
MongoDB Developer Training

Release 3.0

MongoDB, Inc.

June 11, 2015

Contents

1	Introduction	11
1.1	Warm Up	11
	Introductions	11
	Getting to Know You	11
	MongoDB Experience	12
	10gen	12
	Origin of MongoDB	12
1.2	MongoDB Overview	12
	Learning Objectives	12
	MongoDB is a Document Database	13
	An Example MongoDB Document	13
	Vertical Scaling	14
	Scaling with MongoDB	14
	Database Landscape	15
	MongoDB Deployment Models	15
1.3	MongoDB Stores Documents	16
	Learning Objectives	16
	JSON	16
	A Simple JSON Object	17
	JSON Keys and Values	17
	Example Field Values	17
	BSON	18
	BSON Hello World	18
	A More Complex BSON Example	18
	Documents, Collections, and Databases	19
	The <code>_id</code> Field	19
	ObjectId	19
1.4	Storage Engines	20
	Learning Objectives	20
	What is a Database Storage Engine?	20
	How Storage Engines Affect Performance	20
	Storage Engine Journaling	21
	MongoDB Storage Engines	21
	Specifying a MongoDB Storage Engine	21
	Specifying a Location to Store Data Files	21

MMAPv1 Storage Engine	22
MMAPv1 Workloads	22
Power of 2 Sizes Allocation Strategy	22
Compression in MongoDB	23
WiredTiger Storage Engine	23
WiredTiger Compression Options	23
Configuring Compression in WiredTiger	23
Configuring Memory Usage in WiredTiger	23
Journaling in MMAPv1 vs. WiredTiger	24
MMMAPv1 Journaling Mechanics	24
MMAPv1 Journaling Mechanics (Continued)	24
WiredTiger Journaling Mechanics	24
Storage Engine API	24
Conclusion	25
 1.5 Exercise: Installing MongoDB	 25
Learning Objectives	25
Production Releases	25
Installing MongoDB	26
Linux Setup	26
Install on Windows	26
Create a Data Directory on Windows	27
Launch a <code>mongod</code>	27
The MongoDB Data Directory (MMAPv1)	27
The MongoDB Data Directory (WiredTiger)	28
Import Exercise Data	28
Launch a Mongo Shell	29
Explore Databases	29
Exploring Collections	29
Admin Commands	30
 2 CRUD	 31
 2.1 Creating and Deleting Documents	 31
Learning Objectives	31
Creating New Documents	31
Example: Inserting a Document	32
Implicit <code>_id</code> Assignment	32
Example: Assigning <code>_ids</code>	32
Inserts will fail if...	32
Example: Inserts will fail if...	33
Bulk Inserts	33
Ordered Bulk Insert	34
Example: Ordered Bulk Insert	34
Unordered Bulk Insert	34
Example: Unordered Bulk Insert	34
The Shell is a JavaScript Interpreter	35
Exercise: Creating Data in the Shell	35
Deleting Documents	35
Using <code>remove()</code>	35
Example: Removing Documents	36
Dropping a Collection	36
Example: Dropping a Collection	37
Dropping a Database	37

Example: Dropping a Database	37
2.2 Reading Documents	37
Learning Objectives	37
The <code>find()</code> Method	38
Query by Example	38
Example: Querying by Example	38
Querying Arrays	39
Example: Querying Arrays	39
Querying with Dot Notation	39
Example: Querying with Dot Notation	40
Example: Arrays and Dot Notation	40
Projections	41
Projection: Example (Setup)	41
Projection: Example	41
Projection Documents	41
Example: Projections	42
Cursors	42
Example: Introducing Cursors	42
Example: Cursor Objects in the Mongo Shell	43
Cursor Methods	43
Example: Using <code>count()</code>	43
Example: Using <code>sort()</code>	44
The <code>skip()</code> Method	44
The <code>limit()</code> Method	44
The <code>distinct()</code> Method	45
Example: Using <code>distinct()</code>	45
2.3 Query Operators	45
Learning Objectives	45
Comparison Query Operators	46
Example: Comparison Operators (Setup)	46
Example: Comparison Operators	46
Logical Query Operators	47
Example: Logical Operators	47
Example: Logical Operators	47
Element Query Operators	48
Example: Element Operators	48
Array Query Operators	48
Example: Array Operators	48
Example: <code>\$elemMatch</code>	48
2.4 Lab: Finding Documents	49
Exercise: Scores < 65	49
Exercise: Exams and Quizzes	49
Exercise: Highest Quiz Score	49
Exercise: View Count > 1000	50
Exercise: Television or Videos	50
Exercise: News or Images	50
2.5 Updating Documents	50
Learning Objectives	50
The <code>update()</code> Method	51
Parameters to <code>update()</code>	51

\$set and \$unset	51
Example: \$set and \$unset (Setup)	51
Example: \$set and \$unset	52
Example: Update Array Elements by Index	52
Update Operators	53
Example: Update Operators	53
update() Defaults to one Document	53
Updating Multiple Documents	54
Example: Multi-Update	54
Array Operators	54
Example: Array Operators	55
The Positional \$ Operator	55
Example: The Positional \$ Operator	55
Upserts	56
Upsert Mechanics	56
Example: Upserts	56
save()	57
Example: save()	57
Be Careful with save()	57
findAndModify()	58
findAndModify() Example	58
2.6 Lab: Updating Documents	58
Exercise: Letter Grades	58
Exercise: 10 Extra-Credit Points	58
Exercise: Updating Array Elements	59
3 Indexes	60
3.1 Index Fundamentals	60
Learning Objectives	60
Why Indexes?	60
Types of Indexes	61
Exercise: Using explain()	61
Results of explain()	62
Results of explain() - Continued	62
explain() Verbosity Can Be Adjusted	62
explain("executionStats")	63
explain("executionStats") - Continued	63
explain("executionStats") Output	63
Other Operations	64
db.<COLLECTION>.explain()	64
Using explain() for Write Operations	64
Single-Field Indexes	65
Creating an Index	65
Listing Indexes	65
Listing Indexes	65
Indexes and Read/Write Performance	66
Index Limitations	66
Use Indexes with Care	66
Additional Index Options	67
Sparse Indexes in MongoDB	67
Defining Unique Indexes	67
Building Indexes in the Background	67

Additional Index Options	67
Sparse Indexes in MongoDB	68
Defining Unique Indexes	68
Building Indexes in the Background	68
3.2 Compound Indexes	68
Learning Objectives	68
Introduction to Compound Indexes	69
The Order of Fields Matters	69
Designing Compound Indexes	69
Example: A Simple Message Board	70
Load the Data	70
Start with a Simple Index	70
Query Adding <code>username</code>	70
Include <code>username</code> in Our Index	71
<code>totalKeysExamined > n</code>	71
A Different Compound Index	71
<code>totalKeysExamined == n</code>	71
Let Selectivity Drive Field Order	72
Adding in the Sort	72
In-Memory Sorts	72
Avoiding an In-Memory Sort	73
General Rules of Thumb	73
Covered Queries	73
Exercise: Covered Queries	74
3.3 Multikey Indexes	74
Learning Objectives	74
Introduction to Multikey Indexes	74
Example: Array of Numbers	75
Exercise: Array of Documents, Part 1	75
Exercise: Array of Documents, Part 2	76
Exercise: Array of Arrays, Part 1	76
Exercise: Array of Arrays, Part 2	76
How Multikey Indexes Work	77
Multikey Indexes and Sorting	77
Exercise: Multikey Indexes and Sorting	77
Limitations on Multikey Indexes	78
Example: Multikey Indexes on Multiple Fields	78
3.4 Hashed Indexes	78
Learning Objectives	78
What is a Hashed Index?	79
Why Hashed Indexes?	79
Limitations	79
Floating Point Numbers	79
Creating a Hashed Index	80
4 Aggregation	81
4.1 Aggregation Tutorial	81
Learning Objectives	81
Aggregation Basics	81
The Aggregation Pipeline	81

Aggregation Stages	82
The Match Stage	82
Exercise: The Match Stage	82
The Project Stage	82
Exercise: Selecting fields with \$project	83
Exercise: Renaming fields with \$project	83
Exercise: Shaping documents with \$project	83
A Twitter Dataset	84
Tweets Data Model	84
Analyzing Tweets	84
Friends and Followers	85
Exercise: Friends and Followers	85
Exercise: \$match and \$project	86
The Group Stage	86
Group using \$avg	87
Group using \$push	87
Group Aggregation Operators	87
Rank Users by Number of Tweets	87
Process	88
Exercise: Ranking Users by Number of Tweets	88
Exercise: Tweet Source	88
The Unwind Stage	89
Example: User Mentions in a Tweet	89
Using \$unwind	89
Data Processing Pipelines	90
Most Unique User Mentions	90
Same Operator (\$group), Multiple Stages	90
The Sort Stage	91
The Skip Stage	91
The Limit Stage	91
The Out Stage	91
 4.2 Optimizing Aggregation	 92
Learning Objectives	92
Aggregation Options	92
Aggregation Limits	92
Limits Prior to MongoDB 2.6	93
Optimization: Reducing Documents in the Pipeline	93
Optimization: Sorting	93
Automatic Optimizations	94
 4.3 Lab: Aggregating Zip Code Data	 94
Exercise: Average Population	94
Exercise: Cities Starting with a Digit	95
 5 Schema Design	 96
5.1 Schema Design Core Concepts	96
Learning Objectives	96
What is a schema?	96
Example: Normalized Data Model	96
Example: Denormalized Version	97
Schema Design in MongoDB	97
Three Considerations	97

Case Study	97
Author Schema	97
User Schema	98
Book Schema	98
Example Documents: Author	98
Example Documents: User	99
Example Documents: Book	99
Embedded Documents	99
Example: Embedded Documents	99
Embedded Documents: Pros and Cons	100
Linked Documents	100
Example: Linked Documents	100
Linked Documents: Pros and Cons	100
Arrays	100
Array of Scalars	101
Array of Documents	101
Exercise: Users and Book Reviews	101
Solution A: Users and Book Reviews	101
Solution B: Users and Book Reviews	102
Solution C: Users and Book Reviews	102
Store Binary Files in MongoDB with GridFS	102
How GridFS Works	103
Schema Design Use Cases with GridFS	103
5.2 Schema Evolution	103
Learning Objectives	103
Development Phase	103
Development Phase: Known Query Patterns	104
Production Phase	104
Production Phase: Read Patterns	104
Addressing List Books by Last Name	104
Production Phase: Write Patterns	105
Exercise: Recent Reviews	105
Solution: Recent Reviews, Schema	105
Solution: Recent Reviews, Update	106
Solution: Recent Reviews, Read	106
Solution: Recent Reviews, Delete	106
5.3 Common Schema Design Patterns	107
Learning Objectives	107
One-to-One Relationship	107
One-to-One: Linking	107
One-to-One: Embedding	107
One-to-Many Relationship	108
One-to-Many: Array of IDs	108
One-to-Many: Single Field with ID	108
One-to-Many: Array of Documents	108
Many-to-Many Relationship	109
Many-to-Many: Array of IDs on Both Sides	109
Many-to-Many: Array of IDs on Both Sides	109
Many-to-Many: Array of IDs on One Side	109
Many-to-Many: Array of IDs on One Side	110
Tree Structures	110
Allow users to browse by subject	110

Alternative: Parents and Ancestors	110
Find Sub-Categories	111
Summary	111
5.4 Lab: Data Model for an E-Commerce Site	111
Introduction	111
Product Catalog	111
Product Metrics	112
Deliverables	112
6 Replica Sets	113
6.1 Introduction to Replica Sets	113
Learning Objectives	113
Use Cases for Replication	113
High Availability (HA)	113
Disaster Recovery (DR)	114
Functional Segregation	114
Large Replica Sets	114
Replication is Not Designed for Scaling	115
Replica Sets	115
Primary Server	116
Secondaries	116
Heartbeats	116
The Oplog	117
6.2 Write Concern	117
Learning Objectives	117
What happens to the write?	117
Answer	117
Balancing Durability with Performance	118
Defining Write Concern	118
Write Concern: { w : 1 }	118
Example: { w : 1 }	119
Write Concern: { w : 2 }	119
Example: { w : 2 }	120
Other Write Concerns	120
Write Concern: { w : "majority" }	120
Example: { w : "majority" }	120
Quiz: Which write concern?	120
Further Reading	121
6.3 Read Preference	122
What is Read Preference?	122
Use Cases	122
Not for Scaling	122
Read Preference Modes	123
Tag Sets	123
7 Sharding	124
7.1 Introduction to Sharding	124
Learning Objectives	124
Contrast with Replication	124

Sharding is Concerned with Scale	124
Vertical Scaling	125
The Working Set	125
Limitations of Vertical Scaling	125
Sharding Overview	126
A Model that Does Not Scale	126
A Scalable Model	126
Sharding Basics	127
Sharded Cluster Architecture	127
Mongos	128
Config Servers	128
Config Server Hardware Requirements	129
When to Shard	129
Possible Imbalance?	129
Balancing Shards	130
What is a Shard Key?	130
Targeted Query Using Shard Key	130
With a Good Shard Key	131
With a Bad Shard Key	131
Choosing a Shard Key	131
More Specifically	131
Cardinality	132
Non-Monotonic	132
Shards Should be Replica Sets	132
8 MMS & Ops Manager	133
8.1 MongoDB Management Service (MMS) & Ops Manager	133
Learning Objectives	133
MMS and Ops Manager	133
Deployment Options	133
Architecture	133
MMS	133
Ops Manager	134
MMS and Ops Manager Use Cases	134
Creating an MMS Account	135
8.2 Automation	135
Learning Objectives	135
What is Automation?	135
How Does Automation Work?	135
Automation Agents	135
Sample Use Case	135
Upgrades Using Automation	136
Automation: Behind the Scenes	136
Configuration File	137
Automation Goal State	137
Demo	137
8.3 Exercise: Cluster Automation	138
Learning Objectives	138
MMS Automation Support	138
Exercise #1	138
Exercise #2	138

9	Supplementary Material (Time Permitting)	139
9.1	Geospatial Indexes	139
	Learning Objectives	139
	Introduction to Geospatial Indexes	139
	Easiest to Start with 2 Dimensions	139
	Location Field	140
	Find Nearby Documents	140
	Flat vs. Spherical Indexes	140
	Flat Geospatial Index	140
	Spherical Geospatial Index	141
	Creating a 2d Index	141
	Exercise: Creating a 2d Index	141
	Inserting Documents with a 2d Index	141
	Exercise: Inserting Documents with 2d Fields	142
	Querying Documents Using a 2d Index	142
	Example: Find Based on 2d Coords	142
	Creating a 2dsphere Index	143
	The geoJSON Specification	143
	geoJSON Considerations	143
	Simple Types of 2dsphere Objects	143
	Polygons	144
	Other Types of 2dsphere Objects	144
	Exercise: Inserting geoJSON Objects (1)	144
	Exercise: Inserting geoJSON Objects (2)	145
	Exercise: Inserting geoJSON Objects (3)	145
	Exercise: Creating a 2dsphere Index	145
	Querying 2dsphere Objects	146
9.2	TTL Indexes	146
	Learning Objectives	146
	TTL Index Basics	146
	Creating a TTL Index	147
	Exercise: Creating a TTL Index	147
	Exercise: Check the Collection	147
9.3	Text Indexes	147
	Learning Objectives	147
	What is a Text Index?	148
	Creating a Text Index	148
	Exercise: Creating a Text Index	148
	Creating a Text Index with Weighted Fields	148
	Creating a Text Index with Weighted Fields	148
	Text Indexes are Similar to Multikey Indexes	149
	Exercise: Inserting Texts	149
	Querying a Text Index	149
	Exercise: Querying a Text Index	149
	Exercise: Querying Using Two Words	149
	Search for a Phrase	150
	Text Search Score	150

1 Introduction

Warm Up (page 11) Activities to get the class started

MongoDB Overview (page 12) MongoDB philosophy and features.

MongoDB Stores Documents (page 16) The structure of data in MongoDB.

Storage Engines (page 20) MongoDB storage engines.

Exercise: Installing MongoDB (page 25) Install mongodb experiment with a few operations.

1.1 Warm Up

Introductions

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

Note:

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

Getting to Know You

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

Note:

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

MongoDB Experience

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

10gen

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

Origin of MongoDB

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

1.2 MongoDB Overview

Learning Objectives

Upon completing this module students should understand:

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

MongoDB is a Document Database

Documents are associative arrays like:

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

An Example MongoDB Document

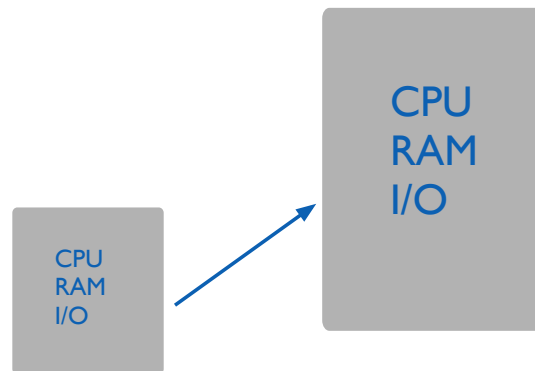
A MongoDB document expressed using JSON syntax.

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

Note:

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

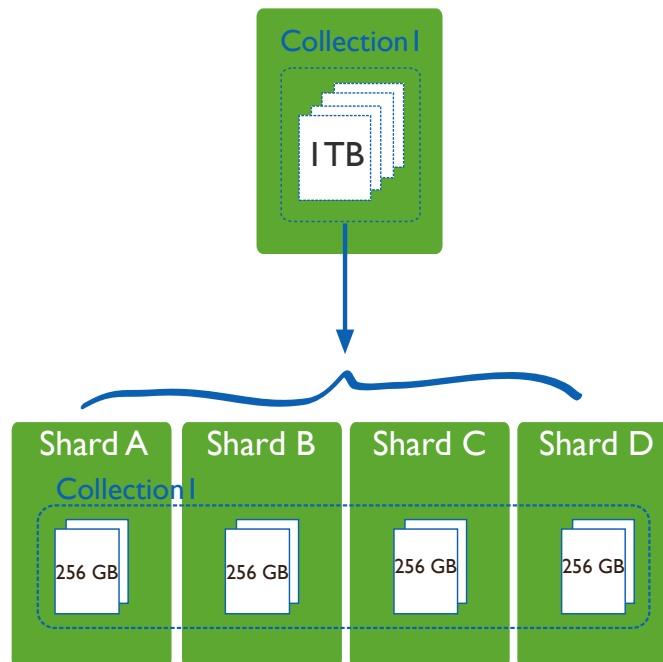
Vertical Scaling



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

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

Scaling with MongoDB

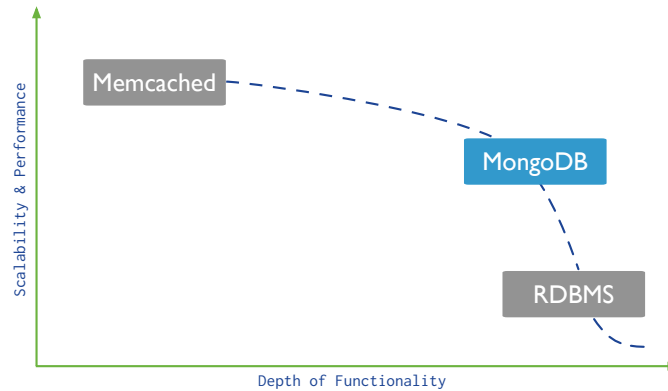


Note:

- MongoDB is designed to be horizontally scalable (linear).
 - MongoDB scales by enabling you to shard your data.
-

- When you need more performance, you just buy another machine and add it to your cluster.
 - MongoDB is highly performant on commodity hardware.
-

Database Landscape



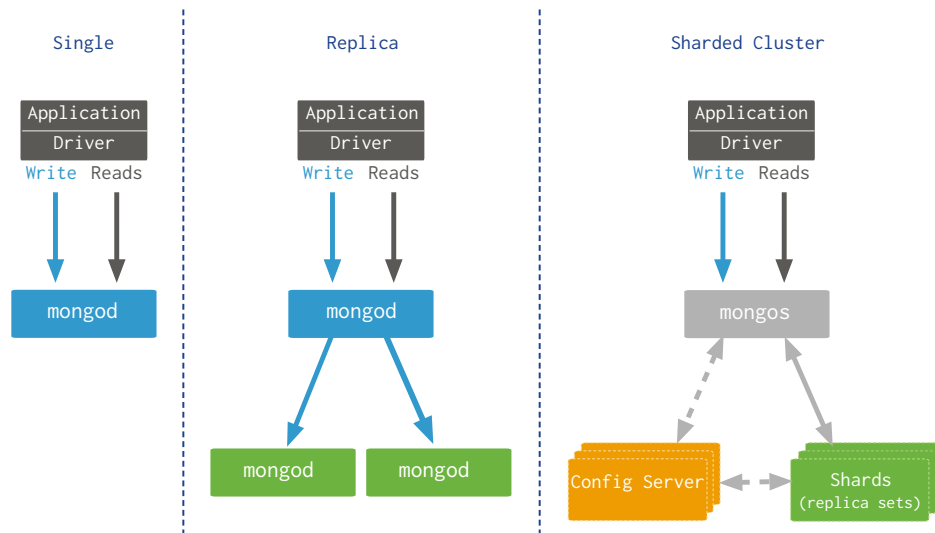
Note:

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

MongoDB Deployment Models

Note:

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



- Your data is distributed across several shards.
 - The shards are themselves replica sets.
 - This provides high availability and redundancy at scale.
-

1.3 MongoDB Stores Documents

Learning Objectives

Upon completing this module, students should understand:

- JSON
- BSON basics
- That documents are organized into collections
- ObjectIds
- 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.org¹.

Example Field Values

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

¹<http://json.org/>

BSON

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

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

BSON Hello World

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

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

Note:

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

A More Complex BSON Example

```
// 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
 \x14\x40\x102\x00\xc2\x07\x00\x00
 \x00\x00"
```

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

Documents, Collections, and Databases

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

The `_id` Field

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

ObjectIds



Note:

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

1.4 Storage Engines

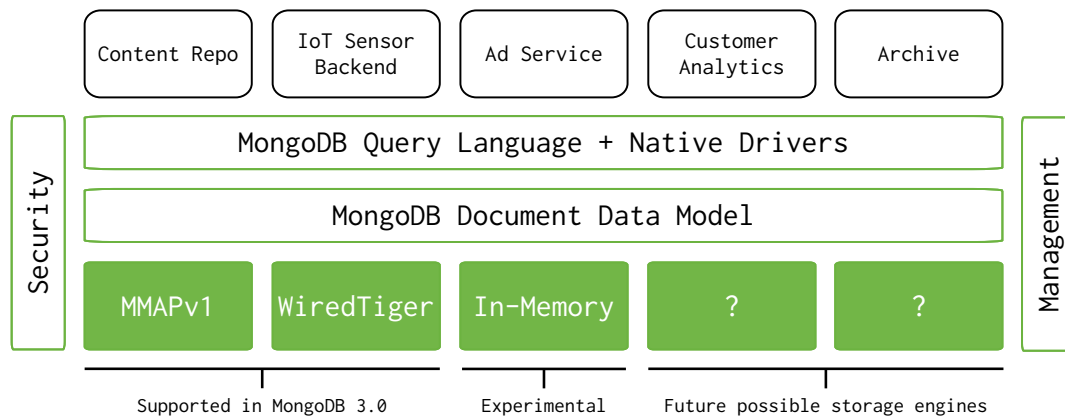
Learning Objectives

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?

A database storage engine is the underlying software component that a database management system uses to create, read, update, and delete data from a database.



Note:

- Talk through the diagram and how storage engines are used to abstract access to the data
-

How Storage Engines Affect Performance

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

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

Storage Engine Journaling

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

MongoDB Storage Engines

With the release of MongoDB 3.0, 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
```

Note:

- mmapv1 is used if `storageEngine` parameter isn't specified
-

Specifying a Location to Store Data Files

- Use the `dbpath` parameter

```
mongod --dbpath /data/db
```
- Other files are also stored here. E.g.,
 - `mongod.lock` file
 - `journal`
- See the MongoDB docs for a complete list of [storage options](http://docs.mongodb.org/manual/reference/program/mongod/#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

Note:

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

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

Note:

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

Journaling in MMAPv1 vs. WiredTiger

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

MMAPv1 Journaling Mechanics

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

MMAPv1 Journaling Mechanics (Continued)

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

WiredTiger Journaling Mechanics

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

Storage Engine API

MongoDB 3.0 introduced a storage engine API:

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

Conclusion

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

Note:

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

1.5 Exercise: Installing MongoDB

Learning Objectives

Upon completing this exercise students should understand:

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

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

Linux Setup

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

```
sudo mkdir -p /data/db
```

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

Note:

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

Install on Windows

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

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

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

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

Create a Data Directory on Windows

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

```
md \data\db
```

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

Launch a mongod

Explore the `mongod` command.

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

Launch a `mongod` with the MMAPv1 storage engine:

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

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
```

Note:

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

The MongoDB Data Directory (MMAPv1)

```
ls /data/db
```

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

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

- `sample.0` is 64 MB
 - `sample.1` is 128 MB
-

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

The MongoDB Data Directory (WiredTiger)

```
ls /data/db
```

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

Import Exercise Data

```
cd usb_drive
unzip sampledata.zip
cd sampledata
mongoimport -d sample -c tweets twitter.json
mongoimport -d sample -c zips zips.json
cd dump
mongorestore -d sample training
mongorestore -d sample digg
```

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

Note:

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

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
```

Note:

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

Exploring Collections

```
show collections
```

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

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

Note:

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

Admin Commands

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

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

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

2 CRUD

Creating and Deleting Documents (page 31) Inserting documents into collections, deleting documents, and dropping collections.

Reading Documents (page 37) The `find()` command, query documents, dot notation, and cursors.

Query Operators (page 45) MongoDB query operators including: comparison, logical, element, and array operators.

Lab: Finding Documents (page 49) Exercises for querying documents in MongoDB.

Updating Documents (page 50) Using `update()` and associated operators to mutate existing documents.

Lab: Updating Documents (page 58) Exercises for updating documents in MongoDB.

2.1 Creating and Deleting Documents

Learning Objectives

Upon completing this module students should understand:

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

Creating New Documents

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

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

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

Example: Inserting a Document

Experiment with the following commands.

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

Note:

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

Implicit `_id` Assignment

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

Example: Assigning `_ids`

Experiment with the following commands.

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

Note:

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

Inserts will fail if...

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

Example: Inserts will fail if...

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

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

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

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

Note:

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

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

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

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

Bulk Inserts

- MongoDB 2.6 introduced bulk inserts.
- You may bulk insert using an array of documents.
- The API has two core concepts:
 - Ordered bulk operations
 - Unordered bulk operations
- The main difference is in the way the operations are executed in bulk.

Note:

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

Ordered Bulk Insert

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

Example: Ordered Bulk Insert

Experiment with the following bulk insert.

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

db.movies.find()
```

Note:

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

Unordered Bulk Insert

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

Example: Unordered Bulk Insert

Experiment with the following bulk insert.

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

db.movies.find()
```

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

Using `remove()`

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

Example: Removing Documents

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

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

db.testcol.remove( { a : 1 } ) // Remove the first document

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

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

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

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

Dropping a Collection

- You can drop an entire collection with `db.<COLLECTION>.drop()`
- The collection and all documents will be deleted.
- It will also remove any metadata associated with that collection.
- Indexes are one type of metadata removed.
- More on meta data later.

Note: Mention that `drop()` is more performant than `remove` because of the lookup costs associated with `remove()`.

Example: Dropping a Collection

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

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

Dropping a Database

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

Example: Dropping a Database

```
use tempDB
db.testcoll1.insert( { a : 1 } )
db.testcoll2.insert( { a : 1 } )

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

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

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

2.2 Reading Documents

Learning Objectives

Upon completing this module students should understand:

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

The `find()` Method

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

Query by Example

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

Example: Querying by Example

Experiment with the following sequence of commands.

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

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

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

Note: Matching Rules:

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

Querying Arrays

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

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

Example: Querying Arrays

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

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

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

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

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

Querying with Dot Notation

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

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

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

Example: Querying with Dot Notation

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

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

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

Example: Arrays and Dot Notation

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

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

Note:

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

Projections

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

Projection: Example (Setup)

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

Projection: Example

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

Projection Documents

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

Example: Projections

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

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

Cursors

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

Example: Introducing Cursors

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

it
it
```

Note:

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

Example: Cursor Objects in the Mongo Shell

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

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

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

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

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

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

Cursor Methods

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

Example: Using `count()`

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

Note:

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

Example: Using sort ()

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

Note:

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

The skip () Method

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

The limit () Method

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

The `distinct()` Method

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

Example: Using `distinct()`

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

.. only:: instructor

.. note::

    Returns

.. code-block:: javascript

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

2.3 Query Operators

Learning Objectives

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

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

Comparison Query Operators

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

Example: 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()

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

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

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

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

Logical Query Operators

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

Example: Logical Operators

```
db.movies.find( { $or : [
  { "category" : "sci-fi" }, { "imdb_rating" : { $gte : 7 } }
] } )

// more complex $or, really good sci-fi movie or medicore family movie
db.movies.find( { $or : [
  { "category" : "sci-fi", "imdb_rating" : { $gte : 8 } },
  { "category" : "family", "imdb_rating" : { $gte : 7 } }
] } )

// find bad movies
db.movies.find( { "imdb_rating" : { $not : { $gt : 7 } } } )
```

Note:

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

Example: Logical Operators

```
// find movies within an imdb_rating range
db.movies.find( { "imdb_rating" : { $gt : 5 , $lte : 7 } } ) // and is implicit

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

Element Query Operators

- `$exists`: Select documents based on the existence of a particular field.
- `$type`: Select documents based on their type.
- See [BSON types](#)⁴ for reference on types.

Example: Element Operators

```
db.movies.find( { "budget" : { $exists : true } } )
```

```
// type 1 is Double  
db.movies.find( { "budget" : { $type : 1 } } )
```

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

Array Query Operators

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

Example: Array Operators

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

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

Example: \$elemMatch

```
db.movies.insert( {  
  "title" : "Raiders of the Lost Ark",  
  "filming_locations" : [  
    { "city" : "Los Angeles", "state" : "CA", "country" : "USA" },  
    { "city" : "Rome", "state" : "Lazio", "country" : "Italy" },  
    { "city" : "Florence", "state" : "SC", "country" : "USA" }  
  ] } )
```

```
// This query is incorrect, it won't return what we want
```

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

```
// $elemMatch is needed, now there are no results, this is expected
```

```
db.movies.find( {  
  "filming_locations" : {  
    $elemMatch : {  
      "city" : "Florence",
```

⁴<http://docs.mongodb.org/manual/reference/bson-types>


```
"country" : "Italy"
} } } )
```

Note:

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

2.4 Lab: Finding Documents

Exercise: Scores < 65

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

Note:

- 832
 - `db.scores.find({ score: { $lt: 65 } })`
-

Exercise: Exams and Quizzes

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

Note:

- 2000
 - `db.scores.find({ type: { $in: [“exam”, “quiz”] } }).count()`
-

Exercise: Highest Quiz Score

Find the highest quiz score.

Note:

- You can `.limit()` with `.sort()`
 - `db.scores.find({ type: “quiz” }).sort({ score: -1 }).limit(1)[0].score`
-

Exercise: View Count > 1000

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

Note:

- Requires querying into subdocuments
 - `db.stories.find({ "shorturl.view_count": { $gt: 1000 } })`
-

Exercise: Television or Videos

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

Note:

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

Exercise: News or Images

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

Note:

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

2.5 Updating Documents

Learning Objectives

Upon completing this module students should understand

- The `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()`.
 - By default only the first document found is updated.
- The update parameter:
 - Take care to simply modify documents if that is what you intend.
 - Replacing documents in their entirety is easy to do by mistake.

\$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

```
db.movies.update( { "title" : "Batman" }, { $set : { "imdb_rating" : 7.7 } } )  
  
db.movies.update( { "title" : "Godzilla" }, { $set : { "budget" : 1 } } )  
  
db.movies.update( { "title" : "Home Alone" },  
                  { $set : { "budget" : 15, "imdb_rating" : 5.5 } } )  
  
// how will this query behave?  
db.movies.update( { "title" : "Batman" }, { "imdb_rating" : 7.7 } )  
  
db.movies.update( { "title" : "Home Alone" }, { $unset : { "budget" : 1 } } )
```

Note:

- Update will only update the first document it finds by default
 - Difference between using \$set and not using \$set is very important
-

Example: Update Array Elements by Index

```
// add a sample document to track mentions per hour  
db.movie_mentions.insert( { "title" : "E.T.",  
                             "day" : ISODate("2015-03-27T00:00:00.000Z"),  
                             "mentions_per_hour" : [ 0, 0, 0, 0, 0, 0, 0, 0,  
                                                       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
                                                       0, 0 ]  
                           } )  
  
// update all mentions for the fifth hour of the day  
db.movie_mentions.update( { "title" : "E.T." },  
                          { $set : { "mentions_per_hour.5" : 2300 } } )
```

Note:

- Cool pattern for time series data
 - Displaying charts is now trivial, can change granularity to by the minute, hour, day, etc.
-

Update Operators

- `$inc`: Increment a field's value by the specified amount.
- `$mul`: Multiply a field's value by the specified amount.
- `$rename`: Rename a field.
- `$set` (already discussed)
- `$unset` (already discussed)
- `$min`: Update only if value is smaller than specified quantity
- `$max`: Update only if value is larger than specified quantity
- `$currentDate`: Set the value of a field to the current date or timestamp.

Example: Update Operators

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

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

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

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

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

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

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

`update()` Defaults to one Document

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

Updating Multiple Documents

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

Example: Multi-Update

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

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

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

Note:

- `db.movies.update({ }, { $set : { "sequels" : 0 } })` only updates one document.
 - `db.movies.update({ }, { $set : { "sequels" : 0 } }, { multi : true })` updates four documents.
-

Array Operators

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

Note:

- These operators may be applied to array fields.
-

Example: Array Operators

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

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

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

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

Note:

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

The Positional \$ Operator

- \$⁵ is a positional operator that specifies an element in an array to update.
- It acts as a placeholder for the first element that matches the query document.
- \$ replaces the element in the specified position with the value given.
- Example:

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

Example: The Positional \$ Operator

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

⁵<http://docs.mongodb.org/manual/reference/operator/update/positional>

Upserts

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

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

Upsert Mechanics

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

Example: Upserts

```
db.movies.update( { "title" : "Jaws" },  
                  { $inc: { "budget" : 5 } },  
                  { upsert: true } )  
  
db.movies.update( { "title" : "Jaws II" },  
                  { $inc: { "budget" : 5 } },  
                  { upsert: true } )  
  
db.movies.update( { "title" : "Jaws III", "category" : [ "horror" ] },  
                  { $set : { "budget" : 1 } },  
                  { upsert: true } )
```

Note:

```
// updates the document with "title" = "Jaws" by incrementing "budget"  
db.movies.update( { "title" : "Jaws" }, { $inc: { "budget" : 5 } }, { upsert: true } )  
  
// 1) creates a new document, 2) assigns an _id, 3) sets "title" to "Jaws II"  
// 4) performs the update  
db.movies.update( { "title" : "Jaws II" }, { $inc: { "budget" : 5 } }, { upsert: true } )  
  
// 1) creates a new document, 2) sets "title" : "Jaws III",  
// 3) Set budget to 1  
db.movies.update( { "title" : "Jaws III" }, { "budget" : 1 }, { upsert: true } )
```

save()

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

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

Example: save()

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

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

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

Note:

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

Be Careful with save()

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

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

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

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

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

doc.imdb_rating = 7.4
doc

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

findAndModify()

Modify a document and return either:

- The pre-modification document
- If “new:true” is set, the modified document

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

findAndModify() Example

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

2.6 Lab: Updating Documents

Exercise: Letter Grades

- Using the `sample.scores` namespace, set the proper “grade” attributes.
- For example, users with scores greater than 90 get an “A”.
- Set the grade to “B” for scores falling between 80 and 90 and so on for grades “C”, “D”, and “F”.

Note:

- `db.scores.update({ score: { $gte: 90 } }, { $set: { grade: "A" } }, { multi: true })`
 - and so on for `{ score: { $gte: 80, $lt: 90 } }` etc.
 - In the shell you could write a JS script with a loop. In SQL you could do it in one statement, note the difference.
-

Exercise: 10 Extra-Credit Points

- You’re being nice, so you decide to add 10 points to every score on every exam where the score is lower than 60.
- How do you do this update?

Note:

```
db.scores.update( { name : "exam", score : { $lt : 60 } },
  { $inc : { score : 10 } }, { multi: true } )
```

Exercise: Updating Array Elements

Insert a document representing product metrics for a backpack:

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

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

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

Note:

- Talk about how this can be used for time series data, real-time graphs/charts
 - `db.product_metrics.update({ "name" : "backpack" }, { $inc : { "purchases_past_7_days.4" : 200 } })`
-

3 Indexes

Index Fundamentals (page 60) An introduction to MongoDB indexes.

Compound Indexes (page 68) Indexes on two or more fields.

Multikey Indexes (page 74) Indexes on array fields.

Hashed Indexes (page 78) Hashed Indexes.

3.1 Index Fundamentals

Learning Objectives

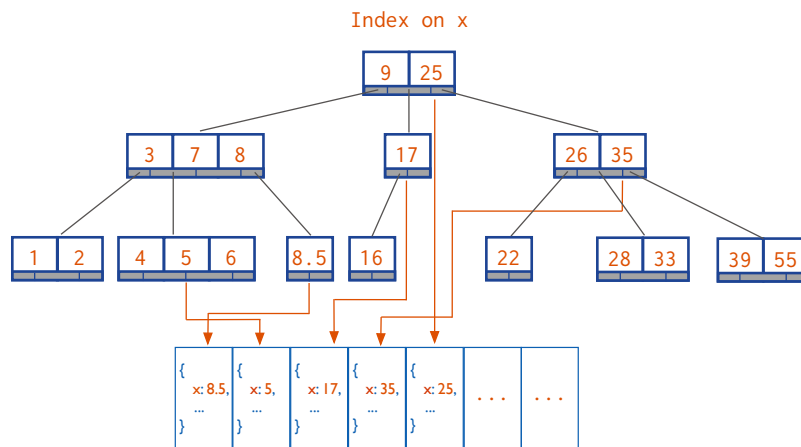
Upon completing this module students should understand:

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

Note:

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

Why Indexes?



Note:

- Without an index, in order to find all documents matching a query, MongoDB must scan every document in the collection.
 - This is murder for read performance, and often write performance, too.
 - If all your documents do not fit into memory, the system will page data in and out in order to scan the entire collection.
-

- An index enables MongoDB to locate exactly which documents match the query and where they are located on disk.
 - MongoDB indexes are based on B-trees.
-

Types of Indexes

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

Note:

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

Exercise: Using `explain()`

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

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

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

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

Note:

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

Results of `explain()`

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

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

Results of `explain()` - Continued

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

`explain()` Verbosity Can Be Adjusted

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

Note:

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

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

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

Now we have query statistics:

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

explain("executionStats") - Continued

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

explain("executionStats") Output

- nReturned 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()
```

Note:

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

Using `explain()` for Write Operations

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

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

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

Note:

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

Single-Field Indexes

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

Creating an Index

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

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

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

Note:

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

Listing Indexes

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

List index keys:

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

Listing Indexes

List indexes for a collection:

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

List index keys:

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

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.

Note:

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

Additional Index Options

- Sparse
- Unique
- Background

Sparse Indexes in MongoDB

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

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

Defining Unique Indexes

- Enforce a unique constraint on the index.
- Prevent duplicate values from being inserted into the database.
- No duplicate values may exist prior to defining the index.

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

Building Indexes in the Background

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

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

Additional Index Options

- Sparse
- Unique
- Background

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 } )
```

3.2 Compound Indexes

Learning Objectives

Upon completing this module students should understand:

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

Introduction to Compound Indexes

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

The Order of Fields Matters

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

- Equality tests, e.g.,

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

- Range queries, e.g.,

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

- Sorting, e.g.,

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

Note:

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

Designing Compound Indexes

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

Example: A Simple Message Board

Requirements:

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

Load the Data

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

Note:

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

A Different Compound Index

Drop the index and build a new one with user.

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

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

totalKeysExamined is 2. n is 2.

totalKeysExamined == n

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

Note:

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

Let Selectivity Drive Field Order

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

Adding in the Sort

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

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

- 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.

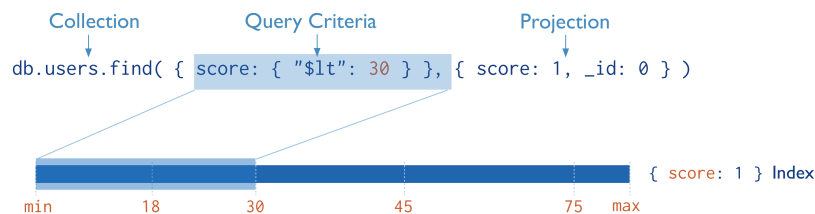
```
db.messages.dropIndex( "myindex" );
db.messages.createIndex( { username : 1, rating : 1, timestamp : 1 },
                        { name : "myindex" } );
db.messages.find( {
    timestamp : { $gte : 2, $lte : 4 },
    username : "anonymous"
} ).sort( { rating : -1 } ).explain();
```

- We no longer have an in-memory sort, but need to examine more keys.
- `totalKeysExamined` is 3 and `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, "budget" : 1 }
                ).explain("executionStats")
```

3.3 Multikey Indexes

Learning Objectives

Upon completing this module, students should understand:

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

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 } )
```

Note:

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

Exercise: Array of Documents, Part 2

For each of the three queries below:

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

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

Note:

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

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

Exercise: Array of Arrays, Part 1

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

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

Note:

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

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

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

How Multikey Indexes Work

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

Multikey Indexes and Sorting

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

Exercise: Multikey Indexes and Sorting

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

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

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

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

Note:

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

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

Limitations on Multikey Indexes

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

Example: Multikey Indexes on Multiple Fields

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

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

3.4 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](#)⁶ for more details.
- We discuss sharding in detail in another module.

Limitations

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

Note:

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

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} .

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

Creating a Hashed Index

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

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


4 Aggregation

Aggregation Tutorial (page 81) An introduction to the the aggregation framework, pipeline concept, and stages.

Optimizing Aggregation (page 92) Resource management in the aggregation pipeline.

Lab: Aggregating Zip Code Data (page 94) Aggregate zip code data.

4.1 Aggregation Tutorial

Learning Objectives

Upon completing this module students should understand:

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

Aggregation Basics

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

The Aggregation Pipeline

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

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

Aggregation Stages

- `$match`: Similar to `find()`
- `$project`: Shape documents
- `$sort`: Like the cursor method of the same name
- `$skip`: Like the cursor method of the same name
- `$limit`: Like the cursor method of the same name
- `$unwind`: Used for working with arrays
- `$group`: Used to aggregate field values from multiple documents
- `$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.

Exercise: The Match Stage

Select only the first two documents using a match stage in an aggregation pipeline.

```
a = [ { _id : 1, a : 1 }, { _id : 2, a : 2 }, { _id : 3, a : 3 },  
      { _id : 4, a : 4 }, { _id : 5, a : 5 } ]  
db.testcol.insert( a )  
  
// 2 docs are output from the aggregation pipeline  
db.testcol.aggregate( [ { $match : { a : { $lte : 2 } } } ] )
```

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.

Exercise: Selecting fields with \$project

Use the \$project operator to pass specific fields in output documents.

```
db.testcol.drop()
for ( var i=1; i<=10; i++ ) {
  db.testcol.insert( { a : i, b : i*2, c : { d : i*4, e : i*8 } } ) }
db.testcol.find()

db.testcol.aggregate( [ { $project : { a : 1 } } ] )

db.testcol.aggregate( [ { $project : { _id : 0, a : 1 } } ] )

db.testcol.aggregate( [ { $project : { a : 1, "c.d": 1 } } ] )
```

Note:

- { \$project : { a : 1 } } : _id is projected implicitly
 - { \$project : { _id : 0, a : 1 } } : suppress projection of _id
 - { \$project : { a : 1, "c.d": 1 } } : use dot notation to specify fields in embedded documents.
-

Exercise: Renaming fields with \$project

Use the \$project operator to rename a field

```
db.testcol.aggregate( [ { $project : { _id : 0,
                                     sequenceNumber : "$a",
                                     b : 1 } } ] )
```

Exercise: Shaping documents with \$project

Experiment with the following projections.

```
db.testcol.aggregate( [ { $project : { a : 1, b : 1, d : "$c.d" } } ] )

db.testcol.aggregate( [ { $project : { sequenceNumber : "$a",
                                     ratio : { $divide : [ "$c.d", "$c.e" ] } } } ] )
```

More about [\\$divide](http://docs.mongodb.org/manual/reference/operator/aggregation/divide/)⁷ in another lesson.

⁷<http://docs.mongodb.org/manual/reference/operator/aggregation/divide/>

A Twitter Dataset

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

Tweets Data Model

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

Analyzing Tweets

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

Note:

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

Friends and Followers

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

Exercise: Friends and Followers

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

Note:

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

Exercise: \$match and \$project

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

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

Note:

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

The Group Stage

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

Group using \$avg

```
db.testcol.aggregate( [ { $group : { _id : { a : "$a" },  
                                     b_avg : { $avg : "$b" } } } ] )
```

Group using \$push

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" : 5 }  
] )
```

Group Aggregation Operators

The complete list of operators available in the group stage:

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

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.
- We will use the aggregation framework to do this.

Process

- Group together all tweets by a user for every user in our collection
- Count the tweets for each user
- Sort in decreasing order

Exercise: Ranking Users by Number of Tweets

Try this aggregation pipeline for yourself.

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

Note:

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

Exercise: 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.
 - Write an aggregation pipeline that identifies the applications most frequently used to publish tweets.
-

Note:

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


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 as within an embedded document for entities.
- This embedded document also lists any urls and hashtags used in the tweet.

Example: User Mentions in a Tweet

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

Using \$unwind

Who includes the most user mentions in their tweets?

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

Note:

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

Data Processing Pipelines

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

Most Unique User Mentions

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

Same Operator (\$group), Multiple Stages

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

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

Note:

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

The Sort Stage

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

The Skip Stage

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

The Limit Stage

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

The 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" }`

4.2 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 Optimization](#)⁸

4.3 Lab: Aggregating Zip Code Data

For the following exercises, import the `zips.json` file from the USB stick:

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

Exercise: Average Population

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

Please note:

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

Note:

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

⁸<http://docs.mongodb.org/manual/core/aggregation-pipeline-optimization/>

Exercise: Cities Starting with a Digit

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

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

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

Note:

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

5 Schema Design

Schema Design Core Concepts (page 96) An introduction to schema design in MongoDB.

Schema Evolution (page 103) Considerations for evolving a MongoDB schema design over an application's lifetime.

Common Schema Design Patterns (page 107) Common design patterns for representing 1-1, 1-M, and M-M relationships and tree structures in MongoDB.

Lab: Data Model for an E-Commerce Site (page 111) Schema design group exercise

5.1 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 GridFS

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

5.2 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("...") } } }
);
```

5.3 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.

5.4 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 a 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.

Note:

- The most difficult part to this exercise is querying on arbitrary attributes, e.g. macbooks with 11 inch screens. Something like "tags": [{ "key": "screen_size", "value": 11 }, { "key": "brand", "value": "Apple" }] can be used, but \$elemMatch is extremely important here!

- Representing the category hierarchy is also interesting
-

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.
-

Note:

- Use arrays here (for hour/day granularity), and aggregate the arrays for other reports
-

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.

6 Replica Sets

Introduction to Replica Sets (page 113) An introduction to replication and replica sets.

Write Concern (page 117) Balancing performance and durability of writes.

Read Preference (page 122) Configuring clients to read from specific members of a replica set.

6.1 Introduction to Replica Sets

Learning Objectives

Upon completing this module, students should understand:

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

Use Cases for Replication

- High Availability
- Disaster Recovery
- Functional Segregation

High Availability (HA)

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

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

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

Disaster Recovery (DR)

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

Functional Segregation

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

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

Note:

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

Large Replica Sets

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

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

Note:

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

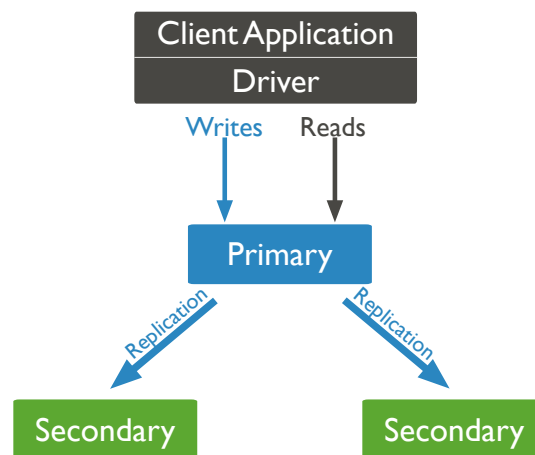
Replication is Not Designed for Scaling

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

Note:

- Eventual consistency: This is often tricky to handle as most developers are used to the strong consistency that most databases (and MongoDB) have by default. It also raises the question of how stale the data can be, and what to do when it crosses the threshold (e.g. fall back to reading from the primary).
 - Potential system overload: For example, consider a 3 data node replica set using secondaries to scale reads. Each node is serving reads at 70% of its capacity, which is a reasonable utilization rate. What happens if one of the secondaries fail or is intentionally taken down for maintenance (e.g. upgrading the OS or MongoDB)? Even if the load splits evenly between the 2 remaining nodes, they will be at $70 + (70/2) = 105\%$ capacity.
-

Replica Sets



Note:

- MongoDB implements replication in the form of replica sets. Don't use the term master-slave as that is what we had before replica sets. It still exists for some corner cases (e.g. > 12 replicas) but should otherwise be avoided.
- A replica set consists of one or more `mongod` servers. Maximum 50 nodes in total and up to 7 with votes.
- There is at most one `mongod` that is "primary" at any one time (though there are edge cases/bugs when there is more than one).
- There are usually two or more other `mongod` instances that are secondaries.
- Secondaries may become primary if there is a failover event of some kind.
- Failover is automatic when correctly configured and a majority of nodes remain.

- The secondaries elect a new primary automatically. A primary may also voluntarily step down, like when it can no longer reach the majority of nodes to avoid a potential split brain scenario.
-

Primary Server

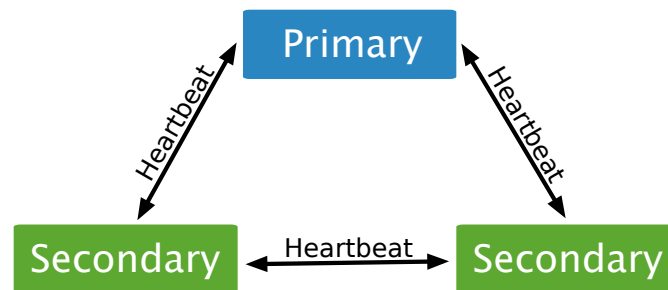
- Clients send writes the primary only.
 - MongoDB, Inc. maintains client drivers in many programming languages like Java, C#, Python, Ruby, and PHP.
 - MongoDB drivers are replica set aware.
-

Note: If the primary for a replica set changes from one node to another, the driver will automatically route writes to the correct `mongod`.

Secondaries

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

Heartbeats



Note:

- The members of a replica set use heartbeats to determine if they can reach every other node.
 - The heartbeats are sent every two seconds.
 - If a node is unreachable, this may indicate server failure, a network partition, or simply too slow to respond. The heartbeat will timeout and retried several times before the state is updated.
-

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.

6.2 Write Concern

Learning Objectives

Upon completing this module students should understand:

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

What happens to the write?

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

Answer

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

Balancing Durability with Performance

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

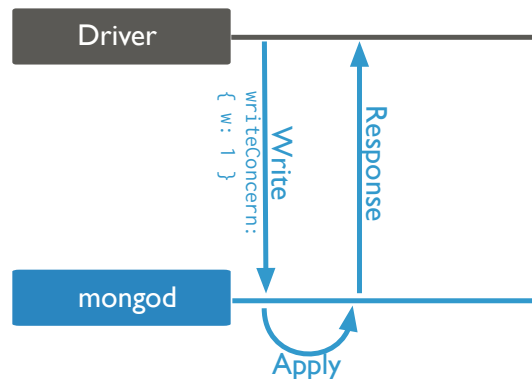
Note:

- MongoDB provides tunable write concern to better address the specific needs of applications.
 - Clients may adjust write concern to ensure that the most important operations persist successfully to an entire MongoDB deployment.
 - For other less critical operations, clients can adjust write concern to ensure faster performance.
-

Defining Write Concern

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

Write Concern: { w : 1 }



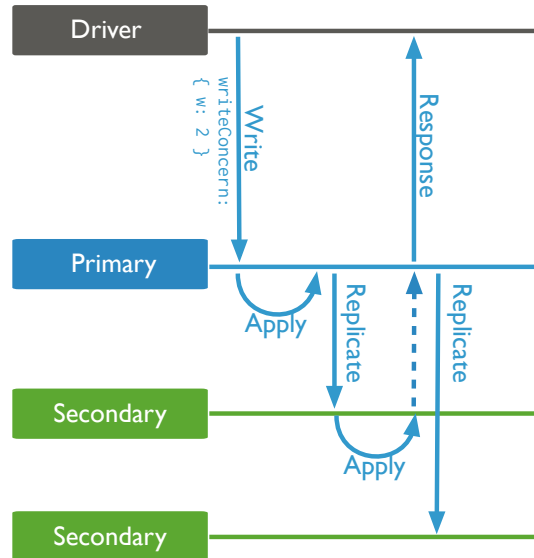
Note:

- We refer to this write concern as “Acknowledged”.
 - This is the default.
 - The primary sends an acknowledgement back to the client that it received the write operation (in RAM).
 - Allows clients to catch network, duplicate key, and other write errors.
-

Example: { w : 1 }

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

Write Concern: { w : 2 }



Note:

- Called “Replica Acknowledged”
 - Ensures the primary completed the write.
 - Ensures at least one secondary replicated the write.
-

Example: { w : 2 }

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

Other Write Concerns

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

Write Concern: { w : "majority" }

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

Example: { w : "majority" }

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

Quiz: Which write concern?

Suppose you have a replica set with 7 data nodes. Your application has critical inserts for which you do not want rollbacks to happen. Secondaries may be taken down from time to time for maintenance, leaving you with a potential 4 server replica set. Which write concern is best suited for these critical inserts?

- { w : 1 }
- { w : 2 }
- { w : 3 }
- { w : 4 }
- { w : "majority" }

Note: Answer: { w : "majority" }. This is the same as 4 for a 7 member replica set.

Further Reading

See [Write Concern Reference](#)⁹ for more details on write concern configurations, including setting timeouts and identifying specific replica set members that must acknowledge writes (i.e. [tag sets](#)¹⁰).

⁹<http://docs.mongodb.org/manual/reference/write-concern>

¹⁰<http://docs.mongodb.org/manual/tutorial/configure-replica-set-tag-sets/#replica-set-configuration-tag-sets>

6.3 Read Preference

What is Read Preference?

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

Use Cases

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

Note:

- If you have application servers in multiple data centers, you may consider having a [geographically distributed replica set](#)¹¹ and using a read preference of `nearest`.
 - This allows the client to read from the lowest-latency members.
 - Use `primaryPreferred` if you want an application to read from the primary under normal circumstances, but to allow possibly stale reads from secondaries during failover.
-

Not for Scaling

- In general, do *not* read from secondaries to provide extra capacity for reads.
- [Sharding](#)¹² increases read and write capacity by distributing operations across a group of machines.
- Sharding is a better strategy for adding capacity.

¹¹<http://docs.mongodb.org/manual/core/replica-set-geographical-distribution>

¹²<http://docs.mongodb.org/manual/sharding>

Read Preference Modes

MongoDB drivers support the following read preferences. Note that `hidden` nodes will never be read from when connected via the replica set.

- **primary**: Default. All operations read from the primary.
- **primaryPreferred**: Read from the primary but if it is unavailable, read from secondary members.
- **secondary**: All operations read from the secondary members of the replica set.
- **secondaryPreferred**: Read from secondary members but if no secondaries are available, read from the primary.
- **nearest**: Read from member of the replica set with the least network latency, regardless of the member's type.

Tag Sets

- There is also the option to use tag sets.
- You may tag nodes such that queries that contain the tag will be routed to one of the servers with that tag.
- This can be useful for running reports, say for a particular data center or nodes with different hardware (e.g. hard disks vs SSDs).

For example, in the mongo shell:

```
conf = rs.conf()
conf.members[0].tags = { dc : "east", use : "production" }
conf.members[1].tags = { dc : "east", use : "reporting" }
conf.members[2].tags = { use : "production" }
rs.reconfig(conf)
```

7 Sharding

Introduction to Sharding (page 124) An introduction to sharding.

7.1 Introduction to Sharding

Learning Objectives

Upon completing this module, students should understand:

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

Contrast with Replication

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

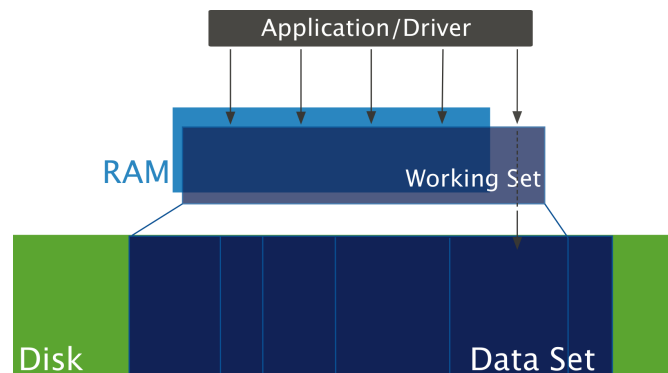
Sharding is Concerned with Scale

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

Vertical Scaling

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

The Working Set



Note:

- The working set for a MongoDB database is the portion of your data that clients access most often.
 - Your working set should stay in memory, otherwise random disk operations will hurt performance.
 - For the fastest processing, ensure that your indexes fit entirely in RAM so that the system can avoid reading the index from disk.
 - In some cases, only recently indexed values must be in RAM.
-

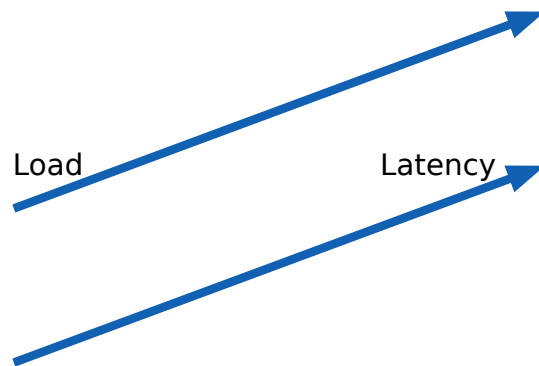
Limitations of Vertical Scaling

- There is a limit to how much RAM one machine can support.
- There are other bottlenecks such as I/O, disk access and network.
- Cost may limit our ability to scale up.
- There may be requirements to have a large working set that no single machine could possibly support.
- This is when it is time to scale horizontally.

Sharding Overview

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

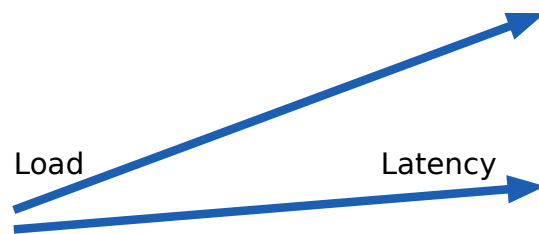
A Model that Does Not Scale



Note:

- Load and latency are related.
 - On a single server, latency tends to increase linearly with load.
 - Eventually latency is too great and the database has become a bottleneck.
-

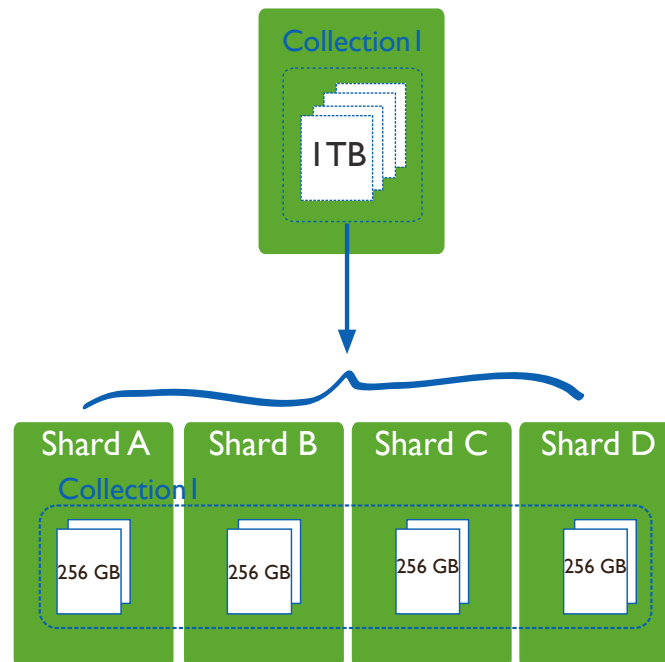
A Scalable Model



Note:

- Latency increases, but not linearly
 - We're not saying there is no increase in latency.
 - Just that it is much smaller than the increase in the server load.
 - Relational databases can scale vertically (i.e., by buying a bigger box).
 - MongoDB scales vertically and horizontally (i.e., spreading out the load across several boxes)
-

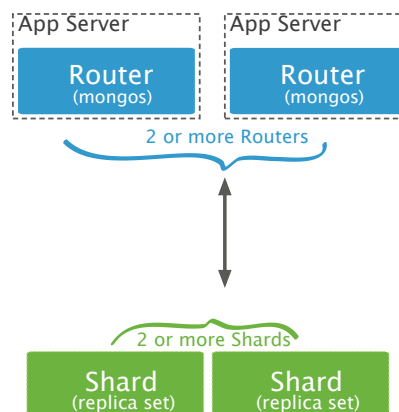
Sharding Basics



Note:

- When you shard a collection it is distributed across several servers.
 - When you perform a read or write operation it will go to a router that will then direct the query to the appropriate server.
 - Depending on the operation and how your cluster is configured you may need to touch only one server to complete the operation.
-

Sharded Cluster Architecture



Note:

- This figure illustrates one possible architecture for a sharded cluster.
- Each shard is a self contained replica set.
- Each replica set holds a partition of the data.
- As many new shards could be added to this sharded cluster as scale requires.
- At this point our cluster can handle a load approaching twice that of a single replica set using the same hardware.
- As mentioned, read/write operations go through a router.
- The server that routes requests is the mongos.

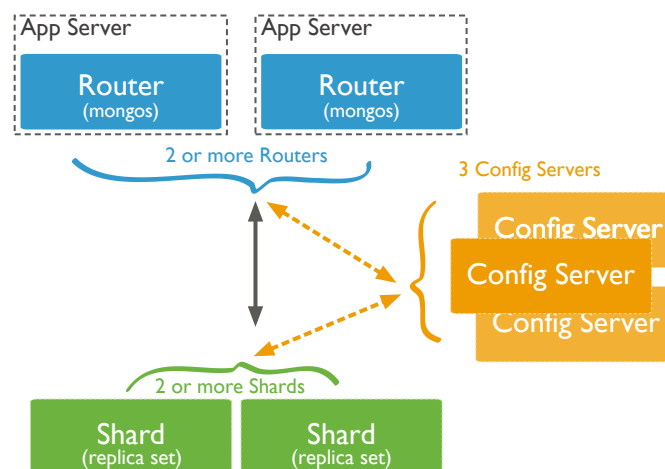
Mongos

- A mongos is responsible for accepting requests and returning results to an application driver.
- In a sharded cluster, nearly all operations go through a mongos.
- A sharded cluster can have as many mongos routers as required.
- It is typical for each application server to have one mongos.
- Always use more than one mongos to avoid a single point of failure.

Note:

- A mongos is typically deployed on an application server.
- There should be one mongos per app server.
- Scale with your app server.
- Very little latency between the application and the router.

Config Servers



Note:

- The previous diagram was incomplete; it was missing config servers.
- Use three config servers in production.

- These hold only metadata about the sharded collections.
 - Where your mongos servers are
 - Any hosts that are not currently available
 - What collections you have
 - How your collections are partitioned across the cluster
 - Mongos processes use them to retrieve the state of the cluster.
 - You can access cluster metadata from a mongos by looking at the `config db`.
-

Config Server Hardware Requirements

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

When to Shard

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

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

- MongoDB divides data into `chunks`.
- This is bookkeeping metadata.
 - There is nothing in a document that indicates its chunk.
 - The document does not need to be updated if its assigned chunk changes.
- If a chunk grows too large MongoDB will split it into two chunks.
- The MongoDB balancer keeps chunks distributed across shards in equal numbers.
- However, a balanced sharded cluster depends on a good shard key.

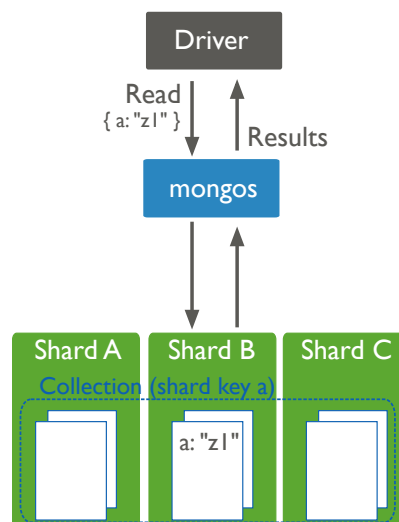
What is a Shard Key?

- You must define a shard key for a sharded collection.
- Based on one or more fields that every document must contain.
- Is immutable.
- The shard key determines where documents are located in the cluster.
- It is used to route operations to the appropriate shard.
- For reads and writes

Note:

- For reads and updates, the shard key determines which shard holds the document and will handle the operation.
 - When you insert a document, the shard key determines which server you will write to.
-

Targeted Query Using Shard Key



With a Good Shard Key

You might easily see that:

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

With a Bad Shard Key

You might see that:

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

Choosing a Shard Key

Generally, you want a shard key:

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

More Specifically

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

Cardinality

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

Non-Monotonic

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

Note:

- Documents will eventually move as chunks are balanced.
 - But in the meantime one server gets hammered while others are idle.
 - And moving chunks has its own performance costs.
-

Shards Should be Replica Sets

- As the number of shards increases, the number of servers in your deployment increases.
- This increases the probability that one server will fail on any given day.
- With redundancy built into each shard you can mitigate this risk.

8 MMS & Ops Manager

MongoDB Management Service (MMS) & Ops Manager (page 133) Learn about what MMS offers

Automation (page 135) MMS Automation

Exercise: Cluster Automation (page 138) Set up a cluster with MMS Automation

8.1 MongoDB Management Service (MMS) & Ops Manager

Learning Objectives

Upon completing this module students should understand:

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

MMS and Ops Manager

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

- Monitoring
- Automation
- Backups

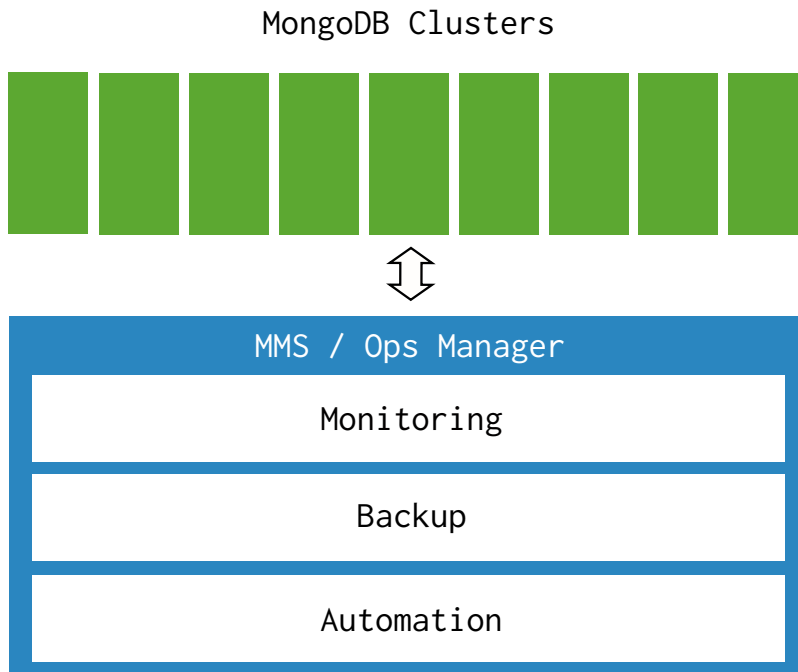
Deployment Options

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

Architecture

MMS

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



Ops Manager

On-premises MMS, with additional features for:

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

MMS and Ops Manager Use Cases

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

Note:

- Use these use cases to get students interested in how MMS can save them a lot of time
-

Creating an MMS Account

Free account at mms.mongodb.com

8.2 Automation

Learning Objectives

Upon completing this module students should understand:

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

What is Automation?

Fully managed MongoDB deployment on your own servers:

- Automated provisioning
- Dynamically add capacity (e.g. add more shards or replica set nodes)
- Upgrades
- Admin tasks (e.g. change the size of the oplog)

How Does Automation Work?

- Automation agent installed on each server in cluster
- Administrator creates design goal for system (through MMS / Ops Manager interface)
- Automation agents periodically check with MMS / Ops Manager to get new design instructions
- Agents create and follow a plan for implementing cluster design
- Minutes later, cluster design is complete, cluster is in goal state

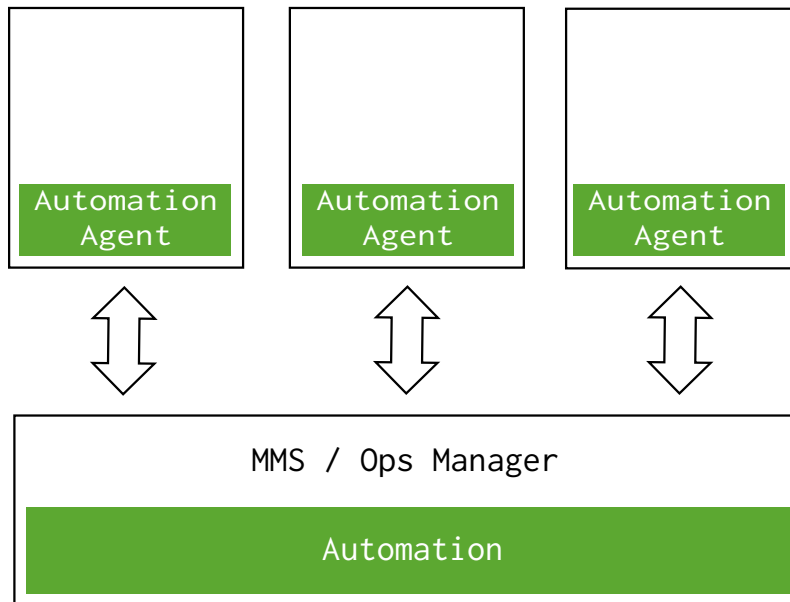
Automation Agents

Sample Use Case

Administrator wants to create a 100 shard cluster, with each shard comprised of a 3 node replica set:

- Administrator installs automation agent on 300 servers
- Cluster design is created in MMS / Ops Manager, then deployed to agents
- Agents execute instructions until 100 shard cluster is complete (usually several minutes)

Machines in Data Center



Upgrades Using Automation

- Upgrades without automation can be a manually intensive process (e.g. 300 servers)
- A lot of edge cases when scripting (e.g. 1 shard has problems, or one replica set is a mixed version)
- One click upgrade with MMS / Ops Manager Automation for the entire cluster

Automation: Behind the Scenes

- Agents ping MMS / Ops Manager for new instructions
- Agents compare their local configuration file with the latest version from MMS / Ops Manager
- Configuration file in json
- All communications over SSL

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


Configuration File

When version number of configuration file on MMS / Ops Manager is greater than local version, agent begins making a plan to implement changes:

```
"replicaSets": [  
  {  
    "_id": "shard_0",  
    "members": [  
      {  
        "_id": 0,  
        "host": "DemoCluster_shard_0_0",  
        "priority": 1,  
        "votes": 1,  
        "slaveDelay": 0,  
        "hidden": false,  
        "arbiterOnly": false  
      },  
      ...  
    ]  
  },  
  ...  
]
```

Automation Goal State

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

Demo

Note:

- Go to your Admin page (within MMS) -> My groups, create a new group, and walk through the process of setting up a small cluster on your laptop
-

8.3 Exercise: Cluster Automation

Learning Objectives

Upon completing this exercise students should understand:

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

MMS Automation Support

Windows machines are not supported at this time.

Exercise #1

Using your personal computer, create a cluster using MMS automation with the following topology:

- 3 shards
- Each shard is a 3 node replica set (2 data bearing nodes, 1 arbiter)
- Version 2.6.8 of MongoDB
- **To conserve space on your local machine, set “smallfiles” = true and “oplogSize” = 10**

Note:

- Windows is not supported, Windows users should work with another person in the class or work on a remote Linux machine
 - The entire cluster should be deployed on a single server (or the students laptop)
 - Registration is free, and won't require a credit card as long as the student stays below 8 servers
-

Exercise #2

Modify the cluster topology from Exercise #1 to the following:

- 4 shards (add one shard)
- Version 3.0.1 of MongoDB (upgrade from 2.6.8 -> 3.0.1)

Note:

- Students may complete this in one or two steps
 - Cluster configuration should be modified, then redeployed
-

9 Supplementary Material (Time Permitting)

Geospatial Indexes (page 139) Geospatial indexes: both those on legacy coordinate pairs and those supporting queries that calculate geometries on an earth-like sphere.

TTL Indexes (page 146) Time-To-Live Indexes.

Text Indexes (page 147) Free text indexes on string fields.

9.1 Geospatial Indexes

Learning Objectives

Upon completing this module, students should understand:

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

Introduction to Geospatial Indexes

We can use geospatial indexes to quickly determine geometric relationships:

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

Easiest to Start with 2 Dimensions

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

Note:

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

Location Field

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

Note:

- Draw several Xs to represent other documents.
-

Find Nearby Documents

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

Flat vs. Spherical Indexes

There are two types of geospatial indexes:

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

Flat Geospatial Index

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

Spherical Geospatial Index

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

Creating a 2d Index

Creating a 2d index:

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

Possible options key-value pairs:

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

Exercise: Creating a 2d Index

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

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

Note: Answer:

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

Inserting Documents with a 2d Index

There are two accepted formats:

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

Exercise: Inserting Documents with 2d Fields

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

Note: Answer:

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

Querying Documents Using a 2d Index

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

Example: Find Based on 2d Coords

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

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

Creating a 2dsphere Index

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

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

The geoJSON Specification

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

geoJSON Considerations

- The coordinates of points are given in degrees (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.

Note:

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

Simple Types of 2dsphere Objects

Point: A single point on the globe

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

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

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

Note:

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

Polygons

Simple Polygon:

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

Polygon with One Hole:

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

Other Types of 2dsphere Objects

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

Exercise: Inserting geoJSON Objects (1)

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

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

Note:

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

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

Exercise: Inserting geoJSON Objects (2)

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

Note:

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

Exercise: Inserting geoJSON Objects (3)

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

Note:

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

Exercise: Creating a 2dsphere Index

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

Note:

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

```
db.flights.createIndex( { destination : "2dsphere" } )  
  
db.flights.getIndexes() // see the indexes.
```

Querying 2dsphere Objects

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

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

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

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

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

9.2 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);
}
```

9.3 Text Indexes

Learning Objectives

Upon completing this module, students should understand:

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

What is a Text Index?

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

Creating a Text Index

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

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

Exercise: Creating a Text Index

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

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

Creating a Text Index with Weighted Fields

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

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

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

Creating a Text Index with Weighted Fields

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

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

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

Text Indexes are Similar to Multikey Indexes

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

Exercise: Inserting Texts

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

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

Querying a Text Index

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

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

Exercise: Querying a Text Index

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

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

Exercise: Querying Using Two Words

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

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

Search for a Phrase

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

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

Text Search Score

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

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