExamined` is 2. n is 2.

```
totalKeysExamined == n
```

username	timestamp
“anonymous”	1
“anonymous”	2
“anonymous”	4
“sam”	2
“martha”	5

Note:

- This illustrates why.
 - There is a fundamental difference in the way the index is structured.
 - This supports a more efficient treatment of our query.
-

Let Selectivity Drive Field Order

- Order fields in a compound index from most selective to least selective.
- Usually, this means equality fields before range fields.
- When dealing with multiple equality values, start with the most selective.
- If a common range query is more selective instead (rare), specify the range component first.

Adding in the Sort

Finally, let's add the sort and run the query

```
db.messages.find( {  
    timestamp : { $gte : 2, $lte : 4 },  
    username : "anonymous"  
} ).sort( { rating : -1 } ).explain("executionStats");
```

- Note that the winningPlan includes a SORT stage
- This means that MongoDB had to perform a sort in memory
- In memory sorts can degrade performance significantly
 - Especially if used frequently
 - In-memory sorts that use > 32 MB will abort

In-Memory Sorts

Let's modify the index again to allow the database to sort for us.

```
db.messages.dropIndex( "myindex" );
db.messages.createIndex( { username : 1 , timestamp : 1, rating : 1 },
                        { name : "myindex" } );
db.messages.find( {
    timestamp : { $gte : 2, $lte : 4 },
    username : "anonymous"
} ).sort( { rating : -1 } ).explain("executionStats");
```

- The explain plan remains unchanged, because the sort field comes after the range fields.
- The index does not store entries in order by rating.
- Note that this requires us to consider a tradeoff.

Avoiding an In-Memory Sort

Rebuild the index as follows.

```
db.messages.dropIndex( "myindex" );
db.messages.createIndex( { username : 1, rating : 1, timestamp : 1 },
                        { name : "myindex" } );
db.messages.find( {
    timestamp : { $gte : 2, $lte : 4 },
    username : "anonymous"
} ).sort( { rating : -1 } ).explain("executionStats");
```

- We no longer have an in-memory sort, but need to examine more keys.
- `totalKeysExamined` is 3 and `n` is 2.
- This is the best we can do in this situation and this is fine.
- However, if `totalKeysExamined` is much larger than `n`, this might not be the best index.

No need for stage : SORT

username	rating	timestamp
“anonymous”	2	4
“anonymous”	3	1
“anonymous”	5	2
“sam”	1	2
“martha”	5	5

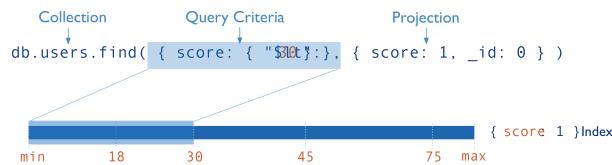
Note:

- general index illustration
 - we can see that the returned results are already sorted.
 - no need to perform an in-memory sort
-

General Rules of Thumb

- Equality before range
- Equality before sorting
- Sorting before range

Covered Queries



- When a query and projection include only the indexed fields, MongoDB will return results directly from the index.
- There is no need to scan any documents or bring documents into memory.
- These covered queries can be very efficient.

Exercise: Covered Queries

```
db.testcol.drop()
for (i=1; i<=20; i++) {
  db.testcol.insertOne({ "_id" : i, "title" : i, "name" : i,
                        "rating" : i, "budget" : i })
}
db.testcol.createIndex( { "title" : 1, "name" : 1, "rating" : 1 } )

// Not covered because _id is present.
db.testcol.find( { "title" : 3 },
                 { "title" : 1, "name" : 1, "rating" : 1 }
               ).explain("executionStats")

// Not covered because other fields may exist in matching docs.
db.testcol.find( { "title" : 3 },
                 { "_id" : 0, "budget" : 0 } ).explain("executionStats")

// Covered query!
db.testcol.find( { "title" : 3 },
                 { "_id" : 0, "title" : 1, "name" : 1, "rating" : 1 }
               ).explain("executionStats")
```

3.3 Lab: Optimizing an Index

Exercise: What Index Do We Need?

Run the the following Javascript file from the handouts.

```
mongo --shell localhost/performance performance.js
```

In the shell that launches execute the following method

```
performance.init()
```

The method above will build a sample data set in the “sensor_readings” collection. What index is needed for this query?

```
db.sensor_readings.find( { tstamp: { $gte: ISODate("2012-08-01"),
                                     $lte: ISODate("2012-09-01") },
                           active: true } ).limit(3)
```

Note:

- Work through method of explaining query with .explain(“executionStats”)
 - Look at differences between (timestamp, active) and (active, timestamp)
-

Exercise: Avoiding an In-Memory Sort

What index is needed for the following query to avoid an in-memory sort?

```
db.sensor_readings.find( { active: true } ).sort( { tstamp : -1 } )
```

Note:

- { active: 1, tstamp: 1 }
-

Exercise: Avoiding an In-Memory Sort, 2

What index is needed for the following query to avoid an in-memory sort?

```
db.sensor_readings.find(
  { x : { $in : [100, 200, 300, 400] } }
).sort( { tstamp : -1 } )
```

Note:

- Trick question, the answer most students will give is { x: 1, tstamp: 1 }, however, the \$in will require an in-memory sort
 - (tstamp) or (tstamp, x) are the only indexes that will prevent an in-memory sort, but aren’t selective at all
-

3.4 Multikey Indexes

Learning Objectives

Upon completing this module, students should understand:

- What a multikey index is
- When MongoDB will use a multikey index to satisfy a query
- How multikey indexes work
- How multikey indexes handle sorting
- Some limitations on multikey indexes

Introduction to Multikey Indexes

- A multikey index is an index on an array.
- An index entry is created on each value found in the array.
- Multikey indexes can support primitives, documents, or sub-arrays.
- There is nothing special that you need to do to create a multikey index.
- You create them using `createIndex()` just as you would with an ordinary single-field index.
- If there is an array as a value for an indexed field, the index will be multikey on that field.

Example: Array of Numbers

```
db.race_results.drop()
db.race_results.createIndex( { "lap_times" : 1 } )
a = [ { "lap_times" : [ 3, 5, 2, 8 ] },
      { "lap_times" : [ 1, 6, 4, 2 ] },
      { "lap_times" : [ 6, 3, 3, 8 ] } ]
db.race_results.insertMany( a )

// Used the index
db.race_results.find( { lap_times : 1 } ).explain()

// One document found.
// Index not used, because it is naive to position.
db.race_results.find( { "lap_times.2" : 3 } ).explain()
```

Exercise: Array of Documents, Part 1

Create a collection and add an index on the x field:

```
db.blog.drop()
b = [ { "comments" : [
    { "name" : "Bob", "rating" : 1 },
    { "name" : "Frank", "rating" : 5.3 },
    { "name" : "Susan", "rating" : 3 } ] },
    { "comments" : [
        { name : "Megan", "rating" : 1 } ] },
    { "comments" : [
        { "name" : "Luke", "rating" : 1.4 },
        { "name" : "Matt", "rating" : 5 },
        { "name" : "Sue", "rating" : 7 } ] }]
db.blog.insertMany(b)

db.blog.createIndex( { "comments" : 1 } )
// vs
db.blog.createIndex( { "comments.rating" : 1 } )

// for this query
db.blog.find( { "comments.rating" : 5 } )
```

Note:

- Note: JSON is a dictionary and doesn't guarantee order, indexing the top level array (comments array) won't work
-

Exercise: Array of Documents, Part 2

For each of the three queries below:

- How many documents will be returned?
- Will it use our multi-key index? Why or why not?
- If a query will not use the index, which index will it use?

```
db.blog.find( { "comments" : { "name" : "Bob", "rating" : 1 } } )
db.blog.find( { "comments" : { "rating" : 1 } } )
db.blog.find( { "comments.rating" : 1 } )
```

Note:

```
// Never do this, won't give you the results expected
// JSON is a dictionary, and won't preserve ordering, second query will return no
// results

db.blog.find( { "comments" : { "name" : "Bob", "rating" : 1 } } )
db.blog.find( { "comments" : { "rating" : 1 } } )
```

Exercise: Array of Arrays, Part 1

Add some documents and create an index simulating a player in a game moving on an X,Y grid.

```
db.player.drop()
db.player.createIndex( { "last_moves" : 1 } )
c = [ { "last_moves" : [ [ 1, 2 ], [ 2, 3 ], [ 3, 4 ] ] },
      { "last_moves" : [ [ 3, 4 ], [ 4, 5 ] ] },
      { "last_moves" : [ [ 4, 5 ], [ 5, 6 ] ] },
      { "last_moves" : [ [ 3, 4 ] ] },
      { "last_moves" : [ [ 4, 5 ] ] } ]
db.player.insertMany(c)
db.player.find()
```

Exercise: Array of Arrays, Part 2

For each of the queries below:

- How many documents will be returned?
- Does the query use the multi-key index? Why or why not?
- If the query does not use the index, what is an index it could use?

```
db.player.find( { "last_moves" : [ 3, 4 ] } )
db.player.find( { "last_moves" : 3 } )
db.player.find( { "last_moves.1" : [ 4, 5 ] } )
db.player.find( { "last_moves.2" : [ 2, 3 ] } )
```

Note:

```
// 3 documents
db.player.find( { "last_moves" : [ 3, 4 ] } )
// Uses the multi-key index
db.player.find( { "last_moves" : [ 3, 4 ] } ).explain()

// No documents
db.player.find( { "last_moves" : 3 } )

// Does not use the multi-key index, because it is naive to position.
db.player.find( { "last_moves.1" : [ 4, 5 ] } )
db.player.find( { "last_moves.2" : [ 2, 3 ] } )
```

How Multikey Indexes Work

- Each array element is given one entry in the index.
- So an array with 17 elements will have 17 entries – one for each element.
- Multikey indexes can take up much more space than standard indexes.

Multikey Indexes and Sorting

- If you sort using a multikey index:
 - A document will appear at the first position where a value would place the document.
 - It will not appear multiple times.
- This applies to array values generally.
- It is not a specific property of multikey indexes.

Exercise: Multikey Indexes and Sorting

```
db.testcol.drop()
a = [ { x : [ 1, 11 ] }, { x : [ 2, 10 ] }, { x : [ 3 ] },
      { x : [ 4 ] }, { x : [ 5 ] } ]
db.testcol.insert(a)

db.testcol.createIndex( { x : 1 } )

// x : [ 1, 11 ] array comes first. It contains the lowest value.
db.testcol.find().sort( { x : 1 } )

// x : [ 1, 11 ] array still comes first. Contains the highest value.
db.testcol.find().sort( { x : -1 } )
```

Note:

```
// x : [ 1, 11 ] array comes first, because it contains the lowest value
db.testcol.find().sort( { x : 1 } )

// x : [ 1, 11 ] array still comes first, because it contains the highest value
db.testcol.find().sort( { x : -1 } )
```

Limitations on Multikey Indexes

- You cannot create a compound index using more than one array-valued field.
- This is because of the combinatorics.
- For a compound index on two array-valued fields you would end up with $N * M$ entries for one document.
- You cannot have a hashed multikey index.
- You cannot have a shard key use a multikey index.
- We discuss shard keys in another module.
- The index on the `_id` field cannot become a multikey index.

Example: Multikey Indexes on Multiple Fields

```
db.testcol.drop()
db.testcol.createIndex( { x : 1, y : 1 } )

// no problems yet
db.testcol.insertOne( { _id : 1, x : 1, y : 1 } )

// still OK
db.testcol.insertOne( { _id : 2, x : [ 1, 2 ], y : 1 } )

// still OK
db.testcol.insertOne( { _id : 3, x : 1, y : [ 1, 2 ] } )

// Won't work
db.testcol.insertOne( { _id : 4, x : [ 1, 2 ], y : [ 1, 2 ] } )
```

3.5 Hashed Indexes

Learning Objectives

Upon completing this module, students should understand:

- What a hashed index is
- When to use a hashed index

What is a Hashed Index?

- Hashed indexes are based on field values like any other index.
- The difference is that the values are hashed and it is the hashed value that is indexed.
- The hashing function collapses sub-documents and computes the hash for the entire value.
- MongoDB can use the hashed index to support equality queries.
- Hashed indexes do not support multi-key indexes, i.e. indexes on array fields.
- Hashed indexes do not support range queries.

Why Hashed Indexes?

- In MongoDB, the primary use for hashed indexes is to support sharding a collection using a hashed shard key.
- In some cases, the field we would like to use to shard data would make it difficult to scale using sharding.
- Using a hashed shard key to shard a collection ensures an even distribution of data and overcomes this problem.
- See [Shard a Collection Using a Hashed Shard Key⁵](#) for more details.
- We discuss sharding in detail in another module.

Limitations

- You may not create compound indexes that have hashed index fields.
- You may not specify a unique constraint on a hashed index.
- You can create both a hashed index and a non-hashed index on the same field.

Note:

- For a field on which there is both a hashed index and a non-hashed index, MongoDB will use the non-hashed index for range queries.

⁵ <http://docs.mongodb.org/manual/tutorial/shard-collection-with-a-hashed-shard-key/>

Floating Point Numbers

- MongoDB hashed indexes truncate floating point numbers to 64-bit integers before hashing.
- Do not use a hashed index for floating point numbers that cannot be reliably converted to 64-bit integers.
- MongoDB hashed indexes do not support floating point values larger than 2^{53} .

Creating a Hashed Index

Create a hashed index using an operation that resembles the following. This operation creates a hashed index for the active collection on the a field.

```
db.active.createIndex( { a: "hashed" } )
```

3.6 Geospatial Indexes

Learning Objectives

Upon completing this module, students should understand:

- Use cases of geospatial indexes
- The two types of geospatial indexes
- How to create 2d geospatial indexes
- How to query for documents in a region
- How to create 2dsphere indexes
- Types of geoJSON objects
- How to query using 2dsphere indexes

Introduction to Geospatial Indexes

We can use geospatial indexes to quickly determine geometric relationships:

- All points within a certain radius of another point
- Whether or not points fall within a polygon
- Whether or not two polygons intersect

Easiest to Start with 2 Dimensions

- Initially, it is easiest to think about geospatial indexes in two dimensions.
- One type of geospatial index in MongoDB is a flat 2d index.
- With a geospatial index we can, for example, search for nearby items.
- This is the type of service that many phone apps provide when, say, searching for a nearby cafe.
- We might have a query location identified by an X in a 2d coordinate system.

Note:

- Instructor, please draw a 2d coordinate system with axes for lat and lon.
 - Draw a red (or some other color) x to represent the query document.
-

Location Field

- A geospatial index is based on a location field within documents in a collection.
- The structure of location values depends on the type of geospatial index.
- We will go into more detail on this in a few minutes.
- We can identify other documents in this collection with Xs in our 2d coordinate system.

Note:

- Draw several Xs to represent other documents.
-

Find Nearby Documents

- A geospatial index enables us to efficiently query a collection based on geometric relationships between documents and the query.
- For example, we can quickly locate all documents within a certain radius of our query location.
- In this example, we've illustrated a `$near` query in a 2d geospatial index.

Flat vs. Spherical Indexes

There are two types of geospatial indexes:

- Flat, made with a `2d` index
- Two-dimensional spherical, made with the `2dsphere` index
 - Takes into account the curvature of the earth
 - Joins any two points using a geodesic or “great circle arc”
 - Deviates from flat geometry as you get further from the equator, and as your points get further apart

Flat Geospatial Index

- This is a Cartesian treatment of coordinate pairs.
- E.g., the index would not reflect the fact that the shortest path from Canada to Siberia is over the North Pole (if units are degrees).
- `2d` indexes can be used to describe any flat surface.
- Recommended if:
 - You have legacy coordinate pairs (MongoDB 2.2 or earlier).
 - You do not plan to use geoJSON objects such as LineStrings or Polygons.
 - You are not going to use points far enough North or South to worry about the Earth’s curvature.

Spherical Geospatial Index

- Spherical indexes model the curvature of the Earth
- If you want to plot the shortest path from the Klondike to Siberia, this will know to go over the North Pole.
- Spherical indexes use geoJSON objects (Points, LineString, and Polygons)
- Coordinate pairs are converted into geoJSON Points.

Creating a 2d Index

Creating a `2d` index:

```
db.<COLLECTION>.createIndex(  
  { field_name : "2d", <optional additional field> : <value> },  
  { <optional options document> } )
```

Possible options key-value pairs:

- `min` : `<lower bound>`
- `max` : `<upper bound>`
- `bits` : `<bits of precision for geohash>`

Exercise: Creating a 2d Index

Create a 2d index on the collection testcol with:

- A min value of -20
- A max value of 20
- 10 bits of precision
- The field indexed should be xy.

Note: Answer:

```
db.testcol.createIndex( { xy : "2d" }, { min : -20, max : 20, bits : 10 } )
```

Inserting Documents with a 2d Index

There are two accepted formats:

- Legacy coordinate pairs
- Document with the following fields specified:
 - lng (longitude)
 - lat (latitude)

Exercise: Inserting Documents with 2d Fields

- Insert 2 documents into the ‘twoD’ collection.
- Assign 2d coordinate values to the xy field of each document.
- Longitude values should be -3 and 3 respectively.
- Latitude values should be 0 and 0.4 respectively.

Note: Answer:

```
db.twoD.insert( { xy : [ -3, 0 ] } ) // legacy coordinate pairs
db.twoD.insert( { xy : { lng : 3, lat : 0.4 } } ) // document with lng, lat
db.twoD.find() // both went in OK
db.twoD.insert( { xy : 5 } ) // insert works fine
// Keep in mind that the index doesn't apply to this document.
db.twoD.insert( { xy : [ 0, -500 ] } )
// Generates an error because -500 isn't between +/-20.
db.twoD.insert( { xy : [ 0, 0.00003 ] } )
db.twoD.find()
// last insert worked fine, even though the position resolution is below
// the resolution of the Geohash.
```

Querying Documents Using a 2d Index

- Use \$near to retrieve documents close to a given point.
- Use \$geoWithin to find documents with a shape contained entirely within the query shape.
- Use the following operators to specify a query shape:
 - \$box
 - \$polygon
 - \$center (circle)

Example: Find Based on 2d Coords

Write a query to find all documents in the testcol collection that have an xy field value that falls entirely within the circle with center at [-2.5, -0.5] and a radius of 3.

```
db.testcol.find( { xy : { $geoWithin : { $center : [ [ -2.5, -0.5 ], 3 ] } } }
```

Creating a 2dsphere Index

You can index one or more 2dsphere fields in an index.

```
db.<COLLECTION>.createIndex( { <location field> : "2dsphere" } )
```

The geoJSON Specification

- The geoJSON format encodes location data on the earth.
- The spec is at <http://geojson.org/geojson-spec.html>
- This spec is incorporated in MongoDB 2dsphere indexes.
- It includes Point, LineString, Polygon, and combinations of these.

geoJSON Considerations

- The coordinates of points are given in degrees (longitude then latitude).
- The LineString that joins two points will always be a geodesic.
- Short lines (around a few hundred kilometers or less) will go about where you would expect them to.
- Polygons are made of a closed set of LineStrings.

Note:

- A geodesic may not go where you think.
- E.g., the LineString that joins the points [90, 5] and [-90, 5]:
 - Does NOT go through the point [0, 5]
 - DOES go through the point [0, 90] (i.e., the North Pole).

Simple Types of 2dsphere Objects

Point: A single point on the globe

```
{ <field_name> : { type : "Point",
                     coordinates : [ <longitude>, <latitude> ] } }
```

LineString: A geodesic line that is defined by its two end Points

```
{ <field_name> : { type : "LineString",
                     coordinates : [ [ <longitude 1>, <latitude 1> ],
                                     [ <longitude 2>, <latitude 2> ],
                                     ...
                                     [ <longitude n>, <latitude n> ] ] } }
```

Note:

- Legacy coordinate pairs are treated as Points by a 2dsphere index.
-

Polygons

Simple Polygon:

```
{ <field_name> : { type : "Polygon",
                     coordinates : [ [ [ <Point1 coordinate pair> ],
                                     [ <Point2 coordinate pair> ],
                                     ...
                                     [ <Point1 coordinate pair again> ] ]
                     ] }
```

Polygon with One Hole:

```
{ <field_name> : { type : "Polygon",
                     coordinates : [ [ <Points that define outer polygon> ],
                                     [ <Points that define inner polygon> ]
                     ] }
```

Other Types of 2dsphere Objects

- **MultiPoint:** One or more Points in one document
- **MultiLine:** One or more LineStrings in one document
- **MultiPolygon:** One or more Polygons in one document
- **GeometryCollection:** One or more geoJSON objects in one document

Exercise: Inserting geoJSON Objects (1)

Create a coordinate pair for each the following airports. Create one variable per airport.

- LaGuardia (New York): 40.7772° N, 73.8726° W
- JFK (New York): 40.6397° N, 73.7789° W
- Newark (New York): 40.6925° N, 74.1686° W
- Heathrow (London): 52.4775° N, 0.4614° W
- Gatwick (London): 51.1481° N, 0.1903° W
- Stansted (London): 51.8850° N, 0.2350° E
- Luton (London): 51.9000° N, 0.4333° W

Note:

```
laguardia = [ -73.8726, 40.7772 ]
jfk = [ -73.7789, 40.6397 ],
newark = [ -74.1686, 40.6925 ]
heathrow = [ -0.4614, 52.4775 ]
gatwick = [ -0.1903, 51.1481 ]
stansted = [ 0.2350, 51.8850 ]
luton = [-0.4333, 51.9000 ]
```

- Remember, we use [latitude, longitude].
 - In this example, we have made North (latitude) and East (longitude) positive.
 - West and South are negative.
-

Exercise: Inserting geoJSON Objects (2)

- Now let's make arrays of these.
- Put all the New York area airports into an array called nyPorts.
- Put all the London area airports into an array called londonPorts.
- Create a third array for flight numbers: "AA4453", "VA3333", "UA2440".

Note:

```
nyPorts = [ laguardia, jfk, newark ]
londonPorts = [ heathrow, gatwick, stansted, luton ]
flightNumbers = [ "AA4453", "VA3333", "UA2440" ]
```

Exercise: Inserting geoJSON Objects (3)

- Create documents for every possible New York to London flight.
- Include a `flightNumber` field for each flight.

Note:

```
for (takeoff in ny_ports) {
    for (landing in london_ports) {
        db.flights.insert(
            { origin : { type : "Point",
                        coordinates : ny_ports[takeoff] },
              destination : { type : "Point",
                              coordinates : london_ports[landing] },
              flightNumber : flightNumbers[takeoff] } )
    }
}
```

Exercise: Creating a 2dsphere Index

- Create two indexes on the collection `flights`.
- Make the first a compound index on the fields:
 - `origin`
 - `destination`
 - `flightNumber`
- Specify 2dsphere indexes on both `origin` and `destination`.
- Specify a simple index on `name`.
- Make the second index just a 2dsphere index on `destination`.

Note:

```
db.flights.createIndex( { origin : "2dsphere",
                           destination : "2dsphere",
                           flightNumber : 1 } )

db.flights.createIndex( { destination : "2dsphere" } )

db.flights.getIndexes() // see the indexes.
```

Querying 2dsphere Objects

\$geoNear: Finds all points, orders them by distance from a position.

```
{ <field name> : { $near : { $geometry : {
    type : "Point",
    coordinates : [ lng, lat ] },
    $maxDistance : <meters> } } }
```

\$near: Just like \$geoNear, except in very edge cases; check the docs.

\$geoWithin: Only returns documents with a location completely contained within the query.

\$geoIntersects: Returns documents with their indexed field intersecting any part of the shape in the query.

3.7 TTL Indexes

Learning Objectives

Upon completing this module students should understand:

- How to create a TTL index
- When a TTL indexed document will get deleted
- Limitations of TTL indexes

TTL Index Basics

- TTL is short for “Time To Live”.
- TTL indexes must be based on a field of type Date (including ISODate) or Timestamp.
- Any Date field older than expireAfterSeconds will get deleted at some point.

Creating a TTL Index

Create with:

```
db.<COLLECTION>.createIndex( { field_name : 1 },
                                { expireAfterSeconds : some_number } )
```

Exercise: Creating a TTL Index

Let's create a TTL index on the `ttl` collection that will delete documents older than 30 seconds. Write a script that will insert documents at a rate of one per second.

```
db.sessions.drop()
db.sessions.createIndex( { "last_user_action" : 1 },
                        { "expireAfterSeconds" : 30 } )

i = 0
while (true) {
    i += 1;
    db.sessions.insertOne( { "last_user_action" : ISODate(), "b" : i } );
    sleep(1000); // Sleep for 1 second
}
```

Exercise: Check the Collection

Then, leaving that window open, open up a new terminal and connect to the database with the mongo shell. This will allow us to verify the TTL behavior.

```
// look at the output and wait. After a ramp-up of up to a minute or so,
// count() will be reset to 30 once/minute.
while (true) {
    print(db.sessions.count());
    sleep(100);
}
```

3.8 Text Indexes

Learning Objectives

Upon completing this module, students should understand:

- The purpose of a text index
- How to create text indexes
- How to search using text indexes
- How to rank search results by relevance score

What is a Text Index?

- A text index is based on the tokens (words, etc.) used in string fields.
- MongoDB supports text search for a number of languages.
- Text indexes drop language-specific stop words (e.g. in English “the”, “an”, “a”, “and”, etc.).
- Text indexes use simple, language-specific suffix stemming (e.g., “running” to “run”).

Creating a Text Index

You create a text index a little bit differently than you create a standard index.

```
db.<COLLECTION>.createIndex( { <field name> : "text" } )
```

Exercise: Creating a Text Index

Create a text index on the “dialog” field of the montyPython collection.

```
db.montyPython.createIndex( { dialog : "text" } )
```

Creating a Text Index with Weighted Fields

- Default weight of 1 per indexed field.
- Weight is relative to other weights in text index.

```
db.<COLLECTION>.createIndex(  
  { "title" : "text", "keywords": "text", "author" : "text" },  
  { "weights" : {  
    "title" : 10,  
    "keywords" : 5  
  }})
```

- Term match in “title” field has 10 times (i.e. 10:1) the impact as a term match in the “author” field.

Creating a Text Index with Weighted Fields

- The default weight is 1 for each indexed field.
- The weight is relative to other weights in a text index.

```
db.<COLLECTION>.createIndex(  
  { "title" : "text", "keywords": "text", "author" : "text" },  
  { "weights" : {  
    "title" : 10,  
    "keywords" : 5  
  }})
```

- Term match in “title” field has 10 times (i.e. 10:1) the impact as a term match in the “author” field.

Text Indexes are Similar to Multikey Indexes

- Continuing our example, you can treat the `dialog` field as a multikey index.
- A multikey index with each of the words in `dialog` as values.
- You can query the field using the `$text` operator.

Exercise: Inserting Texts

Let's add some documents to our `montyPython` collection.

```
db.montyPython.insert( [
{ _id : 1,
  dialog : "What is the air-speed velocity of an unladen swallow?" },
{ _id : 2,
  dialog : "What do you mean? An African or a European swallow?" },
{ _id : 3,
  dialog : "Huh? I... I don't know that." },
{ _id : 45,
  dialog : "You're using coconuts!" },
{ _id : 55,
  dialog : "What? A swallow carrying a coconut?" } ] )
```

Querying a Text Index

Next, let's query the collection. The syntax is:

```
db.<COLLECTION>.find( { $text : { $search : "query terms go here" } } )
```

Exercise: Querying a Text Index

Using the text index, find all documents in the `montyPython` collection with the word “swallow” in it.

```
// Returns 3 documents.
db.montyPython.find( { $text : { $search : "swallow" } } )
```

Exercise: Querying Using Two Words

- Find all documents in the `montyPython` collection with either the word ‘coconut’ or ‘swallow’.
- By default MongoDB ORs query terms together.
- E.g., if you query on two words, results include documents using either word.

```
// Finds 4 documents, 3 of which contain only one of the two words.
db.montyPython.find( { $text : { $search : "coconut swallow" } } )
```

Search for a Phrase

- To match an exact phrase, include search terms in quotes (escaped).
- The following query selects documents containing the phrase “European swallow”:

```
db.montyPython.find( { $text: { $search: "\"European swallow\""} } )
```

Text Search Score

- The search algorithm assigns a relevance score to each search result.
- The score is generated by a vector ranking algorithm.
- The documents can be sorted by that score.

```
db.<COLLECTION>.find(  
    { $text : { $search : "swallow coconut"} },  
    { textScore: { $meta : "textScore" } }  
) .sort(  
    { textScore: { $meta: "textScore" } }  
) )
```

3.9 Lab: Finding and Addressing Slow Operations

Exercise: Determine Indexes Needed

- In a mongo shell run `performance.b()`. This will run in an infinite loop printing some output as it runs various statements against the server.
- Now imagine we have detected a performance problem and suspect there is a slow operation running.
- Find the slow operation and terminate it. Every slow operation is assumed to run for 100ms or more.
- In order to do this, open a second window (or tab) and run a second instance of the mongo shell.
- What indexes can we introduce to make the slow queries more efficient? Disregard the index created in the previous exercise.

Note:

- Look at the logs to find queries over 100ms
 - { “active”: 1 }
 - { “str”: 1, “x”: 1 }
-

3.10 Lab: Using `explain()`

Exercise: `explain("executionStats")`

Drop all indexes from previous exercises:

```
mongo performance
> db.sensor_readings.dropIndexes()
```

Create an index for the “active” field:

```
db.sensor_readings.createIndex({ "active" : 1 })
```

How many index entries and documents are examined for the following query? How many results are returned?

```
db.sensor_readings.find(
    { "active": false, "_id": { $gte: 99, $lte: 1000 } }
).explain("executionStats")
```

4 Storage

Introduction to Storage Engines (page 88) MongoDB storage engines

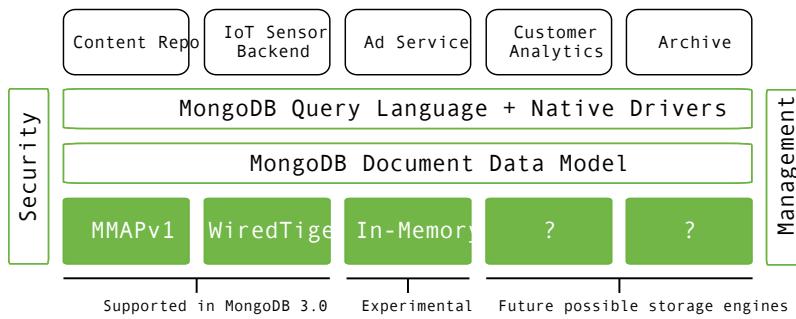
4.1 Introduction to Storage Engines

Learning Objectives

Upon completing this module, students should be familiar with:

- Available storage engines in MongoDB
- MongoDB journaling mechanics
- The default storage engine for MongoDB
- Common storage engine parameters
- The storage engine API

What is a Database Storage Engine?



Note:

- A database storage engine is the underlying software component that a database management system uses to create, read, update, and delete data from a database.
 - Talk through the diagram and how storage engines are used to abstract access to the data
-

How Storage Engines Affect Performance

- Writing and reading documents
- Concurrency
- Compression algorithms
- Index format and implementation
- On-disk format

Note: Can use an extreme example, such as the difference between an in-memory storage engine and mmap/wiredtiger for write performance

Storage Engine Journaling

- Keep track of all changes made to data files
- Stage writes sequentially before they can be committed to the data files
- Crash recovery, writes from journal can be replayed to data files in the event of a failure

MongoDB Storage Engines

With the release of MongoDB 3.2, three storage engine options are available:

- MMAPv1
- WiredTiger (default)
- In-memory storage (Enterprise only)

Specifying a MongoDB Storage Engine

Use the `--storageEngine` parameter to specify which storage engine MongoDB should use. E.g.,

```
mongod --storageEngine mmapv1
```

Note:

- wiredTiger is used if `storageEngine` parameter isn't specified
-

Specifying a Location to Store Data Files

- Use the dbpath parameter

```
mongod --dbpath /data/db
```

- Other files are also stored here. E.g.,
 - mongod.lock file
 - journal
- See the MongoDB docs for a complete list of storage options⁶.

MMAPv1 Storage Engine

- MMAPv1 is MongoDB's original storage engine was the default up to MongoDB 3.0.
- specify the use of the MMAPv1 storage engine as follows:

```
mongod --storageEngine mmapv1
```

- MMAPv1 is based on memory-mapped files, which map data files on disk into virtual memory.
- As of MongoDB 3.0, MMAPv1 supports collection-level concurrency.

MMAPv1 Workloads

MMAPv1 excels at workloads where documents do not outgrow their original record size:

- High-volume inserts
- Read-only workloads
- In-place updates

Note:

- None of the use cases above grow the documents (and potentially force them to move), one flaw with mmapv1

⁶ <http://docs.mongodb.org/manual/reference/program/mongod/#storage-options>

Power of 2 Sizes Allocation Strategy

- MongoDB 3.0 uses power of 2 sizes allocation as the default record allocation strategy for MMAPv1.
- With this strategy, records include the document plus extra space, or padding.
- Each record has a size in bytes that is a power of 2 (e.g. 32, 64, 128, ... 2MB).
- For documents larger than 2MB, allocation is rounded up to the nearest multiple of 2MB.
- This strategy enables MongoDB to efficiently reuse freed records to reduce fragmentation.
- In addition, the added padding gives a document room to grow without requiring a move.
 - Saves the cost of moving a document
 - Results in fewer updates to indexes

Compression in MongoDB

- Compression can significantly reduce the amount of disk space / memory required.
- The tradeoff is that compression requires more CPU.
- MMAPv1 does not support compression.
- WiredTiger does.

WiredTiger Storage Engine

- The WiredTiger storage engine excels at all workloads, especially write-heavy and update-heavy workloads.
- Notable features of the WiredTiger storage engine that do not exist in the MMAPv1 storage engine include:
 - Compression
 - Document-level concurrency
- Default storage engine since MongoDB 3.2.
- For older versions, specify the use of the WiredTiger storage engine as follows.

```
mongod --storageEngine wiredTiger
```

WiredTiger Compression Options

- snappy (default): less CPU usage than zlib, less reduction in data size
- zlib: greater CPU usage than snappy, greater reduction in data size
- no compression

Configuring Compression in WiredTiger

Use the `wiredTigerCollectionBlockCompressor` parameter. E.g.,

```
mongod --storageEngine wiredTiger  
      --wiredTigerCollectionBlockCompressor zlib
```

Configuring Memory Usage in WiredTiger

Use the `wiredTigerCacheSize` parameter to designate the amount of RAM for the WiredTiger storage engine.

- By default, this value is set to the maximum of half of physical RAM or 1GB
- If the database server shares a machine with an application server, it is now easier to designate the amount of RAM the database server can use

Note:

- Unlike MMAPv1, WiredTiger can be configured to use a finite amount of RAM.
-

Journaling in MMAPv1 vs. WiredTiger

- MMAPv1 uses write-ahead journaling to ensure consistency.
- WiredTiger uses a write-ahead transaction log in combination with checkpoints to ensure durability.
- With WiredTiger, the replication process may provide sufficient durability guarantees.

MMAPv1 Journaling Mechanics

- Journal files in <DATA-DIR>/journal are append only
- 1GB per journal file
- Once MongoDB applies all write operations from a journal file to the database data files, it deletes the journal file (or re-uses it)
- Usually only a few journal files in the <DATA-DIR>/journal directory

MMAPv1 Journaling Mechanics (Continued)

- Data is flushed from the shared view to data files every 60 seconds (configurable)
- The operating system may force a flush at a higher frequency than 60 seconds if the system is low on free memory
- Once a journal file contains only flushed writes, it is no longer needed for recovery and can be deleted or re-used

WiredTiger Journaling Mechanics

- WiredTiger will commit a checkpoint to disk every 60 seconds or when there are 2 gigabytes of data to write.
- Between and during checkpoints the data files are always valid.
- The WiredTiger journal persists all data modifications between checkpoints.
- If MongoDB exits between checkpoints, it uses the journal to replay all data modified since the last checkpoint.
- By default, WiredTiger journal is compressed using snappy.

Storage Engine API

MongoDB 3.0 introduced a storage engine API:

- Abstracted storage engine functionality in the code base
- Easier for MongoDB to develop future storage engines
- Easier for third parties to develop their own MongoDB storage engines

Conclusion

- MongoDB 3.0 introduces pluggable storage engines.
- Current options include:
 - MMAPv1 (default)
 - WiredTiger
- WiredTiger introduces the following to MongoDB:
 - Compression
 - Document-level concurrency
- The storage engine API enables third parties to develop storage engines. Examples include:
 - RocksDB
 - An HDFS storage engine

Note:

- Good time to draw what this replica set could look like on the board and talk through even more possibilities

5 Drivers

Introduction to MongoDB Drivers (page 94) An introduction to the MongoDB drivers

Lab: Driver Tutorial (Optional) (page 97) A quick tour through the Python driver

5.1 Introduction to MongoDB Drivers

Learning Objectives

Upon completing this module, students should understand:

- What MongoDB drivers are available
- Where to find MongoDB driver specifications
- Key driver settings

MongoDB Supported Drivers

- C⁷
- C++⁸
- C#⁹
- Java¹⁰
- Node.js¹¹
- Perl¹²
- PHP¹³
- Python¹⁴
- Ruby¹⁵
- Scala¹⁶

⁷ <http://docs.mongodb.org/ecosystem/drivers/c>

⁸ <http://docs.mongodb.org/ecosystem/drivers/cpp>

⁹ <http://docs.mongodb.org/ecosystem/drivers/csharp>

¹⁰ <http://docs.mongodb.org/ecosystem/drivers/java>

¹¹ <http://docs.mongodb.org/ecosystem/drivers/node-js>

¹² <http://docs.mongodb.org/ecosystem/drivers/perl>

¹³ <http://docs.mongodb.org/ecosystem/drivers/php>

¹⁴ <http://docs.mongodb.org/ecosystem/drivers/python>

¹⁵ <http://docs.mongodb.org/ecosystem/drivers/ruby>

¹⁶ <http://docs.mongodb.org/ecosystem/drivers/scala>

MongoDB Community Supported Drivers

35+ different drivers for MongoDB:

Go, Erlang, Clojure, D, Delphi, F#, Groovy, Lisp, Objective C, Prolog, Smalltalk, and more

Driver Specs

To ensure drivers have a consistent functionality, series of publicly available specification documents¹⁷ for:

- Authentication¹⁸
- CRUD operations¹⁹
- Index management²⁰
- SDAM²¹
- Server Selection²²
- Etc.

Driver Settings (Per Operation)

- Read preference
- Write concern
- Maximum operation time (maxTimeMS)
- Batch Size (batchSize)
- Exhaust cursor (exhaust)
- Etc.

Driver Settings (Per Connection)

- Connection timeout
- Connections per host
- Time that a thread will block waiting for a connection (maxWaitTime)
- Socket keep alive
- Sets the multiplier for number of threads allowed to block waiting for a connection
- Etc.

¹⁷ <https://github.com/mongodb/specifications>

¹⁸ <https://github.com/mongodb/specifications/tree/master/source/auth>

¹⁹ <https://github.com/mongodb/specifications/tree/master/source/crud>

²⁰ <https://github.com/mongodb/specifications/blob/master/source/index-management.rst>

21 <https://github.com/mongodb/specifications/tree/master/source/server-discovery-and-monitoring>22 <https://github.com/mongodb/specifications/tree/master/source/server-selection>

Insert a Document with the Java Driver

Connect to a MongoDB instance on localhost:

```
MongoClient mongoClient = new MongoClient();
```

Access the test database:

```
MongoDatabase db = mongoClient.getDatabase("test");
```

Insert a myDocument Document into the test.blog collection:

```
db.getCollection("blog").insertOne(myDocument);
```

Insert a Document with the Python Driver

Connect to a MongoDB instance on localhost:

```
client = MongoClient()
```

Access the test database:

```
db = client['test']
```

Insert a myDocument Document into the test.blog collection:

```
db.blog.insert_one(myDocument);
```

Insert a Document with the C++ Driver

Connect to the “test” database on localhost:

```
mongocxx::instance inst{};  
mongocxx::client conn{};  
  
auto db = conn["test"];
```

Insert a myDocument Document into the test.blog collection:

```
auto res = db["blog"].insert_one(myDocument);
```

5.2 Lab: Driver Tutorial (Optional)

Tutorial

Complete the Python driver tutorial²³ for a concise introduction to:

- Creating MongoDB documents in Python
- Connecting to a database
- Writing and reading documents
- Creating indexes

²³ <http://api.mongodb.org/python/current/tutorial.html>

6 Aggregation

Aggregation Tutorial (page 98) An introduction to the aggregation framework, pipeline concept, and stages

Optimizing Aggregation (page 113) Resource management in the aggregation pipeline

Lab: Aggregation Framework (page 115) Aggregation labs

Lab: Using \$graphLookup (page 118) \$graphLookup lab

6.1 Aggregation Tutorial

Learning Objectives

Upon completing this module students should understand:

- The concept of the aggregation pipeline
- The stages of the aggregation pipeline
- How to use aggregation operators
- The fundamentals of using aggregation for data analysis
- Group aggregation operators
- Using the same operator in multiple stages of an aggregation pipeline

Aggregation Basics

- Use the aggregation framework to transform and analyze data in MongoDB collections.
- For those who are used to SQL, aggregation can be similar to GROUP BY.
- The aggregation framework is based on the concept of a pipeline.

The Aggregation Pipeline

- An aggregation pipeline is analogous to a UNIX pipeline.
- Each stage of the pipeline:
 - Receives a set of documents as input.
 - Performs an operation on those documents.
 - Produces a set of documents for use by the following stage.
- A pipeline has the following syntax:

```
db.<COLLECTION>.aggregate( [ { stage1 }, { stage2 }, ... ],
                           { options } )
```

Aggregation Stages

- `$match`: Similar to `find()`
- `$project`: Shape documents
- `$sort`: Like the cursor method of the same name
- `$skip`: Like the cursor method of the same name
- `$limit`: Like the cursor method of the same name
- `$unwind`: Used for working with arrays
- `$group`: Used to aggregate field values from multiple documents
- `$lookup`: Performs a left outer join to another collection
- `$sample`: Randomly selects the specified number of documents from its input.

Aggregation Stages (continued)

- `$graphLookup`: Performs a recursive search on a collection
- `$indexStats`: Returns statistics regarding the use of each index for the collection
- `$out`: Creates a new collection from the output of an aggregation pipeline

For more information please check the [aggregation framework stages²⁴](#) documentation page.

The Match Stage

- The `$match` operator works like the query phase of `find()`
- Documents in the pipeline that match the query document will be passed to subsequent stages.
- `$match` is often the first operator used in an aggregation stage.
- Like other aggregation operators, `$match` can occur multiple times in a single pipeline.

The Project Stage

- `$project` allows you to shape the documents into what you need for the next stage.
 - The simplest form of shaping is using `$project` to select only the fields you are interested in.
 - `$project` can also create new fields from other fields in the input document.
 - * *E.g.*, you can pull a value out of an embedded document and put it at the top level.
 - * *E.g.*, you can create a ratio from the values of two fields as pass along as a single field.
- `$project` produces 1 output document for every input document it sees.

²⁴ <https://docs.mongodb.com/manual/reference/operator/aggregation/#stage-operators>

A Twitter Dataset

- Let's look at some examples that illustrate the MongoDB aggregation framework.
- These examples operate on a collection of tweets.
 - As with any dataset of this type, it's a snapshot in time.
 - It may not reflect the structure of Twitter feeds as they look today.

Tweets Data Model

```
{  
    "text" : "Something interesting ...",  
    "entities" : {  
        "user_mentions" : [  
            {  
                "screen_name" : "somebody_else",  
                ...  
            }  
        ],  
        "urls" : [ ],  
        "hashtags" : [ ]  
    },  
    "user" : {  
        "friends_count" : 544,  
        "screen_name" : "somebody",  
        "followers_count" : 100,  
        ...  
    },  
}
```

Analyzing Tweets

- Imagine the types of analyses one might want to do on tweets.
- It's common to analyze the behavior of users and the networks involved.
- Our examples will focus on this type of analysis

Note:

- We should also mention that our tweet documents actually contain many more fields.
 - We are showing just those fields relevant to the aggregations we'll do.
-

Friends and Followers

- Let's look again at two stages we touched on earlier:
 - \$match
 - \$project
- In our dataset:
 - friends are those a user follows.
 - followers are others that follow a users.
- Using these operators we will write an aggregation pipeline that will:
 - Ignore anyone with no friends and no followers.
 - Calculate who has the highest followers to friends ratio.

Exercise: Friends and Followers

```
db.tweets.aggregate( [  
    { $match: { "user.friends_count": { $gt: 0 },  
               "user.followers_count": { $gt: 0 } } },  
    { $project: { ratio: { $divide: ["$user.followers_count",  
                                    "$user.friends_count"] },  
                screen_name : "$user.screen_name" } },  
    { $sort: { ratio: -1 } },  
    { $limit: 1 } ] )
```

Note:

- Discuss the \$match stage
 - Discuss the \$project stage as a whole
 - Remember that with project we can pull a value out of an embedded document and put it at the top level.
 - Discuss the ratio projection
 - Discuss screen_name projection
 - Give an overview of other operators we might use in projections
-

Exercise: \$match and \$project

- Of the users in the “Brasilia” timezone who have tweeted 100 times or more, who has the largest number of followers?
- Time zone is found in the “time_zone” field of the user object in each tweet.
- The number of tweets for each user is found in the “statuses_count” field.
- A result document should look something like the following:

```
{ _id : ObjectId('52fd2490bac3fa1975477702'),
  followers : 2597,
  screen_name: 'marbles',
  tweets : 12334
}
```

Note:

```
[ { "$match" : { "user.time_zone" : "Brasilia",
                 "user.statuses_count" : {"$gte" : 100} } },
  { "$project" : { "followers" : "$user.followers_count",
                  "tweets" : "$user.statuses_count",
                  "screen_name" : "$user.screen_name" } },
  { "$sort" : { "followers" : -1 } },
  { "$limit" : 1 } ]
```

The Group Stage

- For those coming from the relational world, \$group is similar to the SQL GROUP BY statement.
- \$group operations require that we specify which field to group on.
- Documents with the same identifier will be aggregated together.
- With \$group, we aggregate values using accumulators²⁵.

Tweet Source

- The tweets in our twitter collection have a field called source.
- This field describes the application that was used to create the tweet.
- Let’s write an aggregation pipeline that identifies the applications most frequently used to publish tweets.

²⁵ <http://docs.mongodb.org/manual/meta/aggregation-quick-reference/#accumulators>

Exercise: Tweet Source

```
db.tweets.aggregate( [  
    { "$group" : { "_id" : "$source",  
                  "count" : { "$sum" : 1 } } },  
    { "$sort" : { "count" : -1 } }  
] )
```

Group Aggregation Accumulators

Accumulators available in the group stage:

- \$sum
- \$avg
- \$first
- \$last
- \$max
- \$min
- \$push
- \$addToSet

Rank Users by Number of Tweets

- One common task is to rank users based on some metric.
- Let's look at who tweets the most.
- Earlier we did the same thing for tweet source.
 - Group together all tweets by a user for every user in our collection
 - Count the tweets for each user
 - Sort in decreasing order
- Let's add the list of tweets to the output documents.
- Need to use an accumulator that works with arrays.
- Can use either \$addToSet or \$push.

Exercise: Adding List of Tweets

For each user, aggregate all their tweets into a single array.

```
db.tweets.aggregate( [  
    { "$group" : { "_id" : "$user.screen_name",  
                  "tweet_texts" : { "$push" : "$text" },  
                  "count" : { "$sum" : 1 } } },  
    { "$sort" : { "count" : -1 } },  
    { "$limit" : 3 }  
]
```

Note:

- \$group operations require that we specify which field to group on.
 - In this case, we group documents based on the user's screen name.
 - With \$group, we aggregate values using arithmetic or array operators.
 - Here we are counting the number of documents for each screen name.
 - We do that by using the \$sum operator
 - This will add 1 to the count field for each document produced by the \$group stage.
 - Note that there will be one document produced by \$group for each screen name.
 - The \$sort stage receives these documents as input and sorts them by the value of the count field
-

The Unwind Stage

- In many situations we want to aggregate using values in an array field.
- In our tweets dataset we need to do this to answer the question:
 - “Who includes the most user mentions in their tweets?”
- User mentions are stored within an embedded document for entities.
- This embedded document also lists any urls and hashtags used in the tweet.

Example: User Mentions in a Tweet

```
...  
"entities" : {  
    "user_mentions" : [  
        {  
            "indices" : [  
                28,  
                44  
            ],  
            "screen_name" : "LatinsUnitedGSX",  
            "name" : "Henry Ramirez",  
            "id" : 102220662  
        }  
    ],  
    "urls" : [ ]  
}
```

```
    "hashtags" : [ ]  
},  
...
```

Using \$unwind

Who includes the most user mentions in their tweets?

```
db.tweets.aggregate(  
  { $unwind: "$entities.user_mentions" },  
  { $group: { _id: "$user.screen_name",  
             count: { $sum: 1 } } },  
  { $sort: { count: -1 } },  
  { $limit: 1 })
```

Note:

- Many tweets contain multiple user mentions.
- We use unwind to produce one document for each user mention.
- Each of these documents is passed to the \$group stage that follows.
- They will be grouped by the user who created the tweet and counted.
- As a result we will have a count of the total number of user mentions made by any one tweeter.

\$unwind also supports this form:

```
{  
  $unwind:  
  {  
    path: <field path>,  
    includeArrayIndex: <string>,  
    preserveNullAndEmptyArrays: <boolean>  
  }  
}
```

\$unwind Behavior

- \$unwind no longer errors on non-array operands.
- If the operand is not:
 - An array,
 - Missing
 - null
 - An empty array
- \$unwind treats the operand as a single element array.

Note:

- Nullish values are null, undefined, empty array, and missing fields

- the `preserveNullAndEmptyArrays` option (see next slide) can keep the nullish values around with the `$unwind`.
-

Data Processing Pipelines

- The aggregation framework allows you to create a data processing pipeline.
- You can include as many stages as necessary to achieve your goal.
- For each stage consider:
 - What input that stage must receive
 - What output it should produce.
- Many tasks require us to include more than one stage using a given operator.

Most Unique User Mentions

- We frequently need multiple group stages to achieve our goal.
- We just looked at a pipeline to find the tweeter that mentioned the most users.
- Let's change this so that it is more of a question about a tweeter's active network.
- We might ask which tweeter has mentioned the most unique users in their tweets.

Same Operator (`$group`), Multiple Stages

Which tweeter has mentioned the most unique users in their tweets?

```
db.tweets.aggregate( [
    { $unwind: "$entities.user_mentions" },
    { $group: {
        _id: "$user.screen_name",
        mset: { $addToSet: "$entities.user_mentions.screen_name" } } },
    { $unwind: "$mset" },
    { $group: { _id: "$_id", count: { $sum: 1 } } },
    { $sort: { count: -1 } },
    { $limit: 1 }
] )
```

Note:

- We begin as we did before by unwinding user mentions.
 - Instead of simple counting them, we aggregate using `$addToSet`.
 - This produces documents that include only unique user mentions.
 - We then do another unwind stage to produce a document for each unique user mention.
 - And count these in a second `$group` stage.
-

The Sort Stage

- Uses the \$sort operator
- Works like the `sort()` cursor method
- 1 to sort ascending; -1 to sort descending
- E.g, `db.testcol.aggregate([{ $sort : { b : 1, a : -1 } }])`

The Skip Stage

- Uses the \$skip operator
- Works like the `skip()` cursor method.
- Value is an integer specifying the number of documents to skip.
- E.g, the following will pass all but the first 3 documents to the next stage in the pipeline.
 - `db.testcol.aggregate([{ $skip : 3 }, ...])`

The Limit Stage

- Used to limit the number of documents passed to the next aggregation stage.
- Works like the `limit()` cursor method.
- Value is an integer.
- E.g., the following will only pass 3 documents to the stage that comes next in the pipeline.
 - `db.testcol.aggregate([{ $limit: 3 }, ...])`

The Lookup Stage

- Pulls documents from a second collection into the pipeline
 - In SQL terms, performs a left outer join
 - * If you `$lookup` then immediately `$unwind` the field, it becomes an inner join
 - The second collection must be in the same database
 - The second collection cannot be sharded
- Documents based on a matching field in each collection
- Previously, you could get this behavior with two separate queries

The Lookup Stage (continued)

- Documents based on a matching field in each collection
- Previously, you could get this behavior with two separate queries
 - One to the collection that contains reference values
 - The other to the collection containing the documents referenced

Note:

- When following with \$unwind, if you use preserveNullAndEmptyArrays: true then it remains a left outer join.
-

Example: Using \$lookup

Create a separate collection for \$lookup

```
db.commentOnEmployees.insertMany( [
    { employeeCount: 405000,
      comment: "Biggest company in the set." },
    { employeeCount: 405000,
      comment: "So you get two comments." },
    { employeeCount: 100000,
      comment: "This is a suspiciously round number." },
    { employeeCount: 99999,
      comment: "This is a suspiciously accurate number." },
    { employeeCount: 99998,
      comment: "This isn't in the data set." }
] )
```

Example: Using \$lookup (Continued)

```
db.companies.aggregate( [
    { $match: { number_of_employees: { $in:
        [ 405000, 388000, 100000, 99999, 99998 ] } } },
    { $project: { _id :0, name: 1, number_of_employees: 1 } },
    { $lookup: {
        from: "commentOnEmployees",
        localField: "number_of_employees",
        foreignField: "employeeCount",
        as: "example_comments"
    } },
    { $sort : { number_of_employees: -1 } } ] )
```

The GraphLookup Stage

- Used to perform a recursive search on a collection, with options for restricting the search by recursion depth and query filter.
- Has the following prototype form:

```
$graphLookup: {  
    from: <collection>,  
    startWith: <expression>,  
    connectFromField: <string>,  
    connectToField: <string>,  
    as: <string>,  
    maxDepth: <number>,  
    depthField: <string>,  
    restrictSearchWithMatch: <document>  
}
```

Note: \$graphLookup Documentation²⁶

\$graphLookup Fields

- **from:** The target collection for \$graphLookup to search
- **startWith:** Expression that specifies the value of the connectFromField with which to start the recursive search
- **connectFromField:** field name whose value \$graphLookup uses to recursively match against the connectToField of other documents in the collection
- **connectToField:** Field name in other documents against which to match the value of the field specified by the connectFromField parameter
- **as:** Name of the array field added to each output document

\$graphLookup Optional Fields

- **maxDepth:** Optional. Non-negative integral number specifying the maximum recursion depth.
- **depthField:** Optional. Name of the field to add to each traversed document in the search path. The value of this field is the recursion depth for the document
- **restrictSearchWithMatch:** Optional. A document specifying additional conditions for the recursive search. The syntax is identical to query filter syntax.

Note: You cannot use any aggregation expression in this filter. For example, a query document such as

```
{ lastName: { $ne: "$lastName" } }
```

will not work in this context to find documents in which the lastName value is different from the lastName value of the input document, because “\$lastName” will act as a string literal, not a field path.

Query Documentation²⁷

²⁶ <https://docs.mongodb.com/manual/reference/operator/aggregation/graphLookup/>

²⁷ <https://docs.mongodb.com/manual/tutorial/query-documents/#read-operations-query-argument>

\$graphLookup Search Process

Input documents flow into the `$graphLookup` stage of an aggregation

- `$graphLookup` targets the search to the collection designated by the `from` parameter

For each input document, the search begins with the value designated by `startWith` - `$graphLookup` matches the `startWith` value against the field designated by the `connectToField` in other documents in the `from` collection

\$graphLookup Search Process (continued)

For each matching document, `$graphLookup` takes the value of the `connectFromField` and checks every document in the `from` collection for a matching `connectToField` value

- For each match, `$graphLookup` adds the matching document in the `from` collection to an array field named by the `as` parameter
- This step continues recursively until no more matching documents are found, or until it reaches the recursion depth specified by `maxDepth`

\$graphLookup Considerations

- The collection specified in `from` cannot be sharded.
- Setting `maxDepth` to 0 is equivalent to `$lookup`
- The `$graphLookup` stage must stay within the 100 megabyte memory limit. `$graphLookup` will ignore `allowDiskUse: true`
- If performing an aggregation that involves multiple views, the views must have the same collation.

Note: Although `$graphLookup` ignores the `allowDiskUse` argument, other stages in the aggregation pipeline will use it.

\$graphLookup Example

Let's illustrate how `$graphLookup` works with an example.

```
use company;
db.employees.insertMany([
  { "_id" : 1, "name" : "Dev" },
  { "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" },
  { "_id" : 3, "name" : "Ron", "reportsTo" : "Eliot" },
  { "_id" : 4, "name" : "Andrew", "reportsTo" : "Eliot" },
  { "_id" : 5, "name" : "Asya", "reportsTo" : "Ron" },
  { "_id" : 6, "name" : "Dan", "reportsTo" : "Andrew" }
])
```

\$graphLookup Example (continued)

With the sample data inserted, perform the following aggregation:

```
db.employees.aggregate([
  {
    $match: { "name": "Dan" }
  },
  {
    $graphLookup: {
      from: "employees",
      startWith: "$reportsTo",
      connectFromField: "reportsTo",
      connectToField: "name",
      as: "reportingHierarchy"
    }
  }
]).pretty()
```

\$graphLookup Example Results

The previous \$graphLookup operation will produce the following:

```
{
  "_id" : 6,
  "name" : "Dan",
  "reportsTo" : "Andrew",
  "reportingHierarchy" : [
    { "_id" : 1, "name" : "Dev" },
    { "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" },
    { "_id" : 4, "name" : "Andrew", "reportsTo" : "Eliot" }
  ]
}
```

The Sample Stage

- Randomized sample of documents
- Useful for calculating statistics
- \$sample provides an efficient means of sampling a data set
- Though if the sample size requested is larger than 5% of the collection \$sample will perform a collection scan
 - Also happens if collection has fewer than 100 documents
- Can use \$sample only as a first stage of the pipeline

Note:

- The exact method is in the documentation.

– [Link here²⁸](#)

²⁸ <https://docs.mongodb.com/manual/reference/operator/aggregation/sample/>

Example: \$sample

```
db.companies.aggregate( [
    { $sample : { size : 5 } },
    { $project : { _id : 0, number_of_employees: 1 } }
] )
```

Note:

- Users will want their sample sizes to be large enough to be useful.
 - 5 is too small for anything
 - A statistician may be required for determining how much is enough; it depends on the distribution of data
-

The IndexStats Stage

- Tells you how many times each index has been used since the server process began
 - Must be the first stage of the pipeline
 - You can use other stages to aggregate the data
 - Returns one document per index
 - The `accesses.ops` field reports the number of times an index was used
-

Note:

- Doesn't include all internal operations
 - TTL deletions or `.explain()` queries, for example
-

Example: \$indexStats

Issue each of the following commands in the mongo shell, one at a time.

```
db.companies.dropIndexes()
db.companies.createIndex( { number_of_employees : 1 } )
db.companies.aggregate( [ { $indexStats: {} } ] )
db.companies.find( { number_of_employees : { $gte : 100 } },
                   { number_of_employees: 1 } ).next()
db.companies.find( { number_of_employees : { $gte : 100 } },
                   { number_of_employees: 1 } ).next()
db.companies.aggregate( [ { $indexStats: {} } ] )
```

Note:

- Point out the “accesses” doc, with ops, is 0 for the new index initially.
 - Ops incremented to 2 from the two `find()` queries.
 - The `.next()` operations are to get the DB to actually execute the query.
 - `_id` did not increment because we weren’t using that index
-

- Neither query changed its “since” field in the “accesses” doc
 - If using replication, the oplog will query on _id when replicating.
-

The Out Stage

- Used to create a new collection from the output of the aggregation pipeline.
- Can only be the last stage in the pipeline.
- If a collection by the name already exists, it replaces that collection.
- Syntax is { \$out : “collection_name” }

6.2 Optimizing Aggregation

Learning Objectives

Upon completing this module students should understand:

- Aggregation pipeline options
- Key aspects of resource management during the aggregation pipeline
- How to order aggregation stages to maximize speed and minimize resource usage
- How MongoDB automatically reorders pipeline stages to improve efficiency
- Changes in the aggregation framework from MongoDB 2.4 to 2.6.

Aggregation Options

- You may pass an options document to aggregate().
- Syntax:

```
db.<COLLECTION>.aggregate( [ { stage1 }, { stage2 }, ... ], { options } )
```
- Following are some of the fields that may be passed in the options document.
 - allowDiskUse : true - permit the use of disk for memory-intensive queries
 - explain : true - display how indexes are used to perform the aggregation.

Aggregation Limits

- An aggregation pipeline cannot use more than 100 MB of RAM.
- `allowDiskUse : true` allows you to get around this limit.
- The following operators do not require the entire dataset to be in memory:
 - `$match`, `$skip`, `$limit`, `$unwind`, and `$project`
 - Stages for these operators are not subject to the 100 MB limit.
 - `$unwind` can, however, dramatically increase the amount of memory used.
- `$group` and `$sort` might require all documents in memory at once.

Limits Prior to MongoDB 2.6

- `aggregate()` returned results in a single document up to 16 MB in size.
- The upper limit on pipeline memory usage was 10% of RAM.

Optimization: Reducing Documents in the Pipeline

- These operators can reduce the number of documents in the pipeline:
 - `$match`
 - `$skip`
 - `$limit`:
- They should be used as early as possible in the pipeline.

Optimization: Sorting

- `$sort` can take advantage of indexes.
- Must be used before any of the following to do this:
 - `$group`
 - `$unwind`
 - `$project`
- After these stages, the fields or their values change.
- `$sort` requires a full scan of the input documents.

Automatic Optimizations

MongoDB will perform some optimizations automatically. For example:

- If a `$project` stage is used late in the pipeline it may be used to eliminate those fields earlier if possible.
- A `$sort` followed by a `$match` will be executed as a `$match` followed by a `$sort` to reduce the number of documents to be sorted.
- A `$skip` followed by a `$limit` will be executed as a `$limit` followed by a `$skip`, with the `$limit` parameter increased by the `$skip` amount to allow `$sort + $limit` coalescence.
- See: [Aggregation Pipeline Optimization²⁹](#)

6.3 Lab: Aggregation Framework

Exercise: Working with Array Fields

Use the aggregation framework to find the name of the individual who has made the most comments on a blog.

Start by importing the necessary data if you have not already.

```
# for version <= 2.6.x
mongoimport -d blog -c posts --drop posts.json
# for version > 3.0
mongoimport -d blog -c posts --drop --batchSize=100 posts.json
```

To help you verify your work, the author with the fewest comments is Mariela Sherer and she commented 387 times.

Note:

```
use blog;
db.posts.aggregate([
  { "$unwind": "$comments" },
  { "$group": {
    {
      _id: "comments.author",
      num_comments: { $sum: 1 }
    }
  },
  { "$sort": { "num_comments": 1 } },
  { "$limit": 10 }
])
```

²⁹ <http://docs.mongodb.org/manual/core/aggregation-pipeline-optimization/>

Exercise: Repeated Aggregation Stages

Import the zips.json file from the data handouts provided:

```
mongoimport -d sample -c zips --drop zips.json
```

Consider together cities in the states of California (CA) and New York (NY) with populations over 25,000. Calculate the average population of this sample of cities.

Please note:

- Different states might have the same city name.
- A city might have multiple zip codes.

Note:

```
db.zips.aggregate( [
    { $match: { state: { $in: ["CA", "NY"] } } },
    { $group: { _id: { state: "$state", city: "$city" },
                pop: { $sum: "$pop" } },
      { $match: { pop: { $gt: 25000} } },
      { $group: { _id: null,
                  pop: { $avg: "$pop" } } }
  ] )
```

Exercise: Projection

Calculate the total number of people who live in a zip code in the US where the city starts with a digit.

\$project can extract the first digit from any field. E.g.,

```
db.zips.aggregate([
    {$project:
        {
            first_char: { $substr: ["$city", 0, 1] }
        }
    }
])
```

Note:

```
db.zips.aggregate( [
    { $project : {
        first_char: { $substr: [ "$city", 0, 1] },
        pop: 1,
        city: "$city",
        zip: "$_id",
        state: 1
    } },
    { $match : {
        first_char: { $in: [ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' ] }
    } },
    { "$group" : { _id: null,
```

```
        population: { $sum : "$pop" } }
    }
])
```

Exercise: Descriptive Statistics

From the `grades` collection, find the class (display the `class_id`) with the highest average student performance on `exams`. To solve this problem you'll want an average of averages.

First calculate the average exam score of each student in each class. Then determine the average class exam score using these values. If you have not already done so, import the `grades` collection as follows.

```
mongoimport -d sample -c grades --drop grades.json
```

Before you attempt this exercise, explore the `grades` collection a little to ensure you understand how it is structured. For additional exercises, consider other statistics you might want to see with this data and how to calculate them.

Note: A solution should be something like them following:

- Unwind all the different scores
- Match based on a score type of “exam”
- Average the based on `class_id` and `student_id`
- Average per class
- Sort into descending order
- Limit to 1

```
db.grades.aggregate([
  {$unwind: "$scores"},
  {$match: {"scores.type": "exam"}},
  {$group: {_id: {student_id: "$student_id", class_id: "$class_id"}, student_avg: {$avg: "$scores.score"}}},
  {$group: {_id: {class_id: "$_id.class_id"}, class_avg: {$avg: "$student_avg"}}},
  {$sort: {class_avg: -1}}
])
```

6.4 Lab: Using \$graphLookup

Exercise: Finding Airline Routes

For this exercise, incorporate the \$graphLookup stage into an aggregation pipeline to find Delta Air Lines routes from JFK to BOI. Find all routes that only have one layover.

- Start by importing the necessary dataset

```
mongoimport -d air -c routes routes.json
mongo air
> db.routes.count()
66985
```

Note:

```
db.routes.aggregate([
  {
    $match: { "airline.alias": "DL", "dst_airport": "BOI" }
  },
  {
    $graphLookup: {
      "startWith": "$dst_airport",
      "from": "routes",
      "connectFromField": "dst_airport",
      "connectToField": "src_airport",
      "as": "connections",
      "restrictSearchWithMatch": { "airline.alias": "DL" }
    }
  },
  {
    $unwind: "$connections"
  },
  {
    $match: { "connections.src_airport": "JFK" }
  },
  {
    $project: {
      src_airport: 1,
      dst_airport: 1,
      connections: 1,
      is_equal: { $eq: ["$src_airport", "$connections.dst_airport"] }
    }
  },
  {
    $match: { is_equal: true }
  }
]).pretty()
```

7 Introduction to Schema Design

Schema Design Core Concepts (page 119) An introduction to schema design in MongoDB

Schema Evolution (page 126) Considerations for evolving a MongoDB schema design over an application's lifetime

Common Schema Design Patterns (page 130) Common design patterns for representing 1-1, 1-M, and M-M relationships and tree structures in MongoDB

7.1 Schema Design Core Concepts

Learning Objectives

Upon completing this module, students should understand:

- Basic schema design principles for MongoDB
- Tradeoffs for embedded documents in a schema
- Tradeoffs for linked documents in a schema
- The use of array fields as part of a schema design

What is a schema?

- Maps concepts and relationships to data
- Sets expectations for the data
- Minimizes overhead of iterative modifications
- Ensures compatibility

Example: Normalized Data Model

User:	Book:	Author:
- username	- title	- firstName
- firstName	- isbn	- lastName
- lastName	- language	
	- createdBy	
	- author	

Example: Denormalized Version

```
User:          Book:  
- username      - title  
- firstName    - isbn  
- lastName     - language  
               - createdBy  
               - author  
               - firstName  
               - lastName
```

Schema Design in MongoDB

- Schema is defined at the application-level
- Design is part of each phase in its lifetime
- There is no magic formula

Three Considerations

- The data your application needs
- Your application's read usage of the data
- Your application's write usage of the data

Case Study

- A Library Web Application
- Different schemas are possible.

Author Schema

```
{  "_id": int,  
  "firstName": string,  
  "lastName": string  
}
```

User Schema

```
{  "_id": int,
  "username": string,
  "password": string
}
```

Book Schema

```
{  "_id": int,
  "title": string,
  "slug": string,
  "author": int,
  "available": boolean,
  "isbn": string,
  "pages": int,
  "publisher": {
    "city": string,
    "date": date,
    "name": string
  },
  "subjects": [ string, string ],
  "language": string,
  "reviews": [ { "user": int, "text": string },
               { "user": int, "text": string } ]
}
```

Example Documents: Author

```
{  _id: 1,
  firstName: "F. Scott",
  lastName: "Fitzgerald"
}
```

Example Documents: User

```
{  _id: 1,
  username: "emily@10gen.com",
  password: "slsjfk4odk84k209dlkdj90009283d"
}
```

Example Documents: Book

```
{  _id: 1,
  title: "The Great Gatsby",
  slug: "9781857150193-the-great-gatsby",
  author: 1,
  available: true,
  isbn: "9781857150193",
  pages: 176,
  publisher: {
    name: "Everyman's Library",
    date: ISODate("1991-09-19T00:00:00Z"),
    city: "London"
  },
  subjects: ["Love stories", "1920s", "Jazz Age"],
  language: "English",
  reviews: [
    { user: 1, text: "One of the best..." },
    { user: 2, text: "It's hard to..." }
  ]
}
```

Embedded Documents

- AKA sub-documents or embedded objects
- What advantages do they have?
- When should they be used?

Example: Embedded Documents

```
{  ...
  publisher: {
    name: "Everyman's Library",
    date: ISODate("1991-09-19T00:00:00Z"),
    city: "London"
  },
  subjects: ["Love stories", "1920s", "Jazz Age"],
  language: "English",
  reviews: [
    { user: 1, text: "One of the best..." },
    { user: 2, text: "It's hard to..." }
  ]
}
```

Embedded Documents: Pros and Cons

- Great for read performance
- One seek to find the document
- At most, one sequential read to retrieve from disk
- Writes can be slow if constantly adding to objects

Linked Documents

- What advantages does this approach have?
- When should they be used?

Example: Linked Documents

```
{ ...  
  author: 1,  
  reviews: [  
    { user: 1, text: "One of the best..." },  
    { user: 2, text: "It's hard to..." }  
  ]  
}
```

Linked Documents: Pros and Cons

- More, smaller documents
- Can make queries by ID very simple
- Accessing linked documents requires extra seeks + reads.
- What effect does this have on the system?

Arrays

- Array of scalars
- Array of documents

Array of Scalars

```
{  
  ...  
  subjects: ["Love stories", "1920s", "Jazz Age"],  
}
```

Array of Documents

```
{  
  ...  
  reviews: [  
    { user: 1, text: "One of the best..." },  
    { user: 2, text: "It's hard to..." }  
  ]  
}
```

Exercise: Users and Book Reviews

Design a schema for users and their book reviews. Usernames are immutable.

- Users
 - username (string)
 - email (string)
- Reviews
 - text (string)
 - rating (integer)
 - created_at (date)

Solution A: Users and Book Reviews

Reviews may be queried by user or book

```
// db.users (one document per user)  
{  
  _id: ObjectId("..."),  
  username: "bob",  
  email: "bob@example.com"  
}  
  
// db.reviews (one document per review)  
{  
  _id: ObjectId("..."),  
  user: ObjectId("..."),  
  book: ObjectId("..."),  
  rating: 5,  
  text: "This book is excellent!",  
  created_at: ISODate("2012-10-10T21:14:07.096Z")  
}
```

Solution B: Users and Book Reviews

Optimized to retrieve reviews by user

```
// db.users, one document per user with all reviews
{
  _id: ObjectId("..."),
  username: "bob",
  email: "bob@example.com",
  reviews: [
    {
      book: ObjectId("..."),
      rating: 5,
      text: "This book is excellent!",
      created_at: ISODate("2012-10-10T21:14:07.096Z")
    }
  ]
}
```

Solution C: Users and Book Reviews

Optimized to retrieve reviews by book

```
// db.users (one document per user)
{
  _id: ObjectId("..."),
  username: "bob",
  email: "bob@example.com"
}

// db.books, one document per book with all reviews
{
  _id: ObjectId("..."),
  // Other book fields...
  reviews: [
    {
      user: ObjectId("..."),
      rating: 5,
      text: "This book is excellent!",
      created_at: ISODate("2014-11-10T21:14:07.096Z")
    }
  ]
}
```

Store Binary Files in MongoDB with GridFS

- Application may have a requirement for binary file storage
- GridFS is a specification for storing files larger than 16MB in MongoDB
- Handled automatically by most drivers
- “mongofiles” is the command line tool for working with GridFS

How GridFS Works

- Files are split into chunks
- Default chunk size is 255k
- fs.files collection stores meta data for the file (name, size, etc.)
- fs.chunks collection stores chunks for binary file

Schema Design Use Cases with GridsFS

- Store large video files and stream chunks to a user
- Enterprise assets, replicated across data centers
- Medical record attachments (x-rays, reports, etc.)

7.2 Schema Evolution

Learning Objectives

Upon completing this module, students should understand the basic philosophy of evolving a MongoDB schema during an application's lifetime:

- Development Phase
- Production Phase
- Iterative Modifications

Development Phase

Support basic CRUD functionality:

- Inserts for authors and books
- Find authors by name
- Find books by basics of title, subject, etc.

Development Phase: Known Query Patterns

```
// Find authors by last name.  
db.authors.createIndex({ "lastName": 1 })  
  
// Find books by slug for detail view  
db.books.createIndex({ "slug": 1 })  
  
// Find books by subject (multi-key)  
db.books.createIndex({ "subjects": 1 })  
  
// Find books by publisher (index on embedded doc)  
db.books.createIndex({ "publisher.name": 1 })
```

Production Phase

Evolve the schema to meet the application's read and write patterns.

Production Phase: Read Patterns

List books by author last name

```
authors = db.authors.find({ lastName: /^f.*$/i }, { _id: 1 });  
  
authorIds = authors.map(function(x) { return x._id; });  
  
db.books.find({author: { $in: authorIds }});
```

Addressing List Books by Last Name

“Cache” the author name in an embedded document.

```
{  
  _id: 1,  
  title: "The Great Gatsby",  
  author: {  
    firstName: "F. Scott",  
    lastName: "Fitzgerald"  
  }  
  // Other fields follow...  
}
```

Queries are now one step

```
db.books.find({ "author.firstName": /^f.*$/i })
```

Production Phase: Write Patterns

Users can review a book.

```
review = {  
    user: 1,  
    text: "I thought this book was great!",  
    rating: 5  
};  
  
db.books.updateOne(  
    { _id: 3 },  
    { $push: { reviews: review } }  
) ;
```

Caveats:

- Document size limit (16MB)
- Storage fragmentation after many updates/deletes

Exercise: Recent Reviews

- Display the 10 most recent reviews by a user.
- Make efficient use of memory and disk seeks.

Solution: Recent Reviews, Schema

Store users' reviews in monthly buckets.

```
// db.reviews (one document per user per month)  
{  
    _id: "bob-201412",  
    reviews: [  
        { _id: ObjectId("..."),  
            rating: 5,  
            text: "This book is excellent!",  
            created_at: ISODate("2014-12-10T21:14:07.096Z")  
        },  
        { _id: ObjectId("..."),  
            rating: 2,  
            text: "I didn't really enjoy this book.",  
            created_at: ISODate("2014-12-11T20:12:50.594Z")  
        }  
    ]  
}
```

Solution: Recent Reviews, Update

Adding a new review to the appropriate bucket

```
myReview = {  
    _id: ObjectId("..."),  
    rating: 3,  
    text: "An average read.",  
    created_at: ISODate("2012-10-13T12:26:11.502Z")  
};  
  
db.reviews.updateOne(  
    { _id: "bob-201210" },  
    { $push: { reviews: myReview } }  
) ;
```

Solution: Recent Reviews, Read

Display the 10 most recent reviews by a user

```
cursor = db.reviews.find(  
    { _id: /^bob-/ },  
    { reviews: { $slice: -10 } }  
) .sort({ _id: -1 }) .batchSize(5);  
  
num = 0;  
  
while (cursor.hasNext() && num < 10) {  
    doc = cursor.next();  
  
    for (var i = 0; i < doc.reviews.length && num < 10; ++i, ++num) {  
        printjson(doc.reviews[i]);  
    }  
}
```

Solution: Recent Reviews, Delete

Deleting a review

```
db.reviews.updateOne(  
    { _id: "bob-201210" },  
    { $pull: { reviews: { _id: ObjectId("...") } } }  
) ;
```

7.3 Common Schema Design Patterns

Learning Objectives

Upon completing this module students should understand common design patterns for modeling:

- One-to-One Relationships
- One-to-Many Relationships
- Many-to-Many Relationships
- Tree Structures

One-to-One Relationship

Let's pretend that authors only write one book.

One-to-One: Linking

Either side, or both, can track the relationship.

```
db.books.findOne()
{
  _id: 1,
  title: "The Great Gatsby",
  slug: "9781857150193-the-great-gatsby",
  author: 1,
  // Other fields follow...
}

db.authors.findOne({ _id: 1 })
{
  _id: 1,
  firstName: "F. Scott",
  lastName: "Fitzgerald"
  book: 1,
}
```

One-to-One: Embedding

```
db.books.findOne()
{
  _id: 1,
  title: "The Great Gatsby",
  slug: "9781857150193-the-great-gatsby",
  author: {
    firstName: "F. Scott",
    lastName: "Fitzgerald"
  }
  // Other fields follow...
}
```

One-to-Many Relationship

In reality, authors may write multiple books.

One-to-Many: Array of IDs

The “one” side tracks the relationship.

- Flexible and space-efficient
- Additional query needed for non-ID lookups

```
db.authors.findOne()
{
  _id: 1,
  firstName: "F. Scott",
  lastName: "Fitzgerald",
  books: [1, 3, 20]
}
```

One-to-Many: Single Field with ID

The “many” side tracks the relationship.

```
db.books.find({ author: 1 })
{
  _id: 1,
  title: "The Great Gatsby",
  slug: "9781857150193-the-great-gatsby",
  author: 1,
  // Other fields follow...
}

{
  _id: 3,
  title: "This Side of Paradise",
  slug: "9780679447238-this-side-of-paradise",
  author: 1,
  // Other fields follow...
}
```

One-to-Many: Array of Documents

```
db.authors.findOne()
{
  _id: 1,
  firstName: "F. Scott",
  lastName: "Fitzgerald",
  books: [
    { _id: 1, title: "The Great Gatsby" },
    { _id: 3, title: "This Side of Paradise" }
  ],
  // Other fields follow...
}
```

Many-to-Many Relationship

Some books may also have co-authors.

Many-to-Many: Array of IDs on Both Sides

```
db.books.findOne()
{
  _id: 1,
  title: "The Great Gatsby",
  authors: [1, 5]
  // Other fields follow...
}

db.authors.findOne()
{
  _id: 1,
  firstName: "F. Scott",
  lastName: "Fitzgerald",
  books: [1, 3, 20]
}
```

Many-to-Many: Array of IDs on Both Sides

Query for all books by a given author.

```
db.books.find({ authors: 1 });
```

Query for all authors of a given book.

```
db.authors.find({ books: 1 });
```

Many-to-Many: Array of IDs on One Side

```
db.books.findOne()
{
  _id: 1,
  title: "The Great Gatsby",
  authors: [1, 5]
  // Other fields follow...
}

db.authors.find({ _id: { $in: [1, 5] } })
{
  _id: 1,
  firstName: "F. Scott",
  lastName: "Fitzgerald"
}
{
  _id: 5,
  firstName: "Unknown",
  lastName: "Co-author"
}
```

Many-to-Many: Array of IDs on One Side

Query for all books by a given author.

```
db.books.find({ authors: 1 });
```

Query for all authors of a given book.

```
book = db.books.findOne(
  { title: "The Great Gatsby" },
  { authors: 1 }
);

db.authors.find({ _id: { $in: book.authors } });
```

Tree Structures

E.g., modeling a subject hierarchy.

Allow users to browse by subject

```
db.subjects.findOne()
{
  _id: 1,
  name: "American Literature",
  sub_category: {
    name: "1920s",
    sub_category: { name: "Jazz Age" }
  }
}
```

- How can you search this collection?
- Be aware of document size limitations
- Benefit from hierarchy being in same document

Alternative: Parents and Ancestors

```
db.subjects.find()
{   _id: "American Literature" }

{   _id : "1920s",
  ancestors: ["American Literature"],
  parent: "American Literature"
}

{   _id: "Jazz Age",
  ancestors: ["American Literature", "1920s"],
  parent: "1920s"
}

{   _id: "Jazz Age in New York",
  ancestors: ["American Literature", "1920s", "Jazz Age"],
```

```
        parent: "Jazz Age"  
    }
```

Find Sub-Categories

```
db.subjects.find({ ancestors: "1920s" })  
{  
    _id: "Jazz Age",  
    ancestors: ["American Literature", "1920s"],  
    parent: "1920s"  
}  
  
{  
    _id: "Jazz Age in New York",  
    ancestors: ["American Literature", "1920s", "Jazz Age"],  
    parent: "Jazz Age"  
}
```

Summary

- Schema design is different in MongoDB.
- Basic data design principles apply.
- It's about your application.
- It's about your data and how it's used.
- It's about the entire lifetime of your application.

8 Replica Sets

Introduction to Replica Sets (page 135) An introduction to replication and replica sets

Elections in Replica Sets (page 139) The process of electing a new primary (automated failover) in replica sets

Replica Set Roles and Configuration (page 145) Configuring replica set members for common use cases

The Oplog: Statement Based Replication (page 147) The process of replicating data from one node of a replica set to another

Lab: Working with the Oplog (page 150) A brief lab that illustrates how the oplog works

Write Concern (page 152) Balancing performance and durability of writes

Read Preference (page 156) Configuring clients to read from specific members of a replica set

Lab: Setting up a Replica Set (page 157) Launching members, configuring, and initiating a replica set

8.1 Introduction to Replica Sets

Learning Objectives

Upon completing this module, students should understand:

- Striking the right balance between cost and redundancy
- The many scenarios replication addresses and why
- How to avoid downtime and data loss using replication

Use Cases for Replication

- High Availability
- Disaster Recovery
- Functional Segregation

High Availability (HA)

- Data still available following:
 - Equipment failure (e.g. server, network switch)
 - Datacenter failure
- This is achieved through automatic failover.

Note: If we lose a server and MongoDB is correctly configured:

- Our database system can still service reads and writes, but by default not during failover period when the election takes place and there is no primary.
 - Without manual intervention as long as there is still a majority of nodes available.
-

Disaster Recovery (DR)

- We can duplicate data across:
 - Multiple database servers
 - Storage backends
 - Datacenters
- Can restore data from another node following:
 - Hardware failure
 - Service interruption

Functional Segregation

There are opportunities to exploit the topology of a replica set:

- Based on physical location (e.g. rack or datacenter location)
- For analytics, reporting, data discovery, system tasks, etc.
- For backups

Note:

- You may direct particular queries to specific nodes (which may have different indexes or hardware) to increase overall performance.
 - Backup data from secondaries to avoid performance penalties on the primary, especially when using tools like `mongodump` which are I/O intensive and evict the working set from memory (significant when data size is larger than RAM and disks are slow).
 - Dedicate secondaries for other purposes such as analytics jobs.
-

Large Replica Sets

Functional segregation can be further exploited by using large replica sets.

- 50 node replica set limit
- Useful for deployments with a large number of data centers or offices
- Read only workloads can position secondaries in data centers around the world (closer to application servers)

Note:

- Sample use case: bank reference data distributed to 20+ data centers around the world, then consumed by the local application server
-

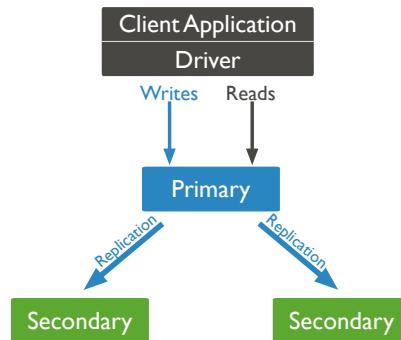
Replication is Not Designed for Scaling

- Can be used for scaling reads, but generally not recommended.
- Drawbacks include:
 - Eventual consistency
 - Not scaling writes
 - Potential system overload when secondaries are unavailable
- Consider sharding for scaling reads and writes.

Note:

- Eventual consistency: This is often tricky to handle as most developers are used to the strong consistency that most databases (and MongoDB) have by default. It also raises the question of how stale the data can be, and what to do when it crosses the threshold (e.g. fall back to reading from the primary).
 - Potential system overload: For example, consider a 3 data node replica set using secondaries to scale reads. Each node is serving reads at 70% of its capacity, which is a reasonable utilization rate. What happens if one of the secondaries fail or is intentionally taken down for maintenance (e.g. upgrading the OS or MongoDB)? Even if the load splits evenly between the 2 remaining nodes, they will be at $70 + (70/2) = 105\%$ capacity.
-

Replica Sets



Note:

- MongoDB implements replication in the form of replica sets. Don't use the term master-slave as that is what we had before replica sets. It still exists for some corner cases (e.g. > 12 replicas) but should otherwise be avoided.
 - A replica set consists of one or more `mongod` servers. Maximum 50 nodes in total and up to 7 with votes.
 - There is at most one `mongod` that is “primary” at any one time (though there are edge cases/bugs when there is more than one).
 - There are usually two or more other `mongod` instances that are secondaries.
 - Secondaries may become primary if there is a failover event of some kind.
 - Failover is automatic when correctly configured and a majority of nodes remain.
 - The secondaries elect a new primary automatically. A primary may also voluntarily step down, like when it can no longer reach the majority of nodes to avoid a potential split brain scenario.
-

Primary Server

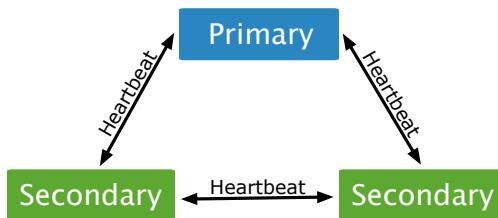
- Clients send writes to the primary only.
- MongoDB, Inc. maintains client drivers in many programming languages like Java, C#, Python, Ruby, and PHP.
- MongoDB drivers are replica set aware.

Note: If the primary for a replica set changes from one node to another, the driver will automatically route writes to the correct mongod.

Secondaries

- A secondary replicates operations from another node in the replica set.
- Secondaries usually replicate from the primary.
- Secondaries may also replicate from other secondaries. This is called replication chaining.
- A secondary may become primary as a result of a failover scenario.

Heartbeats



Note:

- The members of a replica set use heartbeats to determine if they can reach every other node.
- The heartbeats are sent every two seconds.
- If a node is unreachable, this may indicate server failure, a network partition, or simply too slow to respond. The heartbeat will timeout and retry several times before the state is updated.

The Oolog

- The operations log, or oolog, is a special capped collection that is the basis for replication.
- The oolog maintains one entry for each document affected by every write operation.
- Secondaries copy operations from the oolog of their sync source.

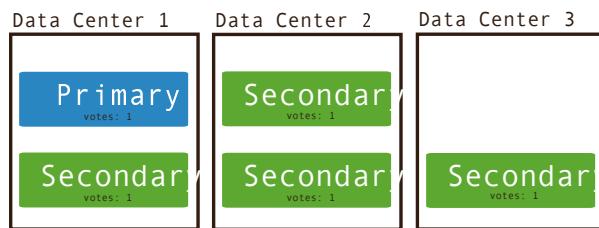
8.2 Elections in Replica Sets

Learning Objectives

Upon completing this module students should understand:

- That elections enable automated failover in replica sets
- How votes are distributed to members
- What prompts an election
- How a new primary is selected

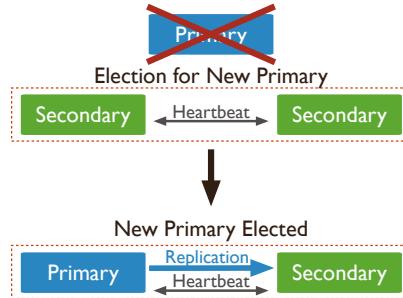
Members and Votes



Note:

- In order for writes to occur, one member of a replica set must be primary.
 - In the event the current primary becomes unavailable, the remaining members elect a new primary.
 - Voting members of replica set each get one vote.
 - Up to seven members may be voting members.
 - This enables MongoDB to ensure elections happen quickly, but enables distribution of votes to different data centers.
 - In order to be elected primary a server must have a true majority of votes.
 - A member must have greater than 50% of the votes in order to be elected primary.
-

Calling Elections



Note:

- MongoDB uses a consensus protocol to determine when an election is required.
 - Essentially, an election will occur if there is no primary.
 - Upon initiation of a new replica set the members will elect a primary.
 - If a primary steps down the set will hold an election.
 - A secondary will call for an election if it does not receive a response to a heartbeat sent to the primary after waiting for 10 seconds.
 - If other members agree that the primary is not available, an election will be held.
-

Selecting a New Primary

Three factors are important in the selection of a primary:

- Priority
- Optime
- Connections

Priority

- The higher its priority, the more likely a member is to become primary.
- The default is 1.
- Servers with a priority of 0 will never become primary.
- Priority values are floating point numbers 0 - 1000 inclusive.

Note:

- Priority is a configuration parameter for replica set members.
- Use priority to determine where writes will be directed by default.
- And where writes will be directed in case of failover.
- Generally all identical nodes in a datacenter should have the same priority to avoid unnecessary failovers. For example, when a higher priority node rejoins the replica set after a maintenance or failure event, it will trigger a failover (during which by default there will be no reads and writes) even though it is unnecessary.

- More on this in a later module.
-

Optime

- Optime: Operation time, which is the timestamp of the last operation the member applied from the oplog.
- To be elected primary, a member must have the most recent optime.
- Only optimates of visible members are compared.

Connections

- Must be able to connect to a majority of the members in the replica set.
- Majority refers to the total number of votes.
- Not the total number of members.

Note: To be elected primary, a replica set member must be able to connect to a majority of the members in the replica set.

When will a primary step down?

- After receiving the `replSetStepDown` or `rs.stepDown()` command.
- If a secondary is eligible for election and has a higher priority.
- If it cannot contact a majority of the members of the replica set.

`replSetStepDown` Behavior

- Primary will attempt to terminate long running operations before stepping down.
- Primary will wait for electable secondary to catch up before stepping down.
- “`secondaryCatchUpPeriodSecs`” can be specified to limit the amount of time the primary will wait for a secondary to catch up before the primary steps down.

Note:

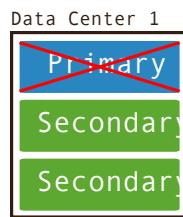
- Ask the class what the tradeoffs could be in setting `secondaryCatchUpPeriodSecs` to a very short amount of time (rollbacks could occur or operations not replicated)
-

Exercise: Elections in Failover Scenarios

- We have learned about electing a primary in replica sets.
- Let's look at some scenarios in which failover might be necessary.

Scenario A: 3 Data Nodes in 1 DC

Which secondary will become the new primary?

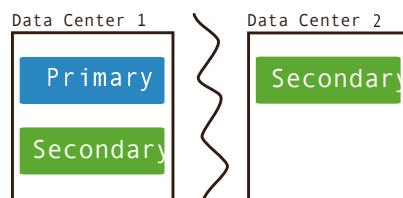


Note:

- It depends on the priorities of the secondaries.
 - And on the optime.
-

Scenario B: 3 Data Nodes in 2 DCs

Which member will become primary following this type of network partition?

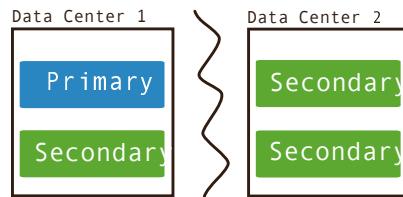


Note:

- The current primary is likely to remain primary.
 - It probably has the highest priority.
 - If DC2 fails, we still have a primary.
 - If DC1 fails, we won't have a primary automatically. The remaining node in DC2 needs to be manually promoted by reconfiguring the replica set.
-

Scenario C: 4 Data Nodes in 2 DCs

What happens following this network partition?

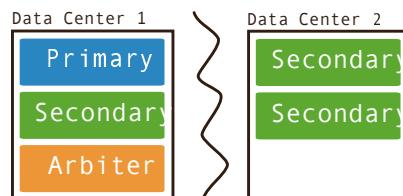


Note:

- We enter a state with no primary.
 - Each side of the network partition has only 2 votes (not a majority).
 - All the servers assume secondary status.
 - This is avoidable.
 - One solution is to add another member to the replica set.
 - If another data node can not be provisioned, MongoDB has a special alternative called an arbiter that requires minimal resources.
 - An arbiter is a `mongod` instance without data and performs only heartbeats, votes, and vetoes.
-

Scenario D: 5 Nodes in 2 DCs

The following is similar to Scenario C, but with the addition of an arbiter in Data Center 1. What happens here?

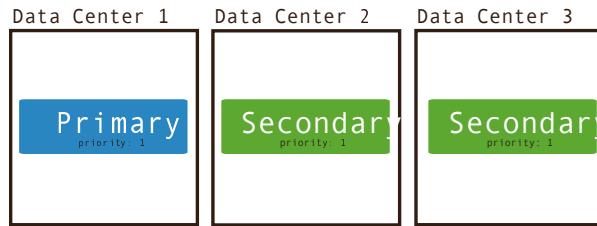


Note:

- The current primary is likely to remain primary.
 - The arbiter helps ensure that the primary can reach a majority of the replica set.
-

Scenario E: 3 Data Nodes in 3 DCs

- What happens here if any one of the nodes/DCs fail?
- What about recovery time?

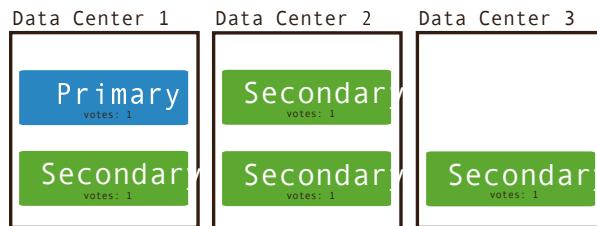


Note:

- The intent is to explain the advantage of deploying to 3 DCs - it's the minimum number of DCs in order for MongoDB to automatically failover if any one DC fails. This is generally what we recommend to customers in our consult and health check reports, though many continue to use 2 DCs due to costs and legacy reasons.
 - To have automated failover in the event of single DC level failure, there must be at least 3 DCs. Otherwise the DC with the minority of nodes must be manually reconfigured.
 - One of the data nodes can be replaced by an arbiter to reduce costs.
-

Scenario F: 5 Data Nodes in 3 DCs

What happens here if any one of the nodes/DCs fail? What about recovery time?



Note:

- Adds another data node to each “main” DC to reduce typically slow and costly cross DC network traffic if an initial sync or similar recovery is needed, as the recovering node can pull from a local replica instead.
 - Depending on the data sizes, operational budget, and requirements, this can be overkill.
 - The data node in DC3 can be replaced by an arbiter to reduce costs.
-

8.3 Replica Set Roles and Configuration

Learning Objectives

Upon completing this module students should understand:

- The use of priority to preference certain members or datacenters as primaries.
- Hidden members.
- The use of hidden secondaries for data analytics and other purposes (when secondary reads are used).
- The use of slaveDelay to protect against operator error.

Example: A Five-Member Replica Set Configuration

- For this example application, there are two datacenters.
- We name the hosts accordingly: dc1-1, dc1-2, dc2-1, etc.
 - This is just a clarifying convention for this example.
 - MongoDB does not care about host names except to establish connections.
- The nodes in this replica set have a variety of roles in this application.

Configuration

```
conf = { // 5 data-bearing nodes
  _id: "mySet",
  members: [
    { _id : 0, host : "dc1-1.example.net:27017", priority : 5 },
    { _id : 1, host : "dc1-2.example.net:27017", priority : 5 },
    { _id : 2, host : "dc2-1.example.net:27017" },
    { _id : 3, host : "dc1-3.example.net:27017", hidden : true },
    { _id : 4, host : "dc2-2.example.net:27017", hidden : true,
      slaveDelay: 7200 }
  ]
}
```

Principal Data Center

```
{ _id : 0, host : "dc1-1.example.net", priority : 5 },
{ _id : 1, host : "dc1-2.example.net", priority : 5 },
```

Note:

- The objective with the priority settings for these two nodes is to prefer DC1 for writes.
- The highest priority member that is up to date will be elected primary.
- Up to date means the member's copy of the oplog is within 10 seconds of the primary.
- If a member with higher priority than the primary is a secondary because it is not up to date, but eventually catches up, it will force an election and win.

Data Center 2

```
{ _id : 2, host : "dc2-1.example.net:27017" },
```

Note:

- Priority is not specified, so it is at the default of 1.
 - dc2-1 could become primary, but only if both dc1-1 and dc1-2 are down.
 - If there is a network partition and clients can only reach DC2, we can manually failover to dc2-1.
-

What about dc1-3 and dc2-2?

```
// Both are hidden.  
// Clients will not distribute reads to hidden members.  
// We use hidden members for dedicated tasks.  
{ _id : 3, host : "dc1-3.example.net:27017", hidden : true },  
{ _id : 4, host : "dc2-2.example.net:27017", hidden : true,  
  slaveDelay: 7200 }
```

Note:

- Will replicate writes normally.
 - We would use this node to pull reports, run analytics, etc.
 - We can do so without paying a performance penalty in the application for either reads or writes.
-

What about dc2-2?

```
{ _id : 4, host : "dc2-2.example.net:27017", hidden : true,  
  slaveDelay : 7200 }
```

Note:

- slaveDelay permits us to specify a time delay (in seconds) for replication.
 - In this case it is 7200 seconds or 2 hours.
 - slaveDelay allows us to use a node as a short term protection against operator error:
 - Fat fingering – for example, accidentally dropping a collection in production.
 - Other examples include bugs in an application that result in corrupted data.
 - Not recommended. Use proper backups instead as there is no optimal delay value. E.g. 2 hours might be too long or too short depending on the situation.
-

8.4 The Oplog: Statement Based Replication

Learning Objectives

Upon completing this module students should understand:

- Binary vs. statement-based replication.
- How the oplog is used to support replication.
- How operations in MongoDB are translated into operations written to the oplog.
- Why oplog operations are idempotent.
- That the oplog is a capped collection and the implications this holds for syncing members.

Binary Replication

- MongoDB replication is statement based.
- Contrast that with binary replication.
- With binary replication we would keep track of:
 - The data files
 - The offsets
 - How many bytes were written for each change
- In short, we would keep track of actual bytes and very specific locations.
- We would simply replicate these changes across secondaries.

Tradeoffs

- The good thing is that figuring out where to write, etc. is very efficient.
- But we must have a byte-for-byte match of our data files on the primary and secondaries.
- The problem is that this couples our replica set members in ways that are inflexible.
- Binary replication may also replicate disk corruption.

Note:

- Some deployments might need to run different versions of MongoDB on different nodes.
 - Different versions of MongoDB might write to different file offsets.
 - We might need to run a compaction or repair on a secondary.
 - In many cases we want to do these types of maintenance tasks independently of other nodes.
-

Statement-Based Replication

- Statement-based replication facilitates greater independence among members of a replica set.
- MongoDB stores a statement for every operation in a capped collection called the oplog.
- Secondaries do not simply apply exactly the operation that was issued on the primary.

Example

Suppose the following command is issued and it deletes 100 documents:

```
db.foo.deleteMany({ age : 30 })
```

This will be represented in the oplog with records such as the following:

```
{ "ts" : Timestamp(1407159845, 5), "h" : NumberLong("-704612487691926908"),  
  "v" : 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 65 } }  
{ "ts" : Timestamp(1407159845, 1), "h" : NumberLong("6014126345225019794"),  
  "v" : 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 333 } }  
{ "ts" : Timestamp(1407159845, 4), "h" : NumberLong("8178791764238465439"),  
  "v" : 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 447 } }  
{ "ts" : Timestamp(1407159845, 3), "h" : NumberLong("-1707391001705528381"),  
  "v" : 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 1033 } }  
{ "ts" : Timestamp(1407159845, 2), "h" : NumberLong("-6814297392442406598"),  
  "v" : 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 9971 } }
```

Replication Based on the Oplog

- One statement per document affected by each write: insert, update, or delete.
- Provides a level of abstraction that enables independence among the members of a replica set:
 - With regard to MongoDB version.
 - In terms of how data is stored on disk.
 - Freedom to do maintenance without the need to bring the entire set down.

Note:

- Can do maintenance without bringing the set down because statement-based replication does not depend on all nodes running the same version of MongoDB or other restrictions that may be imposed by binary replication.
 - In the next exercise, we will see that the oplog is designed so that each statement is idempotent.
 - This feature has several benefits for independent operation of nodes in replica sets.
-

Operations in the Oolog are Idempotent

- Each operation in the oplog is idempotent.
- Whether applied once or multiple times it produces the same result.
- Necessary if you want to be able to copy data while simultaneously accepting writes.

Note: We need to be able to copy while accepting writes when:

- Doing an initial sync for a new replica set member.
 - When a member rejoins a replica set after a network partition a member might end up writing operations it had already received prior to the partition.
-

The Oolog Window

- Oologs are capped collections.
- Capped collections are fixed-size.
- They guarantee preservation of insertion order.
- They support high-throughput operations.
- Like circular buffers, once a collection fills its allocated space:
 - It makes room for new documents.
 - By overwriting the oldest documents in the collection.

Sizing the Oolog

- The oplog should be sized to account for latency among members.
- The default size oplog is usually sufficient.
- But you want to make sure that your oplog is large enough:
 - So that the oplog window is large enough to support replication
 - To give you a large enough history for any diagnostics you might wish to run.

8.5 Lab: Working with the Oplog

Create a Replica Set

Let's take a look at a concrete example. Launch mongo shell as follows.

```
mkdir -p /data/db  
mongo --nodb
```

Create a replica set by running the following command in the mongo shell.

```
replicaSet = new ReplSetTest( { nodes : 3 } )
```

Note: *mongo --nodb* command will start a MongoDB shell without connecting to a host.

This allows the user to perform a set of operations using the javascript handlers and helpers that the mongo shell allows.

A set of those helpers allow the creation of replica set nodes and sharded clusters running on the local machine. The only restriction is that a */data* folder exists and the user running *mongo* has write permission on that folder.

ReplSetTest

- ReplSetTest is useful for experimenting with replica sets as a means of hands-on learning.
- It should never be used in production. Never.
- The command above will create a replica set with three members.
- It does not start the mongods, however.
- You will need to issue additional commands to do that.

Start the Replica Set

Start the mongod processes for this replica set.

```
replicaSet.startSet()
```

Issue the following command to configure replication for these mongods. You will need to issue this while output is flying by in the shell.

```
replicaSet.initiate()
```

Status Check

- You should now have three mongods running on ports 20000, 20001, and 20002.
- You will see log statements from all three printing in the current shell.
- To complete the rest of the exercise, open a new shell.

Connect to the Primary

Open a new shell, connecting to the primary.

```
mongo --port 20000
```

Create some Inventory Data

Use the `store` database:

```
use store
```

Add the following inventory:

```
inventory = [ { _id: 1, inStock: 10 }, { _id: 2, inStock: 20 },
              { _id: 3, inStock: 30 }, { _id: 4, inStock: 40 },
              { _id: 5, inStock: 50 }, { _id: 6, inStock: 60 } ]
db.products.insert(inventory)
```

Perform an Update

Issue the following update. We might issue this update after a purchase of three items.

```
db.products.update({ _id: { $in: [ 2, 5 ] } },
                    { $inc: { inStock : -1 } },
                    { multi: true })
```

View the Oplog

The oplog is a capped collection in the `local` database of each replica set member:

```
use local
db.oplog.rs.find()
{ "ts" : Timestamp(1406944987, 1), "h" : NumberLong(0), "v" : 2, "op" : "n",
  "ns" : "", "o" : { "msg" : "initiating set" } }
...
{ "ts" : Timestamp(1406945076, 1), "h" : NumberLong("-9144645443320713428"),
  "v" : 2, "op" : "u", "ns" : "store.products", "o2" : { "_id" : 2 },
  "o" : { "$set" : { "inStock" : 19 } } }
{ "ts" : Timestamp(1406945076, 2), "h" : NumberLong("-7873096834441143322"),
  "v" : 2, "op" : "u", "ns" : "store.products", "o2" : { "_id" : 5 },
  "o" : { "$set" : { "inStock" : 49 } } }
```

Note:

- Note the last two entries in the oplog.
- These entries reflect the update command issued above.
- Note that there is one operation per document affected.
- More specifically, one operation for each of the documents with the _id values 2 and 5.

One can test students by asking them to insert data into the *store.notes* collection and then go find those writes in the oplog

```
use store
db.notes.insert({'note': 'oh my lord!'})
use local
db.rs.oplog.find({'ns': 'store.notes'})
```

8.6 Write Concern

Learning Objectives

Upon completing this module students should understand:

- How and when rollback occurs in MongoDB.
- The tradeoffs between durability and performance.
- Write concern as a means of ensuring durability in MongoDB.
- The different levels of write concern.

What happens to the write?

- A write is sent to a primary.
- The primary acknowledges the write to the client.
- The primary then becomes unavailable before a secondary can replicate the write

Answer

- Another member might be elected primary.
- It will not have the last write that occurred before the previous primary became unavailable.
- When the previous primary becomes available again:
 - It will note it has writes that were not replicated.
 - It will put these writes into a `rollback` file.
 - A human will need to determine what to do with this data.
- This is default behavior in MongoDB and can be controlled using `write concern`.

Balancing Durability with Performance

- The previous scenario is a specific instance of a common distributed systems problem.
- For some applications it might be acceptable for writes to be rolled back.
- Other applications may have varying requirements with regard to durability.
- Tunable write concern:
 - Make critical operations persist to an entire MongoDB deployment.
 - Specify replication to fewer nodes for less important operations.

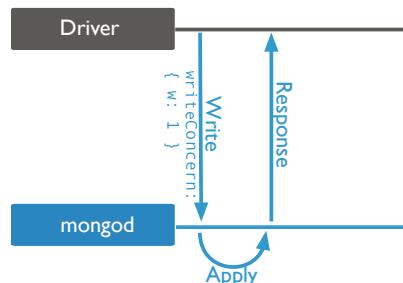
Note:

- MongoDB provides tunable write concern to better address the specific needs of applications.
 - Clients may adjust write concern to ensure that the most important operations persist successfully to an entire MongoDB deployment.
 - For other less critical operations, clients can adjust write concern to ensure faster performance.
-

Defining Write Concern

- Clients may define the write concern per write operation, if necessary.
- Standardize on specific levels of write concerns for different classes of writes.
- In the discussion that follows we will look at increasingly strict levels of write concern.

Write Concern: { w : 1 }



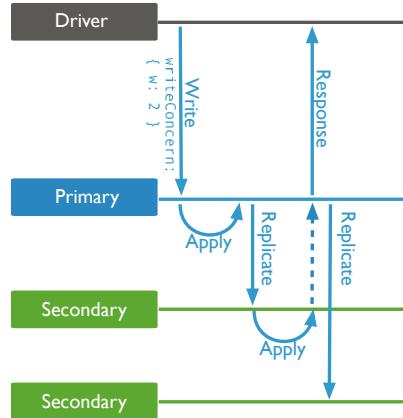
Note:

- We refer to this write concern as “Acknowledged”.
 - This is the default.
 - The primary sends an acknowledgement back to the client that it received the write operation (in RAM).
 - Allows clients to catch network, duplicate key, and other write errors.
-

Example: { w : 1 }

```
db.edges.insertOne( { from : "tom185", to : "mary_p" },
                    { writeConcern : { w : 1 } } )
```

Write Concern: { w : 2 }



Note:

- Called “Replica Acknowledged”
 - Ensures the primary completed the write.
 - Ensures at least one secondary replicated the write.
-

Example: { w : 2 }

```
db.customer.updateOne( { user : "mary_p" },
                      { $push : { shoppingCart:
                                  { _id : 335443, name : "Brew-a-cup",
                                    price : 45.79 } } },
                      { writeConcern : { w : 2 } } )
```

Other Write Concerns

- You may specify any integer as the value of the w field for write concern.
- This guarantees that write operations have propagated to the specified number of members.
- E.g., { w : 3 }, { w : 4 }, etc.

Write Concern: { w : "majority" }

- Ensures the primary completed the write (in RAM).
- Ensures write operations have propagated to a majority of a replica set's **voting** members.
- Avoids hard coding assumptions about the size of your replica set into your application.
- Using majority trades off performance for durability.
- It is suitable for critical writes and to avoid rollbacks.

Example: { w : "majority" }

```
db.products.updateOne({ _id : 335443 },
    { $inc : { inStock : -1 } },
    { writeConcern : { w : "majority" } })
```

Quiz: Which write concern?

Suppose you have a replica set with 7 data nodes. Your application has critical inserts for which you do not want rollbacks to happen. Secondaries may be taken down from time to time for maintenance, leaving you with a potential 4 server replica set. Which write concern is best suited for these critical inserts?

- { w : 1 }
- { w : 2 }
- { w : 3 }
- { w : 4 }
- { w : "majority" }

Note: Answer: { w : "majority" }. This is the same as 4 for a 7 member replica set.

Further Reading

See [Write Concern Reference³⁰](#) for more details on write concern configurations, including setting timeouts and identifying specific replica set members that must acknowledge writes (i.e. [tag sets³¹](#)).

³⁰ <http://docs.mongodb.org/manual/reference/write-concern>

³¹ <http://docs.mongodb.org/manual/tutorial/configure-replica-set-tag-sets/#replica-set-configuration-tag-sets>

8.7 Read Preference

What is Read Preference?

- Read preference allows you to specify the nodes in a replica set to read from.
- Clients only read from the primary by default.
- There are some situations in which a client may want to read from:
 - Any secondary
 - A specific secondary
 - A specific type of secondary
- Only read from a secondary if you can tolerate possibly stale data, as not all writes might have replicated.

Use Cases

- Running systems operations without affecting the front-end application.
- Providing local reads for geographically distributed applications.
- Maintaining availability during a failover.

Note:

- If you have application servers in multiple data centers, you may consider having a [geographically distributed replica set³²](#) and using a read preference of `nearest`.
 - This allows the client to read from the lowest-latency members.
 - Use `primaryPreferred` if you want an application to read from the primary under normal circumstances, but to allow possibly stale reads from secondaries during failover.
-

Not for Scaling

- In general, do *not* read from secondaries to provide extra capacity for reads.
- [Sharding³³](#) increases read and write capacity by distributing operations across a group of machines.
- Sharding is a better strategy for adding capacity.

³² <http://docs.mongodb.org/manual/core/replica-set-geographical-distribution>

³³ <http://docs.mongodb.org/manual/sharding>

Read Preference Modes

MongoDB drivers support the following read preferences. Note that `hidden` nodes will never be read from when connected via the replica set.

- **primary**: Default. All operations read from the primary.
- **primaryPreferred**: Read from the primary but if it is unavailable, read from secondary members.
- **secondary**: All operations read from the secondary members of the replica set.
- **secondaryPreferred**: Read from secondary members but if no secondaries are available, read from the primary.
- **nearest**: Read from member of the replica set with the least network latency, regardless of the member's type.

Tag Sets

- There is also the option to use tag sets.
- You may tag nodes such that queries that contain the tag will be routed to one of the servers with that tag.
- This can be useful for running reports, say for a particular data center or nodes with different hardware (e.g. hard disks vs SSDs).

For example, in the mongo shell:

```
conf = rs.conf()
conf.members[0].tags = { dc : "east", use : "production" }
conf.members[1].tags = { dc : "east", use : "reporting" }
conf.members[2].tags = { use : "production" }
rs.reconfig(conf)
```

8.8 Lab: Setting up a Replica Set

Overview

- In this exercise we will setup a 3 data node replica set on a single machine.
- In production, each node should be run on a dedicated host:
 - To avoid any potential resource contention
 - To provide isolation against server failure

Create Data Directories

Since we will be running all nodes on a single machine, make sure each has its own data directory.

On Linux or Mac OS, run the following in the terminal to create the 3 directories `~/data/rs1`, `~/data/rs2`, and `~/data/rs3`:

```
mkdir -p ~/data/rs{1,2,3}
```

On Windows, run the following command instead in Command Prompt or PowerShell:

```
md c:\data\rs1 c:\data\rs2 c:\data\rs3
```

Launch Each Member

Now start 3 instances of `mongod` in the foreground so that it is easier to observe and shutdown.

On Linux or Mac OS, run *each* of the following commands in its *own terminal window*:

```
mongod --replSet myReplSet --dbpath ~/data/rs1 --port 27017 --oplogSize 200 --  
  ↵smallfiles  
mongod --replSet myReplSet --dbpath ~/data/rs2 --port 27018 --oplogSize 200 --  
  ↵smallfiles  
mongod --replSet myReplSet --dbpath ~/data/rs3 --port 27019 --oplogSize 200 --  
  ↵smallfiles
```

On Windows, run *each* of the following commands in its *own Command Prompt or PowerShell window*:

```
mongod --replSet myReplSet --dbpath c:\data\rs1 --port 27017 --oplogSize 200 --  
  ↵smallfiles  
mongod --replSet myReplSet --dbpath c:\data\rs2 --port 27018 --oplogSize 200 --  
  ↵smallfiles  
mongod --replSet myReplSet --dbpath c:\data\rs3 --port 27019 --oplogSize 200 --  
  ↵smallfiles
```

Status

- At this point, we have 3 `mongod` instances running.
- They were all launched with the same `replSet` parameter of “myReplSet”.
- Despite this, the members are not aware of each other yet.
- This is fine for now.

Note:

- In production, each member would run on a different machine and use service scripts. For example on Linux, modify `/etc/mongod.conf` accordingly and run:

```
sudo service mongod start
```

- To simplify this exercise, we run all members on a single machine.
 - The same configuration process is used for this deployment as for one that is distributed across multiple machines.
-

Connect to a MongoDB Instance

- Connect to the one of the MongoDB instances with the mongo shell.
- To do so run the following command in the terminal, Command Prompt, or PowerShell:

```
mongo // connect to the default port 27017
```

Configure the Replica Set

```
rs.initiate()
// wait a few seconds
rs.add  ('<HOSTNAME>:27018')
rs.addArb('<HOSTNAME>:27019')

// Keep running rs.status() until there's a primary and 2 secondaries
rs.status()
```

Note:

- `rs.initiate()` will use the FQDN. If we simply use `localhost` when adding the data node and arbiter, MongoDB will refuse to mix the two and return an error.
-

Problems That May Occur When Initializing the Replica Set

- `bindIp` parameter is incorrectly set
- Replica set configuration may need to be explicitly specified to use a different hostname:

```
> conf = {
  _id: "<REPLICA-SET-NAME>",
  members: [
    { _id : 0, host : "<HOSTNAME>:27017" },
    { _id : 1, host : "<HOSTNAME>:27018" },
    { _id : 2, host : "<HOSTNAME>:27019",
      "arbiterOnly" : true},
  ]
}
> rs.initiate(conf)
```

Write to the Primary

While still connected to the primary (port 27017) with mongo shell, insert a simple test document:

```
db.testcol.insert({ a: 1 })
db.testcol.count()

exit // Or Ctrl-d
```

Read from a Secondary

Connect to one of the secondaries. E.g.:

```
mongo --port 27018
```

Read from the secondary

```
rs.slaveOk()
db.testcol.find()
```

Review the Oplog

```
use local
db.oplog.rs.find()
```

Changing Replica Set Configuration

To change the replica set configuration, first connect to the primary via mongo shell:

```
mongo --port <PRIMARY_PORT> # e.g. 27017
```

Let's raise the priority of one of the secondaries. Assuming it is the 2nd node (e.g. on port 27018):

```
cfg = rs.conf()
cfg["members"][1]["priority"] = 10
rs.reconfig(cfg)
```

Note:

- Note that `cfg["members"][1]["priority"] = 10` does not actually change the priority.
 - `rs.reconfig(cfg)` does.
-

Verifying Configuration Change

You will see errors like the following, which are expected:

```
2014-10-07T17:01:34.610+0100 DBClientCursor::init call() failed  
2014-10-07T17:01:34.613+0100 trying reconnect to 127.0.0.1:27017 (127.0.0.1) failed  
2014-10-07T17:01:34.617+0100 reconnect 127.0.0.1:27017 (127.0.0.1) ok  
reconnected to server after rs command (which is normal)
```

Verify that the replica set configuration is now as expected:

```
rs.conf()
```

The secondary will now become a primary. Check by running:

```
rs.status()
```

Further Reading

- [Replica Configuration³⁴](#)
- [Replica States³⁵](#)

³⁴ <http://docs.mongodb.org/manual/reference/replica-configuration/>

³⁵ <http://docs.mongodb.org/manual/reference/replica-states/>

9 Sharding

Introduction to Sharding (page 162) An introduction to sharding

Balancing Shards (page 170) Chunks, the balancer, and their role in a sharded cluster

Shard Tags (page 173) How tag-based sharding works

Lab: Setting Up a Sharded Cluster (page 175) Deploying a sharded cluster

9.1 Introduction to Sharding

Learning Objectives

Upon completing this module, students should understand:

- What problems sharding solves
- When sharding is appropriate
- The importance of the shard key and how to choose a good one
- Why sharding increases the need for redundancy

Contrast with Replication

- In an earlier module, we discussed Replication.
- This should never be confused with sharding.
- Replication is about high availability and durability.
 - Taking your data and constantly copying it
 - Being ready to have another machine step in to field requests.

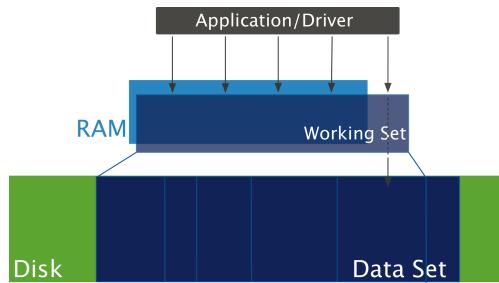
Sharding is Concerned with Scale

- What happens when a system is unable to handle the application load?
- It is time to consider scaling.
- There are 2 types of scaling we want to consider:
 - Vertical scaling
 - Horizontal scaling

Vertical Scaling

- Adding more RAM, faster disks, etc.
- When is this the solution?
- First, consider a concept called the `working set`.

The Working Set



Note:

- The working set for a MongoDB database is the portion of your data that clients access most often.
 - Your working set should stay in memory, otherwise random disk operations will hurt performance.
 - For the fastest processing, ensure that your indexes fit entirely in RAM so that the system can avoid reading the index from disk.
 - In some cases, only recently indexed values must be in RAM.
-

Limitations of Vertical Scaling

- There is a limit to how much RAM one machine can support.
- There are other bottlenecks such as I/O, disk access and network.
- Cost may limit our ability to scale up.
- There may be requirements to have a large working set that no single machine could possibly support.
- This is when it is time to scale horizontally.

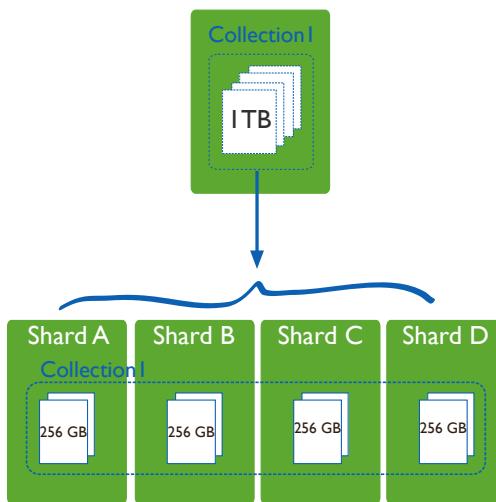
Sharding Overview

- MongoDB enables you to scale horizontally through sharding.
- Sharding is about adding more capacity to your system.
- MongoDB's sharding solution is designed to perform well on commodity hardware.
- The details of sharding are abstracted away from applications.
- Queries are performed the same way as if sending operations to a single server.
- Connections work the same by default.

When to Shard

- If you have more data than one machine can hold on its drives
- If your application is write heavy and you are experiencing too much latency.
- If your working set outgrows the memory you can allocate to a single machine.

Dividing Up Your Dataset



Note:

- When you shard a collection it is distributed across several servers.
 - Each mongod manages a subset of the data.
 - When you perform a read or write operation it will go to a router that will then direct the query to the appropriate server.
 - Depending on the operation and how your cluster is configured you may need to touch only one server to complete the operation.
-

Sharding Concepts

To understand how sharding works in MongoDB, we need to understand:

- Shard Keys
- Chunks

Shard Key

- You must define a shard key for a sharded collection.
- Based on one or more fields (like an index)
- Shard key defines a space of values
- Think of the key space like points on a line
- A key range is a segment of that line

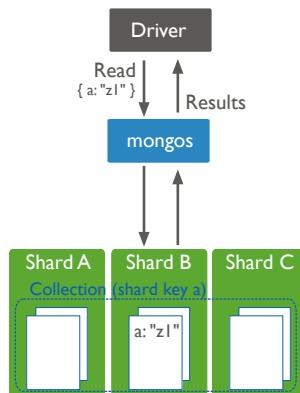
Shard Key Ranges

- A collection is partitioned based on shard key ranges.
- The shard key determines where documents are located in the cluster.
- It is used to route operations to the appropriate shard.
- For reads and writes
- Once a collection is sharded, you cannot change a shard key.

Note:

- For reads and updates, the shard key determines which shard holds the document and will handle the operation.
 - When you insert a document, the shard key determines which server you will write to.
-

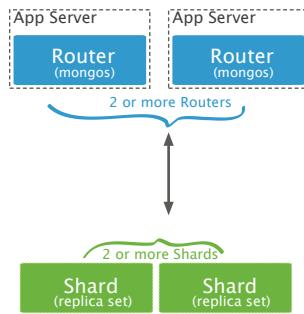
Targeted Query Using Shard Key



Chunks

- MongoDB partitions data into chunks based on shard key ranges.
- This is bookkeeping metadata.
- MongoDB attempts to keep the amount of data balanced across shards.
- This is achieved by migrating chunks from one shard to another as needed.
- There is nothing in a document that indicates its chunk.
- The document does not need to be updated if its assigned chunk changes.

Sharded Cluster Architecture



Note:

- This figure illustrates one possible architecture for a sharded cluster.
 - Each shard is a self-contained replica set.
 - Each replica set holds a partition of the data.
 - As many new shards could be added to this sharded cluster as scale requires.
 - At this point our cluster can handle a load approaching twice that of a single replica set using the same hardware.
 - As mentioned, read/write operations go through a router.
 - The server that routes requests is the mongos.
-

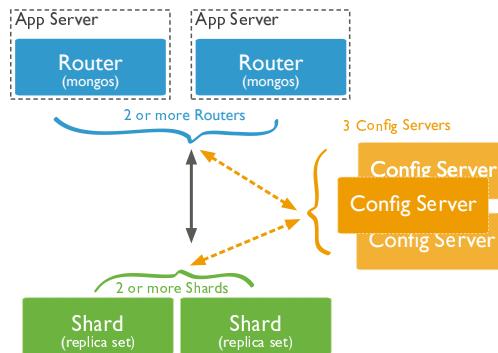
Mongos

- A mongos is responsible for accepting requests and returning results to an application driver.
- In a sharded cluster, nearly all operations go through a mongos.
- A sharded cluster can have as many mongos routers as required.
- It is typical for each application server to have one mongos.
- Always use more than one mongos to avoid a single point of failure.

Note:

- A mongos is typically deployed on an application server.
 - There should be one mongos per app server.
 - Scale with your app server.
 - Very little latency between the application and the router.
-

Config Servers



Note:

- The previous diagram was incomplete; it was missing config servers.
 - Use three config servers in production.
 - These hold only metadata about the sharded collections.
 - Where your mongos servers are
 - Any hosts that are not currently available
 - What collections you have
 - How your collections are partitioned across the cluster
 - Mongos processes use them to retrieve the state of the cluster.
 - You can access cluster metadata from a mongos by looking at the `config db`.
-

Config Server Hardware Requirements

- Quality network interfaces
- A small amount of disk space (typically a few GB)
- A small amount of RAM (typically a few GB)
- The larger the sharded cluster, the greater the config server hardware requirements.

Possible Imbalance?

- Depending on how you configure sharding, data can become unbalanced on your sharded cluster.
 - Some shards might receive more inserts than others.
 - Some shards might have documents that grow more than those in other shards.
- This may result in too much load on a single shard.
 - Reads and writes
 - Disk activity
- This would defeat the purpose of sharding.

Balancing Shards

- If a chunk grows too large MongoDB will split it into two chunks.
- The MongoDB balancer keeps chunks distributed across shards in equal numbers.
- However, a balanced sharded cluster depends on a good shard key.

With a Good Shard Key

You might easily see that:

- Reads hit only 1 or 2 shards per query.
- Writes are distributed across all servers.
- Your disk usage is evenly distributed across shards.
- Things stay this way as you scale.

With a Bad Shard Key

You might see that:

- Your reads hit every shard.
- Your writes are concentrated on one shard.
- Most of your data is on just a few shards.
- Adding more shards to the cluster will not help.

Choosing a Shard Key

Generally, you want a shard key:

- That has high cardinality
- That is used in the majority of read queries
- For which the values read and write operations use are randomly distributed
- For which the majority of reads are routed to a particular server

More Specifically

- Your shard key should be consistent with your query patterns.
- If reads usually find only one document, you only need good cardinality.
- If reads retrieve many documents:
 - Your shard key supports locality
 - Matching documents will reside on the same shard

Cardinality

- A good shard key will have high cardinality.
- A relatively small number of documents should have the same shard key.
- Otherwise operations become isolated to the same server.
- Because documents with the same shard key reside on the same shard.
- Adding more servers will not help.
- Hashing will not help.

Non-Monotonic

- A good shard key will generate new values non-monotonically.
- Datetimes, counters, and ObjectIds make bad shard keys.
- Monotonic shard keys cause all inserts to happen on the same shard.
- Hashing will solve this problem.
- However, doing range queries with a hashed shard key will perform a scatter-gather query across the cluster.

Note:

- Documents will eventually move as chunks are balanced.
 - But in the meantime one server gets hammered while others are idle.
 - And moving chunks has its own performance costs.
-

Shards Should be Replica Sets

- As the number of shards increases, the number of servers in your deployment increases.
- This increases the probability that one server will fail on any given day.
- With redundancy built into each shard you can mitigate this risk.

9.2 Balancing Shards

Learning Objectives

Upon completing this module students should understand:

- Chunks and the balancer
- The status of chunks in a newly sharded collection
- How chunk splits automatically occur
- Advantages of pre-splitting chunks
- How the balancer works

Chunks and the Balancer

- Chunks are groups of documents.
- The shard key determines which chunk a document will be contained in.
- Chunks can be split when they grow too large.
- The balancer decides where chunks go.
- It handles migrations of chunks from one server to another.

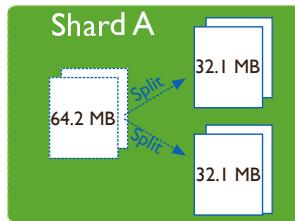
Chunks in a Newly Sharded Collection

- The range of a chunk is defined by the shard key values of the documents the chunk contains.
- When a collection is sharded it starts with just one chunk.
- The first chunk for a collection will have the range:

```
{ $minKey : 1 } to { $maxKey : 1 }
```

- All shard key values from the smallest possible to the largest fall in this chunk's range.

Chunk Splits



Note:

- When a chunk grows larger than the chunk size it will be split in half.
 - The default chunk size is 64MB.
 - A chunk can only be split between two values of a shard key.
 - If every document on a chunk has the same shard key value, it cannot be split.
 - This is why the shard key's cardinality is important
 - Chunk splitting is just a bookkeeping entry in the metadata.
 - No data bearing documents are altered.
-

Pre-Splitting Chunks

- You may pre-split data before loading data into a sharded cluster.
- Pre-splitting is useful if:
 - You plan to do a large data import early on
 - You expect a heavy initial server load and want to ensure writes are distributed

Note:

- A large data import will take time to split and balance without pre-splitting.
-

Start of a Balancing Round

- A balancing round may be initiated by any mongos in the cluster.
- This happens when the difference in the number of chunks between two shards becomes too large.
- Specifically, the difference between the shard with the most chunks and the shard with the fewest.
- A balancing round starts when the imbalance reaches:
 - 2 when the cluster has < 20 chunks
 - 4 when the cluster has 20-79 chunks
 - 8 when the cluster has 80+ chunks

Balancing is Resource Intensive

- Chunk migration requires copying all the data in the chunk from one shard to another.
- MongoDB can migrate only a single chunk at a time.
- MongoDB creates splits only after an insert operation.
- For these reasons, it is possible to define a balancing window to ensure the balancer will only run during scheduled times.

Chunk Migration Steps

1. The balancer process sends the `moveChunk` command to the source shard.
2. The source shard continues to process reads/writes for that chunk during the migration.
3. The destination shard requests documents in the chunk and begins receiving copies.
4. After receiving all documents, the destination shard receives any changes to the chunk.
5. Then the destination shard tells the config db that it has the chunk.
6. The destination shard will now handle all reads/writes.
7. The source shard deletes its copy of the chunk.

Concluding a Balancing Round

- Each chunk will move:
 - From the shard with the most chunks
 - To the shard with the fewest
- A balancing round ends when all shards differ by at most one chunk.

9.3 Shard Tags

Learning Objectives

Upon completing this module students should understand:

- The purpose for shard tags
- Advantages of using shard tags
- Potential drawbacks of shard tags

Tags - Overview

- Shard tags allow you to “tie” data to one or more shards.
- A shard tag describes a range of shard key values.
- If a chunk is in the shard tag range, it will live on a shard with that tag.

Example: DateTime

- Documents older than one year need to be kept, but are rarely used.
- You tag those ranges as “LTS” for Long Term Storage.
- Tag specific shards to hold LTS documents.
- These shards can be on cheaper, slower machines.
- Invest in high-performance servers for more frequently accessed data.

Example: Location

- You are required to keep certain data in its home country.
- You include the country in the shard tag.
- Maintain data centers within each country that house the appropriate shards.
- Meets the country requirement but allows all servers to be part of the same system.

Example: Premium Tier

- You have customers who want to pay for a “premium” tier.
- The shard key permits you to distinguish one customer’s documents from all others.
- Tag the document ranges for each customer so that their documents will be located on shards of the appropriate tier.
- Shards tagged as premium tier run on high performance servers.
- Other shards run on commodity hardware.
- See [Manage Shard Tags³⁶](#)

Note:

- As customers move from one tier to another it will be necessary to execute commands that either add a given customer’s shard key range to the premium tag or remove that range from those tagged as “premium”.
 - During balancing rounds, if the balancer detects that any chunks are not on the correct shards per configured tags, the balancer migrates chunks in tagged ranges to shards associated with those tags.
 - After re-configuring tags with a shard key range, and associating it with a shard or shards, the cluster may take some time to balance the data among the shards.
 - See: [Tiered Storage Models in MongoDB: Optimizing Latency and Cost³⁷](#).
-

Tags - Caveats

- Because tagged chunks will only be on certain servers, if you tag more than those servers can handle, you’ll have a problem.
 - You’re not only worrying about your overall server load, you’re worrying about server load for each of your tags.
- Your chunks will evenly distribute themselves across the available chunks. You cannot control things more fine grained than your tags.

³⁶ <http://docs.mongodb.org/manual/tutorial/administer-shard-tags/>

³⁷ <http://blog.mongodb.org/post/85721044164/tiered-storage-models-in-mongodb-optimizing-latency>

9.4 Lab: Setting Up a Sharded Cluster

Learning Objectives

Upon completing this module students should understand:

- How to set up a sharded cluster including:
 - Replica sets as shards
 - Config Servers
 - Mongos processes
- How to enable sharding for a database
- How to shard a collection
- How to determine where data will go

Our Sharded Cluster

- In this exercise, we will set up a cluster with 3 shards.
- Each shard will be a replica set with 3 members (including one arbiter).
- We will insert some data and see where it goes.

Sharded Cluster Configuration

- Three shards:
 1. A replica set on ports 27107, 27108, 27109
 2. A replica set on ports 27117, 27118, 27119
 3. A replica set on ports 27127, 27128, 27129
- Three config servers on ports 27217, 27218, 27219
- Two mongos servers at ports 27017 and 27018

Build Our Data Directories

On Linux or MacOS, run the following in the terminal to create the data directories we'll need.

```
mkdir -p ~/data/cluster/config/{c0,c1,c2}
mkdir -p ~/data/cluster/shard0/{m0,m1,arb}
mkdir -p ~/data/cluster/shard1/{m0,m1,arb}
mkdir -p ~/data/cluster/shard2/{m0,m1,arb}
mkdir -p ~/data/cluster/{s0,s1}
```

On Windows, run the following commands instead:

```
md c:\data\cluster\config\c0 c:\data\cluster\config\c1 c:\data\cluster\config\c2
md c:\data\cluster\shard0\m0 c:\data\cluster\shard0\m1 c:\data\cluster\shard0\arb
md c:\data\cluster\shard1\m0 c:\data\cluster\shard1\m1 c:\data\cluster\shard1\arb
md c:\data\cluster\shard2\m0 c:\data\cluster\shard2\m1 c:\data\cluster\shard2\arb
md c:\data\cluster\s0 c:\data\cluster\s1
```

Initiate a Replica Set (Linux/MacOS)

```
mongod --replSet shard0 --dbpath ~/data/cluster/shard0/m0 \
    --logpath ~/data/cluster/shard0/m0/mongod.log \
    --fork --port 27107

mongod --replSet shard0 --dbpath ~/data/cluster/shard0/m1 \
    --logpath ~/data/cluster/shard0/m1/mongod.log \
    --fork --port 27108

mongod --replSet shard0 --dbpath ~/data/cluster/shard0/arb \
    --logpath ~/data/cluster/shard0/arb/mongod.log \
    --fork --port 27109

mongo --port 27107 --eval " \
    rs.initiate(); sleep(3000); \
    rs.add('$HOSTNAME:27108'); \
    rs.addArb('$HOSTNAME:27109')"
```

Initiate a Replica Set (Windows)

```
mongod --replSet shard0 --dbpath c:\data\cluster\shard0\m0 \
    --logpath c:\data\cluster\shard0\m0\mongod.log \
    --port 27107 --oplogSize 10

mongod --replSet shard0 --dbpath c:\data\cluster\shard0\m1 \
    --logpath c:\data\cluster\shard0\m1\mongod.log \
    --port 27108 --oplogSize 10

mongod --replSet shard0 --dbpath c:\data\cluster\shard0\arb \
    --logpath c:\data\cluster\shard0\arb\mongod.log \
    --port 27109 --oplogSize 10

mongo --port 27107 --eval " \
    rs.initiate(); sleep(3000); \
    rs.add ('<HOSTNAME>:27108'); \
    rs.addArb('<HOSTNAME>:27109')"
```

Note:

- Students should replace <HOSTNAME> by the local machine name
-

Spin Up a Second Replica Set (Linux/MacOS)

```
mongod --replSet shard1 --dbpath ~/data/cluster/shard1/m0 \
    --logpath ~/data/cluster/shard1/m0/mongod.log \
    --fork --port 27117

mongod --replSet shard1 --dbpath ~/data/cluster/shard1/m1 \
    --logpath ~/data/cluster/shard1/m1/mongod.log \
    --fork --port 27118

mongod --replSet shard1 --dbpath ~/data/cluster/shard1/arb \
    --logpath ~/data/cluster/shard1/arb/mongod.log \
    --fork --port 27119

mongo --port 27117 --eval " \
    rs.initiate(); sleep(3000); \
    rs.add(''$HOSTNAME':27118'); \
    rs.addArb('$HOSTNAME':27119)"
```

Spin Up a Second Replica Set (Windows)

```
mongod --replSet shard1 --dbpath c:\data\cluster\shard1\m0 \
    --logpath c:\data\cluster\shard1\m0\mongod.log \
    --port 27117 --oplogSize 10

mongod --replSet shard1 --dbpath c:\data\cluster\shard1\m1 \
    --logpath c:\data\cluster\shard1\m1\mongod.log \
    --port 27118 --oplogSize 10

mongod --replSet shard1 --dbpath c:\data\cluster\shard1\arb \
    --logpath c:\data\cluster\shard1\arb\mongod.log \
    --port 27119 --oplogSize 10

mongo --port 27117 --eval " \
    rs.initiate(); sleep(3000); \
    rs.add('<HOSTNAME>:27118'); \
    rs.addArb('<HOSTNAME>:27119')"
```

Note:

- Students should replace <HOSTNAME> by the local machine name
-

A Third Replica Set (Linux/MacOS)

```
mongod --replSet shard2 --dbpath ~/data/cluster/shard2/m0 \
    --logpath ~/data/cluster/shard2/m0/mongod.log \
    --fork --port 27127

mongod --replSet shard2 --dbpath ~/data/cluster/shard2/m1 \
    --logpath ~/data/cluster/shard2/m1/mongod.log \
    --fork --port 27128

mongod --replSet shard2 --dbpath ~/data/cluster/shard2/arb \
    --logpath ~/data/cluster/shard2/arb/mongod.log \
    --fork --port 27129

mongo --port 27127 --eval " \
    rs.initiate(); sleep(3000); \
    rs.add(''$HOSTNAME':27128'); \
    rs.addArb('$HOSTNAME':27129)"
```

A Third Replica Set (Windows)

```
mongod --replSet shard2 --dbpath c:\data\cluster\shard2\m0 \
    --logpath c:\data\cluster\shard2\m0\mongod.log \
    --port 27127 --oplogSize 10

mongod --replSet shard2 --dbpath c:\data\cluster\shard2\m1 \
    --logpath c:\data\cluster\shard2\m1\mongod.log \
    --port 27128 --oplogSize 10

mongod --replSet shard2 --dbpath c:\data\cluster\shard2\arb \
    --logpath c:\data\cluster\shard2\arb\mongod.log \
    --port 27129 --oplogSize 10

mongo --port 27127 --eval " \
    rs.initiate(); sleep(3000); \
    rs.add('<HOSTNAME>:27128'); \
    rs.addArb('<HOSTNAME>:27129')"
```

Note:

- Students should replace <HOSTNAME> by the local machine name
-

Status Check

- Now we have three replica sets running.
- We have one for each shard.
- They do not know about each other yet.
- To make them a sharded cluster we will:
 - Build our config databases
 - Launch our mongos processes
 - Add each shard to the cluster
- To benefit from this configuration we also need to:
 - Enable sharding for a database
 - Shard at least one collection within that database

Launch Config Servers (Linux/MacOS)

```
mongod
--dbpath ~/data/cluster/config/c0 \
--logpath ~/data/cluster/config/c0/mongod.log \
--fork --port 27217 --configsvr

mongod
--dbpath ~/data/cluster/config/c1 \
--logpath ~/data/cluster/config/c1/mongod.log \
--fork --port 27218 --configsvr

mongod
--dbpath ~/data/cluster/config/c2 \
--logpath ~/data/cluster/config/c2/mongod.log \
--fork --port 27219 --configsvr
```

Launch Config Servers (Windows)

```
mongod --dbpath c:\data\cluster\config\c0 \
--logpath c:\data\cluster\config\c0\mongod.log \
--port 27217 --configsvr

mongod --dbpath c:\data\cluster\config\c1 \
--logpath c:\data\cluster\config\c1\mongod.log \
--port 27218 --configsvr

mongod --dbpath c:\data\cluster\config\c2 \
--logpath c:\data\cluster\config\c2\mongod.log \
--port 27219 --configsvr
```

Launch the Mongos Processes (Linux/MacOS)

Now our mongos's. We need to tell them about our config servers.

```
mongos --logpath ~/data/cluster/s0/mongos.log --fork --port 27017 \
--configdb $HOSTNAME:27217,$HOSTNAME:27218,$HOSTNAME:27219

mongos --logpath ~/data/cluster/s1/mongos.log --fork --port 27018 \
--configdb $HOSTNAME:27217,$HOSTNAME:27218,$HOSTNAME:27219
```

Note:

- Students should set \$HOSTNAME by the local machine name
 - echo \$HOSTNAME
-

Launch the Mongos Processes (Windows)

Now our mongos's. We need to tell them about our config servers.

```
configseedlist="$HOSTNAME:27217,$HOSTNAME:27218,$HOSTNAME:27219"
mongos --logpath c:\data\cluster\s0\mongos.log --port 27017 \
--configdb $configseedlist

mongos --logpath c:\data\cluster\s1\mongos.log --port 27018 \
--configdb localhost:27217,localhost:27218,localhost:27219
```

Add All Shards

```
echo "sh.addShard( 'shard0/$HOSTNAME:27107' ); \
sh.addShard( 'shard1/$HOSTNAME:27117' ); \
sh.addShard( 'shard2/$HOSTNAME:27127' ); sh.status()" | mongo
```

Note: Instead of doing this through a bash (or other) shell command, you may prefer to launch a mongo shell and issue each command individually.

Enable Sharding and Shard a Collection

Enable sharding for the test database, shard a collection, and insert some documents.

```
mongo --port 27017
```

```
sh.enableSharding("test")
sh.shardCollection("test.testcol", { a : 1, b : 1 } )

for (i=0; i<1000; i++) {
  docArr = [];
  for (j=0; j<1000; j++) {
    docArr.push(
    {
```

Observe What Happens

Connect to either mongos using a mongo shell and frequently issue:

```
sh.status()
```

Note:

- Point out to the students that you can see chunks get created and moved to different shards.
 - Also useful to have students run a query or two.

```
db.testcol.find( { a : { $lte : 100 } } ).explain()
```

10 Application Engineering

MongoMart Introduction (page 182) MongoMart Introduction

Java Driver Labs (MongoMart) (page 184) Build an e-commerce site backed by MongoDB (Java)

Python Driver Labs (MongoMart) (page 185) Build an e-commerce site backed by MongoDB (Python)

10.1 MongoMart Introduction

What is MongoMart

MongoMart is an on-line store for buying MongoDB merchandise. We'll use this application to learn more about interacting with MongoDB through the driver.

MongoMart Demo of Fully Implemented Version

- View Items
- View Items by Category
- Text Search
- View Item Details
- Shopping Cart

Note:

- Mongomart is in the training repo in the /mongomart directory. Distribute the materials in the /mongomart/student subdirectory to students.
 - Ensure you have a fully running version of the instructor MongoMart implementation: /instructor/java/final
 - The instructor version requires a three node replica set (uses MAJORITY writes)
 - Ensure you've loaded the dataset from /mongomart/student/data and create a text index (as described in the /mongomart/student/data/README.md file)
 - **DO NOT WALK THROUGH THIS VERSION OF THE SOURCE CODE WITH THE STUDENTS, ONLY THE STUDENT VERSION**
-

View Items

- <http://localhost:8080>
 - Pagination and page numbers
 - Click on a category
-

Note:

- Click around the site to show them how it works, click through various pages and categories (into next slide)
-

View Items by Category

- <http://localhost:8080/?category=Apparel>
- Pagination and page numbers
- “All” is listed as a category, to return to all items listing

Text Search

- <http://localhost:8080/search?query=shirt>
 - Search for any word or phrase in item title, description or slogan
 - Pagination
-

Note:

- Search for a few strings, such as “shirt”, “mug”, “mongodb”, etc.
-

View Item Details

- <http://localhost:8080/item?id=1>
 - Star rating based on reviews
 - Add a review
 - Related items
 - Add item to cart
-

Note:

- Add reviews, show how stars change, etc.
-

Shopping Cart

- <http://localhost:8080/cart>
 - Adding an item multiple times increments quantity by 1
 - Change quantity of any item
 - Changing quantity to 0 removes item
-

Note:

- Add an item to your cart
 - Update the quantity in the cart
 - Checkout doesn't work, we are only interested in the cart functionality
-

10.2 Java Driver Labs (MongoMart)

Introduction

- In this lab, we'll set up and optimize an application called MongoMart. MongoMart is an on-line store for buying MongoDB merchandise.
-

Note:

- Student skeleton code is located in /student/java
 - Ensure data is correctly loaded from /student/data
-

Lab: Setup and Connect to the Database

- Import the "item" collection to a standalone MongoDB server (without replication) as noted in the README.md file of the /data directory of MongoMart
 - Become familiar with the structure of the Java application in /java/src/main/java/mongomart/
 - Modify the MongoMart.java class to properly connect to your local database instance
-

Note:

- Solution in /instructor/java/lab_1
 - Please type in solution with the class instead of distributing source code
-

Lab: Populate All Necessary Database Queries

- After running the MongoMart.java class, navigate to “localhost:8080” to view the application
- Initially, all data is static and the application does not query the database
- Modify the ItemDao.java and CartDao.java classes to ensure all information comes from the database (do not modify the method return types or parameters)

Note:

- Solution in /instructor/java/lab_2
 - Please type in solution with the class instead of distributing source code
-

Lab: Use a Local Replica Set with a Write Concern

- It is important to use replication for production MongoDB instances, however, Lab 1 advised us to use a standalone server.
- Convert your local standalone mongod instance to a three node replica set named “shard1”
- Modify MongoMart’s MongoDB connection string to include at least two nodes from the replica set
- Modify your application’s write concern to MAJORITY for all writes to the “cart” collection, any writes to the “item” collection should continue using the default write concern of W:1

Note:

- Solution in /instructor/java/lab_3
 - Please type in solution with the class instead of distributing source code
-

10.3 Python Driver Labs (MongoMart)

Introduction

- In this lab, we’ll set up and optimize an application called MongoMart.
- MongoMart is an on-line store for buying MongoDB merchandise.

Note:

- Student skeleton code is located in /student/python
 - Ensure data is correctly loaded from /student/python
-

Lab: Setup and Connect to the Database

- Import the “item” collection to a standalone MongoDB server (without replication) as noted in the README.md file of the /data directory of MongoMart
 - Become familiar with the structure of the Python application in /
 - Start the application by running “python mongomart.py”, stop it by using ctrl-c
 - Modify the mongomart.py file to properly connect to your local database instance
-

Note:

- Solution in /instructor/python/lab_1
 - Please type in solution with the class instead of distributing source code
-

Lab: Populate All Necessary Database Queries

- After running “python mongomart.py”, navigate to “localhost:8080” to view the application
 - Initially, all data is static and the application does not query the database
 - Modify the itemDao.py and cartDao.py files to ensure all information comes from the database (you will not need to modify parameters for each DAO method)
-

Note:

- Solution in /instructor/python/lab_2
 - Please type in solution with the class instead of distributing source code
-

Lab: Use a Local Replica Set with a Write Concern

- It is important to use replication for production MongoDB instances, however, Lab 1 advised us to use a standalone server.
 - Convert your local standalone mongod instance to a three node replica set named “rs0”
 - Modify MongoMart’s MongoDB connection string to include at least two nodes from the replica set
 - Modify your application’s write concern to MAJORITY for all writes to the database
-

Note:

- Solution in /instructor/python/lab_3
 - Please type in solution with the class instead of distributing source code
-

11 Security

Authorization (page 187) Authorization in MongoDB

Lab: Administration Users (page 195) Lab on creating admin users

Lab: Create User-Defined Role (Optional) (page 197) Lab on creating custom user roles

Authentication (page 199) Authentication in MongoDB

Lab: Secure mongod (page 201) Lab on standing up a mongod with authorization enabled

Auditing (page 202) Auditing in MongoDB

Encryption (page 205) Encryption at rest in MongoDB

Lab: Secured Replica Set - KeyFile (Optional) (page 207) Using keyfiles to secure a replica set

Lab: LDAP Authentication & Authorization (Optional) (page 211) Authentication & authorization with LDAP

11.1 Authorization

Learning Objectives

Upon completing this module, students should be able to:

- Outline MongoDB's authorization model
- List authorization resources
- Describe actions users can take in relation to resources
- Create roles
- Create privileges
- Outline MongoDB built-in roles
- Grant roles to users
- Explain LDAP authorization

Authorization vs Authentication

Authorization and Authentication are generally confused and misinterpreted concepts:

- Authorization defines the rules by which users can interact with a given system:
 - Which operations can they perform
 - Over which resources
- Authentication is the mechanism by which users identify and are granted access to a system:
 - Validation of credentials and identities
 - Controls access to the system and operational interfaces

Authorization Basics

- MongoDB enforces a role-based authorization model.
- A user is granted roles that determine the user's access to database resources and operations.

The model determines:

- Which roles are granted to users
- Which privileges are associated with roles
- Which actions can be performed over different resources

Note:

- You can bring up the following questions:
 - What are privileges?
 - What kind of resources can be found on a typical database?
 - Have some open discussion about what defines an action.
 - Also you can take the opportunity to give examples of different roles in a company and how they are organized in terms of procedures and resources.
-

What is a resource?

- Databases?
- Collections?
- Documents?
- Users?
- Nodes?
- Shard?
- Replica Set?

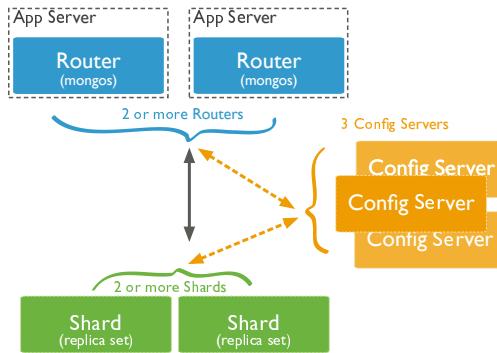
Note:

- A resource is a database, collection, set of collections, or the cluster.
 - If the resource is the cluster, the affiliated actions affect the state of the system rather than a specific database or collection.
-

Authorization Resources

- Databases
- Collections
- Cluster

Cluster Resources



Note:

- Given the distributed nature of our database, MongoDB includes the cluster resource in the authorization module.
 - Replica sets and shards comprise the cluster domain.
-

Types of Actions

Given a resource, we can consider the available actions:

- Query and write actions
- Database management actions
- Deployment management actions
- Replication actions
- Sharding actions
- Server administration actions
- Diagnostic actions
- Internal actions

Note:

- Actions are the operations that one can perform on database resources.
- The actions above are grouped by purpose.
- This organization is logical, not operational.

- Here we can ask the students which common operations they are familiar with while operating with a database and how those translate to MongoDB operations.
-

Specific Actions of Each Type

Query / Write	Database Mgmt	Deployment Mgmt
find	enableProfiler	planCacheRead
insert	createIndex	storageDetails
remove	createCollection	authSchemaUpgrade
update	changeOwnPassword	killOp

See the [complete list of actions³⁸](#) in the MongoDB documentation.

Note: These are just a few examples of the list of actions available. The full list is available in MongoDB docs: <https://docs.mongodb.org/v3.0/reference/privilege-actions/#privilege-actions>

Authorization Privileges

A privilege defines a pairing between a resource as a set of permitted actions.

Resource:

```
{"db": "yourdb", "collection": "mycollection"}
```

Action: find

Privilege:

```
{
  resource: {"db": "yourdb", "collection": "mycollection"},
  actions: ["find"]
}
```

Note:

- We want to explain that we can set a privilege that enables multiple actions on a given resource.
- Also important to highlight that we can set *loose* resources like all databases or all collections

```
{
  resource: {"db": "", "collection": ""},
  actions: ["find", "insert"]
}
```

³⁸ <https://docs.mongodb.com/manual/reference/privilege-actions/>

Authorization Roles

MongoDB grants access to data through a role-based authorization system:

- Built-in roles: pre-canned roles that cover the most common sets of privileges users may require
- User-defined roles: if there is a specific set of privileges not covered by the existing built-in roles you are able to create your own roles

Built-in Roles

Database Admin	Cluster Admin	All Databases
dbAdmin	clusterAdmin	readAnyDatabase
dbOwner	clusterManager	readWriteAnyDatabase
userAdmin	clusterMonitor	userAdminAnyDatabase
	hostManager	dbAdminAnyDatabase

Database User	Backup & Restore
read	backup
readWrite	restore

Superuser	Internal
root	system

Note: Built-in roles have been created given the generic users that interact with a database and their respective tasks.

- Database user roles: should be granted to application-side users;
 - Database administrators: roles conceived for system administrator, DBAs and security officers
 - Cluster Administrator roles: mostly for system administrators and DBAs; individuals that will deal with the overall administration of deployments
 - Backup and Restore: for applications that perform only backup and restore operations (for example: Cloud and Ops Manager)
 - All Database Roles: for global administrators of a deployment. If you want to avoid granting the same role for every single database
 - Superuser: root level operations. Generally the first user that you create on any give system should probably have a root role and then add other specific users.
 - Internal: it's documented, it's public but don't mention it too much. This a backdoor that only the cluster members (other replica set members, or a mongos) should have access to. Do not assign this role to user objects representing applications or human administrators.
-

Built-in Roles

To grant roles while creating an user:

```
use admin
db.createUser(
{
  user: "myUser",
  pwd: "$up3r$3cr7",
  roles: [
    {role: "readAnyDatabase", db: "admin"},
    {role: "dbOwner", db: "superdb"},
    {role: "readWrite", db: "yourdb"}
  ]
})
```

Built-in Roles

To grant roles to existing user:

```
use admin
db.grantRolesToUser(
  "reportsUser",
  [
    { role: "read", db: "accounts" }
  ]
)
```

Note: `grantRolesToUser` also allows to specify a `writeConcern` to ensure the durability of the operation, as any of the remaining authz methods.

```
db.grantRolesToUser(
  "reportsUser",
  [
    { role: "read", db: "accounts" }
  ],
  { w: "majority" , wtimeout: 4000 }
)
```

User-defined Roles

- If no suitable built-in role exists, we can can create a role.
- Define:
 - Role name
 - Set of privileges
 - List of inherit roles (optional)

```
use admin
db.createRole({
```

```
role: "insertAndFindOnlyMyDB",
privileges: [
  {resource: { db: "myDB", collection: "" }, actions: ["insert", "find"]}
],
roles: []})
```

Role Privileges

To check the privileges of any particular role we can get that information using the `getRole` method:

```
db.getRole("insertAndFindOnlyMyDB", {showPrivileges: true})
```

Note: There are many other authorization and user management commands and options that you should get your students acquainted with. All of those can be found in the [security reference](#)³⁹

The output of this slide command is should be similar to the following:

```
{
  "role": "insertAndFindOnlyMyDB",
  "db": "admin",
  "isBuiltin": false,
  "roles": [],
  "inheritedRoles": [],
  "privileges": [
    {
      "resource": {
        "db": "myDB",
        "collection": ""
      },
      "actions": [
        "find",
        "insert"
      ]
    }
  ],
  "inheritedPrivileges": [
    {
      "resource": {
        "db": "myDB",
        "collection": ""
      },
      "actions": [
        "find",
        "insert"
      ]
    }
  ]
}
```

³⁹ <https://docs.mongodb.org/manual/reference/security/>

LDAP Authorization

As of MongoDB 3.4, MongoDB supports *authorization* with LDAP.

How it works:

1. User authenticates via an external mechanism

```
$ mongo --username alice \
    --password secret \
    --authenticationMechanism PLAIN \
    --authenticationDatabase '$external'
```

LDAP Authorization (cont'd)

2. Username is transformed into LDAP query

```
[
  {
    match: "(.+)@ENGINEERING",
    substitution: "cn={0},ou=engineering,dc=example,dc=com"
  },
  {
    match: "(.+)@DBA",
    substitution: "cn={0},ou=dba,dc=example,dc=com"
  }
]
```

LDAP Authorization (cont'd)

3. MongoDB queries the LDAP server

- A single entity's attributes are treated as the user's roles
- Multiple entity's distinguished names are treated as the user's roles

Mongoldap

mongoldap can be used to test configurations between MongoDB and an LDAP server

```
$ mongoldap -f mongod.conf \
    --user "uid=alice,ou=Users,dc=example,dc=com" \
    --password secret
```

11.2 Lab: Administration Users

Premise

Security roles often span different levels:

- Superuser roles
- DBA roles
- System administration roles
- User administration roles
- Application roles

In this lab we will look at several types of administration roles.

User Administration user

- Generally, in complex systems, we need someone to administer users.
- This role should be different from a `root` level user for a few reasons.
- `root` level users should be used has last resort user
- Administration of users is generally related with security officers

Create User Admin user

Create a user that will administer other users:

```
db.createUser({  
    user: "securityofficer",  
    pwd: "doughnuts",  
    customData: { notes: ["admin", "the person that adds other persons"] },  
    roles: [  
        { role: "userAdminAnyDatabase", db: "admin" }  
    ]  
})
```

Note: Make sure that users understand the importance of this role:

- What happens when we need to add a new user?
- What happens when someone loses their password?
- What happens when we need to remove a user?

These are very specific operations. Also make note that we can be more granular if we want a specific user to administer only specific database users using role:

```
{ role: "userAdmin", db: "justthisdb" }
```

Create DBA user

DBAs are generally concerned with maintenance operations in the database.

```
db.createUser(  
{  
  user: "dba",  
  pwd: "i+love+indexes",  
  customData: { notes: ["admin", "the person that admins databases"] },  
  roles: [  
    { role: "dbAdmin", db: "X" }  
  ]  
})
```

If want to make sure this DBA can administer all databases of the system, which role(s) should he have? See the [MongoDB documentation](#)⁴⁰.

Note: The answer for this question is *dbAdminAnyDatabase*

In this section we should explore the existing built-in roles for database administrators. Raise the some of the following questions.

- What differences exist between `dbAdmin` and `dbOwner` roles?
 - When should we apply one or another?
-

Create a Cluster Admin user

Cluster administration is generally an operational role that differs from DBA in the sense that is more focussed on the deployment and cluster node management.

For a team managing a cluster, what roles enable individuals to do the following?

- Add and remove replica nodes
- Manage shards
- Do backups
- Cannot read data from any application database

Note: This question requires a little more thought since the correct answer actually requires two different roles:

```
use admin  
db.createUser(  
{  
  user: "theITguy",  
  pwd: "i+love+networkprotocols",  
  customData: { notes: ["admin", "the person admins machines"] },  
  roles: ["clusterAdmin", "backup"]  
})
```

Students can also come up with a custom user-defined role that should be given credit but discouraged given that we have a set of roles that perform the wanted operation.

⁴⁰ <https://docs.mongodb.com/manual/reference/built-in-roles/>

11.3 Lab: Create User-Defined Role (Optional)

Premise

- MongoDB provides a set of built-in roles.
- Please consider those before generating another role on your system.
- Sometimes it is necessary to create roles match specific the needs of a system.
- For that we can rely on user-defined roles that system administrators can create.
- This function should be carried by `userAdmin` level administration users.

Note: At this point its good moment to ask the question:

- Why can't we just have a `root` level user create the roles?
 - Why should we first have a look to the built-in roles ?
-

Define Privileges

- Roles are sets of privileges that a user is granted.
- Create a role with the following privileges:
 - User can read user details from database `brands`
 - Can list all collections of database `brands`
 - Can update all collections on database `brands`
 - Can write to the collection `automotive` in database `brands`

Create the JSON array that describes the requested set of privileges.

Note: Students should create a JSON object that will contain all of the above privileges

```
privileges: [
  {resource: { db: "brands", collection: "*" },
   actions: ["viewUser", "listCollections", "update"]},
  {resource: { db: "brands", collection: "automotive" },
   actions: ["insert"]}]
```

Here you can reference the set of available privileges that MongoDB has:
<https://docs.mongodb.org/manual/reference/privilege-actions/>

Create Role

- Given the privileges we just defined, we now need to create this role specific to database brands.
- The name of this role should be carlover
- What command do we need to issue?

Note: Students should come up with something very similar to the following:

```
db.createRole({
  role: "carlover",
  privileges: [
    { resource: { db: "brands", collection: "*" }, actions: ["viewUser",
      "listCollections", "update"] },
    { resource: { db: "brands", collection: "automotive" }, actions: ["insert"] }
  ],
  roles: []
})
```

Ask the students why we need the `roles` array and why it should be empty.

Grant Role: Part 1

We now want to grant this role to the user named `ilikecars` on the database brands.

```
use brands;
db.createUser(
{
  user: "ilikecars",
  pwd: "ferrari",
  customData: {notes: ["application user"]},
  roles: [
    {role: "carlover", db: "brands"}
  ]
})
```

Grant Role: Part 2

- We now want to grant greater responsibility to our recently created `ilikecars`!
- Let's grant the `dbOwner` role to the `ilikecars` user.

Note: Students should come up with something similar to this command:

```
use brands
db.grantRolesToUser(
  "ilikecars",
  [
    { role: "dbOwner", db: "brands" }
  ]
)
```

Students can come up with other commands like `db.updateUser("ilikecars", {...})` which is also valid.

Revoke Role

- Let's assume that the role `carlover` is no longer valid for user `ilikecars`.
- How do we revoke this role?

Note: In this section the students should be able to indicate that the command for doing this operation is the following:

```
use brands
db.revokeRolesFromUser(
  "ilikecars",
  [
    { role: "carlover", db: "brands" }
  ]
)
```

11.4 Authentication

Learning Objectives

Upon completing this module, you should understand:

- Authentication mechanisms
- External authentication
- Native authentication
- Internal node authentication
- Configuration of authentication mechanisms

Authentication

- Authentication is concerned with:
 - Validating identities
 - Managing certificates / credentials
 - Allowing accounts to connect and perform authorized operations
- MongoDB provides native authentication and supports X509 certificates, LDAP, and Kerberos as well.

Authentication Mechanisms

MongoDB supports a number of authentication mechanisms:

- SCRAM-SHA-1 (default >= 3.0)
- MONGODB-CR (legacy)
- X509 Certificates
- LDAP (MongoDB Enterprise)
- Kerberos (MongoDB Enterprise)

Note:

- Native: SCRAM-SHA-1 and MongoDB-CR are native mechanisms in the sense that they are fully managed by MongoDB instances.
 - External: LDAP and Kerberos are external authentication mechanisms and are only available with MongoDB Enterprise.
 - X509 can also be considered native in terms of management but they rely on certificates generated by 3rd parties and only enforced by MongoDB.
-

Internal Authentication

For internal authentication purposes (mechanism used by replica sets and sharded clusters) MongoDB relies on:

- Keyfiles
 - Shared password file used by replica set members
 - Hexadecimal value of 6 to 1024 chars length
- X509 Certificates

Simple Authentication Configuration

To get started we just need to make sure we are launching our mongod instances with the `--auth` parameter.

```
mongod --dbpath /data/db --auth
```

For any connections to be established to this mongod instance, the system will require a username and password.

```
mongo -u user -p
←
←
←5
Enter password:                                     MongoDB shell version: 3.2.
```

Note:

- Using the `--auth` parameter will only cause mongod to enable authentication.
- You need to create users separately.
- the shell

- You can take the opportunity to ask:
 - Q: What happens if we just launch a mongod without having any users created?
 - A: Nothing happens, we just can't access the instance.
- cover localhost exception
- create a first user with the following, avoid using root as the role

```
db.createUser({user:'admin',pwd:'pwd',roles:[{role:'userAdminAnyDatabase',db:'admin'}  
← ] })
```

- run the following to see who is authenticated:

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

11.5 Lab: Secure mongod

Premise

It is time for us to get started setting up our first MongoDB instance with authentication enabled!

Note:

- Expected time: 5 minutes
- Prerequisites:
 - Students should have installed MongoDB Enterprise or compiled MongoDB community with –ssl flags.

Launch mongod

Let's start by launching a mongod instance:

```
mkdir /data/secure_instance_dbpath  
mongod --dbpath /data/secure_instance_dbpath --port 28000
```

At this point there is nothing special about this setup. It is just an ordinary mongod instance ready to receive connections.

Root level user

Create a root level user:

```
mongo --port 28000 admin // Puts you in the _admin_ database
```

```
use admin
db.createUser( {
  user: "maestro",
  pwd: "maestro+rules",
  customData: { information_field: "information value" },
  roles: [ {role: "root", db: "admin"} ]
} )
```

Note: *root* is a superuser role so make sure you mention the privileges.

<https://docs.mongodb.org/manual/reference/built-in-roles/#superuser-roles>

Enable Authentication

Launch mongod with auth enabled

```
mongo admin --port 28000 --eval 'db.shutdownServer()'
mongod --port 28000 --dbpath /data/secure_instance_dbpath --auth
```

Note: With these commands, you can mention that:

- *mongo admin --eval 'db.shutdownServer()'* is a clean shutdown of the server
 - *timeoutSecs* is parameter that can be used to control the shutdown operation
 - Especially w/ replica sets, which they'll be using, soon.
-

Connect using the recently created *maestro* user.

```
mongo --port 28000 admin -u maestro -p
```

11.6 Auditing

Learning Objectives

Upon completing this module, you should be able to:

- Outline the auditing capabilities of MongoDB
- Enable auditing
- Summarize auditing configuration options

Auditing

- MongoDB Enterprise includes an auditing capability for mongod and mongos instances.
- The auditing facility allows administrators and users to track system activity
- Important for deployments with multiple users and applications.

Audit Events

Once enabled, the auditing system can record the following operations:

- Schema
- Replica set and sharded cluster
- Authentication and authorization
- CRUD operations (DML, off by default)

Auditing Configuration

The following are command-line parameters to mongod/mongos used to configure auditing.

Enable auditing with --auditDestination.

- --auditDestination: where to write the audit log
 - syslog
 - console
 - file
- --auditPath: audit log path in case we define “file” as the destination

Auditing Configuration (cont'd)

- --auditFormat: the output format of the emitted event messages
 - BSON
 - JSON
- --auditFilter: an expression that will filter the types of events the system records

By default we only audit DDL operations but we can also enable DML (requires auditAuthorizationSuccess set to true)

Note:

- Explain what DML and DDL operations are:
 - DML means data manipulation language (inserts, updates, removes, grant user role ...)
 - DDL means data definition language (create collection, index, drop database ...)
- Q: Why do we not enable DML by default?
 - A: Due to the performance impact of logging all write operations
 - Q: In what circumstances might we want to enable DML?

- A: On highly sensitive namespaces or for given set of users.
-

Auditing Message

The audit facility will launch a message every time an auditable event occurs:

```
{  
    atype: <String>,  
    ts : { "$date": <timestamp> },  
    local: { ip: <String>, port: <int> },  
    remote: { ip: <String>, port: <int> },  
    users : [ { user: <String>, db: <String> }, ... ],  
    roles: [ { role: <String>, db: <String> }, ... ],  
    param: <document>,  
    result: <int>  
}
```

Auditing Configuration

If we want to configure our audit system to generate a *JSON* file we would need express the following command:

```
mongod --auditDestination file --auditPath /some/dir/audit.log --auditFormat JSON
```

If we want to capture events from a particular user *myUser*:

```
mongod --auditDestination syslog --auditFilter '{"users.user": "myUser"}'
```

To enable DML we need to set a specific parameter:

```
mongod --auditDestination console --setParameter auditAuthorizationSuccess=true
```

Note:

- We can define filters on any particular field of the audit message
 - These will work as regular MongoDB query filter expressions, but not all operators will apply.
 - Be creative and ask students to set different filters based on roles or incoming connections.
-

11.7 Encryption

Learning Objectives

Upon completing this module, students should understand:

- The encryption capabilities of MongoDB
- Network encryption
- Native encryption
- Third party integrations

Encryption

MongoDB offers two levels of encryption

- Transport layer
- Encryption at rest (MongoDB Enterprise >=3.2)

Note:

- important to note to students that encryption at rest is an enterprise version feature
-

Network Encryption

- MongoDB enables TLS/SSL for transport layer encryption of traffic between nodes in a cluster.
- Three different network architecture options are available:
 - Encryption of application traffic connections
 - Full encryption of all connections
 - Mixed encryption between nodes

Note:

- mixed encryption means that we can have nodes in a replica set that communicate with some nodes not encrypted and others encrypted
-

Native Encryption

MongoDB Enterprise comes with a encrypted storage engine.

- Native encryption supported by WiredTiger
- Encrypts data at rest
 - AES256-CBC: 256-bit Advanced Encryption Standard in Cipher Block Chaining mode (default)
 - * symmetric key (same key to encrypt and decrypt)
 - AES256-GCM: 256-bit Advanced Encryption Standard in Galois/Counter Mode
 - FIPS is also available
- Enables integration with key management tools

Encryption and Replication

- Encryption is not part of replication:
 - Data is not natively encrypted on the wire
 - * Requires transport encryption to ensure secured transmission
 - Encryption keys are not replicated
 - * Each node should have their own individual keys

Note:

- Important to raise awareness to this point
 - Many students might get the impression that configuring encryption in one of the nodes would be enough when that's not the case
 - Wire data needs to be encrypted through TLS/SSL configuration
 - Encrypted Storage Engine only provides encryption on data at rest
 - We should use different encryption keys for different nodes.
-

Third Party Integration

- Key Management Interoperability Protocol (KMIP)
 - Integrates with Vormetric Data Security Manager (DSM) and SafeNet KeySecure
- Storage Encryption
 - Linux Unified Key Setup (LUKS)
 - IBM Guardium Data Encryption
 - Vormetric Data Security Platform
 - * Also enables Application Level Encryption on per-field or per-document
 - Bitlocker Drive Encryption

Note:

- MongoDB offers some integration options for Key Management and Storage Encryption
 - Key managers are recommended for good security practices like key expiration and rotation
 - Key managers are important if we want to be complaint with HIPAA, PCI-DSS, and FERPA certifications
-

11.8 Lab: Secured Replica Set - KeyFile (Optional)

Premise

Security and Replication are two aspects that are often neglected during the Development phase to favor usability and faster development.

These are also important aspects to take in consideration for your Production environments, since you probably don't want to have your production environment **Unsecured** and without **High Availability**!

This lab is to get fully acquainted with all necessary steps to create a secured replica set using the `keyfile` for cluster authentication mode

Setup Secured Replica Set

A few steps are required to fully setup a secured Replica Set:

1. Instantiate one `mongod` node with no auth enabled
2. Create a `root` level user
3. Create a `clusterAdmin` user
4. Generate a keyfile for internal node authentication
5. Re-instantiate a `mongod` with auth enabled, `keyfile` defined and `replSet` name
6. Add Replica Set nodes

We will also be basing our setup using [MongoDB configuration files](#)⁴¹

Note:

- This might be a good opportunity to have students work in groups.
 - If we can guarantee:
 - connectivity between all students workstations
 - administration rights over the workstations
 - Then we can go ahead and group students together to accomplish these tasks.
-

⁴¹ <https://docs.mongodb.org/manual/reference/configuration-options/>

Instantiate mongod

This is a rather simple operation that requires just a simple instruction:

```
$ pwd  
/data  
$ mkdir -p /data/secure_replset/{1,2,3}; cd secure_replset/1
```

Then go to [this yaml file⁴²](#) and copy it into your clipboard

```
$ pbpaste > mongod.conf; cat mongod.conf
```

Instantiate mongod (cont'd)

```
systemLog:  
  destination: file  
  path: "/data/secure_replset/1/mongod.log"  
  logAppend: true  
storage:  
  dbPath: "/data/secure_replset/1"  
  wiredTiger:  
    engineConfig:  
      cacheSizeGB: 1  
net:  
  port: 28001  
processManagement:  
  fork: true  
# setParameter:  
#   enableLocalhostAuthBypass: false  
# security:  
#   keyFile: /data/secure_replset/1/mongodb-keyfile
```

Instantiate mongod (cont'd)

After defining the basic configuration we just need to call mongod passing the configuration file.

```
mongod -f mongod.conf
```

Note: If not mentioned before this is a good opportunity to have the students review the configuration options that MongoDB configuration files have.

Make sure you emphasize the security options:

<https://docs.mongodb.org/manual/reference/configuration-options/#security-options>

⁴² https://github.com/thatnerd/work-public/blob/master/mongodb_trainings/secure_replset_config.yaml

Create root user

We start by creating our typical `root` user:

```
$ mongo admin --port 28001
```

```
> use admin
> db.createUser(
{
  user: "maestro",
  pwd: "maestro+rules",
  roles: [
    { role: "root", db: "admin" }
  ]
})
```

Create clusterAdmin user

We then need to create a `clusterAdmin` user to enable management of our replica set.

```
$ mongo admin --port 28001
```

```
> db.createUser(
{
  user: "pivot",
  pwd: "i+like+nodes",
  roles: [
    { role: "clusterAdmin", db: "admin" }
  ]
})
```

Generate a keyfile

For internal Replica Set authentication we need to use a keyfile.

```
openssl rand -base64 741 > /data/secure_replset/1/mongodb-keyfile
chmod 600 /data/secure_replset/1/mongodb-keyfile
```

Add keyfile to the configuration file

Now that we have the `keyfile` generated it's time to add that information to our configuration file. Just un-comment the last few lines.

```
systemLog:
  destination: file
  path: "/data/secure_replset/1/mongod.log"
  logAppend: true
storage:
  dbPath: "/data/secure_replset/1"
net:
  port: 28001
processManagement:
  fork: true
```

```
setParameter:  
  enableLocalhostAuthBypass: false  
security:  
  keyFile: /data/secure_replset/1/mongodb-keyfile
```

Note:

- On this configuration we are focusing on getting the internal authentication to work with a keyfile.
 - There are other options that you may want to bring up with the students.
 - It's probably a good time to ask?
 - If I want to use x509 certificates what other settings would I need to be adding?
 - Once I configured the system to use keyfile is that immutable?
-

Configuring Replica Set

- Now it's time to configure our Replica Set
- The desired setup for this Replica Set should be named "VAULT"
- It should consist of 3 data bearing nodes

Note:

We expect the students to first draft a set of instructions that they need to complete:

- Add the replication configuration to the config file
- Connect with *pivot* user to initiate the replica set
- Instantiate the remaining nodes
- Add those nodes to the replica set

The end result should be something similar to the following:

```
> rs.isMaster()  
{  
  "setName" : "VAULT",  
  "setVersion" : 3,  
  "ismaster" : true,  
  "secondary" : false,  
  "hosts" : [  
    "node0:50000",  
    "node1:50000",  
    "node2:50000"  
,  
  ...  
}
```

They will probably have something to add to their config files.

Here's the link⁴³

and it looks like this:

⁴³ https://github.com/thatnerd/work-public/blob/master/mongodb_trainings/secure_replset_config_including_replset_options.yaml

```
replication:  
  oplogSizeMB: 100  
  replSetName: "VAULT"  
  enableMajorityReadConcern: true
```

11.9 Lab: LDAP Authentication & Authorization (Optional)

Premise

- Authentication and authorization with an external service (like LDAP) is an important functionality for large organizations that rely on centralized user management tools.
 - This lab is designed to get you familiar with the procedure to run a mongod with authentication and authorization enabled with an external LDAP service.
-

Note:

- Take this opportunity to setup your OpenLDAP VM which students will connect to.
 1. Download and extract `ldapserver.tar.gz`⁴⁴
 2. Run the following commands:

```
$ vagrant up  
$ vagrant ssh
```

3. You can confirm that everything is up and running correctly by running:

```
$ ldapwhoami -vvv -H ldapi:/// -D 'uid=alice,ou=Users,dc=mongodb,dc=com' -  
-x -w secret
```

Test Connection to LDAP

- An LDAP server is up and running for you to connect to.
 - **Server Info:**
 - **Server Address:** 192.168.19.100:8389
 - **User:** uid=alice,ou=Users,dc=mongodb,dc=com
 - **Password:** secret
-

Note:

- Make sure to update the IP above with your computer's own local IP address.
 - We're using 8389 instead of the default LDAP port of 389 because we need to forward the port from our VM to our host machine so students can access the OpenLDAP server.
-

⁴⁴ <http://s3.amazonaws.com/mongodb-training/vms/ldapserver.tar.gz>

Test Connection to LDAP (cont'd)

- Your goal is to fill in the following configuration file and get mongoldap to successfully talk to the LDAP server with the following command:

```
$ mongoldap --config mongod.conf --user alice --password secret
```

```
...
security:
  authorization: "enabled"
  ldap:
    servers: "XXXXXXXXXXXXXX:8389"
    authz:
      queryTemplate: "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
      userToDNMapping: '[{match: "XXX", substitution:
        ↳"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"}]'
    transportSecurity: "none"
    bind:
      method: "simple"
  setParameter:
    authenticationMechanisms: PLAIN
```

Note:

- The key to this exercise is to build a correct `queryTemplate` and `userToDNMapping`.
- The correct missing parts of the config are:

```
security:
  ldap:
    servers: "192.168.19.100" // students will update with your local IP, ↳
    ↳currently this is the IP of the VM
    authz:
      queryTemplate: "ou=Users,dc=mongodb,dc=com?one?(member={USER})"
      userToDNMapping: '[{match: "(.)", substitution: "uid={0},ou=Users,dc=mongodb,
        ↳dc=com"}]'
```

- Inside `ldapserver.tar.gz`⁴⁵ there is a sample `mongod.conf` that should be able to talk to the LDAP server.
-

Authentication with LDAP

- Once you've successfully connected to LDAP with `mongoldap` you should be able to use the same config file with `mongod`.

```
$ mongod --config mongod.conf
```

- From here you should be able to authenticate with `alice` and `secret`.

```
$ mongo --username alice \
  --password secret \
  --authenticationMechanism PLAIN \
  --authenticationDatabase '$external'
```

⁴⁵ <http://s3.amazonaws.com/mongodb-training/vms/ldapserver.tar.gz>

Note:

- The following command will test your connection to the OpenLDAP server

```
$ mongoldap --config mongod.conf --user alice --password secret
```

Authorization with LDAP

- After successfully *authenticating* with LDAP, you'll need to take advantage of the **localhost exception** to enable *authorization* with LDAP.
- Create a role that allows anyone who is apart of the **cn=admins,ou=Users,dc=mongodb,dc=com** LDAP group to be able to manage users (e.g., inheriting userAdminAnyDatabase).
- To confirm that you've successfully setup authorization the following command should execute without error if you're authenticated as `alice` since she's apart of the group.

```
> use admin
> db.getRoles()
```

Note:

- The following command will create the necessary role.

```
> db.createRole({
    role: 'cn=admins,ou=Users,dc=mongodb,dc=com',
    privileges: [],
    roles:['userAdminAnyDatabase']
});
```

12 Views

Views Tutorial (page 214) Creating and Deleting views

Lab: Vertical Views (page 216) Creating a vertical view lab

Lab: Horizontal Views (page 218) Creating a horizontal view lab

Lab: Reshaped Views (page 219) Creating a reshaped view lab

12.1 Views Tutorial

Learning Objectives

Upon completing this module students should understand:

- What a view is
- What views are useful for
- How to create and drop a view
- Internal mechanisms of a view

What a View is

- A non-materialized collection created from one or more other collections.
- For those who are used to SQL, MongoDB views are equivalent.
- Can be thought of as a predefined aggregation that can be queried.

What Views are useful for

- Views provide an excellent mechanism for data abstraction.
- Views provide an excellent means to protect data
 - Sensitive data from a collection can be projected out of the view
 - Views are read only
 - Combined with role based authorization allows to select information by roles

How to create and drop a view

- Creating a view is a straightforward process.
 - We must give our view a <name>, which will be the name we can access it by
 - We must specify a <source> collection
 - We must define an aggregation <pipeline> to fill our new view with data
 - Optionally, we may also specify a <collation>

Note:

- Views do not have any data, nor do they cache their data.
 - Views can only be created from another view. In order to combine data from multiple collections, users can use the familiar aggregation operator `$lookup`
-

Example - Creating a view

```
// db.createView(<name>, <source>, <pipeline>, <collation>)
db.createView("contact_info", "patients", [
  { $project: {
    _id: 0,
    first_name: 1,
    last_name: 1,
    gender: 1,
    email: 1,
    phone: 1
  }
})
])
```

Note:

- Views do have limits and there are certain operators that can't be used.
 - MapReduce
 - \$text
 - Certain Projection operators
 - * \$
 - * \$slice
 - * \$elemMatch
 - * \$meta
-

Dropping Views

- Views can be dropped like any other collection

```
db.contact_info.drop()
```

Internal mechanisms of a view

Views can be thought of as a predefined aggregation. As such:

- Views do not contain any data nor take disk space by themselves
- Views will benefit greatly from indexes on the source collection in their \$match stage
- Views are considered sharded if their underlying collection is sharded.
- Views are immutable, and cannot be renamed

Note:

- The aggregation to create the view is ran every time the view is queried
- Creating views from views can be slow, as we are cascading aggregations.
 - MongoDB will neither check collection fields nor existence before creating a view.
 - If there is no collection, the view will return an empty cursor

12.2 Lab: Vertical Views

Exercise: Vertical View Creation

It is useful to create vertical views to give us a lens into a subset of our overall data.

- Start by importing the necessary data if you have not already.

```
tar xvzf views_dataset.tar.gz
```

```
# for version >= 3.4
mongoimport -d companies -c complaints --drop views_dataset.json
```

To help you verify your work, there are 404816 entries in this dataset.

Note:

```
use companies;
db.complaints.count() // 404816
```

Exercise : Vertical View Creation Instructions

Once you've verified the data import was successful:

- Create a view that only shows complaints in New York
- Ensure the view shows the most recently submitted complaints by default

Exercise : Vertical View Creation Instructions Result

The resulting data should look like:

```
db.companyComplaintsInNY.findOne()
{
    "complaint_id" : 1416985,
    "product" : "Debt collection",
    "sub-product" : "",
    "issue" : "Cont'd attempts collect debt not owed",
    "sub-issue" : "Debt is not mine",
    "state" : "NY",
    "zip_code" : 11360,
    "submitted_via" : "Web",
    "date_received" : ISODate("2015-06-11T04:00:00Z"),
    "date_sent_to_company" : ISODate("2015-06-11T04:00:00Z"),
    "company" : "Transworld Systems Inc.",
    "company_response" : "In progress",
    "timely_response" : "Yes",
    "consumer_disputed" : ""
}
```

Note:

```
use companies;
db.createView("companyComplaintsInNY", "complaints", [
  { $match: { "state": "NY" } },
  { $sort: { "date_received": -1 } }
])
```

Exercise: Vertical View Creation Validation Instructions

Verify the view is functioning correctly.

- Insert the document on the following slide
- Query your newly created view
- The newly inserted document should be the first in the result set

Exercise: Vertical View Creation Validation Instructions Cont'd

```
db.complaints.insert({
    "complaint_id" : 987654,
    "product" : "Food and Beverage",
    "sub-product" : "Coffee",
    "issue" : "Coffee is too hot",
    "sub-issue" : "",
    "state" : "NY",
    "zip_code" : 11360,
    "submitted_via" : "Web",
    "date_received" : new Date(),
    "date_sent_to_company" : "pending",
    "company" : "CoffeeMerks",
    "company_response" : "",
    "timely_response" : "",
    "consumer_disputed" : ""
})
```

12.3 Lab: Horizontal Views

Exercise: Horizontal View Creation

Horizontal views allow us to provide a selective set of fields of the underlying collection of documents for efficiency and role-based filtering of data.

- Let's go ahead and create a horizontal view of our dataset.
- Start by importing the necessary data if you have not already.

```
mongoimport -d companies -c complaints --drop views_dataset.json
```

To help you verify your work, there are 404816 entries in this dataset.

Note:

```
use companies;
db.complaints.count() // 404816
```

Exercise : Horizontal View Creation Instructions

Once you've verified the data import was successful, create a view that only shows the the following fields:

- product
- company
- state

Exercise : Horizontal View Creation Instructions Result

The resulting data should look like:

```
db.productComplaints.findOne()
{
  "product" : "Debt collection",
  "state" : "FL",
  "company" : "Enhanced Recovery Company, LLC"
}
```

Note:

```
use companies;
db.createView("productComplaints", "complaints", [
  { $project: { _id: 0, product: 1, state: 1, company: 1 } }
])
```

12.4 Lab: Reshaped Views

Exercise: Reshaped View

We can create a reshaped view of a collection to enable more intuitive data queries and make it easier for applications to perform analytics.

It is also possible to create a view from a view.

- Use the aggregation framework to create a reshaped view of our dataset.
- It is necessary to have completed *Lab: Horizontal Views* (page 218)

Exercise : Reshaped View Specification

Create a view that can be queried by company name that shows the amount of complaints by state. The resulting data should look like:

```
db.companyComplaintsByState.find({ "company": "ROCKY MOUNTAIN MORTGAGE COMPANY" })

{
  "company" : "ROCKY MOUNTAIN MORTGAGE COMPANY",
  "states" : [
    {
      "state" : "TX",
      "count" : 4
    }
  ]
}
```

Note:

```
use companies;
db.createView("companyComplaintsByState", "productComplaints", [
  {
```

```
$group: {
    "_id": {
        company: "$company",
        state: "$state"
    },
    count: {
        $sum: 1
    }
},
{
    $group: {
        "_id": "$_id.company",
        "byState": {
            "$push": {
                "state": "$_id.state",
                "count": "$count"
            }
        }
    },
    $project: {
        _id: 0,
        "company": "$_id",
        "states": "$byState"
    }
}
])
```

13 Reporting Tools and Diagnostics

Performance Troubleshooting (page 221) An introduction to reporting and diagnostic tools for MongoDB

13.1 Performance Troubleshooting

Learning Objectives

Upon completing this module students should understand basic performance troubleshooting techniques and tools including:

- mongostat
- mongotop
- db.setProfilingLevel()
- db.currentOp()
- db.<COLLECTION>.stats()
- db.serverStatus()

mongostat and mongotop

- mongostat samples a server every second.
 - See current ops, pagefaults, network traffic, etc.
 - Does not give a view into historic performance; use Ops Manager for that.
- mongotop looks at the time spent on reads/writes in each collection.

Exercise: mongostat (setup)

In one window, perform the following commands.

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    arr = [];
    for (j=1; j<=1000; j++) {
        doc = { _id: (1000 * (i-1) + j), a: i, b: j, c: (1000 * (i-1)+ j) };
        arr.push(doc);
    };
    db.testcol.insertMany(arr);
    var x = db.testcol.find( { b : 255 } );
    x.next();
    var x = db.testcol.find( { _id : 1000 * (i-1) + 255 } );
    x.next();
    var x = "asdf";
    db.testcol.updateOne( { a : i, b : 255 }, { $set : { d : x.pad(1000) } });
    print(i)
}
```

Exercise: mongostat (run)

- In another window/tab, run mongostat.
- You will see:
 - Inserts
 - Queries
 - Updates

Exercise: mongostat (create index)

- In a third window, create an index when you see things slowing down:

```
db.testcol.createIndex( { a : 1, b : 1 } )
```

- Look at mongostat.
- Notice that things are going significantly faster.
- Then, let's drop that and build another index.

```
db.testcol.dropIndexes()
db.testcol.createIndex( { b : 1, a : 1 } )
```

Exercise: mongotop

Perform the following then, in another window, run mongotop.

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    arr = [];
    for (j=1; j<=1000; j++) {
        doc = {_id: (1000*(i-1)+j), a: i, b: j, c: (1000*(i-1)+j)};
        arr.push(doc)
    };
    db.testcol.insertMany(arr);
    var x = db.testcol.find( {b: 255} );
    x.next();
    var x = db.testcol.find( {_id: 1000*(i-1)+255} );
    x.next();
    var x = "asdf";
    db.testcol.updateOne( {a: i, b: 255}, {$set: {d: x.pad(1000)}} );
    print(i)
}
```

Note: Direct the students to the fact that you can see the activity on the server for reads/writes/total.

db.currentOp()

- currentOp is a tool that asks what the db is doing at the moment.
- currentOp is useful for finding long-running processes.
- Fields of interest:
 - microsecs_running
 - op
 - query
 - lock
 - waitingForLock

Exercise: db.currentOp()

Do the following then, connect with a separate shell, and repeatedly run `db.currentOp()`.

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    arr = [];
    for (j=1; j<=1000; j++) {
        doc = {_id: (1000*(i-1)+j), a: i, b: j, c: (1000*(i-1)+j)};
        arr.push(doc)
    };
    db.testcol.insertMany(arr);
    var x = db.testcol.find( {b: 255} );
    x.next();
    var x = db.testcol.find( {_id: 1000*(i-1)+255} );
    x.next();
    var x = "asdf";
    db.testcol.updateOne( {a: i, b: 255}, {$set: {d: x.pad(1000)}} );
    print(i)
}
```

Note: Point out to students that the running time gets longer & longer, on average.

db.<COLLECTION>.stats()

- Used to view the current stats for a collection.
- Everything is in bytes; use the multiplier parameter to view in KB, MB, etc
- You can also use `db.stats()` to do this at scope of the entire database

Exercise: Using Collection Stats

Look at the output of the following:

```
db.testcol.drop()
db.testcol.insertOne( { a : 1 } )
db.testcol.stats()
var x = "asdf"
db.testcol2.insertOne( { a : x.pad(10000000) } )
db.testcol2.stats()
db.stats()
```

The Profiler

- Off by default.
- To reset, db.setProfilingLevel(0)
- At setting 1, it captures “slow” queries.
- You may define what “slow” is.
- Default is 100ms: db.setProfilingLevel(1)
- E.g., to capture 20 ms: db.setProfilingLevel(1, 20)

The Profiler (continued)

- If the profiler level is 2, it captures all queries.
 - This will severely impact performance.
 - Turns all reads into writes.
- Always turn the profiler off when done (set level to 0)
- Creates db.system.profile collection

Exercise: Exploring the Profiler

Perform the following, then look in your db.system.profile.

```
db.setProfilingLevel(0)
db.testcol.drop()
db.system.profile.drop()
db.setProfilingLevel(2)
db.testcol.insertOne( { a : 1 } )
db.testcol.find()
var x = "asdf"
db.testcol.insertOne( { a : x.pad(10000000) } ) // ~10 MB
db.setProfilingLevel(0)
db.system.profile.find().pretty()
```

Note:

- Mention to the students what the fields mean.
- Things to keep in mind:

- op can be command, query, or update
 - ns is sometimes the db.<COLLECTION> namespace
 - * but sometimes db.\$cmd for commands
 - key updates refers to index keys
 - ts (timestamp) is useful for some queries if problems cluster.
-

db.serverStatus()

- Takes a snapshot of server status.
- By taking diffs, you can see system trends.
- Most of the data that MMS gets is from here.

Exercise: Using db.serverStatus()

- Open up two windows. In the first, type:

```
db.testcol.drop()  
var x = "asdf"  
for (i=0; i<=10000000; i++) {  
    db.testcol.insertOne( { a : x.pad(100000) } )  
}
```

- In the second window, type periodically:

```
var x = db.serverStatus(); x.metrics.document
```

Analyzing Profiler Data

- Enable the profiler at default settings.
- Run for 5 seconds.
- Slow operations are captured.
- The issue is there is not a proper index on the message field.
- You will see how fast documents are getting inserted.
- It will be slow b/c the documents are big.

Performance Improvement Techniques

- Appropriate write concerns
- Bulk operations
- Good schema design
- Good Shard Key choice
- Good indexes

Performance Tips: Write Concern

- Increasing the write concern increases data safety.
- This will have an impact on performance, however.
- This is especially true when there are network issues.

Bulk Operations

- Using bulk operations (including `insertMany` and `updateMany`) can improve performance, especially when using write concern greater than 1.
- These enable the server to amortize acknowledgement.
- Can be done with both `insertMany` and `updateMany`.

Exercise: Comparing `insertMany` with `mongostat`

Let's spin up a 3-member replica set:

```
mkdir -p /data/replset/{1,2,3}
mongod --logpath /data/replset/1/mongod.log \
    --dbpath /data/replset/1 --replSet mySet --port 27017 --fork
mongod --logpath /data/replset/2/mongod.log \
    --dbpath /data/replset/2 --replSet mySet --port 27018 --fork
mongod --logpath /data/replset/3/mongod.log \
    --dbpath /data/replset/3 --replSet mySet --port 27019 --fork

echo "conf = {_id: 'mySet', members: [{_id: 0, host: 'localhost:27017'}, \
    {_id: 1, host: 'localhost:27018'}, {_id: 2, host: 'localhost:27019'}]}; \
    rs.initiate(conf)" | mongo
```

mongostat, insertOne with {w: 1}

Perform the following, with writeConcern : 1 and insertOne():

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    for (j=1; j<=1000; j++) {
        db.testcol.insertOne( { _id : (1000 * (i-1) + j),
                               a : i, b : j, c : (1000 * (i-1)+ j) },
                               { writeConcern : { w : 1 } } );
    };
    print(i);
}
```

Run mongostat and see how fast that happens.

Multiple insertOne s with {w: 3}

Increase the write concern to 3 (safer but slower):

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    for (j=1; j<=1000; j++) {
        db.testcol.insertOne(
            { _id: (1000 * (i-1) + j), a: i, b: j, c: (1000 * (i-1)+ j) },
            { writeConcern: { w: 3 } }
        );
    };
    print(i);
}
```

Again, run mongostat.

mongostat, insertMany with {w: 3}

- Finally, let's use insertMany to our advantage:
- Note that writeConcern is still { w: 3 }

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    arr = []
    for (j=1; j<=1000; j++) {
        arr.push(
            { _id: (1000 * (i-1) + j), a: i, b: j, c: (1000 * (i-1)+ j) }
        );
    };
    db.testcol.insertMany( arr, { writeConcern : { w : 3 } } );
    print(i);
}
```

Schema Design

- The structure of documents affects performance.
- Optimize for your application's read/write patterns.
- We want as few requests to the database as possible to perform a given application task.
- See the data modeling section for more information.

Shard Key Considerations

- Choose a shard key that distributes load across your cluster.
- Create a shard key such that only a small number of documents will have the same value.
- Create a shard key that has a high degree of randomness.
- Your shard key should enable a mongos to target a single shard for a given query.

Indexes and Performance

- Reads and writes that don't use an index will cripple performance.
- In compound indexes, order matters:
 - Sort on a field that comes before any range used in the index.
 - You can't skip fields; they must be used in order.
 - Revisit the indexing section for more detail.

14 Backup and Recovery

Backup and Recovery (page 229) An overview of backup options for MongoDB

14.1 Backup and Recovery

Disasters Do Happen



Human Disasters



Terminology: RPO vs. RTO

- **Recovery Point Objective (RPO):** How much data can you afford to lose?
- **Recovery Time Objective (RTO):** How long can you afford to be off-line?

Terminology: DR vs. HA

- **Disaster Recovery (DR)**
- **High Availability (HA)**
- Distinct business requirements
- Technical solutions may converge

Quiz

- Q: What's the hardest thing about backups?
- A: Restoring them!
- **Regularly test that restoration works!**

Backup Options

- Document Level
 - Logical
 - mongodump, mongorestore
- File system level
 - Physical
 - Copy files
 - Volume/disk snapshots

Document Level: `mongodump`

- Dumps collection to BSON files
- Mirrors your structure
- Can be run live or in offline mode
- Does not include indexes (rebuilt during restore)
- `--dbpath` for direct file access
- `--oplog` to record oplog while backing up
- `--query/filter` selective dump

`mongodump`

```
$ mongodump --help
Export MongoDB data to BSON files.

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 ( /s1,s2 for
                         server port. Can also use --host hostname
  --port arg                server port. Can also use --host hostname
  -u [ --username ] arg      username
  -p [ --password ] arg      password
  --dbpath arg                directly access mongod database files in path
  -d [ --db ] arg                database to use
  -c [ --collection ] arg      collection to use (some commands)
  -o [ --out ] arg (=dump) output directory or "-" for stdout
  -q [ --query ] arg          json query
  --oplog                   Use oplog for point-in-time snapshotting
```

File System Level

- Must use journaling!
- Copy /data/db files
- Or snapshot volume (e.g., LVM, SAN, EBS)
- *Seriously, always use journaling!*

Ensure Consistency

Flush RAM to disk and stop accepting writes:

- db.fsyncLock()
- Copy/Snapshot
- db.fsyncUnlock()

File System Backups: Pros and Cons

- Entire database
- Backup files will be large
- Fastest way to create a backup
- Fastest way to restore a backup

Document Level: mongorestore

- mongorestore
- --oplogReplay replay oplog to point-in-time

File System Restores

- All database files
- Selected databases or collections
- Replay Oplog

Backup Sharded Cluster

1. Stop Balancer (and wait) or no balancing window
2. Stop one config server (data R/O)
3. Backup Data (shards, config)
4. Restart config server
5. Resume Balancer

Restore Sharded Cluster

1. Dissimilar # shards to restore to
2. Different shard keys?
3. Selective restores
4. Consolidate shards
5. Changing addresses of config/shards

Tips and Tricks

- mongodump/mongorestore
 - --oplog [Replay]
 - --objcheck/--repair
 - --dbpath
 - --query/--filter
- bsondump
 - inspect data at console
- LVM snapshot time/space tradeoff
 - Multi-EBS (RAID) backup
 - clean up snapshots

15 MongoDB Cloud & Ops Manager Fundamentals

MongoDB Cloud & Ops Manager (page 234) Learn about what Cloud & Ops Manager offers

Automation (page 236) Cloud & Ops Manager Automation

Lab: Cluster Automation (page 239) Set up a cluster with Cloud Manager Automation

Monitoring (page 240) Monitor a cluster with Cloud Manager

Lab: Create an Alert (page 242) Create an alert on Cloud Manager

Backups (page 242) Use Cloud Manager to create and administer backups

15.1 MongoDB Cloud & Ops Manager

Learning Objectives

Upon completing this module students should understand:

- Features of Cloud & Ops Manager
- Available deployment options
- The components of Cloud & Ops Manager

Cloud and Ops Manager

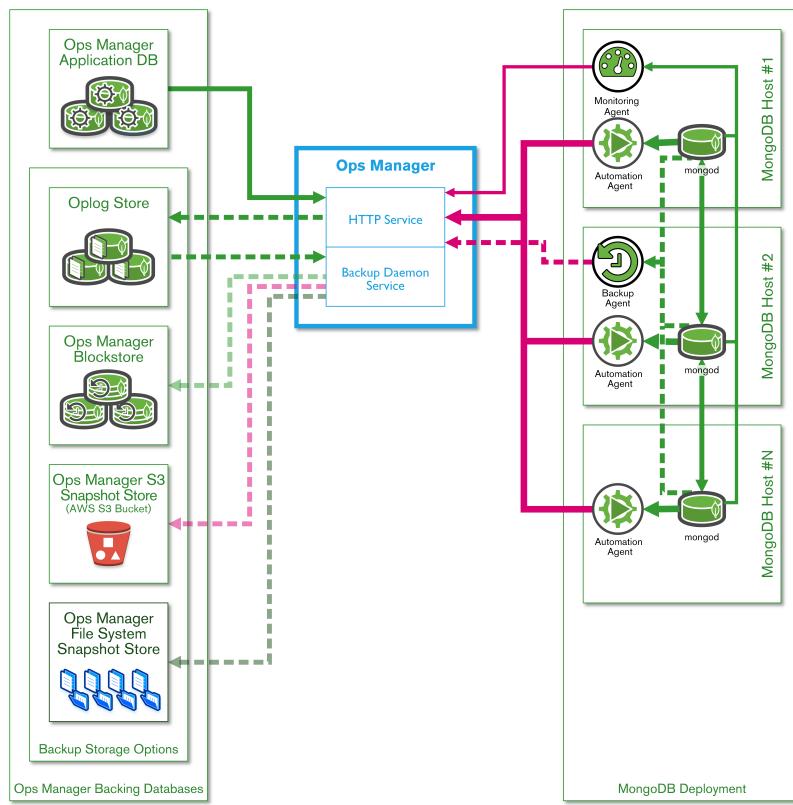
All services for managing a MongoDB cluster or group of clusters:

- Monitoring
- Automation
- Backups

Deployment Options

- Cloud Manager: Hosted, <https://www.mongodb.com/cloud>
- Ops Manager: On-premises

Architecture



Cloud Manager

- Manage MongoDB instances anywhere with a connection to Cloud Manager
- Option to provision servers via AWS integration

Ops Manager

On-premises, with additional features for:

- Alerting (SNMP)
- Deployment configuration (e.g. backup redundancy across internal data centers)
- Global control of multiple MongoDB clusters

Cloud & Ops Manager Use Cases

- Manage a 1000 node cluster (monitoring, backups, automation)
- Manage a personal project (3 node replica set on AWS, using Cloud Manager)
- Manage 40 deployments (with each deployment having different requirements)

Note:

- Use these use cases to get students interested in how Cloud Manager can save them a lot of time
-

Creating a Cloud Manager Account

Free account at <https://www.mongodb.com/cloud>

15.2 Automation

Learning Objectives

Upon completing this module students should understand:

- Use cases for Cloud / Ops Manager Automation
- The Cloud / Ops Manager Automation internal workflow

What is Automation?

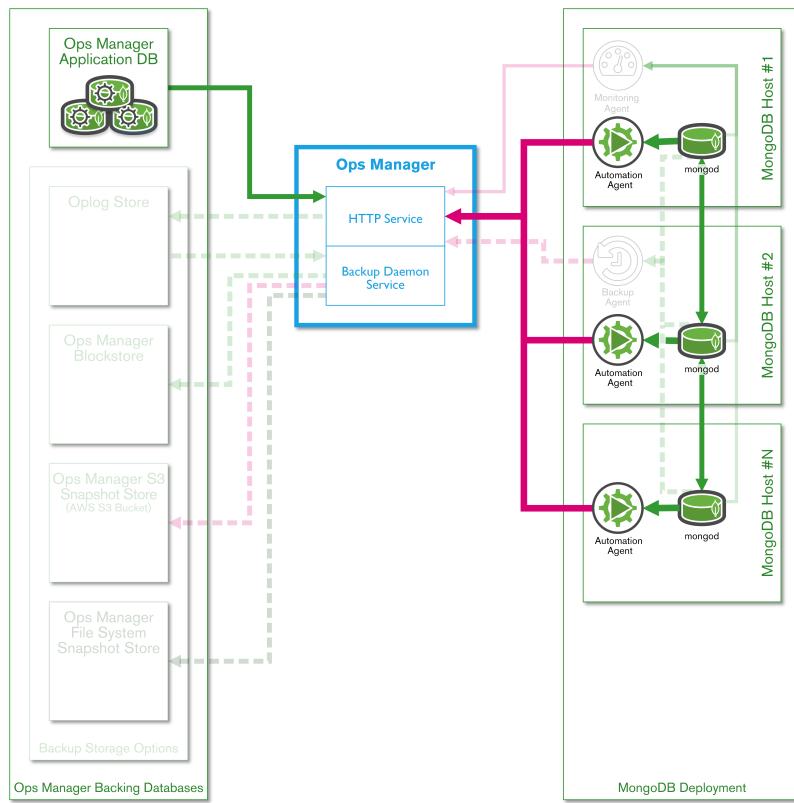
Fully managed MongoDB deployment on your own servers:

- Automated provisioning
- Dynamically add capacity (e.g. add more shards or replica set nodes)
- Upgrades
- Admin tasks (e.g. change the size of the oplog)

How Does Automation Work?

- Automation agent installed on each server in cluster
- Administrator creates design goal for system (through Cloud / Ops Manager interface)
- Automation agents periodically check with Cloud / Ops Manager to get new design instructions
- Agents create and follow a plan for implementing cluster design
- Minutes later, cluster design is complete, cluster is in goal state

Automation Agents



Sample Use Case

Administrator wants to create a 100 shard cluster, with each shard comprised of a 3 node replica set:

- Administrator installs automation agent on 300 servers
- Cluster design is created in Cloud / Ops Manager, then deployed to agents
- Agents execute instructions until 100 shard cluster is complete (usually several minutes)

Upgrades Using Automation

- Upgrades without automation can be a manually intensive process (e.g. 300 servers)
- A lot of edge cases when scripting (e.g. 1 shard has problems, or one replica set is a mixed version)
- One click upgrade with Cloud / Ops Manager Automation for the entire cluster

Automation: Behind the Scenes

- Agents ping Cloud / Ops Manager for new instructions
- Agents compare their local configuration file with the latest version from Cloud / Ops Manager
- Configuration file in JSON
- All communications over SSL

```
{  
  "groupId": "55120365d3e4b0cac8d8a52a737",  
  "state": "PUBLISHED",  
  "version": 4,  
  "cluster": { ...
```

Configuration File

When version number of configuration file on Cloud / Ops Manager is greater than local version, agent begins making a plan to implement changes:

```
"replicaSets": [  
{  
  "_id": "shard_0",  
  "members": [  
    {  
      "_id": 0,  
      "host": "DemoCluster_shard_0_0",  
      "priority": 1,  
      "votes": 1,  
      "slaveDelay": 0,  
      "hidden": false,  
      "arbiterOnly": false  
    },  
    ...
```

Automation Goal State

Automation agent is considered to be in goal state after all cluster changes (related to the individual agent) have been implemented.

Demo

- The instructor will demonstrate using Automation to set up a small cluster locally.
- Reference documentation:
 - [The Automation Agent⁴⁶](#)
 - [The Automation API⁴⁷](#)
 - [Configuring the Automation Agent⁴⁸](#)

Note:

- Go to your Admin page (within Cloud Manager) -> My groups, create a new group, and walk through the process of setting up a small cluster on your laptop
-

15.3 Lab: Cluster Automation

Learning Objectives

Upon completing this exercise students should understand:

- How to deploy, dynamically resize, and upgrade a cluster with Automation

Exercise #1

Create a cluster using Cloud Manager automation with the following topology:

- 3 shards
- Each shard is a 3 node replica set (2 data bearing nodes, 1 arbiter)
- Version 2.6.8 of MongoDB
- **To conserve space, set “smallfiles” = true and “oplogSize” = 10**

Note:

- Windows is not supported, Windows users should work with another person in the class or work on a remote Linux machine
 - The entire cluster should be deployed on a single server (or the students laptop)
 - Registration is free, and won’t require a credit card as long as the student stays below 8 servers
-

⁴⁶ <https://docs.cloud.mongodb.com/tutorial/nav/automation-agent/>

⁴⁷ <https://docs.cloud.mongodb.com/api/>

⁴⁸ <https://docs.cloud.mongodb.com/reference/automation-agent/>

Exercise #2

Modify the cluster topology from Exercise #1 to the following:

- 4 shards (add one shard)
- Version 3.0.1 of MongoDB (upgrade from 2.6.8 -> 3.0.1)

Note:

- Students may complete this in one or two steps
 - Cluster configuration should be modified, then redeployed
-

15.4 Monitoring

Learning Objectives

Upon completing this module students should understand:

- Cloud / Ops Manager monitoring fundamentals
- How to set up alerts in Cloud / Ops Manager

Monitoring in Cloud / Ops Manager

- Identify cluster performance issues
- Identify individual nodes in cluster with performance issues
- Visualize performance through graphs and overlays
- Configure and set alerts

Monitoring Use Cases

- Alert on performance issues, to catch them before they turn into an outage
- Diagnose performance problems
- Historical performance analysis
- Monitor cluster health
- Capacity planning and scaling requirements

Monitoring Agent

- Requests metrics from each host in the cluster
- Sends those metrics to Cloud / Ops Manager server
- Must be able to contact every host in the cluster (agent can live in a private network)
- Must have access to contact Cloud / Ops Manager website with metrics from hosts

Agent Configuration

- Can use HTTP proxy
- Can gather hardware statistics via munin-node
- Agent can optionally gather database statistics, and record slow queries (sampled)

Agent Security

- SSL certificate for SSL clusters
- LDAP/Kerberos supported
- Agent must have “clusterMonitor” role on each host

Monitoring Demo

Visit <https://www.mongodb.com/cloud>

Note:

- The 10gen mongo-perf group may be interesting for demo’ing to the class

Navigating Cloud Manager Charts

- Add charts to view by clicking the name of the chart at the bottom of the host’s page
- “i” icon next to each chart title can be clicked to learn what the chart means
- Holding down the left mouse button and dragging on top of the chart will let you zoom in

Metrics

- Minute-level metrics for 48 hours
- Hourly metrics for about 3 months
- Daily metrics for the life of the cluster

Alerts

- Every chart can be alerted on
- Changes to the state of the cluster can trigger alerts (e.g. a failover)
- Alerts can be sent to email, SMS, HipChat, or PagerDuty

15.5 Lab: Create an Alert

Learning Objectives

Upon completing this exercise students should understand:

- How to create an alert in Cloud Manager

Exercise #1

Create an alert through Cloud Manager for any node within your cluster that is down.

After the alert has been created, stop a node within your cluster to verify the alert.

Note:

- This alert can be created by going to Activity -> Alert Settings -> Create
-

15.6 Backups

Learning Objectives

Upon completing this module students should understand:

- How Cloud / Ops Manager Backups work
- Advantages to Cloud / Ops Manager Backups

Methods for Backing Up MongoDB

- mongodump
- File system backups
- Cloud / Ops Manager Backups

Comparing MongoDB Backup Methods

Considerations	Mongodump	File System	Cloud Backup	Ops Manager
Initial Complexity	Medium	High	Low	High
Replica Set PIT	Yes**	Yes**	Yes	Yes
Sharded Snapshot	No	Yes**	Yes	Yes
Restore Time	Slow	Fast	Medium	Medium

**Requires advanced scripting

Note:

- Instructors should spend a lot of time on this chart, lots of considerations here
-

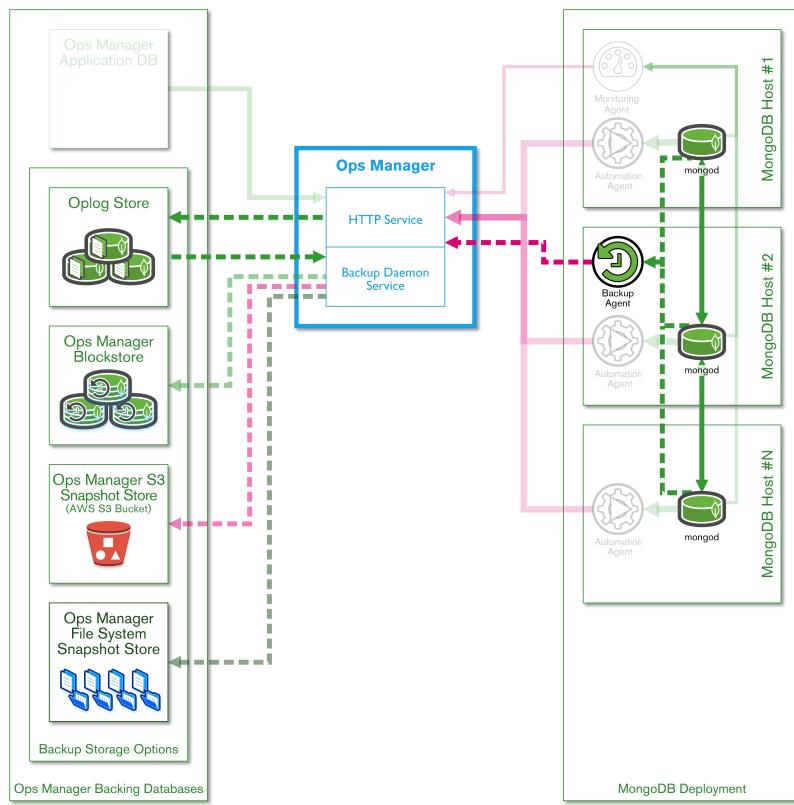
Cloud / Ops Manager Backups

- Based off oplogs (even for the config servers)
- Point-in-time recovery for replica sets, snapshots for sharded clusters
- Oplog on config server for sharded cluster backup
- Ability to exclude collections, databases (such as logs)
- Retention rules can be defined

Restoring from Cloud / Ops Manager

- Specify which backup to restore
- SCP push or HTTPS pull (one time use link) for data files

Architecture



Note:

- Can talk about the daemon process, HEAD database, and backup database with this diagram

Snapshotting

- Local copy of every replica set stored by Cloud / Ops Manager
- Oolog entries applied on top of local copy
- Local copy is used for snapshotting
- Very little impact to the cluster (equivalent to adding another secondary)

Backup Agent

- Backup agent (can be managed by Automation agent)
- Backup agent sends oplog entries to Cloud / Ops Manager service to be apply on local copy

16 MongoDB Cloud & Ops Manager Under the Hood

API (page 246) Using the Cloud & Ops Manager API

Lab: Cloud Manager API (page 247) Cloud & Ops Manager API exercise

Architecture (Ops Manager) (page 249) Ops Manager

Security (Ops Manager) (page 251) Ops Manager Security

Lab: Install Ops Manager (page 252) Install Ops Manager

16.1 API

Learning Objectives

Upon completing this module students should understand:

- Overview of the Cloud / Ops Manager API
- Sample use cases for the Cloud / Ops Manager API

What is the Cloud / Ops Manager API?

Allows users to programmatically:

- Access monitoring data
- Backup functionality (request backups, change snapshot schedules, etc.)
- Automation cluster configuration (modify, view)

API Documentation

<https://docs.mms.mongodb.com/core/api/> <<https://docs.mms.mongodb.com/core/api/>>

Note:

- Open the link in a browser and walk through some of the API calls, such as monitoring stats for a group
-

Sample API Uses Cases

- Ingest Cloud / Ops Manager monitoring data
- Programmatically restore environments
- Configuration management

Ingest Monitoring Data

The monitoring API can be used to ingest monitoring data into another system, such as Nagios, HP OpenView, or your own internal dashboard.

Note:

- Most large companies use other systems for monitoring, point out how the API can help here
-

Programmatically Restore Environments

Use the backup API to programmatically restore an integration or testing environment based on the last production snapshot.

Note:

- Fairly common practice for DBAs, except a lot of them are doing this manually
-

Configuration Management

Use the automation API to integrate with existing configuration management tools (such as Chef or Puppet) to automate creating and maintaining environments.

16.2 Lab: Cloud Manager API

Learning Objectives

Upon completing this exercise students should understand:

- Have a basic understanding of working with the Cloud Manager API (or Ops Manager if the student chooses)

Using the Cloud Manager API

If Ops Manager is installed, it may be used in place of Cloud Manager for this exercise.

Exercise #1

Navigate the Cloud Manager interface to perform the following:

- Generate an API key
- Add your personal machine to the API whitelist

Note:

- Admin -> API Keys and Whitelists
-

Exercise #2

Modify and run the following curl command to return alerts for your Cloud Manager group:

```
curl -u "username:apiKey" --digest -i  
"https://mms.mongodb.com/api/public/v1.0/groups/<GROUP-ID>/alerts"
```

Note:

- Make sure the group id matches a group within the student's account
 - Ensure the IP address the request was coming from is whitelisted
 - Ensure the username/apiKey is correct
-

Exercise #3

How would you find metrics for a given host within your Cloud Manager account? Create an outline for the API calls needed.

Note:

- First, query the API for a list of hosts: <https://docs.mms.mongodb.com/reference/api/hosts/>
 - Next, query the metrics API call with the host id and name of the metric (or for all metrics): <https://docs.mms.mongodb.com/reference/api/hosts/>
-

16.3 Architecture (Ops Manager)

Learning Objectives

Upon completing this module students should understand:

- Ops Manager overview
- Ops Manager components
- Considerations for sizing an Ops Manager environment

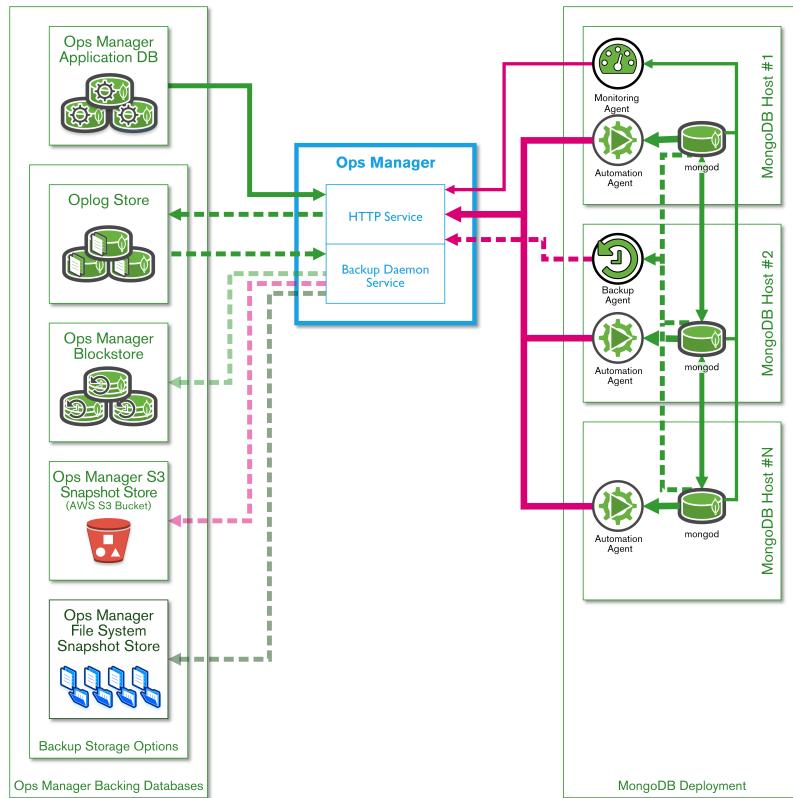
MongoDB Ops Manager

- On-premises version of Cloud Manager
- Everything stays within private network

Components

- Application server(s): web interface
- Ops Manager application database: monitoring metrics, automation configuration, etc.
- Backup infrastructure: cluster backups and restores

Architecture



Note:

- These deployments can get complex, talk about all the pieces
 - Multiple backup daemons/backup DBs for large clusters
-

Application Server

- 15GB RAM, 50GB of disk space are required
- Equivalent to a m3.xlarge AWS instance

Application Database

- All monitoring metrics, automation configurations, etc. stored here
- Replica set, however, a standalone MongoDB node can also be used

Backup Infrastructure

- Backup database (blockstore, oplog, sync)
- Backup daemon process (manages applying oplog entries, creating snapshots, etc.)

Backup Database

- 3 sections: - blockstore for blocks - oplog - sync for initial sync slices
- Replica set, a standalone MongoDB node can also be used
- Must be sized carefully
- All snapshots are stored here
- Block level de-duping, the same block isn't stored twice (significantly reduces database size for deployment with low/moderate writes)

Backup Daemon Process

- The “workhorse” of the backup infrastructure
- Creates a local copy of the database it is backing up (references “HEAD” database)
- Requires 2-3X data space (of the database it is backing up)
- Can run multiple daemons, pointing to multiple backup databases (for large clusters)

16.4 Security (Ops Manager)

Learning Objectives

Upon completing this module students should understand:

- Ops Manager security overview
- Security and authentication options for Ops Manager

Ops Manager User Authentication

- Two-Factor authentication can be enabled (uses Google Authenticator)
- LDAP authentication option

Authentication for the Backing Ops Manager Databases

Ops Manager application database and backup database:

- MongoDB-CR (SCRAM-SHA1)
- LDAP
- Kerberos

Authenticating Between an Ops Manager Agent and Cluster

- LDAP
- MongoDB-CR
- Kerberos (Linux only)

Note:

- This is a good time to draw a diagram on the whiteboard to work through the various authentication options between the Ops Manager components
-

Encrypting Communications

- All communications can be encrypted over SSL.

Ops Manager Groups

- Users can belong to many different groups
- Users have different levels of access per group

User Roles By Group

- Read Only
- User Admin
- Monitoring Admin
- Backup Admin
- Automation Admin
- Owner

Global User Roles

- Global Read Only
- Global User Admin
- Global Monitoring Admin
- Global Backup Admin
- Global Automation Admin
- Global Owner

16.5 Lab: Install Ops Manager

Learning Objectives

Upon completing this exercise students should understand:

- The components needed for Ops Manager
- How to successfully install Ops Manager

Install Ops Manager

A Linux machine with at least 15GB of RAM is required

Install Ops Manager

We will follow an outline of the installation instructions here:

<https://docs.opsmanager.mongodb.com/current/tutorial/install-basic-deployment/>

Exercise #1

Prepare your environment for running all Ops Manager components: Monitoring, Automation, and Backups

- Set up a 3 node replica set for the Ops Manager application database (2 data bearing nodes, 1 arbiter)
- Set up a 3 node replica set for Ops Manager backups (2 data bearing nodes, 1 arbiter)
- Verify both replica sets have been installed and configured correctly

Note:

- This is sometimes the hardest part of setting up Ops Manager for users, ensuring everything has been set up correctly behind the scenes
-

Exercise #2

Install the Ops Manager application

- Ops Manager application requires a license for commercial use
- Download the Ops manager application (after completing form): <http://www.mongodb.com/download>
- Installation instructions (from above): [docs.opsmanager.mongodb.com](https://docs.opsmanager.mongodb.com/current/tutorial/install-basic-deployment/)
- Verify Ops Manager is running successfully

Note:

- The Ops Manager application can be installed via an RPM, once the configuration file is updated, the application can be started (that is pretty much it to installing Ops Manager monitoring/automation)
-

Exercise #3

Install the Ops Manager Backup Daemon

- The Ops Manager backup daemon is required for using Ops Manager for backups
- Download and install the backup daemon (using the link from the past exercise)
- Verify the installation was successful by looking at the logs in: <install_dir>/logs

Note:

- Users sometimes get caught on permissions issues, especially for the HEAD directory
-

Exercise #4

Verify the Ops Manager installation was successful:

<https://docs.opsmanager.mongodb.com/current/tutorial/test-new-deployment/>

Exercise #5

Use Ops Manager to backup a test cluster:

- Create a 1 node replica set via Ops Manager automation
- Add sample data to the replica set:

```
> for (var i=0; i<10000; i++) { db.blog.insert( { "name" : i } ) }
WriteResult({ "nInserted" : 1 })
> db.blog.count()
10000
```

- Use Ops Manager to backup the test cluster
- Perform a restore via Ops Manager of the test cluster