MongoDB Administrator Training

MongoDB Administrator Training

Release 3.0

MongoDB, Inc.

June 11, 2015

Contents

1 Introduction

1.1 Warm Up

Introductions

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

Notes:

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?

Notes:

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?

Notes:

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!"

Notes:

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.

Notes:

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

Notes:

MongoDB is a Document Database

Documents are associative arrays like:

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

Notes:

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?" }
  ]
}
```

Notes:

Vertical Scaling
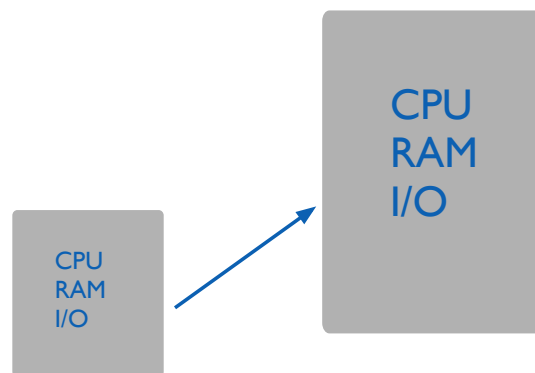
Notes:

Scaling with MongoDB

Notes:
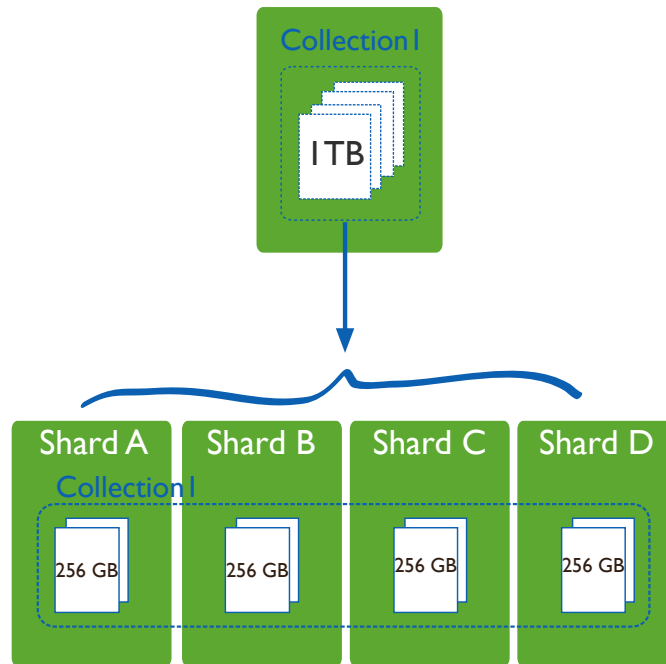
Database Landscape
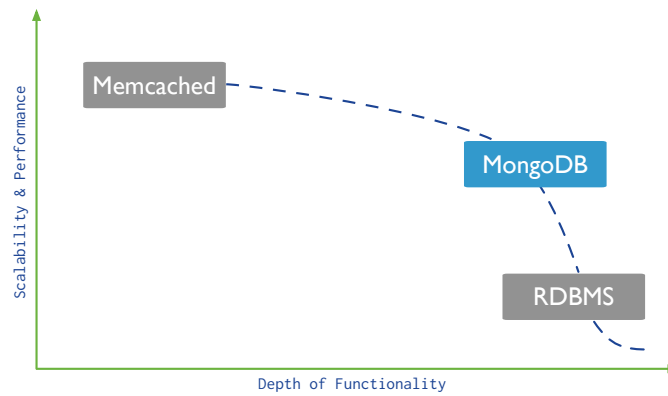
Notes:

MongoDB Deployment Models
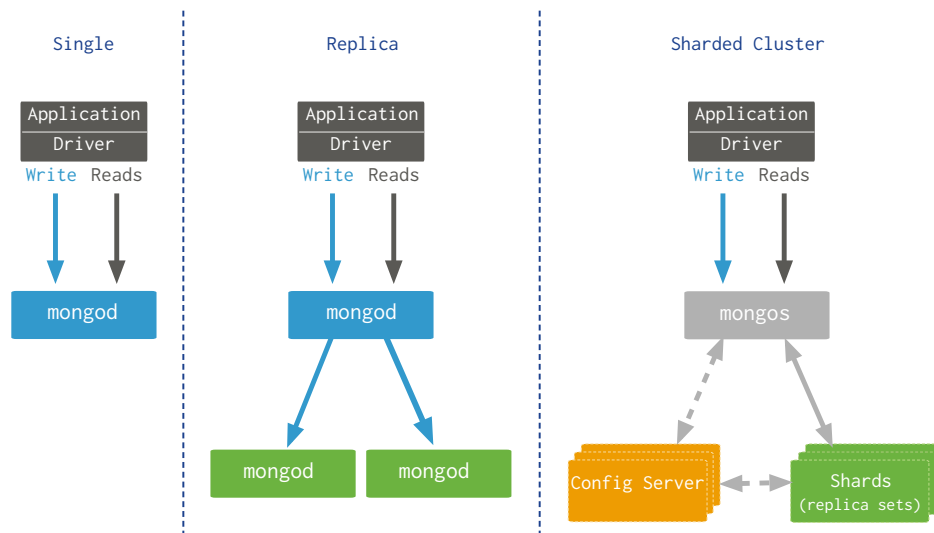
Notes:

1.3 MongoDB Stores Documents

Learning Objectives

Upon completing this module, students should understand:

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

Notes:

JSON

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

Notes:

A Simple JSON Object

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

Notes:

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.org[1].

Notes:

---

[1] http://json.org/

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" ]
  ]
}
```

Notes:

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.org[2].

Notes:

---

[2]http://bsonspec.org/#/specification

BSON Hello World

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

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

Notes:

A More Complex BSON Example

```
// JSON
{ "BSON" : [ "awesome", 5.05, 1986 ] }

// BSON
"\x3b\x00\x00\x00\x04BSON\x00\x26\x00
 \x00\x00\x020\x00\x08\x00\x00
 \x00awesome\x00\x011\x00\x33\x33\x33\x33\x33\x33
 \x14\x40\x102\x00\xc2\x07\x00\x00
 \x00\x00"
```

Notes:

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

Notes:

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).

Notes:

ObjectIds

```
                      Date      ┊  MAC address  ┊   PID   ┊    Counter
         ObjectId: ___  ___  ___  ___┊___  ___  ___┊___  ___┊___  ___  ___

                           12 byte Hex String
```

Notes:

1.4 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

Notes:

What is a Database Storage Engine?

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.

```
┌──────────────┐ ┌──────────────┐ ┌──────────────┐ ┌──────────────┐ ┌──────────────┐
│ Content Repo │ │  IoT Sensor  │ │  Ad Service  │ │   Customer   │ │   Archive    │
│              │ │   Backend    │ │              │ │  Analytics   │ │              │
└──────────────┘ └──────────────┘ └──────────────┘ └──────────────┘ └──────────────┘

┌──────────┐  ┌─────────────────────────────────────────────────────────────┐  ┌────────────┐
│          │  │        MongoDB Query Language + Native Drivers               │  │            │
│          │  └─────────────────────────────────────────────────────────────┘  │            │
│ Security │  ┌─────────────────────────────────────────────────────────────┐  │ Management │
│          │  │            MongoDB Document Data Model                        │  │            │
│          │  └─────────────────────────────────────────────────────────────┘  │            │
│          │   MMAPv1      WiredTiger    In-Memory       ?            ?         │            │
└──────────┘                                                                    └────────────┘
```

Supported in MongoDB 3.0   Experimental   Future possible storage engines
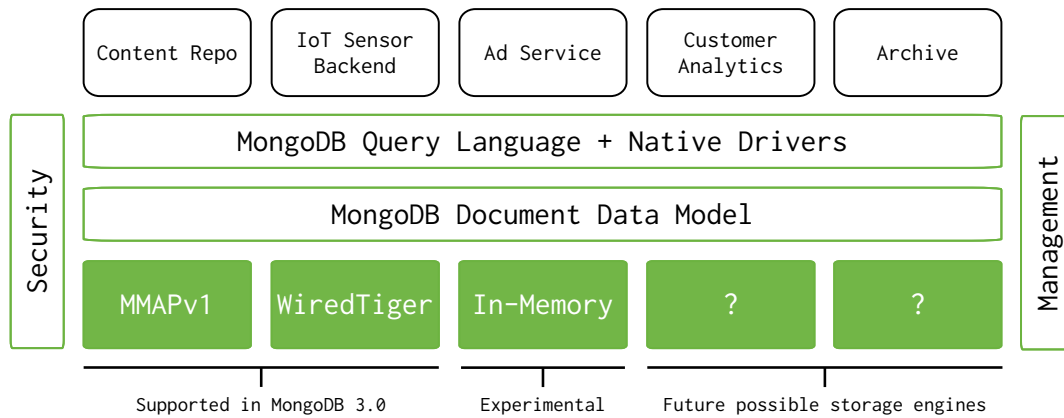
Notes:

How Storage Engines Affect Performance

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

Notes:

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

Notes:

MongoDB Storage Engines

With the release of MongoDB 3.0, two storage engine options are available:

- MMAPv1 (default)

- WiredTiger

Notes:

Specifying a MongoDB Storage Engine

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

```
mongod --storageEngine wiredTiger
```

Notes:

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 options[3].

---

[3]http://docs.mongodb.org/manual/reference/program/mongod/#storage-options

MMAPv1 Storage Engine

- MMAPv1 is MongoDB's original storage engine and currently the default.

```
mongod
```

- This is equivalent to the following command:

```
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

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

Notes:

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.

Notes:

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
- Specify the use of the WiredTiger storage engine as follows.

  ```
  mongod --storageEngine wiredTiger
  ```

Notes:

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

Notes:

Configuring Compression in WiredTiger

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

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

Notes:

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

Notes:

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.

MMMAPv1 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

Notes:

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

Notes:

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

Notes:

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

Notes:

1.5 Exercise: Installing 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

Notes:

Production Releases

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

- Windows
- OSX
- Linux
- Solaris

Notes:

Installing MongoDB

- Visit http://docs.mongodb.org/manual/installation/. Click on the appropriate link, such as "Install on Windows" or "Install on OS X" and follow the instructions there.
- 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.

Notes:

Linux Setup

```
PATH=$PATH:<path to mongodb>/bin

sudo mkdir -p /data/db

sudo chmod -R 777 /data/db
```

Notes:

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"

Notes:

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
  ```

Notes:

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
```

Alternatively, launch with the WiredTiger storage engine.

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

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

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

Notes:

The MongoDB Data Directory (MMAPv1)

`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.

Notes:

The MongoDB Data Directory (WiredTiger)

`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

Notes:

Import Exercise Data

```
cd usb_drive

unzip sampledata.zip

cd sampledata

mongoimport -d sample -c tweets twitter.json

mongoimport -d sample -c zips zips.json

cd dump

mongorestore -d sample training
```

```
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.

Notes:

Launch a Mongo Shell

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

```
mongo
```

Display available commands.

```
help
```

Notes:

Explore Databases

Display available databases.

```
show dbs
```

To use a particular database we can type the following.

```
use <database_name>
```

```
db
```

Notes:

Exploring Collections

```
show collections

db.<COLLECTION>.help()

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

Notes:

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.

Notes:

2 CRUD

Creating and Deleting Documents **(page 36)** Inserting documents into collections, deleting documents, and dropping collections.

Reading Documents **(page 43)** The find() command, query documents, dot notation, and cursors.

Query Operators **(page 52)** MongoDB query operators including: comparison, logical, element, and array operators.

Lab: Finding Documents **(page 57)** Exercises for querying documents in MongoDB.

Updating Documents **(page 58)** Using update() and associated operators to mutate existing documents.

Lab: Updating Documents **(page 67)** 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 remove documents from a collection
- How to remove a collection from a database
- How to remove a database from a MongoDB deployment

Notes:

Creating New Documents

- Create documents using insert().
- For example:

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

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

Notes:

Example: Inserting a Document

Experiment with the following commands.

```
use sample

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

db.movies.find()
```

Notes:

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.

Notes:

Example: Assigning _ids

Experiment with the following commands.

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

db.movies.find()
```

Notes:

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.

Notes:

Example: Inserts will fail if...

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

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

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

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

Notes:

Bulk Inserts

- MongoDB 2.6 introduced bulk inserts.

- You may bulk insert using an array of documents.

- The API has two core concepts:
    - Ordered bulk operations
    - Unordered bulk operations

- The main difference is in the way the operations are executed in bulk.

Notes:

Ordered Bulk Insert

- 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>.insert` is an ordered insert.

- See the next exercise for an example.

Notes:

Example: Ordered Bulk Insert

Experiment with the following bulk insert.

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

db.movies.find()
```

Notes:

Unordered Bulk Insert

- Pass `{ ordered :  false }` to insert to perform unordered inserts.
- If any given insert fails, MongoDB will still attempt the others.
- The inserts may be executed in a different order from the way in which you specified them.
- 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.

Notes:

Example: Unordered Bulk Insert

Experiment with the following bulk insert.

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

Notes:

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

Notes:

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()
```

Notes:

Deleting Documents

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

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

Notes:

Using `remove()`

- Remove documents from a collection using `remove()`.
- This command has one required parameter, a query document.
- All documents in the collection matching the query document will be removed.
- Pass an empty document to remove all documents.
- Prior to MongoDB 2.6 calling `remove()` with no parameters would remove all documents.
- Limit `remove()` to one document using `justOne`.

Notes:

Example: Removing Documents

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

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

db.testcol.remove( { a : 1 } )  // Remove 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.remove( { a : { $lt : 5 } } )  // Remove three more

db.testcol.remove( { a : { $lt : 10 } },
                   { justOne : true } )  // Remove one more

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

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

Notes:

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.

- More on meta data later.

Notes:

Example: Dropping a Collection

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

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

Notes:

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.

Notes:

Example: Dropping a Database

```
use tempDB
db.testcol1.insert( { a : 1 } )
db.testcol2.insert( { 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
```

Notes:

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()`

Notes:

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.

Notes:

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.

Notes:

Example: Querying by Example

Experiment with the following sequence of commands.

```
db.movies.drop()
db.movies.insert([ { "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" } )
```

Notes:

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.

Notes:

Example: Querying Arrays

```
db.movies.drop()
db.movies.insert(
 [{ "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" ] } )
```

Notes:

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.

Notes:

Example: Querying with Dot Notation

```
db.movies.insert(
    [ {
          "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
```

Notes:

Example: Arrays and Dot Notation

```
db.movies.insert( [
    { "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" }
          ] },
    { type : "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
```

Notes:

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()`.

Notes:

Projection: Example (Setup)

```
db.movies.insert(
{
   "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 Anegeles", "state" : "CA", "country" : "USA" }
   ],
   "box_office" : {
     "gross" : 557,
     "opening_weekend" : 24,
     "budget" : 55
   }
 })
```

Notes:

Projection: Example

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

Notes:

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.

Notes:

Example: Projections

```
for (i=1; i<=20; i++) {
    db.movies.insert( { "_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 } )
```

Notes:

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.

Notes:

Example: Introducing Cursors

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

it
it
```

Notes:

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
```

Notes:

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.

Notes:

Example: Using `count()`

```
db.testcol.drop()
for (i=1; i<=100; i++) { db.testcol.insert( { 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(  )
```

Notes:

Example: Using `sort()`

```
db.testcol.drop()
for (i=1; i<=20; i++) {
    db.testcol.insert( { 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 } )
```

Notes:

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.

Notes:

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.

Notes:

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)

Notes:

Example: Using `distinct()`

```
db.movie_reviews.drop()
db.movie_reviews.insert( [ { "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" )

.. only:: instructor

   .. note::

      Returns

      .. code-block:: javascript

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

Notes:

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

Notes:

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

Notes:

Example: Comparison Operators (Setup)

```
// insert sample data
db.movies.insert( [
  {
    "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
  }
] )
```

Notes:

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" ] } } )
```

Notes:

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.

Notes:

Example: Logical Operators

```
db.movies.find( { $or : [
   { "category" : "sci-fi" }, { "imdb_rating" : { $gte : 7 } }
 ] } )
// more complex $or, really good sci-fi movie or medicore 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 } } } )
```

Notes:

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 } }
   ] }
 ] } )
```

Notes:

Element Query Operators

- `$exists`: Select documents based on the existence of a particular field.

- `$type`: Select documents based on their type.

- See BSON types[4] for reference on types.

Notes:

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 } } )
```

Notes:

---

[4]http://docs.mongodb.org/manual/reference/bson-types

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

Notes:

Example: Array Operators

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

db.movies.find( { "category" : { $size : 3 } } )
```

Notes:

Example: $elemMatch

```
db.movies.insert( {
    "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 : {
        "city" : "Florence",
        "country" : "Italy"
      } } } )
```

Notes:

2.4 Lab: Finding Documents

Exercise: Scores < 65

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

Notes:

Exercise: Exams and Quizzes

In the sample database, how many documents in the scores collection have the type "exam" or "quiz"?

Notes:

Exercise: Highest Quiz Score

Find the highest quiz score.

Notes:

Exercise: View Count > 1000

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

Notes:

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.

Notes:

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.)

Notes:

2.5 Updating Documents

Learning Objectives

Upon completing this module students should understand

- The `update()` method
- The required parameters for `update()`
- Field update operators
- Array update operators
- The concept of an upsert and use cases.
- The findAndModify method

Notes:

The `update()` Method

- Mutate documents in MongoDB using `update()`.
- `update()` requires two parameters:
    - A query document used to select documents to be updated
    - An update document that specifies how selected documents will change
- `update()` cannot delete a document.

Notes:

Parameters to `update()`

- Keep the following in mind regarding the required parameters for `update()`
- The query parameter:
    - Use the same syntax as with `find()`.
    - By default only the first document found is updated.
- The update parameter:
    - Take care to simply modify documents if that is what you intend.
    - Replacing documents in their entirety is easy to do by mistake.

Notes:

`$set` and `$unset`

- Update one or more fields using the `$set` operator.
- If the field already exists, using `$set` will change its value.
- If the field does not exist, `$set` will create it and set it to the new value.
- Any fields you do not specify will not be modified.
- You can remove a field using `$unset`.

Notes:

Example: `$set` and `$unset` (Setup)

```
db.movies.insert( [
  {
    "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
  }
] )
```

Notes:

Example: `$set` and `$unset`

```
db.movies.update( { "title" : "Batman" }, { $set : { "imdb_rating" : 7.7 } } )

db.movies.update( { "title" : "Godzilla" }, { $set : { "budget" : 1 } } )

db.movies.update( { "title" : "Home Alone" },
                  { $set : { "budget" : 15, "imdb_rating" : 5.5 } } )

// how will this query behave?
db.movies.update( { "title" : "Batman" }, { "imdb_rating" : 7.7 } )

db.movies.update( { "title" : "Home Alone" }, { $unset :  { "budget" : 1 } } )
```

Notes:

Example: Update Array Elements by Index

```
// add a sample document to track mentions per hour
db.movie_mentions.insert( { "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,
                              0, 0 ]
                          } )

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

Notes:

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` (already discussed)

- `$unset` (already discussed)

- `$min`: Update only if value is smaller than specified quantity

- `$max`: Update only if value is larger than specified quantity

- `$currentDate`: Set the value of a field to the current date or timestamp.

Notes:

Example: Update Operators

```
db.movies.update( { "title" : "Batman" }, { $inc : { "imdb_rating" : 2 } } )

db.movies.update( { "title" : "Home Alone" }, { $inc : { "budget" : 5 } } )

db.movies.update( { "title" : "Batman" }, { $mul : { "imdb_rating" : 4 } } )

db.movies.update( { "title" : "Batman" },
                  { $rename : { "budget" : "estimated_budget" } } )

db.movies.update( { "title" : "Home Alone" }, { $min : { "budget" : 5 } } )

db.movies.update( { "title" : "Home Alone" },
                  { $currentDate : { "last_updated" : { $type : "timestamp" } } } )

// increment movie mentions by 10
db.movie_mentions.update( { "title" : "E.T." } ,
                          { $inc :  { "mentions_per_hour.5" : 10 } } )
```

Notes:

`update()` Defaults to one Document

- By default, `update()` modifies the first document found that matches the query.
- The default use case is one where there is only one document that fits the query.
- This is to reduce the chances of unintended collection scans for updates.

Notes:

Updating Multiple Documents

- In order to update multiple documents, we use the third (optional) parameter to `update()`.
- The third parameter is an options document.
- Specify `multi:   true` as one field in this document.
- Bear in mind that without an appropriate index, you may scan every document in the collection.

Notes:

Example: Multi-Update

Use `db.testcol.find()` after each of these updates.

```
// let's start tracking the number of sequals for each movie
db.movies.update( { }, { $set : { "sequels" : 0 } } )

// we need { multi : true } to change all documents
db.movies.update( { }, { $set : { "sequels" : 0 } },
                  { multi : true } )
```

Notes:

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.

Notes:

Example: Array Operators

```
db.movies.update( { "title" : "Batman" },
                  { $push : { "category" : "superhero" } } )
db.movies.update( { "title" : "Batman" },
                  { $pushAll : { "category" : [ "villain", "comicbased" ] } } )

db.movies.update( { "title" : "Batman" },
                  { $pop : { "category" : 1 } } )

db.movies.update( { "title" : "Batman" },
                  { $pull : { "category" : "action" } } )
db.movies.update( { "title" : "Batman" },
                  { $pullAll : { "category" : [ "villain", "comicbased" ] } } )

db.movies.update( { "title" : "Batman" }, { $addToSet : { "category" : "action" } } )
db.movies.update( { "title" : "Batman" }, { $addToSet : { "category" : "action" } } )
```

Notes:

The Positional **$** Operator

- $\$$[5] 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>.update(
    { <array> : value ... },
    { <update operator> : { "<array>.$" : value } }
)
```


Notes:

Example: The Positional **$** Operator

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


Notes:

Upserts

- By default, if no document matches an update query, the `update()` method does nothing.

- By specifying `upsert:   true`, `update()` will insert a new document if no matching document exists.

- The `db.<COLLECTION>.save()` method is syntactic sugar that performs an upsert if the _id is not yet present

- Syntax:

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

---

[5]http://docs.mongodb.org/manual/reference/operator/update/postional

Upsert Mechanics

- Will update as usual if documents matching the query document exist.
- Will be an upsert if no documents match the query document.
    - MongoDB creates a new document using equality conditions in the query document.
    - Adds an _id if the query did not specify one.
    - Performs an update on the new document.

Example: Upserts

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

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

db.movies.update( { "title" : "Jaws III", "category" : [ "horror" ] },
                  { $set : { "budget" : 1 } },
                  { upsert: true } )
```

save()

- Updates the document if the _id is found, inserts it otherwise
- Syntax:

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

Notes:

Example: `save()`

- If the document does not contain an _id field, then the save() method calls the insert() method. During the operation, the mongo shell will create an ObjectId and assign it to the _id field.
- If the document contains an _id field, then the save() method is equivalent to an update with the upsert option set to true and the query predicate on the _id field.

```
// 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 })
```

Notes:

Be Careful with `save()`

Be careful that you are not modifying stale data when using save(). For example:

```
db.movies.drop()
db.movies.insert( { "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.update( { "title" : "Jaws"  }, { $inc: { "imdb_rating" : 2 } } )
db.movies.find()

doc.imdb_rating = 7.4
doc

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

findAndModify()

Modify a document and return either:

- The pre-modification document

- If "new:true" is set, the modified document

Helpful for making changes to a document and reading the document in the state before or after the update occurred.

Notes:

findAndModify() Example

```
db.worker_queue.findAndModify({
    "query" : { "state" : "unprocessed" },
    "update" : { $set: { "worker_id" : 123, "state" : "processing" } },
})
```

2.6 Lab: Updating Documents

Exercise: Letter Grades

- Using the `sample.scores` namespace, set the proper "grade" attributes.

- For example, users with scores greater than 90 get an "A".

- Set the grade to "B" for scores falling between 80 and 90 and so on for grades "C", "D", and "F".

Notes:

Exercise: 10 Extra-Credit Points

- You're being nice, so you decide to add 10 points to every score on every exam where the score is lower than 60.

- How do you do this update?

Notes:

Exercise: Updating Array Elements

Insert a document representing product metrics for a backpack:

```
db.product_metrics.insert(
   { 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.

Notes:

3 Indexes

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

Notes:

Why Indexes?
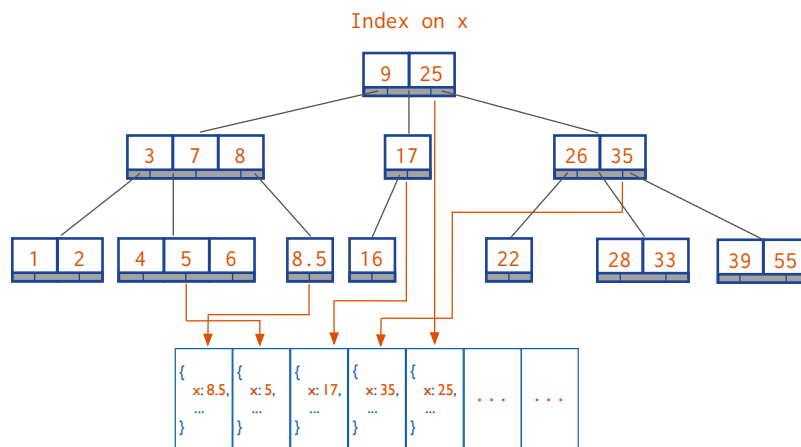
Notes:

Types of Indexes

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

Notes:

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()
```

Notes:

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

Notes:

`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

Notes:

`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
      }
    },
```

Notes:

`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
  }
  ...
}
```

Notes:

`explain("executionStats")` Output

- `nReturned` displays the number of documents that match the query.

- `totalDocsExamined` displays the number of documents the retrieval engine considered during the query.

- `totalKeysExamined` displays how many documents in an existing index were scanned.

- A `totalKeysExamined` or `totalDocsExamined` value much higher than `nReturned` indicates we need a different index.

- Given `totalDocsExamined`, this query will benefit from an index.

Notes:

Other Operations

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

- aggregate
- count
- group
- remove
- update

Notes:

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()
```

Notes:

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 } } )
```

Notes:

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.

Notes:

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.

Notes:

Listing Indexes

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

List index keys:

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

Listing Indexes

List indexes for a collection:

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

List index keys:

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

Notes:

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
    - Is deleted
    - Is updated in such a way that its indexed field changes
    - If an update causes a document to move on disk

Notes:

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.

Notes:

Use Indexes with Care

- Every query should use an index.

- Every index should be used by a query.

- Any write operation that touches an indexed field will require each index to be updated.

- Indexes require RAM.

- Be judicious about the choice of key.

Notes:

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 } )
```

Defining Unique Indexes

- Enforce a unique constraint on the index.
- Prevent duplicate values from being inserted into the database.
- 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 } )
```

Additional Index Options

- Sparse
- Unique
- Background

Notes:

Sparse Indexes in MongoDB

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

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

Notes:

Defining Unique Indexes

- Enforce a unique constraint on the index.

- Prevent duplicate values from being inserted into the database.

- No duplicate values may exist prior to defining the index.

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

Notes:

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 } )
  ```

Notes:

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.

Notes:

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.

Notes:

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 } )
  ```

Notes:

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.

Notes:

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.

Notes:

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.insert(a)
```

Notes:

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()
```

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.

Notes:

Query Adding `username`

Let's add the `user` field to our query.

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

`totalKeysExamined` > n.

Notes:

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()
```

`totalKeysExamined` is still > n. Why?

Notes:

`totalKeysExamined > n`

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

Notes:

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()
```

`totalKeysExamined` is 2. n is 2.

Notes:

`totalKeysExamined == n`

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

Notes:

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.

Notes:

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();
```

- Note that the `winningPlan` includes a `SORT` stage.

- This means that MongoDB had to perform a sort in memory.

- In memory sorts for queries that retrieve large numbers of documents can degrade performance significantly.

- This is especially true if they are used frequently.

Notes:

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();
```

- 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();
```

- We no longer have an in-memory sort, but need to examine more keys.

- `totalKeysExamined` is 3 and 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.

Notes:

General Rules of Thumb

- Equality before range

- Equality before sorting

- Sorting before range

Notes:

Covered Queries
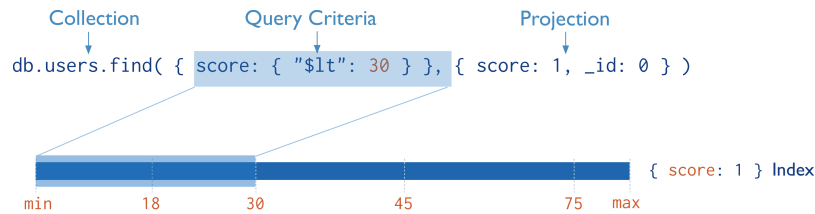
- When a query and projection include only the indexed fields, MongoDB will return results directly from the index.

- There is no need to scan any documents or bring documents into memory.

- These covered queries can be very efficient.

Notes:

Exercise: Covered Queries

```
db.testcol.drop()
for (i=1; i<=20; i++) {
  db.testcol.insert({ "_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, "budget" : 1 }
                 ).explain("executionStats")
```

Notes:

3.3 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

Notes:

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.

Notes:

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.insert( 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()
```

Notes:

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.insert(b)

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

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


Notes:

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 } )
```


Notes:

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.insert(c)
db.player.find()
```

Notes:

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 ] } )
```

Notes:

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.

Notes:

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.

Notes:

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 } )
```

Notes:

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.

Notes:

Example: Multikey Indexes on Multiple Fields

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

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

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

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

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

Notes:

3.4 Hashed Indexes

Learning Objectives

Upon completing this module, students should understand:

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

Notes:

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.

Notes:

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 Key[6] for more details.

- We discuss sharding in detail in another module.

Notes:

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.

Notes:

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}.

Notes:

---

[6]http://docs.mongodb.org/manual/tutorial/shard-collection-with-a-hashed-shard-key/

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" } )
```

Notes:

3.5 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

Notes:

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

Notes:

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.

Notes:

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.

Notes:

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.

Notes:

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

Notes:

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.

Notes:

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.

Notes:

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>`

Notes:

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`.

Notes:

Inserting Documents with a 2d Index

There are two accepted formats:

- Legacy coordinate pairs

- Document with the following fields specified:

  - `lng` (longitude)

  - `lat` (latitude)

Notes:

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.

Notes:

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)

Notes:

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 ] } } } )
```

Notes:

Creating a 2dsphere Index

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

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

Notes:

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.

Notes:

geoJSON Considerations

- The coordinates of points are given in degrees (latitude then longitude).
- 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.

Notes:

Simple Types of 2dsphere Objects

Point: A single point on the globe

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

Notes: **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> ] ] } }
```

Notes:

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> ] ]
              } }
```

Notes:

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

Notes:

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

Notes:

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".

Notes:

Exercise: Inserting geoJSON Objects (3)

- Create documents for every possible New York to London flight.

- Include a `flightNumber` field for each flight.

Notes:

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.

Notes:

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.

Notes:

3.6 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

Notes:

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.

Notes:

Creating a TTL Index

Create with:

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

Notes:

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.insert( { "last_user_action" : ISODate(), "b" : i } );
    sleep(1000);  // Sleep for 1 second
}
```

Notes:

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);
}
```

Notes:

3.7 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.

Notes:

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.

Notes:

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?" } ] )
```

Notes:

Querying a Text Index

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

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

Notes:

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" } } )
```

Notes:

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" } } )
```

Notes:

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\"" } } )
```

Notes:

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" } }
) )
```

Notes:

3.8 Lab: Building and Examining Indexes

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)
```

Notes:

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 } )
```

Notes:

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 })
```

Notes:

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.

Notes:

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")
```

Notes:

4 Replica Sets

Introduction to Replica Sets **(page 108)** An introduction to replication and replica sets.

Elections in Replica Sets **(page 112)** The process of electing a new primary (automated failover) in replica sets.

Replica Set Roles and Configuration **(page 118)** Configuring replica set members for common use cases.

The Oplog: Statement Based Replication **(page 120)** The process of replicating data from one node of a replica set to another.

Write Concern **(page 126)** Balancing performance and durability of writes.

Read Preference **(page 131)** Configuring clients to read from specific members of a replica set.

Exercise: Setting up a Replica Set **(page 132)** Launching members, configuring, and initiating a replica set.

4.1 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

Notes:

Use Cases for Replication

- High Availability
- Disaster Recovery
- Functional Segregation

Notes:

High Availability (HA)

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

Notes:

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

Notes:

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

Notes:

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)

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.

Notes:

Replica Sets
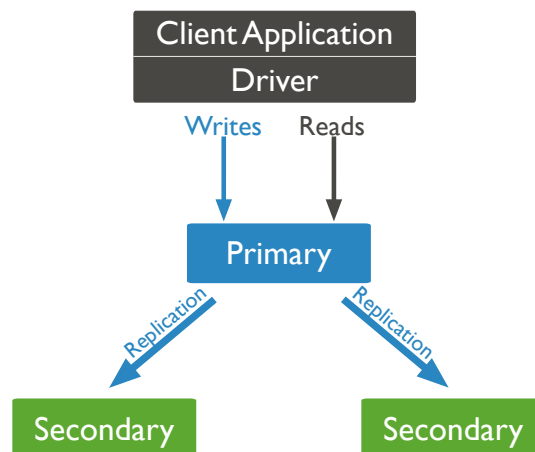
Notes:

Primary Server

- Clients send writes 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.

Notes:

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.

Notes:

Heartbeats
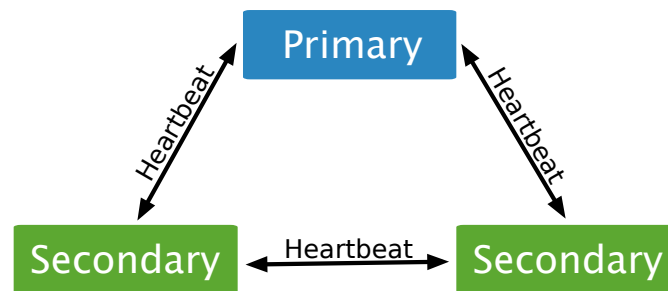
Notes:

The Oplog

- The operations log, or oplog, is a special capped collection that is the basis for replication.
- The oplog maintains one entry for each document affected by every write operation.
- Secondaries copy operations from the oplog of their sync source.

Notes:

4.2 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

Notes:

Members and Votes
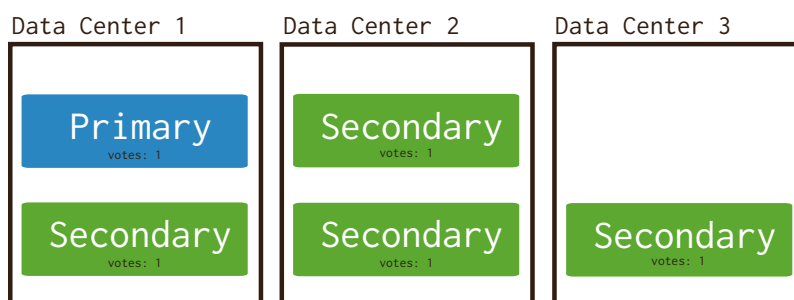
Notes:

Calling Elections

Election for New Primary

New Primary Elected
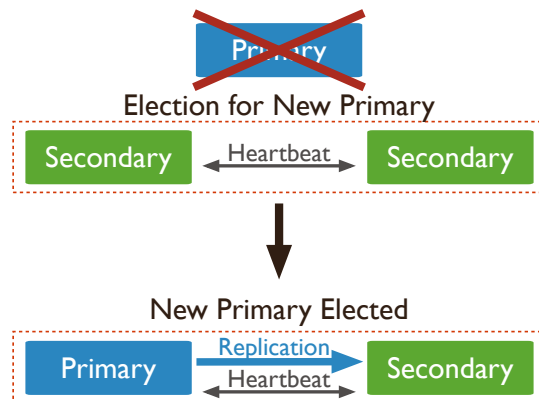
Notes:

Selecting a New Primary

Three factors are important in the selection of a primary:

- Priority
- Optime
- Connections

Notes:

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.

Notes:

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 optimes of visible members are compared.

Notes:

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.

Notes:

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.

Notes:

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

Notes:

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.

Notes:

Scenario A: 3 Data Nodes in 1 DC

Which secondary will become the new primary?
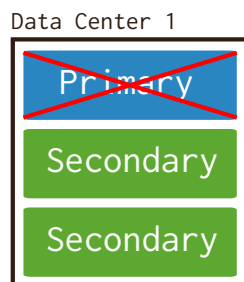
Notes:

Scenario B: 3 Data Nodes in 2 DCs

Which member will become primary following this type of network partition?
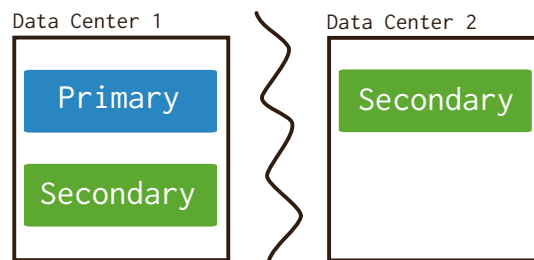
Notes:

Scenario C: 4 Data Nodes in 2 DCs

What happens following this network partition?
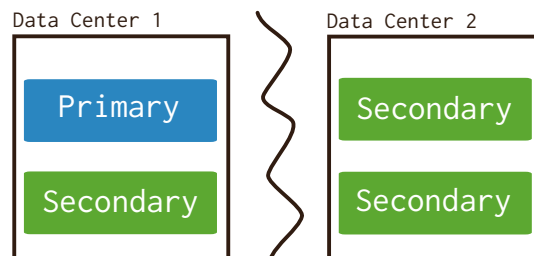
Notes:

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?
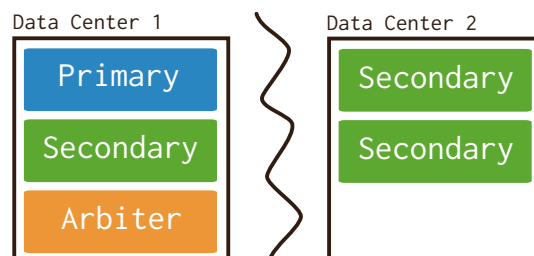
Notes:

Scenario E: 3 Data Nodes in 3 DCs

- What happens here if any one of the nodes/DCs fail?
- What about recovery time?

Data Center 1	Data Center 2	Data Center 3
Primary priority: 1	**Secondary** priority: 1	**Secondary** priority: 1
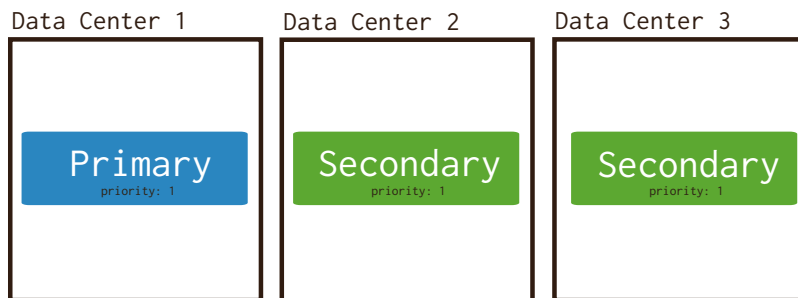
Notes:

Scenario F: 5 Data Nodes in 3 DCs

What happens here if any one of the nodes/DCs fail? What about recovery time?

Data Center 1	Data Center 2	Data Center 3
Primary votes: 1	**Secondary** votes: 1	
Secondary votes: 1	**Secondary** votes: 1	**Secondary** votes: 1
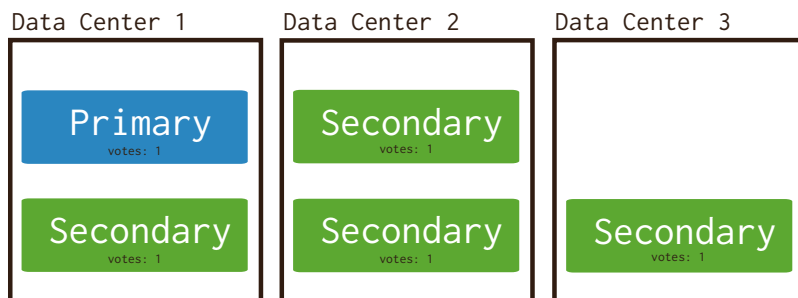
Notes:

4.3 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.

Notes:

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.

Notes:

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 }
  ]
}
```

Notes:

Principal Data Center

```
{ _id : 0, host : "dc1-1.example.net", priority : 5 },
{ _id : 1, host : "dc1-2.example.net", priority : 5 },
```

Notes:

Data Center 2

```
{ _id : 2, host : "dc2-1.example.net:27017" },
```

Notes:

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 }
```

Notes:

What about `dc2-2`?

```
{ _id : 4, host : "dc2-2.example.net:27017", hidden : true,
  slaveDelay : 7200 }
```

Notes:

4.4 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.

Notes:

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.

Notes:

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.

Notes:

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.

Notes:

Example

Suppose the following remove is issued and it deletes 100 documents:

```
db.foo.remove({ 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 } }
```

Notes:

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.

Notes:

Create a Replica Set

Let's take a look at a concrete example. Launch mongo shell as follows.

```
mongo --nodb
```

Create a replica set by running the following command in the mongo shell.

```
replicaSet = new ReplSetTest( { nodes : 3 } )
```

Notes:

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.

Notes:

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()
```

Notes:

Status Check

- You should now have three mongods running on ports 31000, 31001, and 31002.
- You will see log statements from all three printing in the current shell.
- To complete the rest of the exercise, open a new shell.

Notes:

Connect to the Primary

Open a new shell, connecting to the primary.

```
mongo --port 31000
```

Notes:

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)
```

Notes:

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 })
```

Notes:

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 } } }
```

Notes:

Operations in the Oplog 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.

Notes:

The Oplog Window

- Oplogs 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.

Notes:

Sizing the Oplog

- 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.

Notes:

4.5 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.

Notes:

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

Notes:

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`.

Notes:

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.

Notes:

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.

Notes:

Write Concern: `{ w : 1 }`
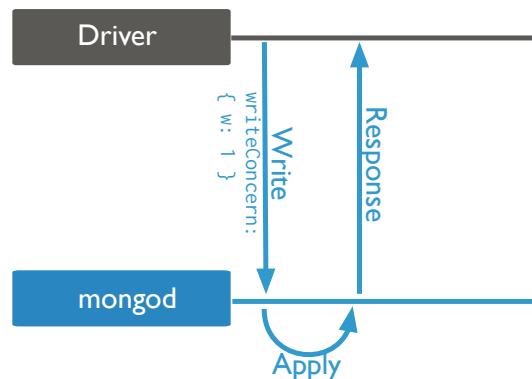
Notes:

Example: `{ w :  1 }`

```
db.edges.insert( { from : "tom185", to : "mary_p" },
                 { writeConcern : { w : 1 } } )
```

Notes:

Write Concern: `{ w :  2 }`
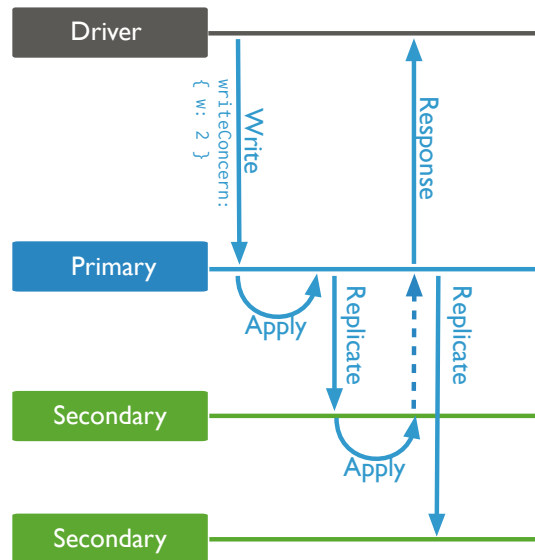
Notes:

Example: `{ w :  2 }`

```
db.customer.update( { user : "mary_p" },
                    { $push : { shoppingCart:
                      { _id : 335443, name : "Brew-a-cup",
                      price : 45.79 } } },
                    { writeConcern : { w : 2 } } )
```

Notes:

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.

Notes:

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.

Notes:

Example: `{ w :  "majority" }`

```
db.products.update({ _id : 335443 },
                   { $inc : { inStock : -1 } },
                   { writeConcern : { w : "majority" }})
```

Notes:

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 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" }

Notes:

Further Reading

See Write Concern Reference[7] for more details on write concern configurations, including setting timeouts and identifying specific replica set members that must acknowledge writes (i.e. tag sets[8]).

---

[7]http://docs.mongodb.org/manual/reference/write-concern
[8]http://docs.mongodb.org/manual/tutorial/configure-replica-set-tag-sets/#replica-set-configuration-tag-sets

4.6 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.

Notes:

Use Cases

- Running systems operations without affecting the front-end application.
- Providing local reads for geographically distributed applications.
- Maintaining availability during a failover.

Notes:

Not for Scaling

- In general, do *not* read from secondaries to provide extra capacity for reads.
- Sharding[9] increases read and write capacity by distributing operations across a group of machines.
- Sharding is a better strategy for adding capacity.

Notes:

---

[9]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.

Notes:

Tag Sets

- There is also the option to used 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)
```

Notes:

4.7 Exercise: 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.

Notes:

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
```


Notes:

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
```


Notes:

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.

Notes:

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
```

Notes:

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()
```

Notes:

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
```

Notes:

Read from a Secondary

Connect to one of the secondaries. E.g.:

```
mongo --port 27018
```

Read from the secondary

```
rs.slaveOk()
db.testcol.find()
```

Notes:

Review the Oplog

```
use local
db.oplog.rs.find()
```

Notes:

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 3rd node (e.g. on port 27019):

```
cfg = rs.conf()
cfg["members"][2]["priority"] = 10
rs.reconfig(cfg)
```

Notes:

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()
```

Notes:

Further Reading

- Replica Configuration[10]
- Replica States[11]

---
[10]http://docs.mongodb.org/manual/reference/replica-configuration/
[11]http://docs.mongodb.org/manual/reference/replica-states/

5 Sharding

5.1 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

Notes:

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.

Notes:

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

Notes:

Vertical Scaling

- Adding more RAM, faster disks, etc.
- When is this the solution?
- First, consider a concept called the `working set`.

Notes:

The Working Set
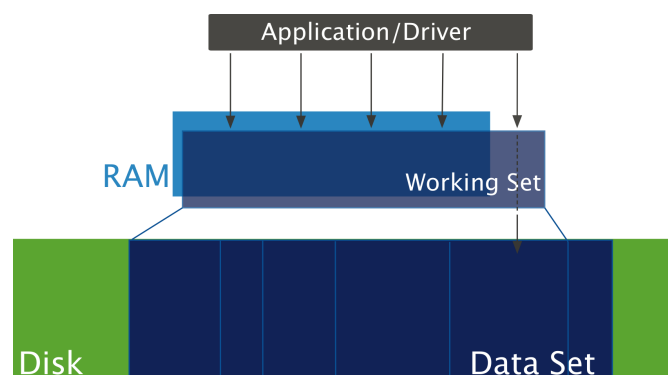
Notes:

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 possible support.
- This is when it is time to scale horizontally.

Notes:

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.

Notes:

A Model that Does Not Scale

Load   Latency
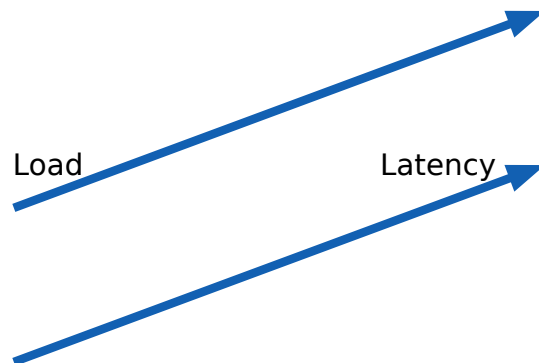
Notes:

A Scalable Model

Load   Latency

Notes:

Sharding Basics

Collection I

I TB

Shard A   Shard B   Shard C   Shard D

Collection I

256 GB   256 GB   256 GB   256 GB
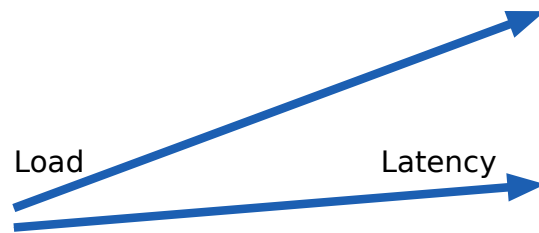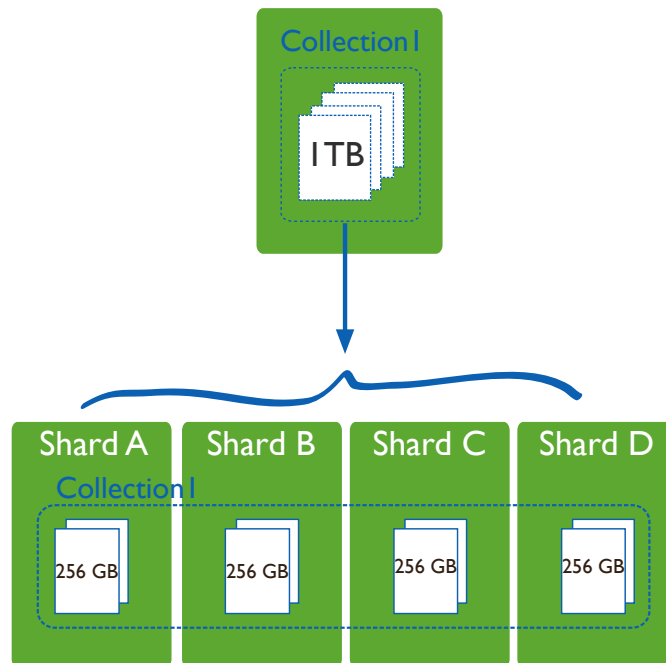
Notes:

Sharded Cluster Architecture
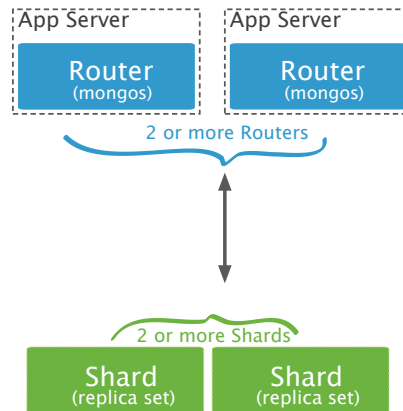
Notes:

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.

Notes:

Config Servers

Notes:
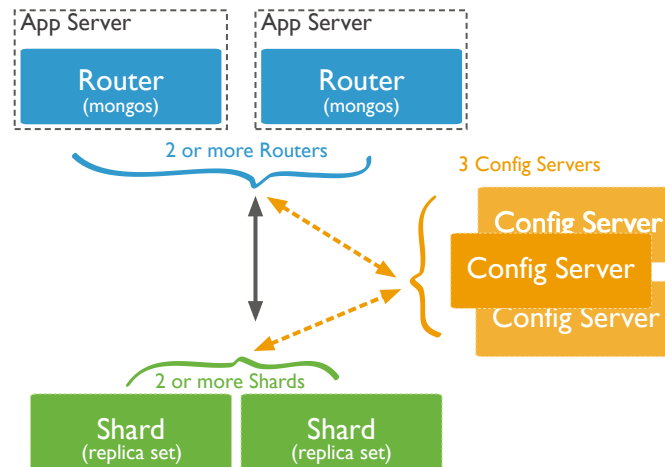
Config Server Hardware Requirements

- 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.

Notes:

When to Shard

- If you have more data than one machine can hold on its drives
- If your application is write heavy and you experiencing too much latency.
- If your working set outgrows the memory you can allocate to a single machine.

Notes:

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.

Notes:

Balancing Shards

- MongoDB divides data into `chunks`.
- This is bookkeeping metadata.
  - There is nothing in a document that indicates its chunk.
  - The document does not need to be updated if its assigned chunk changes.
- 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.

Notes:

What is a Shard Key?

- You must define a shard key for a sharded collection.
- Based on one or more fields that every document must contain.
- Is immutable.
- 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

Notes:

Targeted Query Using Shard Key
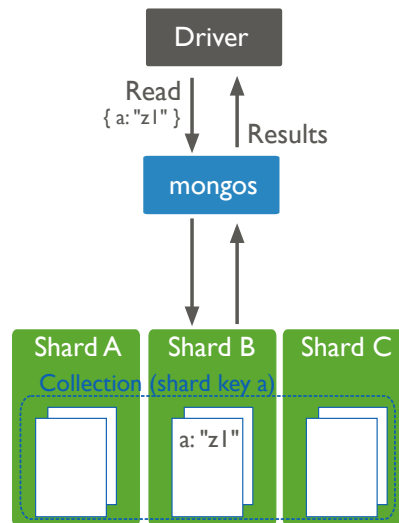
Notes:

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.

Notes:

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.

Notes:

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 or reads are routed to a particular server

Notes:

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.

Notes:

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.

Notes:

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.

Notes:

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.

Notes:

5.2 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

Notes:

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.

Notes:

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

Notes:
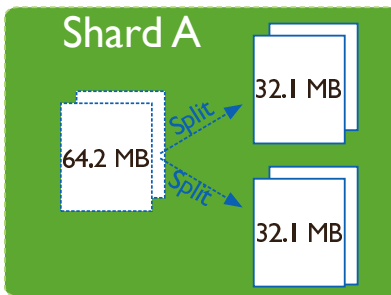
Chunk Splits

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.

Notes:

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 to 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

Notes:

Balancing is Resource Intensive

- Chunk migration requires copying all the data in the chunk from one shard to another.

- MonogDB 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.

Notes:

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.

Notes:

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.

Notes:

5.3 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

Notes:

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.

Notes:

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.

Notes:

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.

Notes:

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 Tags[12]

Notes:

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.

Notes:

---

[12]http://docs.mongodb.org/manual/tutorial/administer-shard-tags/

5.4 Exercise: 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

Notes:

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.

Notes:

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

Notes:

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
```

Notes:

Initiate a Replica Set

```
mongod --replSet shard0 --smallfiles --nojournal --noprealloc \
       --dbpath ~/data/cluster/shard0/m0 \
       --logpath ~/data/cluster/shard0/m0/mongod.log \
       --fork --port 27107

mongod --replSet shard0 --smallfiles --nojournal --noprealloc \
       --dbpath ~/data/cluster/shard0/m1 \
       --logpath ~/data/cluster/shard0/m1/mongod.log \
       --fork --port 27108

mongod --replSet shard0 --smallfiles --nojournal --noprealloc \
       --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')"
```

Notes:

Spin Up a Second Replica Set

```
mongod --replSet shard1 --smallfiles --nojournal --noprealloc \
       --dbpath ~/data/cluster/shard1/m0 \
       --logpath ~/data/cluster/shard1/m0/mongod.log \
       --fork --port 27117

mongod --replSet shard1 --smallfiles --nojournal --noprealloc \
       --dbpath ~/data/cluster/shard1/m1 \
       --logpath ~/data/cluster/shard1/m1/mongod.log \
       --fork --port 27118

mongod --replSet shard1 --smallfiles --nojournal --noprealloc \
       --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')"
```

Notes:

A Third Replica Set

```
mongod --replSet shard2 --smallfiles --nojournal --noprealloc \
       --dbpath ~/data/cluster/shard2/m0 \
       --logpath ~/data/cluster/shard2/m0/mongod.log \
       --fork --port 27127

mongod --replSet shard2 --smallfiles --nojournal --noprealloc \
       --dbpath ~/data/cluster/shard2/m1 \
       --logpath ~/data/cluster/shard2/m1/mongod.log \
       --fork --port 27128

mongod --replSet shard2 --smallfiles --nojournal --noprealloc \
       --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')"
```

Notes:

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

Notes:

Launch Config Servers

```
mongod --smallfiles --nojournal --noprealloc \
      --dbpath ~/data/cluster/config/c0 \
      --logpath ~/data/cluster/config/c0/mongod.log \
      --fork --port 27217 --configsvr

mongod --smallfiles --nojournal --noprealloc \
      --dbpath ~/data/cluster/config/c1 \
      --logpath ~/data/cluster/config/c1/mongod.log \
      --fork --port 27218 --configsvr

mongod --smallfiles --nojournal --noprealloc \
      --dbpath ~/data/cluster/config/c2 \
      --logpath ~/data/cluster/config/c2/mongod.log \
      --fork --port 27219 --configsvr
```

Notes:

Launch the Mongos Processes

Now our mongos's. We need to tell them about our config servers.

```
mongos --logpath ~/data/cluster/s0/mongos.log --fork --port 27017 \
       --configdb localhost:27217,localhost:27218,localhost:27219

mongos --logpath ~/data/cluster/s1/mongos.log --fork --port 27018 \
       --configdb localhost:27217,localhost:27218,localhost:27219
```

Notes:

Add All Shards

```
echo 'sh.addShard( "shard0/localhost:27107" ); \
      sh.addShard("shard1/localhost:27117" ); \
      sh.addShard( "shard2/localhost: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.

Notes:

Enable Sharding and Shard a Collection

Enable sharding for the test database, shard a collection, and insert some documents.

```
echo 'sh.enableSharding("test"); \
      sh.shardCollection("test.testcol", { a : 1, b : 1 } )' | mongo

echo 'for (i=0; i<10000; i++) { docArr = []; for (j=0; j<1000; j++) { \
    docArr.push( { a : i, b : j, c : "Filler String 00000000000000000000 \
    0000000000000000000000000000000000000000000000000000000000000000 \
    000000000000000" } )  }; db.testcol.insert(docArr) }' | mongo
```

Notes:

Observe What Happens

Connect to either mongos using a mongo shell and frequently issue:

```
sh.status()
```

Notes:

6 Security

Security **(page 159)** An overview of security options for MongoDB.

6.1 Security

Learning Objectives

Upon completing this module students should understand:

- Security options for MongoDB
- Basics of native auth for MongoDB
- User roles in MongoDB
- How to manage user roles in MongoDB

Notes:

Overview
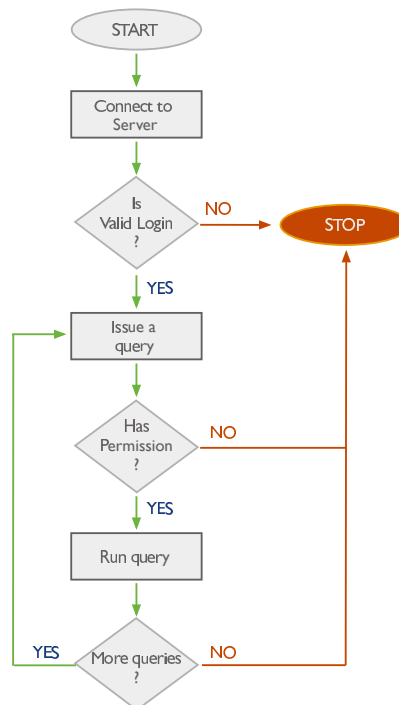
Notes:

Authentication Options

- Chanllenge/response authentication using SCRAM-SHA-1 (username & password)
- x.509 Authentication (using x.509 Certificates)
- Kerberos (through an Enterprise subscription)
- LDAP

Notes:

Authorization via MongoDB

- Each user has a set of potential roles
    - read, readWrite, dbAdmin, etc.
- Each role applies to *one* database
    - A single user can have roles on each database
    - Some roles apply to all databases
    - You can also create custom roles.

Notes:

Network Exposure Options

- bindIp limits the ip addresses the server listens on.
- Using a non-standard port can provide a layer of obscurity.
- MongoDB should still be run only in a trusted environment.

Notes:

Encryption (SSL)

- MongoDB can be configured at build time to run with SSL.
- To get it, build from the source code with –ssl.
- Alternatively, use MongoDB Enterprise.
- Allows you to use public key encryption.
- You can also validate with x.509 certificates.

Notes:

Native MongoDB Auth

- Uses SCRAM-SHA-1 for challenge/response
- Sometimes called MongoDB-CR
- Start a mongod instance with `--auth` to enable this feature
- You can initially login using localhost
    - Called the "localhost exception".
    - Stops working when you create a user.

Notes:

Exercise: Create an Admin User, Part 1

- Launch a mongo shell.
- Create a user with the role, `userAdminAnyDatabase`
- Use name "roland" and password "12345".
- Enable this user to login on the admin database.

Notes:

Exercise: Create an Admin User, Part 2

- Launch a mongo shell without logging in.
- Attempt to create a user.
- Exit the shell.
- Log in again as roland.
- Ensure that you can create a user.

Notes:

Using MongoDB Roles

- Each user logs in on *one* database.
- The user inputs their password on login.
    - Use the -u flag for username.
    - Use the -p flag to enter the password.
- userAdmins may create other users
- But they cannot read/write without other roles.

Notes:

Exercise: Creating a `readWrite` User, Part 1

- Create a user named *vespa*.
- Give *vespa* readWrite access on the *test* and *druidia* databases.
- Create this user so that the login database is *druidia*.

Notes:

Exercise: Creating a `readWrite` User, Part 2

Log in with the user you just created.

Notes:

MongoDB Custom User Roles

- You can create custom user roles in MongoDB.
- You do this by modifying the system.roles collection.
- You can also inherit privileges from other roles into a given role.
- You won't remember how to do this, so if you need it, consult the docs[13].

Notes:

---

[13]http://docs.mongodb.org/manual/core/security-introduction/

7 MMS & Ops Manager

7.1 MongoDB Management Service (MMS) & Ops Manager

Learning Objectives

Upon completing this module students should understand:

- Features of the MongoDB Management Service (MMS) & Ops Manager
- Available deployment options
- The components of MMS & Ops Manager
- MMS demo

Notes:

MMS and Ops Manager

All services for managing a MongoDB cluster or group of clusters:

- Monitoring
- Automation
- Backups

Notes:

Deployment Options

- MMS: Hosted, http://mms.mongodb.com
- Ops Manager: On-premises

Notes:

Architecture

MongoDB Clusters
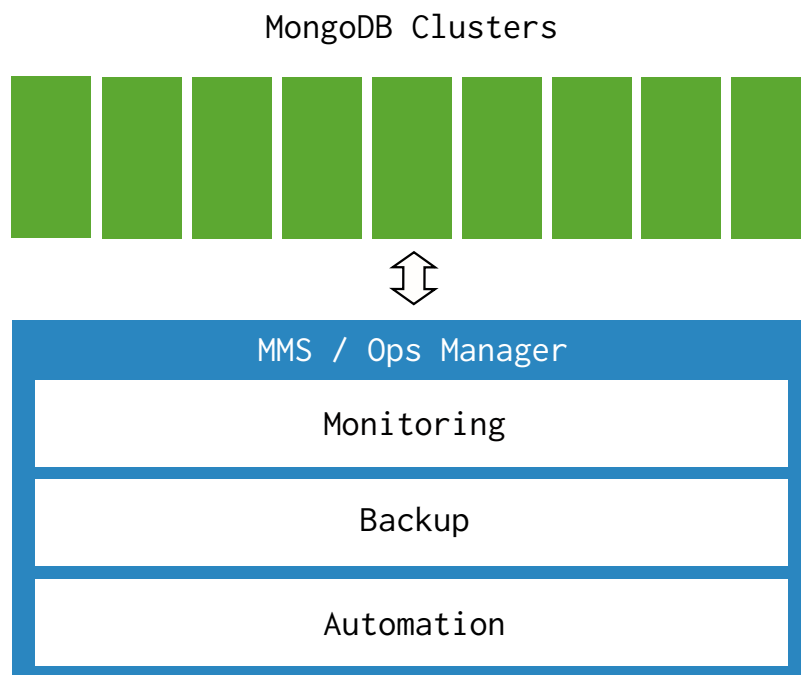
Notes:

MMS

- Manage MongoDB instances anywhere with a connection to MMS
- Option to provision servers via AWS integration

Ops Manager

On-premises MMS, with additional features for:

- Alerting (SNMP)
- Deployment configuration (e.g. backup redundancy across internal data centers)
- Global control of multiple MongoDB clusters

Notes:

MMS and Ops Manager Use Cases

- Manage a 1000 node cluster (monitoring, backups, automation)
- Manage a personal project (3 node replica set on AWS, using MMS)
- Manage 40 deployments (with each deployment having different requirements)

Notes:

Creating an MMS Account

Free account at mms.mongodb.com

Notes:

7.2 Automation

Learning Objectives

Upon completing this module students should understand:

- Use cases for MMS / Ops Manager Automation
- The MMS / Ops Manager Automation internal workflow

Notes:

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)

Notes:

How Does Automation Work?

- Automation agent installed on each server in cluster
- Administrator creates design goal for system (through MMS / Ops Manager interface)
- Automation agents periodically check with MMS / 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

Notes:

Automation Agents

Machines in Data Center

Automation Agent	**Automation Agent**	**Automation Agent**

⇕ ⇕ ⇕

MMS / Ops Manager
Automation
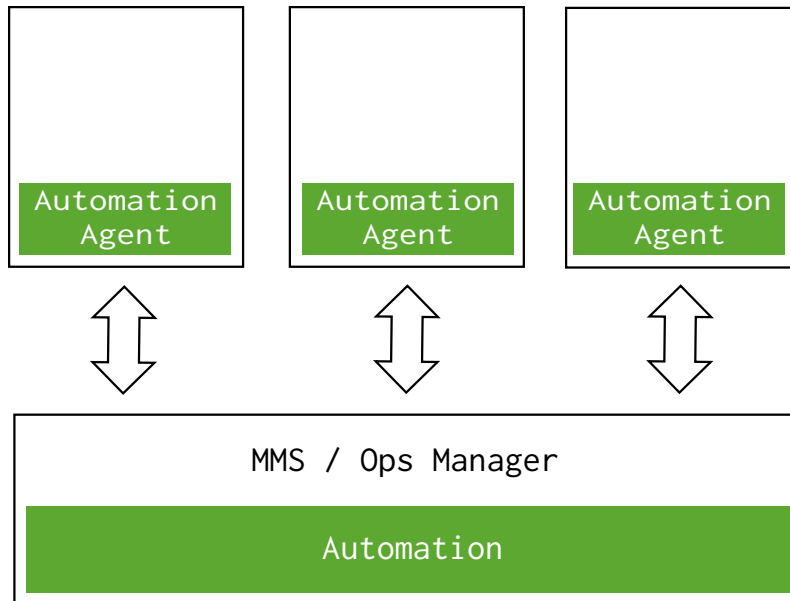
Notes:

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 MMS / Ops Manager, then deployed to agents
- Agents execute instructions until 100 shard cluster is complete (usually several minutes)

Notes:

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 MMS / Ops Manager Automation for the entire cluster

Notes:

Automation: Behind the Scenes

- Agents ping MMS / Ops Manager for new instructions

- Agents compare their local configuration file with the latest version from MMS / Ops Manager

- Configuration file in json

- All communications over SSL

```
{
    "groupId": "55120365d3e4b0cac8d8a52a737",
    "state": "PUBLISHED",
    "version": 4,
    "cluster": { ...
```

Notes:

Configuration File

When version number of configuration file on MMS / 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
        },
    ...
```

Notes:

Automation Goal State

Automation agent is considered to be in goal state after all cluster changes (related to the individual agent) have been implemented.

Notes:

Demo

Notes:

7.3 Exercise: Cluster Automation

Learning Objectives

Upon completing this exercise students should understand:

- How to deploy, dynamically resize, and upgrade a cluster with MMS Automation

Notes:

MMS Automation Support

Windows machines are not supported at this time.

Notes:

Exercise #1

Using your personal computer, create a cluster using MMS 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 on your local machine, set "smallfiles" = true and "oplogSize" = 10**

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)

Notes:

7.4 Monitoring

Learning Objectives

Upon completing this module students should understand:

- MMS / Ops Manager monitoring fundamentals

- How to set up alerts in MMS / Ops Manager

Notes:

Monitoring in MMS / Ops Manager

- Identify cluster performance issues
- Identify individual nodes in cluster with performance issues
- Visualize performance through graphs and overlays
- Configure and set alerts

Notes:

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

Notes:

Monitoring Agent

- Requests metrics from each host in the cluster
- Sends those metrics to MMS / 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 MMS / Ops Manager website with metrics from hosts

Notes:

Agent Configuration

- Can use HTTP proxy

- Can gather hardware statistics via munin-node

- Agent can optionally gather database statistics, and record slow queries (sampled)

Notes:

Agent Security

- SSL certificate for SSL clusters

- LDAP/Kerberos supported

- Agent must have "clusterMonitor" role on each host

Notes:

Monitoring Demo

Visit mms.mongodb.com

Notes:

Navigating MMS 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

Notes:

Metrics

- Minute-level metrics for 48 hours
- Hourly metrics for about 3 months
- Daily metrics for the life of the cluster

Notes:

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

Notes:

7.5 Exercise: Create an Alert

Learning Objectives

Upon completing this exercise students should understand:

- How to create an alert in MMS

Notes:

Exercise #1

Create an alert through MMS 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.

7.6 Backups

Learning Objectives

Upon completing this module students should understand:

- How MMS / Ops Manager Backups work
- Advantages to MMS / Ops Manager Backups

Notes:

Methods for Backing Up MongoDB

- mongodump
- File system backups
- MMS / Ops Manager Backups

Notes:

Comparing MongoDB Backup Methods

Considerations	Mongodump	File System	MMS Backup (Cloud)	Ops Manager (On-prem MMS)
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

Notes:

MMS / 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

Notes:

Restoring from MMS / Ops Manager

- Specify which backup to restore

- SCP push or HTTPS pull (one time use link) for data files

Notes:

Architecture

MongoDB Cluster

P

S

S

Backup Agent
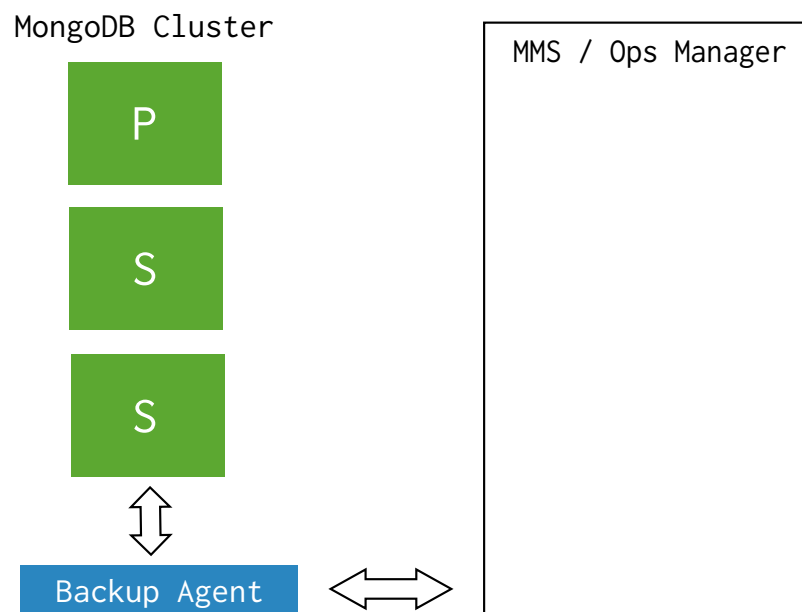
MMS / Ops Manager

Notes:

Snapshotting

- Local copy of every replica set stored by MMS / Ops Manager
- Oplog entries applied on top of local copy
- Local copy is used for snapshotting
- Very little impact to the cluster (equivalent to adding another secondary)

Notes:

Backup Agent

- Backup agent (can be managed by Automation agent)
- Backup agent sends oplog entries to MMS / Ops Manager service to be apply on local copy

Notes:

7.7 API

Learning Objectives

Upon completing this module students should understand:

- Overview of the MMS / Ops Manager API
- Sample use cases for the MMS / Ops Manager API

Notes:

What is the MMS / Ops Manager API?

Allows users to programmatically:

- Access monitoring data

- Backup functionality (request backups, change snapshot schedules, etc.)

- Automation cluster configuration (modify, view)

Notes:

API Documentation

Sample API Uses Cases

- Ingest MMS / Ops Manager monitoring data

- Programmatically restore environments

- Configuration management

Notes:

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

Notes:

Programatically Restore Environments

Use the backup API to programmatically restore an integration or testing environment based on the last production snapshot.

Notes:

Configuration Management

Use the automation API to integrate with existing configuration management tools (such as Chef or Puppet) to automate creating and maintaining environments.

Notes:

7.8 Exercise: MMS API

Learning Objectives

Upon completing this exercise students should understand:

- Have a basic understanding of working with the MMS API (or Ops Manager if the student chooses)

Notes:

Using the MMS API

If Ops Manager is installed, it may be used in place of MMS for this exercise.

Notes:

Exercise #1

Navigate the MMS interface to perform the following:

- Generate an API key
- Add your personal machine to the API whitelist

Notes:

Exercise #2

Modify and run the following curl command to return alerts for your MMS group:

```
curl -u "username:apiKey" --digest -i
"https://mms.mongodb.com/api/public/v1.0/groups/<GROUP-ID>/alerts"
```

Notes:

Exercise #3

How would you find metrics for a given host within your MMS account? Create an outline for the API calls needed.

Notes:

7.9 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

Notes:

MongoDB Ops Manager

- On-premises version of MMS
- Everything stays within private network

Notes:

Components

- Application server(s): web interface
- Ops Manager application database: monitoring metrics, automation configuration, etc.
- Backup infrastructure: cluster backups and restores

Notes:

Architecture
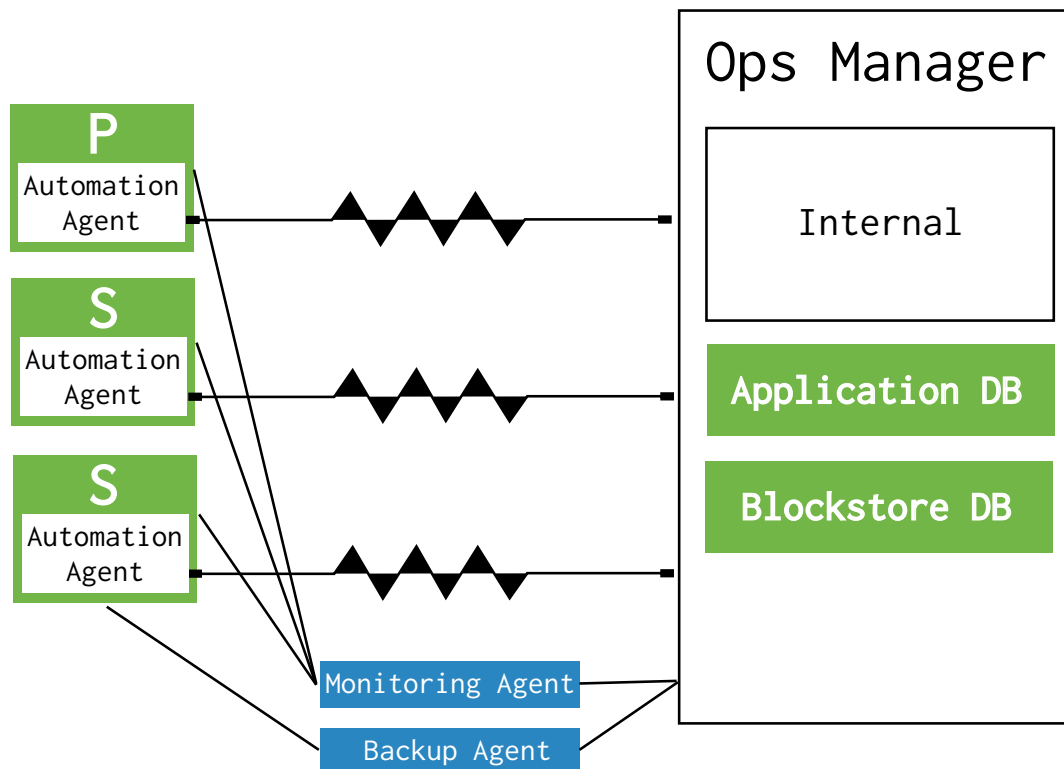
Notes:

Application Server

- 15GB RAM, 50GB of disk space are required
- Equivalent to a m3.xlarge AWS instance

Notes:

Application Database

- All monitoring metrics, automation configurations, etc. stored here
- Replica set, however, a standalone MongoDB node can also be used

Notes:

Backup Infrastructure

- Blockstore database
- Backup daemon process (manages applying oplog entries, creating snapshots, etc.)

Notes:

Blockstore Database

- 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)

Notes:

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 blockstores (for large clusters)

Notes:

7.10 Security (Ops Manager)

Learning Objectives

Upon completing this module students should understand:

- Ops Manager security overview

- Security and authentication options for Ops Manager

Notes:

Ops Manager User Authentication

- Two-Factor authentication can be enabled (uses Google Authenticator)
- LDAP authentication option

Notes:

Authentication for the Backing Ops Manager Databases

Ops Manager application database and blockstore database:

- MongoDB-CR (SCRAM-SHA1)
- LDAP
- Kerberos

Notes:

Authenticating Between an Ops Manager Agent and Cluster

- LDAP
- MongoDB-CR
- Kerberos (Linux only)

Notes:

Encrypting Communications

- All communications can be encrypted over SSL
- However, currently, if you are using Automation to manage a cluster, that cluster cannot use SSL or advanced auth.

Ops Manager Groups

- Users can belong to many different groups
- Users have different levels of access per group

Notes:

User Roles By Group

- Read Only
- User Admin
- Monitoring Admin
- Backup Admin
- Automation Admin
- Owner

Notes:

Global User Roles

- Global Read Only
- Global User Admin
- Global Monitoring Admin
- Global Backup Admin
- Global Automation Admin
- Global Owner

Notes:

7.11 Exercise: Install Ops Manager

Learning Objectives

Upon completing this exercise students should understand:

- The components needed for Ops Manager
- How to successfully install Ops Mananger

Notes:

Install Ops Manager

A Linux machine with at least 15GB of RAM is required

Notes:

Install Ops Manager

We will follow an outline of the installation instructions here:

https://docs.opsmanager.mongodb.com/current/tutorial/install-basic-deployment/

Notes:

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

Notes:

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
- Verify Ops Manager is running successfully

Notes:

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

Notes:

Exercise #4

Verify the Ops Manager installation was successful:

https://docs.opsmanager.mongodb.com/current/tutorial/test-new-deployment/

Notes:

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

Notes:

8 Performance Troubleshooting

Performance Troubleshooting **(page 189)** An introduction to reporting and diagnostic tools for MongoDB.

8.1 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()`

Notes:

`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 MMS for that.
- `mongotop` looks at the time spent on reads/writes in each collection.

Notes:

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.insert(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.update( { a : i, b : 255 }, { $set : { d : x.pad(1000) } });
    print(i)
}
```

Notes:

Exercise: `mongostat` (run)

- In another window/tab, run mongostat.
- You will see:
    - Inserts
    - Queries
    - Updates

Notes:

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 } )
```

Notes:

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.insert(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.update( {a: i, b: 255}, {$set: {d: x.pad(1000)}});
    print(i)
}
```

Notes:

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

Notes:

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.insert(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.update( {a: i, b: 255}, {$set: {d: x.pad(1000)}});
    print(i)
}
```

Notes:

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

Notes:

Exercise: Using Collection Stats

Look at the output of the following:

```
db.testcol.drop()
db.testcol.insert( { a : 1 } )
db.testcol.stats()
var x = "asdf"
db.testcol2.insert( { a : x.pad(10000000) } )
db.testcol2.stats()
db.stats()
```

Notes:

The Profiler

- Off by default.

- To reset, db.setProfilerLevel(0)

- At setting 1, it captures "slow" queries.

- You may define what "slow" is.

- Default is 100ms: db.setProfilerLevel(1)

- E.g., to capture 20 ms: db.setProfilerLevel(1, 20)

Notes:

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

Notes:

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.insert( { a : 1 } )
db.testcol.find()
var x = "asdf"
db.testcol.insert( { a : x.pad(10000000) } )  // ~10 MB
db.setProfilingLevel(0)
db.system.profile.find().pretty()
```

Notes:

`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.

Notes:

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.insert( { a : x.pad(100000) } )
}
```

- In the second window, type periodically:

```
var x = db.serverStatus(); x.metrics.document
```

Notes:

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.

- Allow class to discover this as the data is examined.

- You will see how fast documents are getting inserted.

- It will be slow b/c the documents are big.

Notes:

Performance Improvement Techniques

- Appropriate write concerns

- Bulk operations

- Good schema design

- Good Shard Key choice

- Good indexes

Notes:

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.

Notes:

Bulk Operations

- Using bulk operations can improve performance, especially when using write concern greater than 1.

- These enable the server to bulk write and bulk acknowledge.

- Can be done with both inserts and updates.

Notes:

Exercise: Comparing bulk inserts 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
```

Notes:

mongostat, **bulk inserts with** `{w:  1}`

Perform the following, with writeConcern : 1 and no bulk inserts:

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    for (j=1; j<=1000; j++) {
        db.testcol.insert( { _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.

Notes:

Bulk inserts 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.insert(
            { _id: (1000 * (i-1) + j), a: i, b: j, c: (1000 * (i-1)+ j)},
            { writeConcern: { w: 3 } }
        );
    };
    print(i);
}
```

Again, run mongostat.

Notes:

- Finally, let's use bulk inserts 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.insert( arr, { writeConcern : { w : 3 } } );
    print(i);
}
```

Notes:

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.

Notes:

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.

Notes:

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.

Notes:

9 Backup and Recovery

Backup and Recovery **(page 200)** An overview of file system backup procedures.

9.1 Backup and Recovery

Disasters Do Happen

Notes:

Human Disasters

Notes:

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?

Notes:

Terminology: DR vs. HA

- **Disaster Recovery** (DR)
- **High Availability** (HA)
- Distinct business requirements
- Technical solutions may converge

Notes:

Quiz

- Q: What's the hardest thing about backups?
- A: Restoring them!
- **Regularly test that restoration works!**

Notes:

Backup Options

- Document Level
  - Logical
  - `mongodump`, `mongorestore`
- File system level
  - Physical
  - Copy files
  - Volume/disk snapshots

Notes:

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

Notes:

`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
  --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
```

Notes:

File System Level

- **Must use journaling!**
- Copy `/data/db` files
- Or snapshot volume (e.g., LVM, SAN, EBS)
- *Seriously, always use journaling!*

Notes:

Ensure Consistency

Flush RAM to disk and stop accepting writes:

- `db.fsyncLock()`
- Copy/Snapshot
- `db.fsyncUnlock()`

Notes:

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

Notes:

Document Level: `mongorestore`

- `mongorestore`
- `--oplogReplay` replay oplog to point-in-time

Notes:

File System Restores

- All database files
- Selected databases or collections
- Replay Oplog

Notes:

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

Notes:

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

Notes:

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

Notes:

mongoDB

Find out more	**Having trouble?**	**Follow us on twitter**
mongodb.com | mongodb.org	File a JIRA ticket:	@MongoDBInc
university.mongodb.com	jira.mongodb.org	@MongoDB