MongoDB
UNIVERSITY

MongoDB Developer Training

MongoDB Developer Training

Release 3.0

MongoDB, Inc.

March 22, 2016

Contents

1 Introduction

1.1 Warm Up

Introductions

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

Getting to Know You

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

MongoDB Experience

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

10gen

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

Origin of MongoDB

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

1.2 MongoDB Overview

Learning Objectives

Upon completing this module students should understand:

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

MongoDB is a Document Database

Documents are associative arrays like:

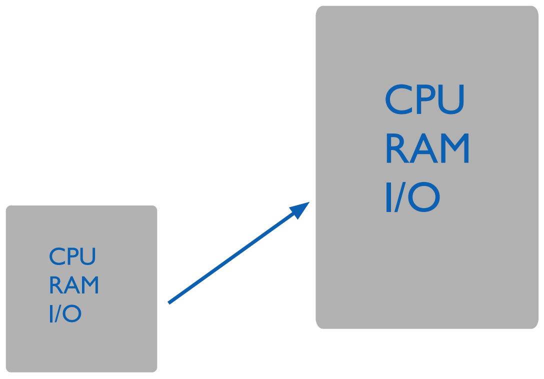
- Python dictionaries
- Ruby hashes
- PHP arrays
- JSON objects

An Example MongoDB Document

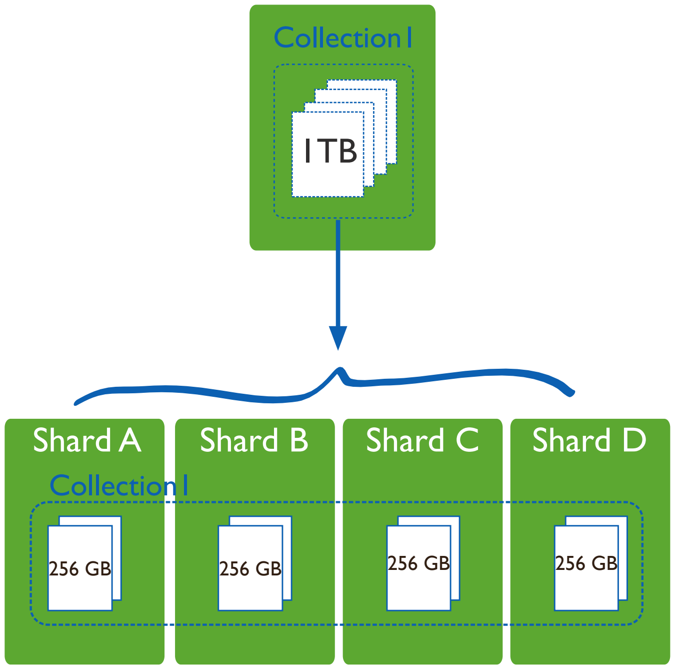
A MongoDB document expressed using JSON syntax.

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

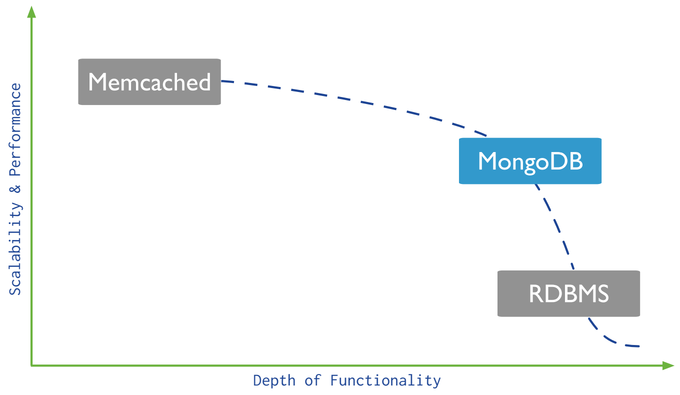
Vertical Scaling



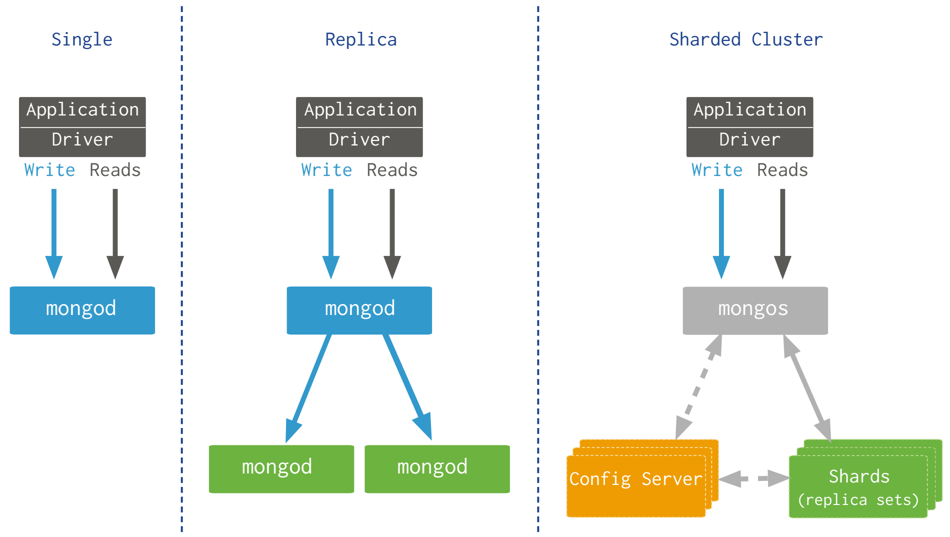
Scaling with MongoDB



Database Landscape



MongoDB Deployment Models



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

JSON

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

A Simple JSON Object

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

JSON Keys and Values

- Keys must be strings.
- Values may be any of the following:
  - string (e.g., “Thomas”)
  - number (e.g., 29, 3.7)
  - true / false
  - null
  - array (e.g., [88.5, 91.3, 67.1])
  - object
- More detail at json.org[1].

Example Field Values

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

[1] http://json.org/

BSON

- MongoDB stores data as Binary JSON (BSON).
- MongoDB drivers send and receive data in this format.
- They map BSON to native data structures.
- BSON provides support for all JSON data types and several others.
- BSON was designed to be lightweight, traversable and efficient.
- See bsonspec.org[2].

BSON Hello World

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

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

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"
```

Documents, Collections, and Databases

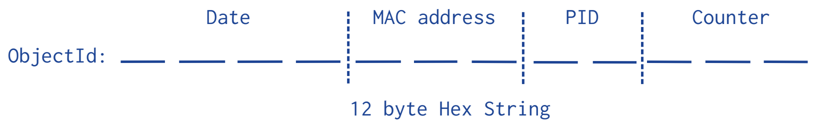
- Documents are stored in collections.
- Collections are contained in a database.
- Example:
  - Database: products
  - Collections: books, movies, music
- Each database-collection combination defines a namespace.
  - products.books
  - products.movies
  - products.music

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

The `_id` Field

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

ObjectIds



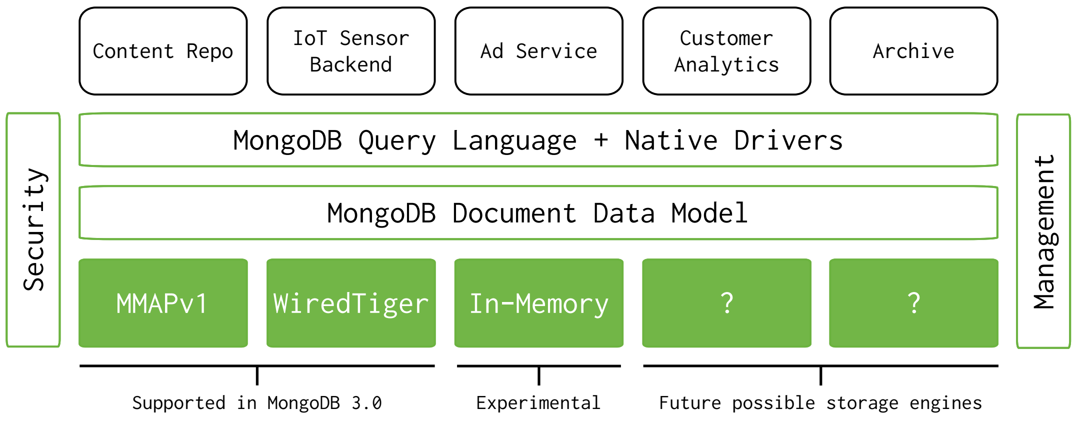
1.4 Storage Engines

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

What is a Database Storage Engine?



How Storage Engines Affect Performance

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

Storage Engine Journaling

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

MongoDB Storage Engines

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

- MMAPv1 (default)
- WiredTiger

Specifying a MongoDB Storage Engine

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

```
mongod --storageEngine wiredTiger
```

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

Compression in MongoDB

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

WiredTiger Storage Engine

- The WiredTiger storage engine excels at all workloads, especially write-heavy and update-heavy workloads.
- Notable features of the WiredTiger storage engine that do not exist in the MMAPv1 storage engine include:
  - Compression
  - Document-level concurrency
- Specify the use of the WiredTiger storage engine as follows.

  ```
  mongod --storageEngine wiredTiger
  ```

WiredTiger Compression Options

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

Configuring Compression in WiredTiger

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

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

Configuring Memory Usage in WiredTiger

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

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

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

MMAPv1 Journaling Mechanics (Continued)

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

WiredTiger Journaling Mechanics

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

Storage Engine API

MongoDB 3.0 introduced a storage engine API:

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

Conclusion

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

1.5 Lab: Installing and Configuring MongoDB

Learning Objectives

Upon completing this exercise students should understand:

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

Production Releases

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

- Windows
- OSX
- Linux
- Solaris

Installing MongoDB

- Visit 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.
- Versions:
  - Even-numbered builds are production releases, e.g., 2.4.x, 2.6.x.
  - Odd-numbers indicate development releases, e.g., 2.5.x, 2.7.x.

Linux Setup

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

sudo mkdir -p /data/db

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

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”

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
  ```

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
```

The MMAPv1 Data Directory

```
ls /data/db
```

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

The WiredTiger Data Directory

```
ls /data/db
```

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

Import Exercise Data

```
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

mongorestore -d air air
```

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

Launch a Mongo Shell

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

```
mongo
```

Display available commands.

```
help
```

Explore Databases

Display available databases.

```
show dbs
```

To use a particular database we can type the following.

```
use <database_name>

db
```

Exploring Collections

```
show collections
```

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

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

Admin Commands

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

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

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

2 MongoDB Drivers

2.1 Introduction to MongoDB Drivers

Learning Objectives

Upon completing this module, students should understand:

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

MongoDB Supported Drivers

- C[4]
- C++[5]
- C#[6]
- Java[7]
- Node.js[8]
- Perl[9]
- PHP[10]
- Python[11]
- Ruby[12]
- Scala[13]

[4] http://docs.mongodb.org/ecosystem/drivers/c
[5] http://docs.mongodb.org/ecosystem/drivers/cpp
[6] http://docs.mongodb.org/ecosystem/drivers/csharp
[7] http://docs.mongodb.org/ecosystem/drivers/java
[8] http://docs.mongodb.org/ecosystem/drivers/node-js
[9] http://docs.mongodb.org/ecosystem/drivers/perl
[10] http://docs.mongodb.org/ecosystem/drivers/php
[11] http://docs.mongodb.org/ecosystem/drivers/python
[12] http://docs.mongodb.org/ecosystem/drivers/ruby
[13] http://docs.mongodb.org/ecosystem/drivers/scala

MongoDB Community Supported Drivers

35+ different drivers for MongoDB:

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

Driver Specs

To ensure drivers have a consistent functionality, series of publicly available specification documents[14] for:

- Authentication[15]
- CRUD operations[16]
- Index management[17]
- SDAM[18]
- Server Selection[19]
- Etc.

Driver Settings (Per Operation)

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

Driver Settings (Per Connection)

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

[14] https://github.com/mongodb/specifications
[15] https://github.com/mongodb/specifications/tree/master/source/auth
[16] https://github.com/mongodb/specifications/tree/master/source/crud
[17] https://github.com/mongodb/specifications/blob/master/source/index-management.rst
[18] https://github.com/mongodb/specifications/tree/master/source/server-discovery-and-monitoring
[19] https://github.com/mongodb/specifications/tree/master/source/server-selection

2.2 MongoDB C++ Driver Introduction

Learning Objectives

Upon completing this module students should understand

- The different versions of MongoDB C++ Driver
- A basic understanding of main `namespaces`
- Different API interfaces
- Main classes

MongoDB C++ Driver

- The MongoDB C++ driver is a native C++ implementation library
- Released under Apache 2.0[20]
- Maintained and supported by MongoDB, Inc staff.
  - https://docs.mongodb.org/ecosystem/drivers/cpp/
- Freely available on github[21]

MongoDB C++ Driver Versions

Currently we support the following versions:

- legacy-1.1.0+[22]
- C++ 11[23]

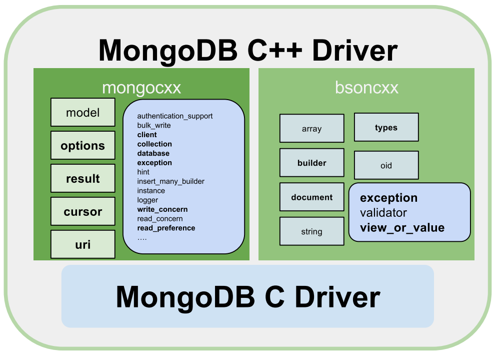
Driver Version	MongoDB 2.4	MongoDB 2.6	MongoDB 3.0	MongoDB 3.2
C++11 3.0.0	X	X	X	X
legacy-1.1.0+	X	X	X	X
legacy-1.0.0+	X	X	X	

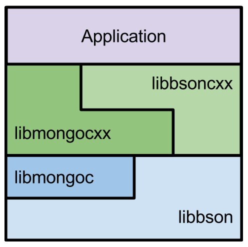
[20] http://www.apache.org/licenses/LICENSE-2.0
[21] https://github.com/mongodb/mongo-cxx-driver/
[22] http://api.mongodb.org/cxx/current/
[23] http://api.mongodb.org/cxx11/current/

MongoDB C++ Driver

For this particular course we will be focussing exclusively on the new C++11 Driver implementation.

Driver Architecture





`mongocxx`

In this namespace we will find the classes and structures that will deal with operations at server / database level.

- Client
- Connection configuration
- Database level operations
- Collection level operations
- Instruction options and exceptions handling
- Authentication Mechanism

`mongocxx`

```cpp
/// Class representing a MongoDB connection string URI.
using namespace mongocxx::uri;
uri m_uri("mongodb://training.mongodb.course:9999");

/// Class representing a client connection to MongoDB.
using namespace mongocxx::client;
client mc{m_uri};

/// Class representing a MongoDB database.
using namespace mongocxx::database;
database db = mongo_client["database_name"];

/// Class representing server side document groupings (a.k.a collections)
/// within a MongoDB database.
using namespace mongocxx::collection;
collection coll = db["collection_name"];
/// collections serve the data operations
coll.find_one({});
coll.insert_many([{}]);
coll.delete_many({});
```

mongocxx

```
/// Options is a specific namespace that determines the different operation options.
/// write_concern, read_preferences, skip, limit, sort
using namespace mongocxx::options;
find find_opts;
find_opts.sort(...);

insert insert_opts;
insert_opts.write_concern(...);

/// Result is a namespace the encloses all different types of write operation
/// response messages
using namespace mongocxx::result;
insert_many im_result;
im_result = coll.insert_many([ ... ], insert_opts);

/// Class representing a pointer to the result set of a query on a MongoDB server.
using namespace mongocxx::cursor;
cursor cur;
cur = coll.find({}, find_opts);
// ... iterate over cursor elements
```

bsoncxx

In this namespace we will find the classes and structures that will deal with document and BSON level instruction set API.

- Build documents
- Data types
- Document validation operations

bsoncxx Document Builder

For the vast majority of the exercises in this course we will be using `bsoncxx::builder` to create and parse BSON objects.

- The API offers 2 types of builders
  - basic
  - stream
- Over the examples and exercises we will be using mainly `bsoncxx::builder::stream` builder.

bsoncxx Document Builder

```
/// A streaming interface for constructing a BSON document.
using namespace bsoncxx::builder::stream;
document query;
query << "first_name" << "Marc" << "last_name" << "Schwering";

/// A traditional builder-style interface for constructing a BSON document.
using namespace bsoncxx::builder::basic;
document query;
query.append( kvp("first_name", "Marc") );
query.append( kvp("last_name", "Schwering") );
```

Document vs Views

In this driver there is a clear distinction between the document builders, the objects that we will be using for creating queries, documents, update statements ... and the actual object sent through the driver.

- builder creates the document structure
- view is sent to the driver operation.
  - this is a read only parsed version of the built document.

```
document query;
query << "singer" << "Jennifer Lopez";
cursor cur = coll.find( query.view()  );
```

2.3 Lab: Installing C++ MongoDB Driver

Learning Objectives

Upon completing this exercise students should understand:

- Prerequisites of MongoDB C++ Driver
- How to build and install the C++ Driver
- First *hellomongo.cpp* exercise

Latest Release

Current MongoDB C++11 Driver 3.0.0 is the first stable release of the C++11 series.

This new driver is **incompatible** with all previous MongoDB C++ drivers.

You can find the new release on the following link:

https://github.com/mongodb/mongo-cxx-driver/releases/tag/r3.0.0

Dependencies

The MongoDB C++11 Driver has some system requirements:

- GCC 4.8.2 OR clang-3.5 OR VS 2015 Update 1 or Apple Clang 5.1+
- CMake 3.2+
- MongoDB C driver version 1.3.1+

Before Installing the C Driver

After validating that compiler and make dependencies are met, ensure the following C driver dependencies are installed.

- automake
- autoconf
- libtool

Install MongoDB C Driver

Clone from GitHub and build

```
git clone https://github.com/mongodb/mongo-c-driver.git
cd mongo-c-driver
git checkout r1.3
./autogen.sh --with-libbson=bundled
make
sudo make install

For specific OS install `follow these instructions <http://api.mongodb.org/c/current/installing.html
```

Install MongoDB C++ Driver

To install the C++ driver, we will complete the following steps

1. Clone the repository and use tag r3.0.0
2. Build from source
3. Test the installation

Clone the C++ Driver Repository

Clone repository

```
git clone -b r3.0.0 https://github.com/mongodb/mongo-cxx-driver
cd mongo-cxx-driver
```

Build and Install the C++ Driver

Build and Install

```
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=/usr/local ..
sudo make && sudo make install
```

Test the C++ Driver

Let's edit *hellomongo.ccp*

```
void run(){
  mongocxx::client conn{mongocxx::uri{}};
  auto cursor = conn.list_databases();
  for (auto&& x : cursor) {
    std::cout << x["name"].get_utf8().value << std::endl;
  }
}

int main(int, char**) {
    try{
      run();
      std::cout << "connected ok" << std::endl;
    } catch( const mongocxx::exception &e ) {
      std::cout << "caught " << e.what() << std::endl;
    }
    return EXIT_SUCCESS;
}
```

Connect To MongoDB

Let's compile and run *hellomongo.cpp*

```
c++ --std=c++11 hellomongo.cpp -o hellomongo $(pkg-config --cflags --libs libmongocxx)
```

And execute the newly build *hellomongo* executable:

```
./hellomongo
```

3 CRUD

3.1 Creating and Deleting Documents (C++ Driver)

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

Baseline

For the extent of this lab we will be using same baseline code snippet called *crud.cpp*

Creating New Documents

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

```
// instantiate a connection
client conn{mongocxx::uri{}};
//instantiate a collection
collection coll = conn["sample"]["cpp"];

//create a document
auto doc0 = document{} << "name" << "Dwight" << finalize;

//insert one document
coll.insert_one(doc0.view());
```

Implicit `_id` Assignment

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

Example: Assigning _ids

Experiment with the following commands.

```
//assign `_id` field
auto doc1 = document{} << "_id" << 1 << "name" << "Dwight" << finalize;

//and fetch back the results
auto cursor = coll.find({});
for (auto&& x : cursor) {
  std::cout << "_id: " << x["_id"].get_int32() << std::endl;
}
```

Inserts will fail if...

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

Example: Inserts will fail if...

```
 //1 - fails to insert document
 auto doc2 = document{} << "_id" << open_array << 0 << 1 << 2<< close_array<< finalize;
 coll.insert_one(doc2.view());

//2 - succeeds
auto doc3 = document{} << "_id" << "Eliot" << finalize;
coll.insert_one(doc3.view());

//3 - fails because of duplicate id
auto doc4 = document{} << "_id" << "Eliot" << finalize;
coll.insert_one(doc4.view());

//4 - malformed document => compilation error!
auto doc5 = document{} << "hello" << finalize;
coll.insert_one(doc5.view());
```

Bulk Inserts

- MongoDB allows applications to perform 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.

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 `coll.insert_many()` is an ordered insert.
  - *options::insert{}*
- See the next exercise for an example.

Example: Ordered Bulk Insert

Experiment with the following bulk insert.

```
std::vector<bsoncxx::document::view> docs{};

auto b0 = document{} << "_id" << 0 << "name" << "Soccer" << finalize;
auto b1 = document{} << "_id" << 1 << "name" << "Rugby" << finalize;
auto b2 = document{} << "_id" << 0 << "name" << "Tennis" << finalize;
auto b3 = document{} << "_id" << 2 << "name" << "Football" << finalize;

docs.push_back(b0.view());
docs.push_back(b1.view());
docs.push_back(b2.view());
docs.push_back(b3.view());

auto result = coll.insert_many(docs, options::insert{});
```

Unordered Bulk Insert

- Pass `mongocxx::options::insert` with *ordered* flag set to false.
- 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

  ```
  auto options = options::insert{};
  options.ordered(false);
  ```

- One insert will fail, but all the rest will succeed.

Example: Unordered Bulk Insert

Experiment with the following bulk insert.

```
auto b0 = document{} << "_id" << 0 << "name" << "Badminton" << finalize;
auto b1 = document{} << "_id" << 1 << "name" << "Snooker" << finalize;
auto b2 = document{} << "_id" << 0 << "name" << "Karts" << finalize;
auto b3 = document{} << "_id" << 2 << "name" << "Javellin throw" << finalize;

docs.push_back(b0.view());
docs.push_back(b1.view());
docs.push_back(b2.view());
docs.push_back(b3.view());


auto options = options::insert{};
//insert unordered
options.ordered(false);
coll.insert_many(docs, options);
```

The Shell is a JavaScript Interpreter

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

Exercise: Creating Data in the Shell

Experiment with the following commands.

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

db.stuff.find()
```

Deleting Documents

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

- Use `delete_one` or `delete_many` to delete documents matching a specific set of conditions.
- Drop an entire collection.
- Drop a database.

Using `collection::delete` methods

- Remove documents from a collection using `delete_one()`.
- This command has one required parameter, a query document.
- `delete_one()` will remove one document in the collection matching the query document.
- If one passes an empty document it will still just remove one document.
- To remove all documents matching the query document one needs to use `delete_many()`

Example: Removing Documents

Using the previous generated dataset, do a `find()` after each `delete` command below.

```
// Remove the first document
delete_query << "a" << 1;
coll.delete_one(delete_query.view());

// $lt is a query operator that enables us to select documents that
// are less than some value. More on operators soon.
// Remove three more
delete_query << "a" << open_document << "$lt" << 5 << close_document ;
coll.delete_many(delete_query.view());


// Remove one more
delete_query << "a" << open_document << "$lt" << 10 << close_document ;
coll.delete_one(delete_query.view());

// All documents removed
coll.delete_many({});
```

Dropping a Collection

- You can drop an entire collection with:

  ```
  collection coll = conn["sample"]["tobedropped"];
  coll.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.

Example: Dropping a Collection

```
void drop_collection(){
  // instantiate a connection
  client conn{mongocxx::uri{}};
  //instantiate a collection
  collection coll = conn["sample"]["names"];

  auto b0 = document{} << "_id" << 0 << "name" << "Meghan" << finalize;
  auto b1 = document{} << "_id" << 1 << "name" << "Elizabeth" << finalize;
  auto b2 = document{} << "_id" << 2 << "name" << "Beatriz" << finalize;
  auto b3 = document{} << "_id" << 3 << "name" << "Anna" << finalize;

  coll.drop();
}
```

Dropping a Database

- You can drop an entire database with

  ```
  mongocxx::database db = conn["toBeDropped"];
  db.drop();
  ```

- This drops the database on which the method is called.
- It also deletes the associated data files from disk, freeing disk space.

Example: Dropping a Database

```
//insert some data
//...
// list all collections
auto cursor = db.list_collections();
auto count = std::distance(cursor.begin(), cursor.end());
std::cout << "Number of collections: " << count << std::endl;

db.drop();

cursor = db.list_collections();
count = std::distance(cursor.begin(), cursor.end());
std::cout << "Number of collections: " << count << std::endl;
```

3.2 Reading Documents (C++ Driver)

Learning Objectives

Upon completing this module students should understand:

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

The `find()` Method

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

Query by Example

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

Example: Querying by Example

Experiment with the following sequence of commands.

```
collection coll = conn["sample"]["sports"];
coll.drop();
//insert documents
std::vector<bsoncxx::document::view> docs{};
auto b0 = document{} << "_id" << 0 << "name" << "Badminton" << "indoor" << true << finalize;
auto b1 = document{} << "_id" << 1 << "name" << "Snooker" << "indoor" << true << finalize;
docs.push_back(b0.view());
docs.push_back(b1.view());
coll.insert_many(docs);
// find all documents in collection - db.sports.find()
coll.find({});
// find documents that match name = Snooker
document query = document{} << "name" << "Snooker";
coll.find(query);

// Multiple indoor sports but only one that is called Badminton
```

```
document query = document{} << "name" << "Snooker" << "indoor" << true;
coll.find(query);
```

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.

Example: Querying Arrays

Let’s assume the following dataset:

```
auto b0 = document{} << "name" << "Badminton"
    << "category" << open_array
      << "field" << "indoor" << "pairs" << "individual" << close_array
    << finalize;
auto b1 = document{} << "name" << "Javelin throw"
    << "category" << open_array
      << "individual" << "track" << "outdoors" << close_array
    << finalize;
auto b2 = document{} << "name" << "Football (not soccer!)"
    << "category" << open_array
      << "team" << "field" << "outdoors" << close_array
    << finalize;
std::vector<bsoncxx::document::view> docs{};
docs.push_back(b0.view());
docs.push_back(b1.view());
docs.push_back(b2.view());
auto options = options::insert{};
options.ordered(true);
coll.insert_many(docs, options );
```

Example: Querying Arrays

```
// Match documents where "category" contains the value specified
auto query0 = document{} << "category" << "field" << finalize;
// Should return 2 documents
auto cur = coll.find(query0.view());


// Match documents where "category" equals the value specified
auto query1 = document{} << "category" << open_array
  << "team" << close_array
<< finalize;
// no documents
auto cur = coll.find(query1.view());

// only the second document
auto query2 = document{} << "category" << open_array
  << "individual" << "track" << "outdoors" << close_array
<< finalize;
auto cur = coll.find(query2.view());
```

Querying with Dot Notation

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

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

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

Example: Querying with Dot Notation

Using the following documents

```
auto b0 = document{} << "name" << "Football (not soccer!)"
    << "worldcup" << open_document
      << "host" << "Brazil"
      << "teams" << 32
      << "champion" << "Germany"
    << close_document
  << finalize;
auto b1 = document{} << "name" << "Rugby"
    << "worldcup" << open_document
      << "host" << "England"
      << "teams" << 20
      << "champion" << "New Zeland"
    << close_document
  << finalize;
coll.insert_one(b0.view());
coll.insert_one(b1.view());
// find worldcup with 40 teams
auto query0 = document{} << "worldcup.teams" << 40 << finalize;
coll.find(query0.view()) // no values
```

Example: Querying with Dot Notation

```
// find worldcup is {teams : 32}
auto query0 = document{} << "worldcup"<< open_document
    << "teams" << 32
  << close_document
<< finalize;
iterate_cursor(coll.find(query0.view()), "query0");
// how many values should the above query return ?

 // dot notation
 auto query1 = document{} << "worldcup.teams" << 32 << finalize;
 coll.find(query1.view())
```

Example: Arrays and Dot Notation

Using the *mongo* shell we can create the following dataset:

```
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" }
         ] },
    { "title": "Star Wars",
      "filming_locations" :
         [ { "city" : "Ajim", "state" : "Jerba", "country" : "Tunisia" },
           { "city" : "Yuma", "state" : "AZ", "country" : "USA" }
         ] } ] )
```

Example: Arrays and Dot Notation

```
// db.movies.find( { "filming_locations.country" : "USA" } )
auto query = document{} << "filming_locations.country"<< "USA"<< finalize;
coll.find(query.view()); // two documents
```

Projections

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

Projection: Example (Setup)

```
db.movies.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
   }
 })
```

Projection: Example

```
document query;
document projection;
//db.movies.findOne( { "title" : "Forrest Gump" }, { "title" : 1, "imdb_rating" : 1 } )
query << "title" << "Forrest Gump";
projection <<  "title" << 1 << "imdb_rating" << 1;
options::find opts;
opts.projection( projection.view());
coll.find(query.view(), opts);
/*
  {
   "_id" : {
       "$oid" : "56e99cf3d7de9ca24ffc54e6"
   },
   "title" : "Forrest Gump",
   "imdb_rating" : 8.8
   }
*/
```

Projection Documents

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

Example: Projections

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

Cursors

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

Example: Introducing Cursors

```
collection coll = conn["sample"]["iterate"];
coll.drop();
for (int i=1; i<=10000; i++) {
    document b0;
    b0 << "i" << i;
    coll.insert_one(b0.view());
}

cursor c = coll.find({});

for ( auto&& d : c ){
  std::cout << bsoncxx::to_json(d) << std::endl;
}
```

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
```

Shell Cursor Methods

The *mongo* shell offers some cursor helper 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.

These are not present in the **c++** cursor object!

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(  )
```

To perform a count in **c++** you should use the following method:

```
cursor c = coll.find({});
std::distance(c.begin(), c.end());
```

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 } )
```

Example: Using `sort()` C++

To perform `sort` operations in **c++** we need to use the following:

```
mongocxx::options::find opts;
// sort a descending
auto sortby = document{} << "a" << -1 << finalize;

// sort a ascending
auto sortby = document{} << "a" << 1 << finalize;

// sort by natural insert order
auto sortby = document{} << "$natural" << 1 << finalize;

opts.sort(sortby.view());
coll.find({}, opts);
```

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.

The `limit()` Method

- Limits the number of documents in a result set to the first `k`.
- Specify `k` as the argument to `limit()`
- Helps reduce resources consumed by queries.

The `distinct()` Method

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

Example: Using `distinct()` shell

```
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" )
```

Example: Using `distinct()` C++

```
collection coll = conn["sample"]["movie_reviews"];
cursor c = coll.distinct("name", {})
for ( auto&& d : c ){
  std::cout << bsoncxx::to_json(d) << std::endl;
}
```

3.3 Query Operators (C++ Driver)

Learning Objectives

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

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

Comparison Query Operators

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

Example: 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
  }
] )
```

Example: Comparison Operators

```
// db.movies.find()
coll.find( {} );

// db.movies.find( { "imdb_rating" : { $gte : 7 } } )
query << "imdb_rating" << open_document << "$gte" << 7 << close_document;

// db.movies.find( { "category" : { $ne : "family" } } )
query << "category" << open_document << "$ne" << "family" << close_document;

// db.movies.find( { "title" : { $in : [ "Batman", "Godzilla" ] } } )
query << "title" << open_document << "$in"
  << open_array << "Batman" << "Godzilla" << close_array<< close_document;

// db.movies.find( { "title" : { $nin : [ "Batman", "Godzilla" ] } } )
query << "title" << open_document << "$nin"
  << open_array << "Batman" << "Godzilla" << close_array<< close_document;

coll.find( query.view() );
```

Logical Query Operators

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

Example: Logical Operators

```
/* db.movies.find( { $or : [
{ "category" : "sci-fi" }, { "imdb_rating" : { $gte : 7 } }] } )
*/
query << "$or"
  << open_array
    << open_document << "category" << "sci-fi" << close_document
    << open_document << "imdb_rating"
      << open_document << "$gte" << 7 << close_document
    << close_document
  << close_array;
```

Example: Logical Operators

```
// 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" : { $gte : 7 } }] } )
*/
document category_scifi, category_family;
category_scifi << "category" << "sci-fi"
  << "imdb_rating" << open_document  << "$gte" << 8 << close_document;
category_family << "category" << "sci-fi"
  << "imdb_rating" << open_document  << "$gte" << 7 << close_document;

b_document or1{category_scifi.view()};
b_document or2{category_family.view()};

query << "$or"
  << open_array
    << or1 << or2
  << close_array;
```

Example: Logical Operators

```
// find bad movies
//db.movies.find( { "imdb_rating" : { $not : { $gt : 7 } } } )
query << "imdb_rating"
  << open_document << "$not"
    << open_document << "$gt" << 7 << close_document
  << close_document;
```

Element Query Operators

- `$exists`: Select documents based on the existence of a particular field.
- `$type`: Select documents based on their type.
- See BSON types[24] for reference on types.

Example: Element Operators

```
using namespace builder::basic;
//db.movies.find( { "budget" : { $exists : true } } )
builder::basic::document query;
query.append(kvp("budget", [](sub_document sd) {
              sd.append(kvp("$exists", true));}));

// type 1 is Double
//db.movies.find( { "budget" : { $type : 1 } } )
builder::basic::document query;
query.append(kvp("budget", [](sub_document sd) {
              sd.append(kvp("$type", 1));}));

// type 3 is Object (embedded document)
```

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

```
//db.movies.find( { "budget" : { $type : 3 } } )
builder::basic::document query;
query.append(kvp("budget", [](sub_document sd) {
              sd.append(kvp("$type", 3));}));
```

Array Query Operators

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

Example: Array Operators

```
//db.movies.find( { "category" : { $all : [ "sci-fi", "action" ] } } )
builder::basic::document query;
query.append(kvp("category", [](sub_document sd) {
              sd.append(kvp("$all",[](sub_array sa) {
                sa.append("sci-fi", "action"); }));}
            ));

//db.movies.find( { "category" : { $size : 3 } } )
query.append(kvp("category", [](sub_document sd) {
              sd.append(kvp("$size", 3));}));
```

Example: $elemMatch

Let's assume we have the following document:

```
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" }
    ] } )
```

Example: $elemMatch

If we want to find a movie that was shoot in the city of Florence, country Italy

```
document query;
//db.movies.find( {"filming_locations.city" : "Florence","filming_locations.country" : "Italy"} )
query << "filming_locations.city" << "Florence"
<< "filming_locations.country" << "Italy";
// This query is incorrect, it won't return what we want


// $elemMatch is needed, now there are no results, this is expected
//db.movies.find( { "filming_locations" : { $elemMatch : { "city" : "Florence", "country" : "Italy"}
query << "filming_locations"
<< open_document << "$elemMatch"
  << open_document
```

```
    << "city" << "Florence" << "country" << "Italy"
  << close_document
<< close_document
```

3.4 Lab: Finding Documents (C++ Driver)

Exercise: Airline Country

In the air database, how many documents in airlines collection are registered in Spain?

Exercise: Airline < 10

In the air.airlines collection, find all airlines that have *airline* id lower than 10

Exercise: German or Italian Airlines

Find all airlines that are either German or Italian. Skip the first 5 and limit the results to 10.

Exercise: Route New York -> Barcelona

From the air database, how many routes depart from New York airports (NYC, LGA, EWR) with destination Barcelona (BCN)?

Exercise: Air France with Destination NYC

Find all *airplanes* that support the Air France routes with destination NYC.

3.5 Updating Documents (C++ Driver)

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

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.

Parameters to `update()`

- Keep the following in mind regarding the required parameters for `update`
- The query parameter:
  - Use the same syntax as with `find()`.
    - To update only one matching query we use `update_one`
    - To update all members matching query `update_many`
- 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.

`$set` and `$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`.

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
  }
] )
```

Example: `$set` and `$unset`

```
document criteria, update;
//db.movies.update( { "title" : "Batman" }, { $set : { "imdb_rating" : 7.7 } } )
criteria << "title" << "Batman";
update << "$set" << open_document << "imdb_rating" << 7.7 << close_document;

//db.movies.update( { "title" : "Godzilla" }, { $set : { "budget" : 1 } } )
criteria << "title" << "Godzilla";
update << "$set" << open_document << "budget" << 1 << close_document;

//db.movies.update( { "title" : "Home Alone" },
//                  { $set : { "budget" : 15, "imdb_rating" : 5.5 } } )
criteria << "title" << "Godzilla";
update << "$set" << open_document
 << "budget" << 15 << "imdb_rating" << 5.5
<< close_document;

//db.movies.update( { "title" : "Home Alone" }, { $unset :  { "budget" : 1 } } )
criteria << "title" << "Home Alone";
update << "$unset" << open_document << "budget" << 1 << close_document;
coll.update_one(criteria.view(), update.view());
```

Example: `$set` and `$unset`

How will this query behave?

```
//db.movies.update( { "title" : "Batman" }, { "imdb_rating" : 7.7 } )
criteria << "title" << "Batman";
update << "imdb_rating" << 7.7 ;
```

Example: Update Array Elements by Index

Mongo shell

```
// 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, 0 ]
                           } )
```

C++ instruction

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

Update Operators

- `$inc`: Increment a field's value by the specified amount.
- `$mul`: Multiply a field's value by the specified amount.
- `$rename`: Rename a field.
- `$set`: Update one or more fields.
- `$unset` Delete a field.
- `$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.

Example: Update Operators

```
criteria << "title" << "Batman";
//db.movies.update( { "title" : "Batman" }, { $inc : { "imdb_rating" : 2 } } )
update << "$inc"
<< open_document << "imdb_rating" << 2 << close_document;

//db.movies.update( { "title" : "Batman" }, { $inc : { "budget" : -5 } } )
update << "$inc"
<< open_document << "budget" << -5 << close_document;

// increment movie mentions by 10
// db.movie_mentions.update( { "title" : "Batman" } ,
//                           { $inc :  { "mentions_per_hour.5" : 10 } } )
update << "$inc"
<< open_document << "mentions_per_hour.5" << 10 << close_document;
```

Example: Update Operators

```
//db.movies.update( { "title" : "Batman" }, { $mul : { "imdb_rating" : 4 } } )
update << "$mul"
<< open_document << "imdb_rating" << 4 << close_document;

//db.movies.update( { "title" : "Batman" },
//                  { $rename : { "budget" : "estimated_budget" } } )
update << "$rename"
<< open_document << "budget" << "estimated_budget" << close_document;

//db.movies.update( { "title" : "Batman" }, { $min : { "budget" : 5 } } )
update << "$min"
<< open_document << "budget" << 5 << close_document;

//db.movies.update( { "title" : "Batman" },
//                  { $currentDate : { "last_updated" :
//                                     { $type : "timestamp" } } } )
update << "$currentDate"
<< open_document << "last_updated"
   << open_document << "$type" << "timestamp" << close_document
<< close_document;
```

Updating Multiple Documents

- In order to update multiple documents, `update_many()`.
- Bear in mind that without an appropriate index, you may scan every document in the collection.

Example: Multi-Update

Use `coll.find({});` after each of these updates.

```
// let's start tracking the number of sequals for each movie
// in the shell, we need { multi : true } to change all documents
//db.movies.update( { }, { $set : { "sequels" : 0 } },{ multi : true } )
// in c++ we just need to call update_many
update << "$set" << open_document << "sequels" : 0 << close_document;
coll.update_many({}, update.view());
```

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.

Example: Array Operators

```
criteria << "title" << "Batman";
//db.movies.update( { "title" : "Batman" },
//                  { $push : { "category" : "superhero" } } )
update << "$push"
<< open_document << "category" << "superhero" << close_document ;

//db.movies.update( { "title" : "Batman" },
//                  { $pushAll : { "category" : [ "villain", "comicbased" ] } } )
update << "$pushAll"
<< open_document << "category"
   << open_array <<  "villain" << "comicbased" << close_array
<< close_document ;

//db.movies.update( { "title" : "Batman" },{ $pop : { "category" : 1 } } )
update << "$pop"
<< open_document << "category" << 1 << close_document ;
```

Example: Array Operators

```
//db.movies.update( { "title" : "Batman" },{ $pull : { "category" : "action" } } )
update << "$pull"
<< open_document << "category" << "action" << close_document ;

//db.movies.update( { "title" : "Batman" },
//                  { $pullAll : { "category" : [ "villain", "comicbased" ] } } )
update << "$pullAll"
<< open_document << "category"
   << open_array <<  "villain" << "comicbased" << close_array
<< close_document ;

//db.movies.update( { "title" : "Batman" },
//                  { $addToSet : { "category" : "action" } } )
update << "$addToSet"
<< open_document << "category" << "action" << close_document ;
//db.movies.update( { "title" : "Batman" },
//                  { $addToSet : { "category" : "action" } } )
update << "$addToSet"
<< open_document << "category" << "action" << close_document ;
```

The Positional $ Operator

- $[25] is a positional operator that specifies an element in an array to update.
- It acts as a placeholder for the first element that matches the query document.
- `$` replaces the element in the specified position with the value given.
- Example:

  ```
  update << "$UPDATE_OPERATOR"
  << open_document << "array.$" << value << close_document;
  ```

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 } )
criteria << "category" << "action";
update << "$set"
<< open_document << "category.$" << "action-adventure" << close_document

coll.update_many(criteria.view, update.view());
```

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

Upserts

- By default, if no document matches an update query, the `update` methods does nothing.
- By specifying `options::update::upsert()`, `update` will insert a new document if no matching document exists.
- In the mongo shell `db.<COLLECTION>.save()` method is syntactic sugar that performs an upsert if the _id is not yet present
- Syntax:

  ```
  options::update opts;
  opts.upsert(true);
  coll.update_one( criteria.view() , update.view(), opts );
  ```

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

```
options::update opts;
opts.upsert(true);
//db.movies.update( { "title" : "Jaws" },{ $inc: { "budget" : 5 } },
//                  { upsert: true } )
criteria << "title" << "Jaws";
update << "$inc"
<< open_document << "budget" << 5 << close_document;
coll.update_one(criteria.view(), update.view(), opts);

//db.movies.update( { "title" : "Jaws II" },
//                  { $inc: { "budget" : 5 } }, { upsert: true } )
criteria << "title" << "Jaws II";
update << "$inc"
<< open_document << "budget" << 5 << close_document;
coll.update_one(criteria.view(), update.view(), opts);

//db.movies.update( { "title" : "Jaws III", "category" : [ "horror" ] },
//                  { $set : { "budget" : 1 } },{ upsert: true } )
criteria
<< "title" << "Jaws II" << "category" << open_array << "horror"<< close_array;
update << "$set"
<< open_document << "budget" << 1 << close_document;
coll.update_one(criteria.view(), update.view(), opts);
```

Find and Modify

Modify a document and return either:

- The pre-modification document
- If `options::return_document::k_after` 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.

`find_one_and_update`

```
options::find_one_and_update opts;
opts.return_document(options::return_document::k_after);

criteria < "state" << "unprocessed";
update << "$set"
<< open_document
  << "worker_id" << 123 << "state" << "processing"
<< close_document;

auto doc = coll.find_one_and_update(criteria.view(), update.view(), opts);

std::cout << bsoncxx::to_json(doc) << std::endl;
```

3.6 Lab: Updating Documents (C++ Driver)

Exercise: Set Points

- Using the `air.airlines` namespace, set a “points” attribute to airlines.
- For example, airlines “Emirates” and “Qatar Airways” should have *90*.
- Set *points* to *75* for airlines “Iberia Airlines” and “Air France”.
- All the remaining should get *50* points

Exercise: 10 Extra-Credit Points

- You’re being nice, so you decide to add *10* points to every airline that is based out of *Portugal*.
- How do you do this update?

Exercise: Updating Array Elements

Insert a document representing flight bookings metrics for a route:

```
db.bookings.insert(
   { dst_airport: "JFK", src_airport: "LHR",
     bookingPast7Days: [ 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 bookings on Friday by 200.

4 Indexes

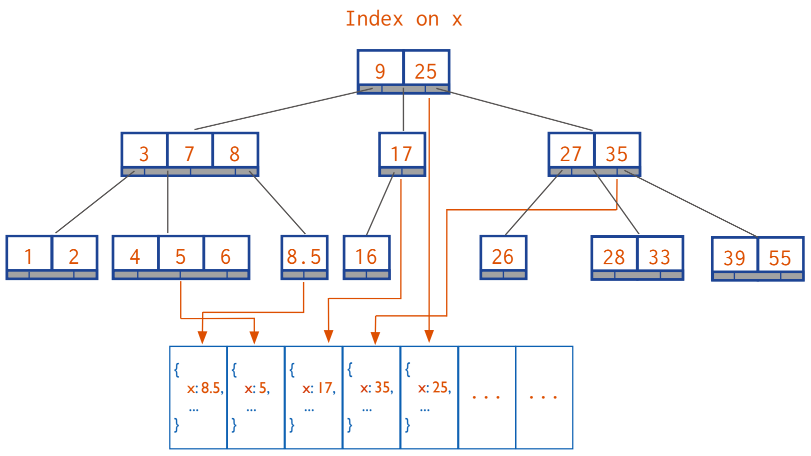
4.1 Index Fundamentals

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

Why Indexes?



Types of Indexes

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

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()
```

Results of `explain()`

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

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

Results of `explain()` - Continued

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

`explain()` Verbosity Can Be Adjusted

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

`explain("executionStats")`

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

Now we have query statistics:

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

`explain("executionStats")` - Continued

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

`explain("executionStats")` Output

- `nReturned` 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.

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`

`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()
```

Using `explain()` for Write Operations

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

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

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

Single-Field Indexes

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

Creating an Index

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

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

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

Listing Indexes

List indexes for a collection:

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

List index keys:

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

Indexes and Read/Write Performance

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

Index Limitations

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

Use Indexes with Care

- Every query should use an index.
- Every index should be used by a query.
- Any write operation that touches an indexed field will require each index to be updated.
- Indexes require RAM.
- Be judicious about the choice of key.

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 } )
```

4.2 Compound Indexes and Troubleshooting Index Performance

Learning Objectives

Upon completing this module students should understand:

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

Introduction to Compound Indexes

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

The Order of Fields Matters

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

- Equality tests, e.g.,

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

- Range queries, e.g.,

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

- Sorting, e.g.,

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

Designing Compound Indexes

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

Example: A Simple Message Board

Requirements:

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

Load the Data

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

Start with a Simple Index

Start by building an index on { timestamp : 1 }

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

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

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

Analysis:

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

Query Adding `username`

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

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

`totalKeysExamined` > `n`.

Include `username` in Our Index

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

`totalKeysExamined` is still > `n`. Why?

`totalKeysExamined` > `n`

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

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.

`totalKeysExamined == n`

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

Let Selectivity Drive Field Order

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

Adding in the Sort

Finally, let’s add the sort and run the query.

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

- 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.

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.

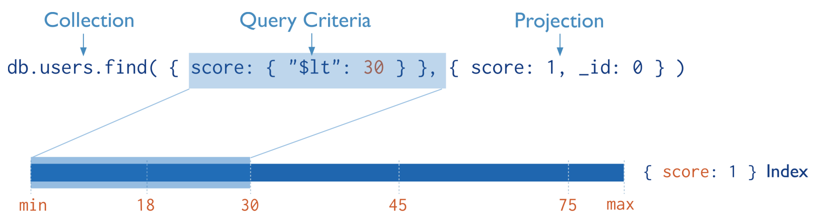
```
db.messages.dropIndex( "myindex" );
db.messages.createIndex( { username : 1, rating : 1, timestamp : 1 },
                         { name : "myindex" } );
db.messages.find( {
                    timestamp : { $gte : 2, $lte : 4 },
                    username : "anonymous"
                  } ).sort(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

General Rules of Thumb

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

Covered Queries



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

Exercise: Covered Queries

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

4.3 Lab: Optimizing an Index

Exercise: What Index Do We Need?

Run the the following Javascript file from the handouts.

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

In the shell that launches execute the following method

```
performance.init()
```

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

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

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 } )
```

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 })
```

4.4 Multikey Indexes

Learning Objectives

Upon completing this module, students should understand:

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

Introduction to Multikey Indexes

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

Example: Array of Numbers

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

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 })
```

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 } )
```

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()
```

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 ] } )
```

How Multikey Indexes Work

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

Multikey Indexes and Sorting

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

Exercise: Multikey Indexes and Sorting

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

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

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

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

Limitations on Multikey Indexes

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

Example: Multikey Indexes on Multiple Fields

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

// no problems yet
db.testcol.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 ] } )
```

4.5 Hashed Indexes

Learning Objectives

Upon completing this module, students should understand:

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

What is a Hashed Index?

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

Why Hashed Indexes?

- In MongoDB, the primary use for hashed indexes is to support sharding a collection using a hashed shard key.
- In some cases, the field we would like to use to shard data would make it difficult to scale using sharding.
- Using a hashed shard key to shard a collection ensures an even distribution of data and overcomes this problem.
- See Shard a Collection Using a Hashed Shard Key[26] for more details.
- We discuss sharding in detail in another module.

Limitations

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

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}.

[26] 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" } )
```

4.6 Geospatial Indexes

Learning Objectives

Upon completing this module, students should understand:

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

Introduction to Geospatial Indexes

We can use geospatial indexes to quickly determine geometric relationships:

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

Easiest to Start with 2 Dimensions

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

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.

Find Nearby Documents

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

Flat vs. Spherical Indexes

There are two types of geospatial indexes:

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

Flat Geospatial Index

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

Spherical Geospatial Index

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

Creating a 2d Index

Creating a 2d index:

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

Possible options key-value pairs:

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

Exercise: Creating a 2d Index

Create a 2d index on the collection `testcol` with:

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

Inserting Documents with a 2d Index

There are two accepted formats:

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

Exercise: Inserting Documents with 2d Fields

- Insert 2 documents into the `testcol` 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.

Querying Documents Using a 2d Index

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

Example: Find Based on 2d Coords

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

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

Creating a 2dsphere Index

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

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

The geoJSON Specification

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

geoJSON Considerations

- The coordinates of points are given in degrees (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.

Simple Types of 2dsphere Objects

Point: A single point on the globe

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

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

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

Polygons

Simple Polygon:

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

Polygon with One Hole:

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

Other Types of 2dsphere Objects

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

Exercise: Inserting geoJSON Objects (1)

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

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

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".

Exercise: Inserting geoJSON Objects (3)

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

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.

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.

4.7 TTL Indexes

Learning Objectives

Upon completing this module students should understand:

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

TTL Index Basics

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

Creating a TTL Index

Create with:

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

Exercise: Creating a TTL Index

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

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

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

Exercise: Check the Collection

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

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

4.8 Text Indexes

Learning Objectives

Upon completing this module, students should understand:

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

What is a Text Index?

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

Creating a Text Index

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

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

Exercise: Creating a Text Index

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

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

Creating a Text Index with Weighted Fields

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

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

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

Creating a Text Index with Weighted Fields

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

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

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

Text Indexes are Similar to Multikey Indexes

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

Exercise: Inserting Texts

Let's add some documents to our montyPython collection.

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

Querying a Text Index

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

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

Exercise: Querying a Text Index

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

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

Exercise: Querying Using Two Words

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

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

Search for a Phrase

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

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

Text Search Score

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

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

4.9 Lab: Finding and Addressing Slow Operations

Exercise: Determine Indexes Needed

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

4.10 Lab: Using `explain()`

Exercise: explain("executionStats")

Drop all indexes from previous exercises:

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

Create an index for the "active" field:

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

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

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

5 Aggregation

5.1 An Aggregation Tutorial Using Twitter Data

Learning Objectives

Upon completing this module students should understand:

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

Aggregation Basics

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

The Aggregation Pipeline

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

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

Aggregation Stages

- `$match`: Similar to `find()`
- `$project`: Shape documents
- `$sort`: Like the cursor method of the same name
- `$skip`: Like the cursor method of the same name
- `$limit`: Like the cursor method of the same name
- `$unwind`: Used for working with arrays
- `$group`: Used to aggregate field values from multiple documents
- `$out`: Creates a new collection from the output of an aggregation pipeline

The Match Stage

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

The Project Stage

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

A Twitter Dataset

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

Tweets Data Model

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

Analyzing Tweets

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

Friends and Followers

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

Exercise: Friends and Followers

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

Exercise: $match and $project

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

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

The Group Stage

- For those coming from the relational world, `$group` is similar to the SQL `GROUP BY` statement.
- $group operations require that we specify which field to group on.
- Documents with the same identifier will be aggregated together.
- With $group, we aggregate values using accumulators[27].

Tweet Source

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

[27]http://docs.mongodb.org/manual/meta/aggregation-quick-reference/#accumulators

Exercise: Tweet Source

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

Group Aggregation Accumulators

Accumulators available in the group stage:

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

Rank Users by Number of Tweets

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

Exercise: Adding List of Tweets

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

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

The Unwind Stage

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

Example: User Mentions in a Tweet

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

Using $unwind

Who includes the most user mentions in their tweets?

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

Data Processing Pipelines

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

Most Unique User Mentions

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

Same Operator ($group), Multiple Stages

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

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

The Sort Stage

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

The Skip Stage

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

The Limit Stage

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

The Out Stage

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

5.2 Optimizing Aggregation

Learning Objectives

Upon completing this module students should understand:

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

Aggregation Options

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

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

- Following are some of the fields that may be passed in the options document.
  - `allowDiskUse : true` - permit the use of disk for memory-intensive queries
  - `explain : true` - display how indexes are used to perform the aggregation.

Aggregation Limits

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

Limits Prior to MongoDB 2.6

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

Optimization: Reducing Documents in the Pipeline

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

Optimization: Sorting

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

Automatic Optimizations

MongoDB will perform some optimizations automatically. For example:

- If a `$project` stage is used late in the pipeline it may be used to eliminate those fields earlier if possible.
- A `$sort` followed by a `$match` will be executed as a `$match` followed by a `$sort` to reduce the number of documents to be sorted.
- A `$skip` followed by a `$limit` will be executed as a `$limit` followed by a `$skip`, with the $limit parameter increased by the $skip amount to allow $sort + $limit coalescence.
- See: Aggregation Pipeline Optimation[28]

5.3 Lab: Working with Array Fields

Lab: Working with Array Fields

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

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

```
mongoimport -d blog -c posts --drop posts.json
```

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

5.4 Lab: Repeated Aggregation Stages

Lab: Repeated Aggregation Stages

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

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

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

Please note:

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

[28] http://docs.mongodb.org/manual/core/aggregation-pipeline-optimization/

5.5 Lab: Descriptive Statistics

Lab: Descriptive Statistics

From the `grades` collection, find the class (display the `class_id`) with the best average student performance.

If you have not already done so, import the grades collection as follows.

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

Before you attempt this exercise, explore the `grades` collection a little to ensure you understand how it is structured.

For additional exercises, consider other statistics you might want to see with this data and how to calculate them.

6 Schema Design

6.1 Schema Design Core Concepts

Learning Objectives

Upon completing this module, students should understand:

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

What is a schema?

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

Example: Normalized Data Model

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

Example: Denormalized Version

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

Schema Design in MongoDB

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

Three Considerations

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

Case Study

- A Library Web Application
- Different schemas are possible.

Author Schema

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

User Schema

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

Book Schema

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

Example Documents: Author

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

Example Documents: User

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

Example Documents: Book

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

Embedded Documents

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

Example: Embedded Documents

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

Embedded Documents: Pros and Cons

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

Linked Documents

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

Example: Linked Documents

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

Linked Documents: Pros and Cons

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

Arrays

- Array of scalars
- Array of documents

Array of Scalars

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

Array of Documents

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

Exercise: Users and Book Reviews

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

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

Solution A: Users and Book Reviews

Reviews may be queried by user or book

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

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

Solution B: Users and Book Reviews

Optimized to retrieve reviews by user

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

Solution C: Users and Book Reviews

Optimized to retrieve reviews by book

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

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

Store Binary Files in MongoDB with GridFS

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

How GridFS Works

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

Schema Design Use Cases with GridsFS

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

6.2 Schema Evolution

Learning Objectives

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

- Development Phase
- Production Phase
- Iterative Modifications

Development Phase

Support basic CRUD functionality:

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

Development Phase: Known Query Patterns

```
// Find authors by last name.
db.authors.createIndex({ "lastName": 1 })

// Find books by slug for detail view
db.books.createIndex({ "slug": 1 })

// Find books by subject (multi-key)
db.books.createIndex({ "subjects": 1 })

// Find books by publisher (index on embedded doc)
db.books.createIndex({ "publisher.name": 1 })
```

Production Phase

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

Production Phase: Read Patterns

List books by author last name

```
authors = db.authors.find({ lastName: /^f.*/i }, { _id: 1 });

authorIds = authors.map(function(x) { return x._id; });

db.books.find({author: { $in: authorIds }});
```

Addressing List Books by Last Name

"Cache" the author name in an embedded document.

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

Queries are now one step

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

Production Phase: Write Patterns

Users can review a book.

```
review = {
    user: 1,
    text: "I thought this book was great!",
    rating: 5
};

db.books.update(
    { _id: 3 },
    { $push: { reviews: review }}
);
```

Caveats:

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

Exercise: Recent Reviews

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

Solution: Recent Reviews, Schema

Store users' reviews in monthly buckets.

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

Solution: Recent Reviews, Update

Adding a new review to the appropriate bucket

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

db.reviews.update(
      { _id: "bob-201210" },
      { $push: { reviews: myReview }}
);
```

Solution: Recent Reviews, Read

Display the 10 most recent reviews by a user

```
cursor = db.reviews.find(
    { _id: /^bob-/ },
    { reviews: { $slice: -10 }}
).sort({ _id: -1 }).batchSize(5);

num = 0;

while (cursor.hasNext() && num < 10) {
    doc = cursor.next();

    for (var i = 0; i < doc.reviews.length && num < 10; ++i, ++num) {
        printjson(doc.reviews[i]);
    }
}
```

Solution: Recent Reviews, Delete

Deleting a review

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

6.3 Common Schema Design Patterns

Learning Objectives

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

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

One-to-One Relationship

Let's pretend that authors only write one book.

One-to-One: Linking

Either side, or both, can track the relationship.

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

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

One-to-One: Embedding

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

One-to-Many Relationship

In reality, authors may write multiple books.

One-to-Many: Array of IDs

The "one" side tracks the relationship.

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

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

One-to-Many: Single Field with ID

The "many" side tracks the relationship.

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

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

One-to-Many: Array of Documents

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

Many-to-Many Relationship

Some books may also have co-authors.

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

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

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

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

Query for all books by a given author.

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

Query for all authors of a given book.

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

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

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

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

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

Query for all books by a given author.

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

Query for all authors of a given book.

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

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

Tree Structures

E.g., modeling a subject hierarchy.

Allow users to browse by subject

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

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

Alternative: Parents and Ancestors

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

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

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

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

Find Sub-Categories

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

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

Summary

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

6.4 Lab: Data Model for an E-Commerce Site

Introduction

- In this group exercise, we're going to take what we've learned about MongoDB and develop a basic but reasonable data model for an e-commerce site.
- For users of RDBMSs, the most challenging part of the exercise will be figuring out how to construct a data model when joins aren't allowed.
- We're going to model for several entities and features.

Product Catalog

- **Products.** Products vary quite a bit. In addition to the standard production attributes, we will allow for variations of product type and custom attributes. E.g., users may search for blue jackets, 11-inch macbooks, or size 12 shoes. The product catalog will contain millions of products.
- **Product pricing.** Current prices as well as price histories.
- **Product categories.** Every e-commerce site includes a category hierarchy. We need to allow for both that hierarchy and the many-to-many relationship between products and categories.
- **Product reviews.** Every product has zero or more reviews and each review can receive votes and comments.

Product Metrics

- **Product views and purchases.** Keep track of the number of times each product is viewed and when each product is purchased.
- **Top 10 lists.** Create queries for top 10 viewed products, top 10 purchased products.
- **Graph historical trends.** Create a query to graph how a product is viewed/purchased over the past.
- **30 days with 1 hour granularity.** This graph will appear on every product page, the query must be very fast.

Deliverables

- Sample document and schema for each collection
- Queries the application will use
- Index definitions

Break into groups of two or three and work together to create these deliverables.

7 Replica Sets

7.1 Introduction to Replica Sets

Learning Objectives

Upon completing this module, students should understand:

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

Use Cases for Replication

- High Availability
- Disaster Recovery
- Functional Segregation

High Availability (HA)

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

Disaster Recovery (DR)

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

Functional Segregation

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

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

Large Replica Sets

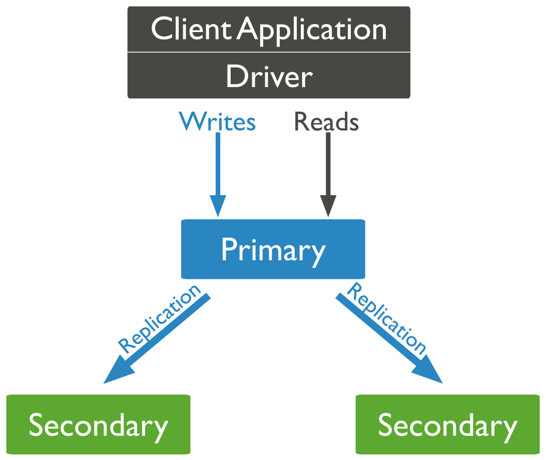
Functional segregation can be further exploited by using large replica sets.

- 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.

Replica Sets



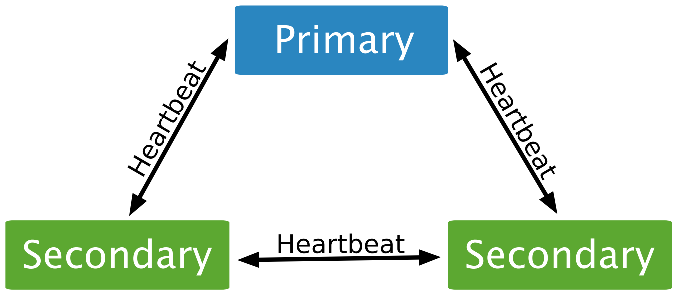
Primary Server

- 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.

Secondaries

- A secondary replicates operations from another node in the replica set.
- Secondaries usually replicate from the primary.
- Secondaries may also replicate from other secondaries. This is called replication chaining.
- A secondary may become primary as a result of a failover scenario.

Heartbeats



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.

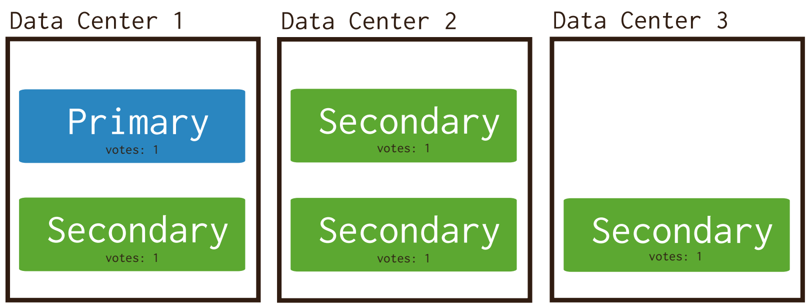
7.2 Elections in Replica Sets

Learning Objectives

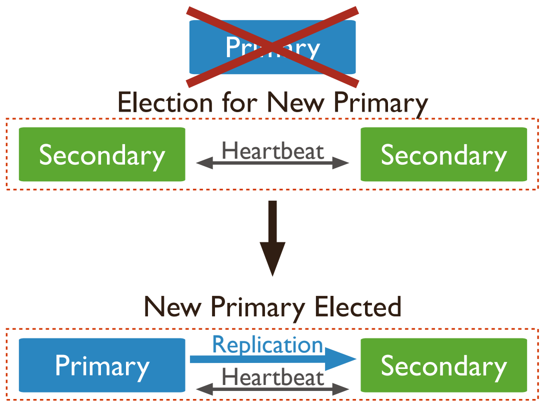
Upon completing this module students should understand:

- That elections enable automated failover in replica sets
- How votes are distributed to members
- What prompts an election
- How a new primary is selected

Members and Votes



Calling Elections



Selecting a New Primary

Three factors are important in the selection of a primary:

- Priority
- Optime
- Connections

Priority

- The higher its priority, the more likely a member is to become primary.
- The default is 1.
- Servers with a priority of 0 will never become primary.
- Priority values are floating point numbers 0 - 1000 inclusive.

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.

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.

When will a primary step down?

- After receiving the `replSetStepDown` or `rs.stepDown()` command.
- If a secondary is eligible for election and has a higher priority.
- If it cannot contact a majority of the members of the replica set.

replSetStepDown Behavior

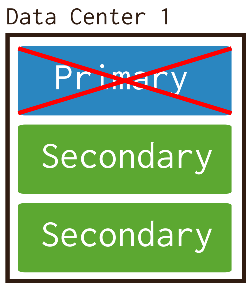
- Primary will attempt to terminate long running operations before stepping down.
- Primary will wait for electable secondary to catch up before stepping down.
- "secondaryCatchUpPeriodSecs" can be specified to limit the amount of time the primary will wait for a secondary to catch up before the primary steps down.

Exercise: Elections in Failover Scenarios

- We have learned about electing a primary in replica sets.
- Let's look at some scenarios in which failover might be necessary.

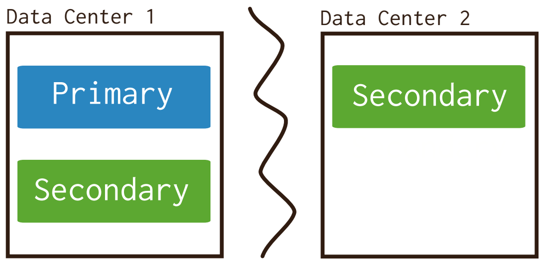
Scenario A: 3 Data Nodes in 1 DC

Which secondary will become the new primary?



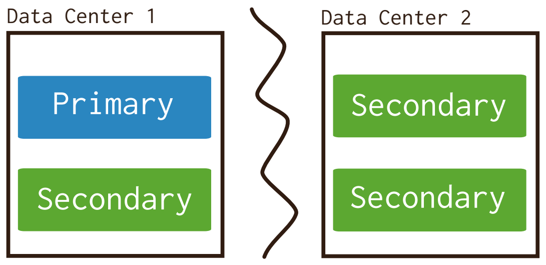
Scenario B: 3 Data Nodes in 2 DCs

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



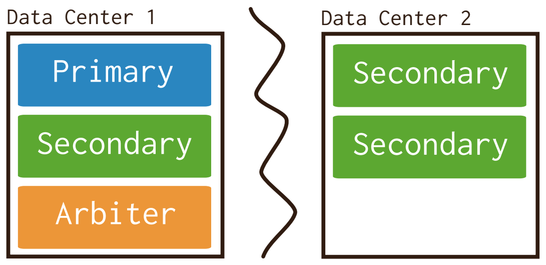
Scenario C: 4 Data Nodes in 2 DCs

What happens following this network partition?



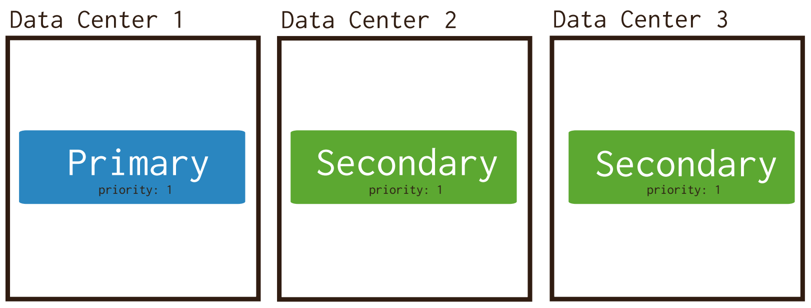
Scenario D: 5 Nodes in 2 DCs

The following is similar to Scenario C, but with the addition of an arbiter in Data Center 1. What happens here?



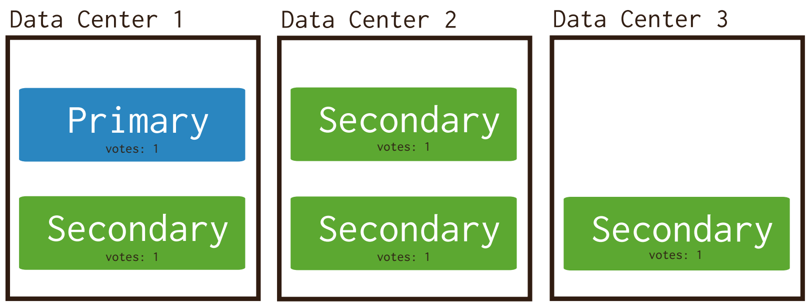
Scenario E: 3 Data Nodes in 3 DCs

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



Scenario F: 5 Data Nodes in 3 DCs

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



7.3 Replica Set Roles and Configuration

Learning Objectives

Upon completing this module students should understand:

- The use of priority to preference certain members or datacenters as primaries.
- Hidden members.
- The use of hidden secondaries for data analytics and other purposes (when secondary reads are used).
- The use of slaveDelay to protect against operator error.

Example: A Five-Member Replica Set Configuration

- For this example application, there are two datacenters.
- We name the hosts accordingly: `dc1-1`, `dc1-2`, `dc2-1`, etc.
  - This is just a clarifying convention for this example.
  - MongoDB does not care about host names except to establish connections.
- The nodes in this replica set have a variety of roles in this application.

Configuration

```
conf = {              // 5 data-bearing nodes
  _id: "mySet",
  members: [
    { _id : 0, host : "dc1-1.example.net:27017", priority : 5 },
    { _id : 1, host : "dc1-2.example.net:27017", priority : 5 },
    { _id : 2, host : "dc2-1.example.net:27017" },
    { _id : 3, host : "dc1-3.example.net:27017", hidden : true },
    { _id : 4, host : "dc2-2.example.net:27017", hidden : true,
      slaveDelay: 7200 }
  ]
}
```

Principal Data Center

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

Data Center 2

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

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 }
```

What about dc2-2?

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

7.4 The Oplog: Statement Based Replication

Learning Objectives

Upon completing this module students should understand:

- Binary vs. statement-based replication.
- How the oplog is used to support replication.
- How operations in MongoDB are translated into operations written to the oplog.
- Why oplog operations are idempotent.
- That the oplog is a capped collection and the implications this holds for syncing members.

Binary Replication

- MongoDB replication is statement based.
- Contrast that with binary replication.
- With binary replication we would keep track of:
  - The data files
  - The offsets
  - How many bytes were written for each change
- In short, we would keep track of actual bytes and very specific locations.
- We would simply replicate these changes across secondaries.

Tradeoffs

- The good thing is that figuring out where to write, etc. is very efficient.
- But we must have a byte-for-byte match of our data files on the primary and secondaries.
- The problem is that this couples our replica set members in ways that are inflexible.
- Binary replication may also replicate disk corruption.

Statement-Based Replication

- Statement-based replication facilitates greater independence among members of a replica set.
- MongoDB stores a statement for every operation in a capped collection called the `oplog`.
- Secondaries do not simply apply exactly the operation that was issued on the primary.

Example

Suppose the following 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 } }
```

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.

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.

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.

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.

7.5 Lab: Working with the Oplog

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 } )
```

`ReplSetTest`

- ReplSetTest is useful for experimenting with replica sets as a means of hands-on learning.
- It should never be used in production. Never.
- The command above will create a replica set with three members.
- It does not start the mongods, however.
- You will need to issue additional commands to do that.

Start the Replica Set

Start the mongod processes for this replica set.

```
replicaSet.startSet()
```

Issue the following command to configure replication for these mongods. You will need to issue this while output is flying by in the shell.

```
replicaSet.initiate()
```

Status Check

- You should now have three mongods running on ports 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.

Connect to the Primary

Open a new shell, connecting to the primary.

```
mongo --port 31000
```

Create some Inventory Data

Use the `store` database:

```
use store
```

Add the following inventory:

```
inventory = [ { _id: 1, inStock: 10 }, { _id: 2, inStock: 20 },
              { _id: 3, inStock: 30 }, { _id: 4, inStock: 40 },
              { _id: 5, inStock: 50 }, { _id: 6, inStock: 60 } ]
db.products.insert(inventory)
```

Perform an Update

Issue the following update. We might issue this update after a purchase of three items.

```
db.products.update({ _id: { $in: [ 2, 5 ] } },
                   { $inc: { inStock : -1 } },
                   { multi: true })
```

View the Oplog

The oplog is a capped collection in the `local` database of each replica set member:

```
use local
db.oplog.rs.find()
{ "ts" : Timestamp(1406944987, 1), "h" : NumberLong(0), "v" : 2, "op" : "n",
  "ns" : "", "o" : { "msg" : "initiating set" } }
...
{ "ts" : Timestamp(1406945076, 1), "h" : NumberLong("-9144645443320713428"),
  "v" : 2, "op" : "u", "ns" : "store.products", "o2" : { "_id" : 2 },
  "o" : { "$set" : { "inStock" : 19 } } }
{ "ts" : Timestamp(1406945076, 2), "h" : NumberLong("-7873096834441143322"),
  "v" : 2, "op" : "u", "ns" : "store.products", "o2" : { "_id" : 5 },
  "o" : { "$set" : { "inStock" : 49 } } }
```

7.6 Write Concern

Learning Objectives

Upon completing this module students should understand:

- How and when rollback occurs in MongoDB.
- The tradeoffs between durability and performance.
- Write concern as a means of ensuring durability in MongoDB.
- The different levels of write concern.

What happens to the write?

- A write is sent to a primary.
- The primary acknowledges the write to the client.
- The primary then becomes unavailable before a secondary can replicate the write

Answer

- Another member might be elected primary.
- It will not have the last write that occurred before the previous primary became unavailable.
- When the previous primary becomes available again:
  - It will note it has writes that were not replicated.
  - It will put these writes into a `rollback file`.
  - A human will need to determine what to do with this data.
- This is default behavior in MongoDB and can be controlled using `write concern`.

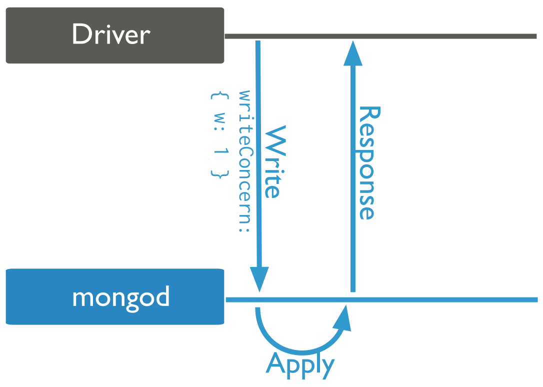
Balancing Durability with Performance

- The previous scenario is a specific instance of a common distributed systems problem.
- For some applications it might be acceptable for writes to be rolled back.
- Other applications may have varying requirements with regard to durability.
- Tunable write concern:
  - Make critical operations persist to an entire MongoDB deployment.
  - Specify replication to fewer nodes for less important operations.

Defining Write Concern

- Clients may define the write concern per write operation, if necessary.
- Standardize on specific levels of write concerns for different classes of writes.
- In the discussion that follows we will look at increasingly strict levels of write concern.

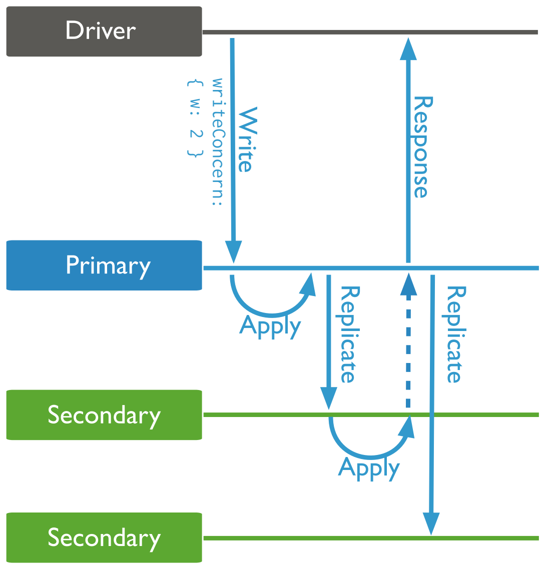
Write Concern: `{ w : 1 }`



Example: `{ w : 1 }`

```
//db.edges.insert( { from : "tom185", to : "mary_p" }, { writeConcern : { w : 1 } } )
client conn{mongocxx::uri{}};
collection coll = conn["sample"]["messages"];

document doc;
doc << "from" << "bryan" << "to" << "meghan";

write_concern wc{};
wc.nodes(1);

options::insert opts;
opts.write_concern(wc);

coll.insert_one( doc.view(), opts);
```

Write Concern: `{ w : 2 }`



Example: `{ w : 2 }`

```
client conn{mongocxx::uri{}};
collection coll = conn["sample"]["messages"];

write_concern wc{};
wc.nodes(2);
//we can set the write concern at collection level
coll.write_concern(wc);

document doc;
doc << "from" << "raul" << "to" << "john";

coll.insert_one( doc.view(), opts);
```

Other Write Concerns

- You may specify any integer as the value of the `w` field for write concern.
- This guarantees that write operations have propagated to the specified number of members.
- E.g., `{ w : 3 }`, `{ w : 4 }`, etc.

Write Concern: `{ w : "majority" }`

- Ensures the primary completed the write (in RAM).
- Ensures write operations have propagated to a majority of a replica set's **voting** members.
- Avoids hard coding assumptions about the size of your replica set into your application.
- Using majority trades off performance for durability.
- It is suitable for critical writes and to avoid rollbacks.

Example: `{ w : "majority" }`

```
client conn{mongocxx::uri{}};
write_concern wc{};
// majority of nodes with confirmation within 100 milliseconds
wc.majority(std::chrono::milliseconds(100));
conn.write_concern(wc);

collection coll = conn["sample"]["messages"];

document criteria, update;
criteria << "from" << "raul" ;
update << "$set"
<< open_document << "from" << "raul marin" << close_document;

coll.update_many( criteria.view(), update.view());
```

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” }

Further Reading

See Write Concern Reference[29] for more details on write concern configurations, including setting timeouts and identifying specific replica set members that must acknowledge writes (i.e. tag sets[30]).

[29] http://docs.mongodb.org/manual/reference/write-concern

[30] http://docs.mongodb.org/manual/tutorial/configure-replica-set-tag-sets/#replica-set-configuration-tag-sets

7.7 Read Preference

What is Read Preference?

- Read preference allows you to specify the nodes in a replica set to read from.
- Clients only read from the primary by default.
- There are some situations in which a client may want to read from:
  - Any secondary
  - A specific secondary
  - A specific type of secondary
- Only read from a secondary if you can tolerate possibly stale data, as not all writes might have replicated.

Use Cases

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

Not for Scaling

- In general, do *not* read from secondaries to provide extra capacity for reads.
- Sharding[31] increases read and write capacity by distributing operations across a group of machines.
- Sharding is a better strategy for adding capacity.

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.

[31] http://docs.mongodb.org/manual/sharding

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)
```

7.8 Lab: Setting up a Replica Set

Overview

- In this exercise we will setup a 3 data node replica set on a single machine.
- In production, each node should be run on a dedicated host:
  - To avoid any potential resource contention
  - To provide isolation against server failure

Create Data Directories

Since we will be running all nodes on a single machine, make sure each has its own data directory.

On Linux or Mac OS, run the following in the terminal to create the 3 directories `~/data/rs1`, `~/data/rs2`, and `~/data/rs3`:

```
mkdir -p ~/data/rs{1,2,3}
```

On Windows, run the following command instead in Command Prompt or PowerShell:

```
md c:\data\rs1 c:\data\rs2 c:\data\rs3
```

Launch Each Member

Now start 3 instances of mongod in the foreground so that it is easier to observe and shutdown.

On Linux or Mac OS, run *each* of the following commands in its *own terminal window*:

```
mongod --replSet myReplSet --dbpath ~/data/rs1 --port 27017 --oplogSize 200 --smallfiles
mongod --replSet myReplSet --dbpath ~/data/rs2 --port 27018 --oplogSize 200 --smallfiles
mongod --replSet myReplSet --dbpath ~/data/rs3 --port 27019 --oplogSize 200 --smallfiles
```

On Windows, run *each* of the following commands in its *own Command Prompt or PowerShell window*:

```
mongod --replSet myReplSet --dbpath c:\data\rs1 --port 27017 --oplogSize 200 --smallfiles
mongod --replSet myReplSet --dbpath c:\data\rs2 --port 27018 --oplogSize 200 --smallfiles
mongod --replSet myReplSet --dbpath c:\data\rs3 --port 27019 --oplogSize 200 --smallfiles
```

Status

- At this point, we have 3 mongod instances running.
- They were all launched with the same replSet parameter of "myReplSet".
- Despite this, the members are not aware of each other yet.
- This is fine for now.

Connect to a MongoDB Instance

- Connect to the one of the MongoDB instances with the mongo shell.
- To do so run the following command in the terminal, Command Prompt, or PowerShell:

  ```
  mongo    // connect to the default port 27017
  ```

Configure the Replica Set

```
rs.initiate()
// wait a few seconds
rs.add   ('<HOSTNAME>:27018')
rs.addArb('<HOSTNAME>:27019')

// Keep running rs.status() until there's a primary and 2 secondaries
rs.status()
```

Problems That May Occur When Initializing the Replica Set

- bindIp parameter is incorrectly set
- Replica set configuration may need to be explicitly specified to use a different hostname:

```
> conf = {
    _id: "<REPLICA-SET-NAME>",
    members: [
        { _id : 0, host : "<HOSTNAME>:27017"},
        { _id : 1, host : "<HOSTNAME>:27018"},
        { _id : 2, host : "<HOSTNAME>:27019",
          "arbiterOnly" : true},
      ]
    }
> rs.initiate(conf)
```

Write to the Primary

While still connected to the primary (port 27017) with mongo shell, insert a simple test document:

```
db.testcol.insert({ a: 1 })
db.testcol.count()

exit    // Or Ctrl-d
```

Read from a Secondary

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

```
mongo --port 27018
```

Read from the secondary

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

Review the Oplog

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

Changing Replica Set Configuration

To change the replica set configuration, first connect to the primary via mongo shell:

```
mongo --port <PRIMARY_PORT>   # e.g. 27017
```

Let's raise the priority of one of the secondaries. Assuming it is the 3rd node (e.g. on port 27019):

```
cfg = rs.conf()
cfg["members"][1]["priority"] = 10
rs.reconfig(cfg)
```

Verifying Configuration Change

You will see errors like the following, which are expected:

```
2014-10-07T17:01:34.610+0100 DBClientCursor::init call() failed
2014-10-07T17:01:34.613+0100 trying reconnect to 127.0.0.1:27017 (127.0.0.1) failed
2014-10-07T17:01:34.617+0100 reconnect 127.0.0.1:27017 (127.0.0.1) ok
reconnected to server after rs command (which is normal)
```

Verify that the replica set configuration is now as expected:

```
rs.conf()
```

The secondary will now become a primary. Check by running:

```
rs.status()
```

Further Reading

- Replica Configuration[32]
- Replica States[33]

[32] http://docs.mongodb.org/manual/reference/replica-configuration/
[33] http://docs.mongodb.org/manual/reference/replica-states/

8 Sharding

8.1 Introduction to Sharding

Learning Objectives

Upon completing this module, students should understand:

- What problems sharding solves
- When sharding is appropriate
- The importance of the shard key and how to choose a good one
- Why sharding increases the need for redundancy

Contrast with Replication

- In an earlier module, we discussed Replication.
- This should never be confused with sharding.
- Replication is about high availability and durability.
  - Taking your data and constantly copying it
  - Being ready to have another machine step in to field requests.

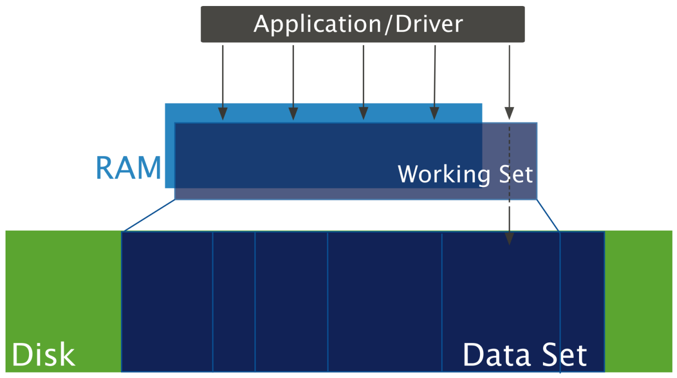
Sharding is Concerned with Scale

- What happens when a system is unable to handle the application load?
- It is time to consider scaling.
- There are 2 types of scaling we want to consider:
  - Vertical scaling
  - Horizontal scaling

Vertical Scaling

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

The Working Set



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.

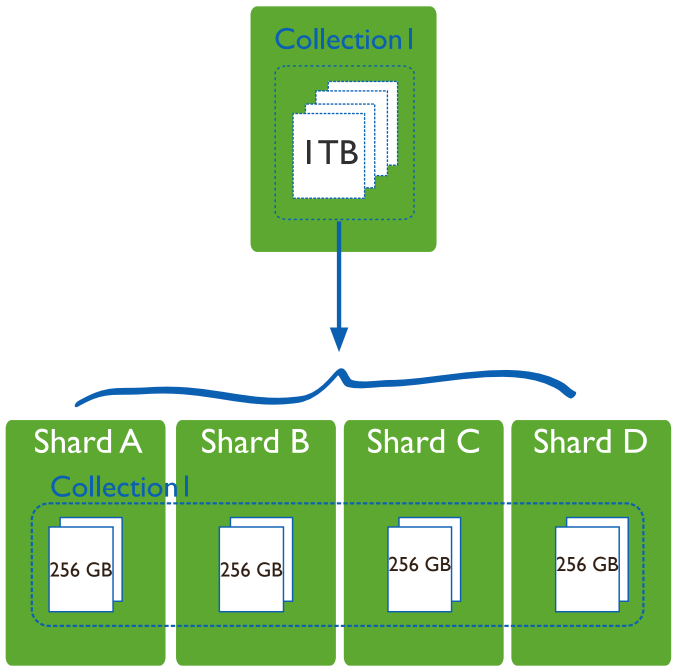
Sharding Overview

- MongoDB enables you to scale horizontally through sharding.
- Sharding is about adding more capacity to your system.
- MongoDB's sharding solution is designed to perform well on commodity hardware.
- The details of sharding are abstracted away from applications.
- Queries are performed the same way as if sending operations to a single server.
- Connections work the same by default.

When to Shard

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

Dividing Up Your Dataset



Sharding Concepts

To understanding how sharding works in MongoDB, we need to understand:

- Shard Keys
- Chunks

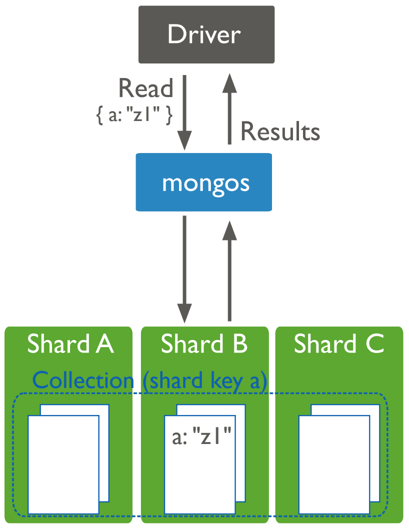
Shard Key

- You must define a shard key for a sharded collection.
- Based on one or more fields (like an index)
- Shard key defines a space of values
- Think of the key space like points on a line
- A key range is a segment of that line

Shard Key Ranges

- A collection is partitioned based on shard key ranges.
- The shard key determines where documents are located in the cluster.
- It is used to route operations to the appropriate shard.
- For reads and writes
- Once a collection is sharded, you cannot change a shard key.

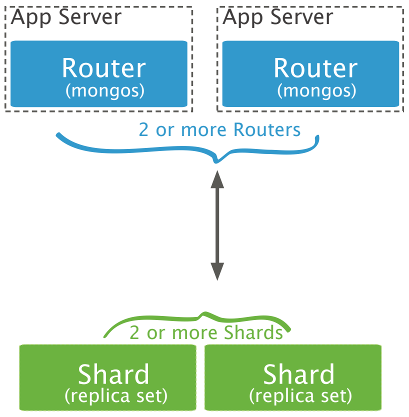
Targeted Query Using Shard Key



Chunks

- MongoDB partitions data into `chunks` based on shard key ranges.
- This is bookkeeping metadata.
- MongoDB attempts to keep the amount of data balanced across shards.
- This is achieved by migrating chunks from one shard to another as needed.
- There is nothing in a document that indicates its chunk.
- The document does not need to be updated if its assigned chunk changes.

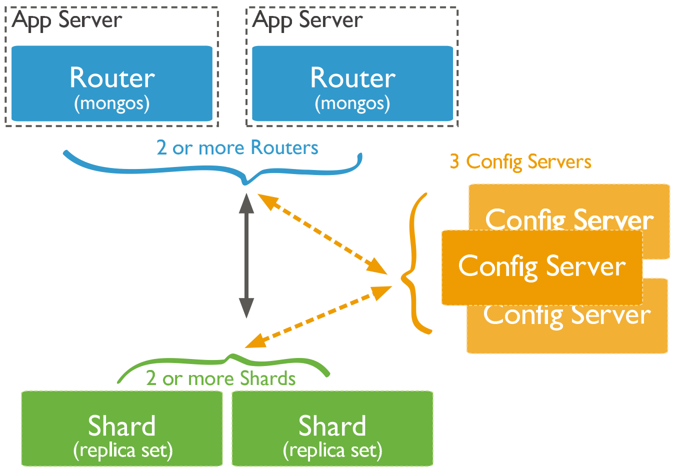
Sharded Cluster Architecture



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.

Config Servers



Config Server Hardware Requirements

- Quality network interfaces
- A small amount of disk space (typically a few GB)
- A small amount of RAM (typically a few GB)
- The larger the sharded cluster, the greater the config server hardware requirements.

Possible Imbalance?

- Depending on how you configure sharding, data can become unbalanced on your sharded cluster.
  - Some shards might receive more inserts than others.
  - Some shards might have documents that grow more than those in other shards.
- This may result in too much load on a single shard.
  - Reads and writes
  - Disk activity
- This would defeat the purpose of sharding.

Balancing Shards

- If a chunk grows too large MongoDB will split it into two chunks.
- The MongoDB balancer keeps chunks distributed across shards in equal numbers.
- However, a balanced sharded cluster depends on a good shard key.

With a Good Shard Key

You might easily see that:

- Reads hit only 1 or 2 shards per query.
- Writes are distributed across all servers.
- Your disk usage is evenly distributed across shards.
- Things stay this way as you scale.

With a Bad Shard Key

You might see that:

- Your reads hit every shard.
- Your writes are concentrated on one shard.
- Most of your data is on just a few shards.
- Adding more shards to the cluster will not help.

Choosing a Shard Key

Generally, you want a shard key:

- That has high cardinality
- That is used in the majority of read queries
- For which the values read and write operations use are randomly distributed
- For which the majority of reads are routed to a particular server

More Specifically

- Your shard key should be consistent with your query patterns.
- If reads usually find only one document, you only need good cardinality.
- If reads retrieve many documents:
  - Your shard key supports locality
  - Matching documents will reside on the same shard

Cardinality

- A good shard key will have high cardinality.
- A relatively small number of documents should have the same shard key.
- Otherwise operations become isolated to the same server.
- Because documents with the same shard key reside on the same shard.
- Adding more servers will not help.
- Hashing will not help.

Non-Monotonic

- A good shard key will generate new values non-monotonically.
- Datetimes, counters, and ObjectIds make bad shard keys.
- Monotonic shard keys cause all inserts to happen on the same shard.
- Hashing will solve this problem.
- However, doing range queries with a hashed shard key will perform a scatter-gather query across the cluster.

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.

8.2 Balancing Shards

Learning Objectives

Upon completing this module students should understand:

- Chunks and the balancer
- The status of chunks in a newly sharded collection
- How chunk splits automatically occur
- Advantages of pre-splitting chunks
- How the balancer works

Chunks and the Balancer

- Chunks are groups of documents.
- The shard key determines which chunk a document will be contained in.
- Chunks can be split when they grow too large.
- The balancer decides where chunks go.
- It handles migrations of chunks from one server to another.

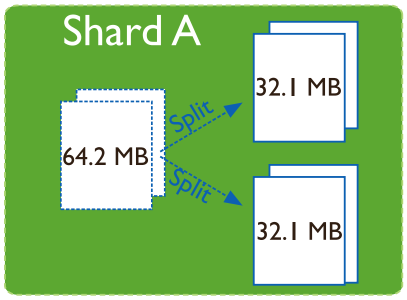
Chunks in a Newly Sharded Collection

- The range of a chunk is defined by the shard key values of the documents the chunk contains.
- When a collection is sharded it starts with just one chunk.
- The first chunk for a collection will have the range:

  ```
  { $minKey : 1 } to { $maxKey : 1 }
  ```

- All shard key values from the smallest possible to the largest fall in this chunk's range.

Chunk Splits



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

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

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.

Chunk Migration Steps

1. The balancer process sends the `moveChunk` command to the source shard.
2. The source shard continues to process reads/writes for that chunk during the migration.
3. The destination shard requests documents in the chunk and begins receiving copies.
4. After receiving all documents, the destination shard receives any changes to the chunk.
5. Then the destination shard tells the config db that it has the chunk.
6. The destination shard will now handle all reads/writes.
7. The source shard deletes its copy of the chunk.

Concluding a Balancing Round

- Each chunk will move:
  - From the shard with the most chunks
  - To the shard with the fewest
- A balancing round ends when all shards differ by at most one chunk.

8.3 Shard Tags

Learning Objectives

Upon completing this module students should understand:

- The purpose for shard tags
- Advantages of using shard tags
- Potential drawbacks of shard tags

Tags - Overview

- Shard tags allow you to “tie” data to one or more shards.
- A shard tag describes a range of shard key values.
- If a chunk is in the shard tag range, it will live on a shard with that tag.

Example: DateTime

- Documents older than one year need to be kept, but are rarely used.
- You tag those ranges as “LTS” for Long Term Storage.
- Tag specific shards to hold LTS documents.
- These shards can be on cheaper, slower machines.
- Invest in high-performance servers for more frequently accessed data.

Example: Location

- You are required to keep certain data in its home country.
- You include the country in the shard tag.
- Maintain data centers within each country that house the appropriate shards.
- Meets the country requirement but allows all servers to be part of the same system.

Example: Premium Tier

- You have customers who want to pay for a “premium” tier.
- The shard key permits you to distinguish one customer’s documents from all others.
- Tag the document ranges for each customer so that their documents will be located on shards of the appropriate tier.
- Shards tagged as premium tier run on high performance servers.
- Other shards run on commodity hardware.
- See Manage Shard Tags[34]

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

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.

8.4 Lab: Setting Up a Sharded Cluster

Learning Objectives

Upon completing this module students should understand:

- How to set up a sharded cluster including:
  - Replica Sets as Shards
  - Config Servers
  - Mongos processes
- How to enable sharding for a database
- How to shard a collection
- How to determine where data will go

Our Sharded Cluster

- In this exercise, we will set up a cluster with 3 shards.
- Each shard will be a replica set with 3 members (including one arbiter).
- We will insert some data and see where it goes.

Sharded Cluster Configuration

- Three shards:
  1. A replica set on ports 27107, 27108, 27109
  2. A replica set on ports 27117, 27118, 27119
  3. A replica set on ports 27127, 27128, 27129
- Three config servers on ports 27217, 27218, 27219
- Two mongos servers at ports 27017 and 27018

Build Our Data Directories

On Linux or MacOS, run the following in the terminal to create the data directories we'll need.

```
mkdir -p ~/data/cluster/config/{c0,c1,c2}
mkdir -p ~/data/cluster/shard0/{m0,m1,arb}
mkdir -p ~/data/cluster/shard1/{m0,m1,arb}
mkdir -p ~/data/cluster/shard2/{m0,m1,arb}
mkdir -p ~/data/cluster/{s0,s1}
```

On Windows, run the following commands instead:

```
md c:\data\cluster\config\c0 c:\data\cluster\config\c1 c:\data\cluster\config\c2
md c:\data\cluster\shard0\m0 c:\data\cluster\shard0\m1 c:\data\cluster\shard0\arb
md c:\data\cluster\shard1\m0 c:\data\cluster\shard1\m1 c:\data\cluster\shard1\arb
md c:\data\cluster\shard2\m0 c:\data\cluster\shard2\m1 c:\data\cluster\shard2\arb
md c:\data\cluster\s0 c:\data\cluster\s1
```

Initiate a Replica Set (Linux/MacOS)

```
mongod --replSet shard0 --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')"
```

Initiate a Replica Set (Windows)

```
mongod --replSet shard0 --smallfiles --nojournal --noprealloc \
       --dbpath c:\data\cluster\shard0\m0 \
       --logpath c:\data\cluster\shard0\m0\mongod.log \
       --port 27107 --oplogSize 10

mongod --replSet shard0 --smallfiles --nojournal --noprealloc \
       --dbpath c:\data\cluster\shard0\m1 \
       --logpath c:\data\cluster\shard0\m1\mongod.log \
       --port 27108 --oplogSize 10

mongod --replSet shard0 --smallfiles --nojournal --noprealloc \
       --dbpath c:\data\cluster\shard0\arb \
       --logpath c:\data\cluster\shard0\arb\mongod.log \
       --port 27109 --oplogSize 10
```

```
mongo --port 27107 --eval "\
    rs.initiate(); sleep(3000);\
    rs.add   ('<HOSTNAME>:27108');\
    rs.addArb('<HOSTNAME>:27109')"
```

Spin Up a Second Replica Set (Linux/MacOS)

```
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')"
```

Spin Up a Second Replica Set (Windows)

```
mongod --replSet shard1 --smallfiles --nojournal --noprealloc \
       --dbpath c:\data\cluster\shard1\m0 \
       --logpath c:\data\cluster\shard1\m0\mongod.log \
       --port 27117 --oplogSize 10

mongod --replSet shard1 --smallfiles --nojournal --noprealloc \
       --dbpath c:\data\cluster\shard1\m1 \
       --logpath c:\data\cluster\shard1\m1\mongod.log \
       --port 27118 --oplogSize 10

mongod --replSet shard1 --smallfiles --nojournal --noprealloc \
       --dbpath c:\data\cluster\shard1\arb \
       --logpath c:\data\cluster\shard1\arb\mongod.log \
       --port 27119 --oplogSize 10

mongo --port 27117 --eval "\
    rs.initiate(); sleep(3000);\
    rs.add   ('<HOSTNAME>:27118');\
    rs.addArb('<HOSTNAME>:27119')"
```

A Third Replica Set (Linux/MacOS)

```
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')"
```

A Third Replica Set (Windows)

```
mongod --replSet shard2 --smallfiles --nojournal --noprealloc \
       --dbpath c:\data\cluster\shard2\m0 \
       --logpath c:\data\cluster\shard2\m0\mongod.log \
       --port 27127 --oplogSize 10

mongod --replSet shard2 --smallfiles --nojournal --noprealloc \
       --dbpath c:\data\cluster\shard2\m1 \
       --logpath c:\data\cluster\shard2\m1\mongod.log \
       --port 27128 --oplogSize 10

mongod --replSet shard2 --smallfiles --nojournal --noprealloc \
       --dbpath c:\data\cluster\shard2\arb \
       --logpath c:\data\cluster\shard2\arb\mongod.log \
       --port 27129 --oplogSize 10

mongo --port 27127 --eval "\
    rs.initiate(); sleep(3000);\
    rs.add   ('<HOSTNAME>:27128');\
    rs.addArb('<HOSTNAME>:27129')"
```

Status Check

- Now we have three replica sets running.
- We have one for each shard.
- They do not know about each other yet.
- To make them a sharded cluster we will:
  - Build our config databases
  - Launch our mongos processes
  - Add each shard to the cluster
- To benefit from this configuration we also need to:
  - Enable sharding for a database
  - Shard at least one collection within that database

Launch Config Servers (Linux/MacOS)

```
mongod --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
```

Launch Config Servers (Windows)

```
mongod --smallfiles --nojournal --noprealloc \
       --dbpath c:\data\cluster\config\c0 \
       --logpath c:\data\cluster\config\c0\mongod.log \
       --port 27217 --configsvr

mongod --smallfiles --nojournal --noprealloc \
       --dbpath c:\data\cluster\config\c1 \
       --logpath c:\data\cluster\config\c1\mongod.log \
       --port 27218 --configsvr

mongod --smallfiles --nojournal --noprealloc \
       --dbpath c:\data\cluster\config\c2 \
       --logpath c:\data\cluster\config\c2\mongod.log \
       --port 27219 --configsvr
```

Launch the Mongos Processes (Linux/MacOS)

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

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

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

Launch the Mongos Processes (Windows)

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

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

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

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.

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 000000000000000000000 \
     000000000000000000000000000000000000000000000000000000000000000000000 \
     000000000000000" } )  }; db.testcol.insert(docArr) }' | mongo
```

Observe What Happens

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

```
sh.status()
```