MongoDB Essentials Training

Release 2.6

MongoDB, Inc.

November 24, 2014

Contents

1 Introduction 12

1.1 Warm Up . . . 12
- Introductions . . . 12
- Getting to Know You . . . 12
- MongoDB Experience . . . 13
- 10gen . . . 13
- Origin of MongoDB . . . 13

1.2 MongoDB Overview . . . 14
- Learning Objectives . . . 14
- MongoDB is a Document Database . . . 14
- An Example MongoDB Document . . . 15
- Vertical Scaling . . . 16
- Scaling with MongoDB . . . 16
- Database Landscape . . . 17
- MongoDB Deployment Models . . . 17

1.3 MongoDB Stores Documents . . . 18
- Learning Objectives . . . 18
- JSON . . . 18
- A Simple JSON Object . . . 18
- JSON Keys and Values . . . 19
- Example Field Values . . . 19
- BSON . . . 20
- BSON Hello World . . . 20
- A More Complex BSON Example . . . 20
- Documents, Collections, and Databases . . . 21
- The `_id` Field . . . 21
- ObjectIds . . . 22
- Storing BSON Documents . . . 22
- Padding Factor . . . 22
- `usePowerOf2Sizes` . . . 23

1.4 Exercise: Installing MongoDB . . . 23
- Learning Objectives . . . 23
- Production Releases . . . 24
- Installing MongoDB . . . 24
- Setup . . . 24

Launch a `mongod` . . . 25
Import Exercise Data . . . 25
Launch a Mongo Shell . . . 25
Explore Databases . . . 26
Exploring Collections . . . 26
Admin Commands . . . 26
The MongoDB Data Directory . . . 27

2 CRUD 28

2.1 Creating and Deleting Documents . . . 28
Learning Objectives . . . 28
Creating New Documents . . . 28
Exercise: Inserting a Document . . . 29
Implicit `_id` Assignment . . . 29
Exercise: Assigning _ids . . . 29
Inserts will fail if... . . . 30
Exercise: Inserts will fail if... . . . 30
Bulk Inserts . . . 30
Ordered Bulk Insert . . . 31
Exercise: Ordered Bulk Insert . . . 31
Unordered Bulk Insert . . . 31
Exercise: Unordered Bulk Insert . . . 32
The Shell is a JavaScript Interpreter . . . 32
Exercise: Creating Data in the Shell . . . 32
Deleting Documents . . . 33
Using `remove()` . . . 33
Exercise: Removing Documents . . . 33
Dropping a Collection . . . 34
Exercise: Dropping a Collection . . . 34
Dropping a Database . . . 34
Exercise: Dropping a Database . . . 35

2.2 Reading Documents . . . 35
Learning Objectives . . . 35
The `find()` Method . . . 36
Query by Example . . . 36
Exercise: Querying by Example . . . 36
Querying Arrays . . . 37
Exercise: Querying Arrays . . . 37
Querying with Dot Notation . . . 37
Exercise: Querying with Dot Notation . . . 38
Exercise: Arrays and Dot Notation . . . 38
Cursors . . . 39
Exercise: Introducing Cursors . . . 39
Exercise: Cursor Objects in the Mongo Shell . . . 39
Cursor Methods . . . 40
Exercise: Using `count()` . . . 40
Exercise: Using `sort()` . . . 41
The `skip()` Method . . . 41
The `limit()` Method . . . 41
The `distinct()` Method . . . 42
Exercise: Using `distinct()` . . . 42

2.3 Query Operators . . . 42
Learning Objectives . . . 42
Comparison Query Operators . . . 43
Exercise: Comparison Operators . . . 43
Logical Query Operators . . . 44
Exercise: Logical Operators (Setup) . . . 44
Exercise: Logical Operators . . . 44
Element Query Operators . . . 45
Exercise: Element Operators . . . 45
Array Query Operators . . . 46
Exercise: Array Operators . . . 46

2.4 Updating Documents . . . 46
Learning Objectives . . . 46
The `update()` Method . . . 47
Parameters to `update()` . . . 47
`$set` and `$unset` . . . 48
Exercise: `$set` and `$unset` . . . 48
Update Operators . . . 48
Exercise: Update Operators . . . 49
`update()` Defaults to one Document . . . 49
Updating Multiple Documents . . . 50
Exercise: Multi-Update . . . 50
Array Operators . . . 50
Exercise: Array Operators . . . 51
The Positional `$` Operator . . . 51
Exercise: The Positional `$` Operator . . . 52
Upserts . . . 52
Upsert Mechanics . . . 52
Exercise: Upserts . . . 53
`save()` . . . 53
Exercise: `save()` . . . 53
Be Careful with `save()` . . . 54

3 Indexes 55

3.1 Index Fundamentals . . . 55
Learning Objectives . . . 55
Why Indexes? . . . 55
Types of Indexes . . . 56
Exercise: Using `explain()` . . . 56
Results of `explain()` . . . 57
Understanding `explain()` Output . . . 57
Single-Field Indexes . . . 58
Creating an Index . . . 58
Indexes and Read/Write Performance . . . 58
Index Limitations . . . 59
Use Indexes with Care . . . 59

3.2 Compound Indexes . . . 59
Learning Objectives . . . 59
Introduction to Compound Indexes . . . 60
The Order of Fields Matters . . . 60
Designing Compound Indexes . . . 60

Example: A Simple Message Board . . . 61
Load the Data . . . 61
Start with a Simple Index . . . 61
Query Adding `username` . . . 62
Include `username` in Our Index . . . 62
`ncanned` > n . . . 62
A Different Compound Index . . . 63
`nscanned` == `n` == 2 . . . 63
Let Selectivity Drive Field Order . . . 63
Adding in the Sort . . . 64
In-Memory Sorts . . . 64
Avoiding an In-Memory Sort . . . 64
General Rules of Thumb . . . 65

3.3 Multikey Indexes . . . 65
Learning Objectives . . . 65
Introduction to Multikey Indexes . . . 65
Example: Array of Numbers . . . 66
Exercise: Array of Documents, Part 1 . . . 66
Exercise: Array of Documents, Part 2 . . . 67
Exercise: Array of Arrays, Part 1 . . . 67
Exercise: Array of Arrays, Part 2 . . . 67
How Multikey Indexes Work . . . 68
Multikey Indexes and Sorting . . . 68
Exercise: Multikey Indexes and Sorting . . . 68
Limitations on Multikey Indexes . . . 69
Example: Multikey Indexes on Multiple Fields . . . 69

3.4 Hashed Indexes . . . 70
Learning Objectives . . . 70
What is a Hashed Index? . . . 70
Why Hashed Indexes? . . . 70
Limitations . . . 71
Floating Point Numbers . . . 71
Creating a Hashed Index . . . 71

3.5 Geospatial Indexes . . . 72
Learning Objectives . . . 72
Introduction to Geospatial Indexes . . . 72
Easiest to Start with 2 Dimensions . . . 72
Location Field . . . 73
Find Nearby Documents . . . 73
Flat vs. Spherical Indexes . . . 73
Flat Geospatial Index . . . 74
Spherical Geospatial Index . . . 74
Creating a 2d Index . . . 74
Exercise: Creating a 2d Index . . . 75
Inserting Documents with a 2d Index . . . 75
Exercise: Inserting Documents with 2d Fields . . . 75
Querying Documents Using a 2d Index . . . 76
Example: Find Based on 2d Coords . . . 76
Creating a 2dsphere Index . . . 76
The geoJSON Specification . . . 77
geoJSON Considerations . . . 77

Simple Types of 2dsphere Objects . . . 77
Polygons . . . 78
Other Types of 2dsphere Objects . . . 78
Exercise: Inserting geoJSON Objects (1) . . . 78
Exercise: Inserting geoJSON Objects (2) . . . 79
Exercise: Inserting geoJSON Objects (3) . . . 79
Exercise: Creating a 2dsphere Index . . . 79
Querying 2dsphere Objects . . . 80

3.6 TTL Indexes . . . 80
Learning Objectives . . . 80
TTL Index Basics . . . 80
Creating a TTL Index . . . 81
Exercise: Creating a TTL Index . . . 81
Exercise: Check the Collection . . . 81

3.7 Text Indexes . . . 82
Learning Objectives . . . 82
What is a Text Index? . . . 82
Creating a Text Index . . . 82
Exercise: Creating a Text Index . . . 83
Text Indexes are Similar to Multikey Indexes . . . 83
Exercise: Inserting Texts . . . 83
Querying a Text Index . . . 84
Exercise: Querying a Text Index . . . 84
Exercise: Querying Using Two Words . . . 84
Search for a Phrase . . . 85
Text Search Score . . . 85

4 Aggregation 86

4.1 Aggregation Tutorial . . . 86
Learning Objectives . . . 86
Aggregation Basics . . . 86
The Aggregation Pipeline . . . 87
Aggregation Stages . . . 87
The Match Stage . . . 88
Exercise: The Match Stage . . . 88
The Project Stage . . . 88
Exercise: Selecting fields with $project . . . 89
Exercise: Renaming fields with $project . . . 89
Exercise: Shaping documents with $project . . . 89
A Twitter Dataset . . . 90
Tweets Data Model . . . 90
Analyzing Tweets . . . 91
Friends and Followers . . . 91
Exercise: Friends and Followers . . . 91
Exercise: $match and $project . . . 92
The Group Stage . . . 92
Group using $avg . . . 92
Group using $push . . . 93
Group Aggregation Operators . . . 93
Rank Users by Number of Tweets . . . 94
Process . . . 94

Exercise: Ranking Users by Number of Tweets . . . 94
Exercise: Tweet Source . . . 95
The Unwind Stage . . . 95
Example: User Mentions in a Tweet . . . 95
Using $unwind . . . 96
Data Processing Pipelines . . . 96
Most Unique User Mentions . . . 96
Same Operator ($group), Multiple Stages . . . 97
The Sort Stage . . . 97
The Skip Stage . . . 97
The Limit Stage . . . 98
The Out Stage . . . 98

4.2 Optimizing Aggregation . . . 98
Learning Objectives . . . 98
Aggregation Options . . . 99
Aggregation Limits . . . 99
Limits Prior to MongoDB 2.6 . . . 99
Optimization: Reducing Documents in the Pipeline . . . 100
Optimization: Sorting . . . 100
Automatic Optimizations . . . 100

5 Schema Design 102

5.1 Schema Design Core Concepts . . . 102
Learning Objectives . . . 102
What is a schema? . . . 102
Example: Normalized Data Model . . . 102
Example: Denormalized Version . . . 103
Schema Design in MongoDB . . . 103
Three Considerations . . . 103
Case Study . . . 103
Author Schema . . . 103
User Schema . . . 104
Book Schema . . . 104
Example Documents: Author . . . 104
Example Documents: User . . . 104
Example Documents: Book . . . 105
Embedded Documents . . . 105
Example: Embedded Documents . . . 105
Embedded Documents: Pros and Cons . . . 106
Linked Documents . . . 106
Example: Linked Documents . . . 106
Linked Documents: Pros and Cons . . . 106
Arrays . . . 107
Array of Scalars . . . 107
Array of Documents . . . 107
Exercise: Users and Book Reviews . . . 108
Solution A: Users and Book Reviews . . . 108
Solution B: Users and Book Reviews . . . 108
Solution C: Users and Book Reviews . . . 109

5.2 Schema Evolution . . . 109
Learning Objectives . . . 109

Development Phase . . . 109
Development Phase: Known Query Patterns . . . 110
Production Phase . . . 110
Production Phase: Read Patterns . . . 110
Addressing List Books by Last Name . . . 111
Production Phase: Write Patterns . . . 111
Exercise: Recent Reviews . . . 112
Solution: Recent Reviews, Schema . . . 112
Solution: Recent Reviews, Update . . . 112
Solution: Recent Reviews, Read . . . 113
Solution: Recent Reviews, Delete . . . 113

5.3 Common Schema Design Patterns . . . 114
Learning Objectives . . . 114
1-1 Relationship . . . 114
1-1: Linking . . . 114
1-1: Embedding . . . 115
1-M Relationship . . . 115
1-M: Array of IDs . . . 115
1-M: Single Field with ID . . . 116
1-M: Array of Documents . . . 116
M-M Relationship . . . 117
M-M: Array of IDs on Both Sides . . . 117
M-M: Array of IDs on Both Sides . . . 117
M-M: Array of IDs on One Side . . . 118
M-M: Array of IDs on One Side . . . 118
Tree Structures . . . 119
Allow users to browse by subject . . . 119
Alternative: Parents and Ancestors . . . 119
Find Sub-Categories . . . 120
Summary . . . 120

6 Replica Sets 121

6.1 Introduction to Replica Sets . . . 121
Learning Objectives . . . 121
Use Cases for Replication . . . 121
High Availability (HA) . . . 122
Disaster Recovery (DR) . . . 122
Functional Segregation . . . 122
Replication is Not Designed for Scaling . . . 123
Replica Sets . . . 123
Primary Server . . . 124
Secondaries . . . 124
Heartbeats . . . 124
The Oplog . . . 125

6.2 Elections in Replica Sets . . . 125
Learning Objectives . . . 125
Members and Votes . . . 125
Calling Elections . . . 126
Selecting a New Primary . . . 126
Priority . . . 126
Optime . . . 127

Connections . . . 127
When will a primary step down? . . . 127
Exercise: Elections in Failover Scenarios . . . 128
Scenario A: 3 Data Nodes in 1 DC . . . 128
Scenario B: 3 Data Nodes in 2 DCs . . . 128
Scenario C: 4 Data Nodes in 2 DCs . . . 129
Scenario D: 5 Nodes in 2 DCs . . . 129
Scenario E: 3 Data Nodes in 3 DCs . . . 130
Scenario F: 5 data nodes in 3 DCs . . . 130

6.3 Replica Set Roles and Configuration . . . 131
Learning Objectives . . . 131
Example: A Five-Member Replica Set Configuration . . . 131
Configuration . . . 131
Principal Data Center . . . 132
Data Center 2 . . . 132
What about `dc1-3` and `dc2-2`? . . . 132
What about `dc2-2`? . . . 132

6.4 The Oplog: Statement Based Replication . . . 133
Learning Objectives . . . 133
Binary Replication . . . 133
Tradeoffs . . . 134
Statement-Based Replication . . . 134
Example . . . 134
Replication Based on the Oplog . . . 135
Create a Replica Set . . . 135
`ReplSetTest` . . . 135
Start the Replica Set . . . 136
Status Check . . . 136
Connect to the Primary . . . 136
Create some Inventory Data . . . 137
Perform an Update . . . 137
View the Oplog . . . 137
Operations in the Oplog are Idempotent . . . 138
The Oplog Window . . . 138
Sizing the Oplog . . . 138

6.5 Write Concern . . . 139
Learning Objectives . . . 139
What happens to the write? . . . 139
Answer . . . 139
Balancing Durability with Performance . . . 140
Defining Write Concern . . . 140
Write Concern: `{ w : 1 }` . . . 140
Example: `{ w : 1 }` . . . 141
Write Concern: `{ w : 2 }` . . . 141
Example: `{ w : 2 }` . . . 141
Other Write Concerns . . . 142
Write Concern: `{ w : "majority" }` . . . 142
Example: `{ w : "majority" }` . . . 142
Quiz: Which write concern? . . . 143
Further Reading . . . 143

6.6 Read Preference . . . 144
What is Read Preference? . . . 144
Use Cases . . . 144
Not for Scaling . . . 144
Read Preference Modes . . . 145
Tag Sets . . . 145

6.7 Exercise: Setting up a Replica Set . . . 146
Overview . . . 146
Create Data Directories . . . 146
Launch Each Member . . . 146
Status . . . 147
Connect to a MongoDB Instance . . . 147
Configure the Replica Set . . . 147
Write to the Primary . . . 148
Read from a Secondary . . . 148
Review the Oplog . . . 148
Changing Replica Set Configuration . . . 149
Verifying Configuration Change . . . 149
Further Reading . . . 149

7 Sharding 150

7.1 Introduction to Sharding . . . 150
Learning Objectives . . . 150
Contrast with Replication . . . 150
Sharding is Concerned with Scale . . . 151
Vertical Scaling . . . 151
The Working Set . . . 151
Limitations of Vertical Scaling . . . 152
Sharding Overview . . . 152
A Model that Does Not Scale . . . 153
A Scalable Model . . . 153
Sharding Basics . . . 153
Sharded Cluster Architecture . . . 154
Mongos . . . 155
Config Servers . . . 155
Config Server Hardware Requirements . . . 156
When to Shard . . . 156
Possible Imbalance? . . . 156
Balancing Shards . . . 157
What is a Shard Key? . . . 157
Targeted Query Using Shard Key . . . 157
With a Good Shard Key . . . 158
With a Bad Shard Key . . . 158
Choosing a Shard Key . . . 159
More Specifically . . . 159
Cardinality . . . 159
Non-Monotonic . . . 160
Shards Should be Replica Sets . . . 160

7.2 Balancing Shards . . . 160
Learning Objectives . . . 160
Chunks and the Balancer . . . 161

Chunks in a Newly Sharded Collection . . . 161
Chunk Splits . . . 161
Pre-Splitting Chunks . . . 162
Start of a Balancing Round . . . 162
Balancing is Resource Intensive . . . 162
Chunk Migration Steps . . . 163
Concluding a Balancing Round . . . 163

7.3 Shard Tags . . . 163
Learing Objectives . . . 163
Tags - Overview . . . 164
Example: DateTime . . . 164
Example: Location . . . 164
Example: Premium Tier . . . 165
Tags - Caveats . . . 165

7.4 Exercise: Setting Up a Sharded Cluster . . . 165
Learning Objectives . . . 165
Our Sharded Cluster . . . 166
Sharded Cluster Configuration . . . 166
Build Our Data Directories . . . 166
Initiate a Replica Set . . . 167
Spin Up a Second Replica Set . . . 167
A Third Replica Set . . . 168
Status Check . . . 168
Launch Config Servers . . . 169
Launch the Mongos Processes . . . 169
Add All Shards . . . 169
Enable Sharding and Shard a Collection . . . 170
Observe What Happens . . . 170

8 Security 171

8.1 Security . . . 171
Learning Objectives . . . 171
Overview . . . 171
Authentication Options . . . 172
Authorization via MongoDB . . . 172
Network Exposure Options . . . 172
Encryption (SSL) . . . 173
Native MongoDB Auth . . . 173
Exercise: Create an Admin User, Part 1 . . . 173
Exercise: Create an Admin User, Part 2 . . . 174
Using MongoDB Roles . . . 174
Exercise: Creating a `readWrite` User, Part 1 . . . 174
Exercise: Creating a `readWrite` User, Part 2 . . . 175
MongoDB Custom User Roles . . . 175

9 Reporting Tools and Diagnostics 176

9.1 Performance Troubleshooting . . . 176
Learning Objectives . . . 176
`mongostat` and `mongotop` . . . 176
Exercise: `mongostat` (setup) . . . 177

Exercise: mongostat (run) . . . 177
Exercise: mongostat (create index) . . . 178
Exercise: mongotop . . . 178
db.currentOp() . . . 179
Exercise: db.currentOp() . . . 179
db.collection.stats() . . . 180
Exercise: Using Collection Stats . . . 180
The Profiler . . . 180
The Profiler (continued) . . . 181
Exercise: Exploring the Profiler . . . 181
db.serverStatus() . . . 181
Exercise: Using db.serverStatus() . . . 182
Analyzing profiler data . . . 182
Performance Improvement Techniques . . . 182
Performance Tips: Write Concern . . . 183
Bulk Operations . . . 183
Exercise: Comparing bulk inserts with mongostat . . . 183
mongostat, bulk inserts with {w: 1} . . . 184
Bulk inserts with {w: 3} . . . 184
mongostat, bulk inserts with {w: 3} . . . 185
Schema Design . . . 185
Shard Key Considerations . . . 185
Indexes and Performance . . . 186

10 Backup and Recovery 187

10.1 Backup and Recovery . . . 187
Disasters Do Happen . . . 187
Human Disasters . . . 188
Terminology: RPO vs. RTO . . . 188
Terminology: DR vs. HA . . . 189
Quiz . . . 189
Backup Options . . . 189
Document Level Backups - mongodump . . . 190
mongodump . . . 190
File System Level . . . 191
Ensure Consistency . . . 191
File System Backups: Pros and Cons . . . 191
Document Level - mongorestore . . . 192
File System Restores . . . 192
Backup Sharded Cluster . . . 192
Restore Sharded Cluster . . . 193
Tips and Tricks . . . 193
Backup Options . . . 194
MMS Backup . . . 194
Sharded Clusters . . . 194
Under the Hood . . . 195
Key Benefits . . . 195
Point in Time Backups . . . 195
Easy to Restore . . . 196
Unlimited Restores . . . 196
Fully Managed . . . 196
Getting Started . . . 196

1 Introduction

Warm Up **(page 12)** Activities to get the class started

MongoDB Overview **(page 14)** MongoDB philosophy and features.

MongoDB Stores Documents **(page 18)** The structure of data in MongoDB.

Exercise: Installing MongoDB **(page 23)** Install mongodb experiment with a few operations.

1.1 Warm Up

Introductions

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

Notes:

Getting to Know You

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

Notes:

MongoDB Experience

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

Notes:

10gen

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

Notes:

Origin of MongoDB

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

Notes:

1.2 MongoDB Overview

Learning Objectives

Upon completing this module students should understand:

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

Notes:

MongoDB is a Document Database

Documents are associative arrays like:

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

Notes:

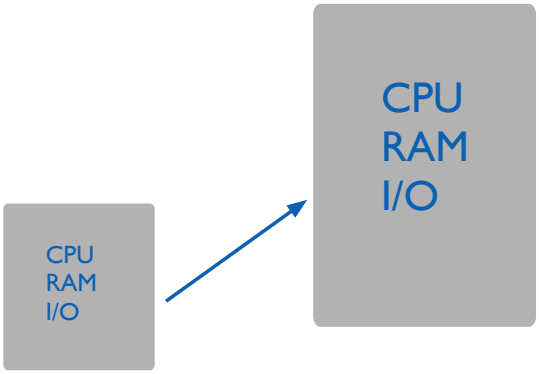
An Example MongoDB Document

A MongoDB document expressed using JSON syntax.

```
{
    "a" : 3,
    "b" : [3, 2, 7],
    "c" : {
        "d" : 4 ,
        "e" : "asdf",
        "f" : true,
        "h" : ISODate("2014-10-23T01:19:40.732Z")
    }
}
```

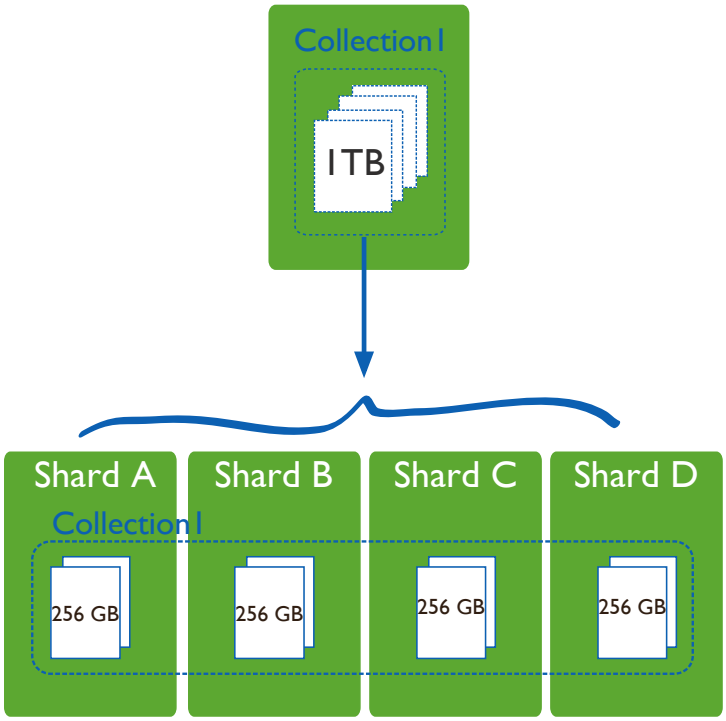
Notes:

Vertical Scaling



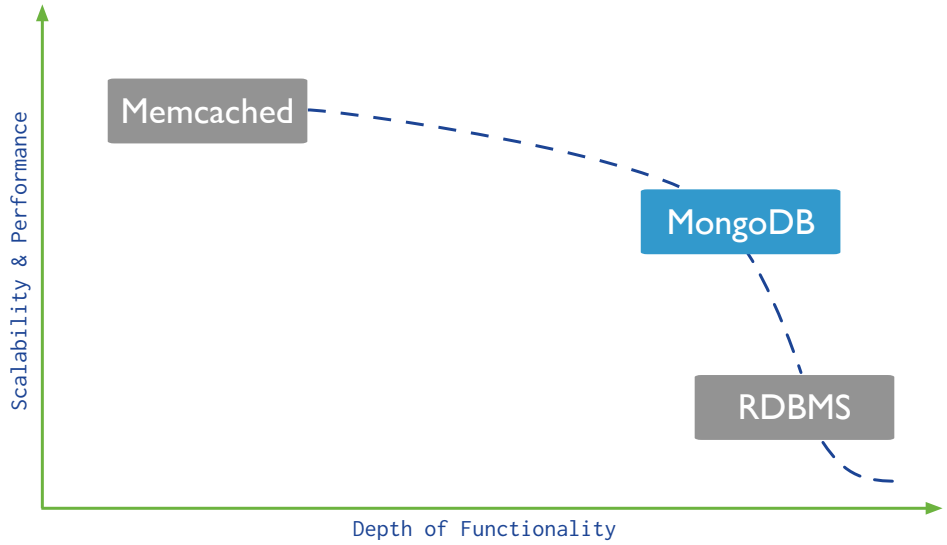
Notes:

Scaling with MongoDB



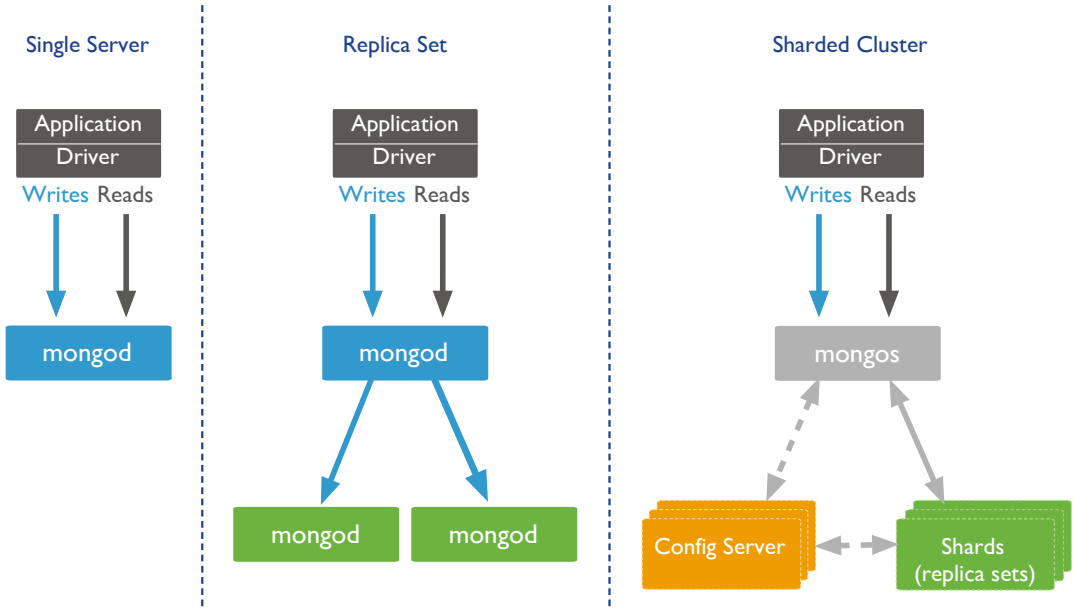
Notes:

Database Landscape



Notes:

MongoDB Deployment Models



Notes:

1.3 MongoDB Stores Documents

Learning Objectives

Upon completing this module, students should understand:

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

Notes:

JSON

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

Notes:

A Simple JSON Object

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

Notes:

JSON Keys and Values

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

Notes:

Example Field Values

```
{
    "first key" : "value" ,
    "second key" : {
        "first embedded key" : "first embedded value",
        "second embedded key" : "second embedded value"
     },
     "third key" : [
         "first array element",
         "second element",
         { "embedded key" : "embedded value" },
         [ 1, 2 ]
     ]
}
```

Notes:

[1] http://json.org/

BSON

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

Notes:

BSON Hello World

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

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

Notes:

A More Complex BSON Example

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

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

Notes:

[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

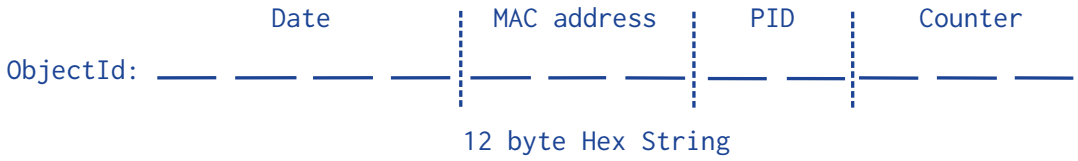
Notes:

The `_id` Field

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

Notes:

ObjectIds



Notes:

Storing BSON Documents

- Each document may be a different size from the others.
- The maximum BSON document size is 16 megabytes.
- Documents are physically adjacent to each other on disk and in memory.
- If a document is updated in a way that makes it larger, MongoDB may move the document.
- This may cause fragmentation, resulting in unnecessary I/O.
- Strategies to reduce the effects of document growth:
  - Padding factor
  - `usePowerOf2Sizes`

Notes:

Padding Factor

Notes:

`usePowerOf2Sizes`

- When a document must move to a new location this leaves a fragment.
- MongoDB will attempt to fill this fragment with a new document eventually.
- As of MongoDB 2.6, collections have a setting called `usePowerOf2Sizes` enabled by default for newly created collections.
- This setting will round the size of the document up to the next power of 2.
- E.g, a document that 118 bytes will be allocated 128 bytes.
- If moved, the space can be filled with two 64-byte documents, four 32-byte documents, etc.

Notes:

1.4 Exercise: Installing MongoDB

Learning Objectives

Upon completing this exercise students should understand:

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

Notes:

Production Releases

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

- Windows
- OSX
- Linux
- Solaris

Notes:

Installing MongoDB

- Visit http://www.mongodb.org/downloads
- Download and install the appropriate package for your machine.
- 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.
- 32-bit versions should NOT be used in production (limited to 2GB of data). They are acceptable for training classes.

Notes:

Setup

```
PATH=$PATH:path_to_mongodb/bin

sudo mkdir -p /data/db

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

Notes:

Launch a `mongod`

```
/<path_to_mongodb>/bin/mongod --help
```

```
/<path_to_mongodb>/bin/mongod
```

Notes:

Import Exercise Data

```
cd usb_drive
```

```
unzip sampledata.zip
```

```
cd sampledata
```

```
mongoimport -d sample -c tweets twitter.json
```

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

```
cd dump
```

```
mongorestore -d sample training
```

```
mongorestore -d sample digg
```

Notes:

Launch a Mongo Shell

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

```
mongo
```

Display available commands.

```
help
```

Notes:

Explore Databases

Display available databases.

```
show dbs
```

To use a particular database we can type the following.

```
use <database_name>
```

```
db
```

Notes:

Exploring Collections

```
show collections
```

```
db.collection.help()
```

```
db.collection.find()
```

Notes:

Admin Commands

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

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

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

Notes:

The MongoDB Data Directory

```
ls /data/db
```

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

Notes:

2 CRUD

Creating and Deleting Documents **(page 28)** Inserting documents into collections, deleting documents, and dropping collections.

Reading Documents **(page 35)** The find() command, query documents, dot notation, and cursors.

Query Operators **(page 42)** MongoDB query operators including: comparison, logical, element, and array operators.

Updating Documents **(page 46)** Using `update()` and associated operators to mutate existing documents.

2.1 Creating and Deleting Documents

Learning Objectives

Upon completing this module students should understand:

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

Notes:

Creating New Documents

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

```
db.collection.insert( { "name" : "susan" } )
```

Notes:

Exercise: Inserting a Document

Experiment with the following commands.

```
use sample
```

```
db.hellos.insert( { a : "hello, world!" } )
```

```
db.hellos.find()
```

Notes:

Implicit `_id` Assignment

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

Notes:

Exercise: Assigning _ids

Experiment with the following commands.

```
db.hellos.insert( { _id : 253, a : "a string" } )
```

```
db.hellos.find()
```

Notes:

Inserts will fail if...

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

Notes:

Exercise: Inserts will fail if...

```
// fails because _id can't have an array value
db.hellos.insert( { _id : [ 1, 2, 3 ] } )

// succeeds
db.hellos.insert( { _id : 3 } )

// fails because of duplicate id
db.hellos.insert( { _id : 3 } )

// malformed document
db.hellos.insert( { "hello" } )
```

Notes:

Bulk Inserts

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

Notes:

Ordered Bulk Insert

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

Notes:

Exercise: Ordered Bulk Insert

Experiment with the following bulk insert.

```
db.things.insert( [ { _id : 19, type : "atom", symbol : "K" },
                    { _id : 20, type : "car", color : "red" },
                    { _id : 20, type : "planet", name : "Saturn" },
                    { type : "office",
                      street : "229 West 43rd Street, 5th Floor",
                      city : "New York",
                      state : "NY" } ] )

db.things.find()
```

Notes:

Unordered Bulk Insert

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

Notes:

Exercise: Unordered Bulk Insert

Experiment with the following bulk insert.

```
db.otherThings.insert( [ { _id : 19, type : "atom", symbol : "K" },
                         { _id : 20, type : "car", color : "red" },
                         { _id : 20, type : "planet", name : "Saturn" },
                         { type : "office",
                           street : "229 West 43rd Street, 5th Floor",
                           city : "New York",
                           state : "NY" } ],
                       { ordered : false } )
db.otherThings.find()
```

Notes:

The Shell is a JavaScript Interpreter

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

Notes:

Exercise: Creating Data in the Shell

Experiment with the following commands.

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

db.stuff.find()
```

Notes:

Deleting Documents

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

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

Notes:

Using `remove()`

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

Notes:

Exercise: Removing Documents

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

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

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

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

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

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

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

Notes:

Dropping a Collection

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

Notes:

Exercise: Dropping a Collection

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

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

Notes:

Dropping a Database

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

Notes:

Exercise: Dropping a Database

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

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

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

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

Notes:

2.2 Reading Documents

Learning Objectives

Upon completing this module students should understand:

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

Notes:

The `find()` Method

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

Notes:

Query by Example

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

Notes:

Exercise: Querying by Example

Experiment with the following sequence of commands.

```
db.testcol.drop()
db.testcol.insert( [ { _id : 1, a : 5, b : 3 },
                     { _id : 3, b : 5, c : 12 },
                     { a : 7, b : 3 },
                     { c : 5, b : 7 } ] )
db.testcol.find()

db.testcol.find( { a : 5 } )

db.testcol.find( { b : 3, a : 7 } )
```

Notes:

Querying Arrays

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

Notes:

Exercise: Querying Arrays

Experiment with the following sequence of commands.

```
db.testcol.drop()
db.testcol.insert( [ { a : [ 1, 2, 3 ] },
                     { a : [ 3, 4, 5 ] },
                     { a : [ 5, 6, 7 ] } ] )

 // These match documents where a contains the value specified
 db.testcol.find( { a : 3 } )
 db.testcol.find( { a : 5 } )

 // These match documents where a equals the value specified
 db.testcol.find( { a : [ 3, 5 ] } )  // no documents
 db.testcol.find( { a : [ 3, 4, 5 ] } )  // only the second document
```

Notes:

Querying with Dot Notation

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

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

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

Notes:

Exercise: Querying with Dot Notation

```
db.buildings.insert(
    [ {
          type : "house",
          location : { streetNumber : 123,
                       street : "7th Ave" } },
      {
          type : "office",
          location : { streetNumber : 234,
                       street : "7th Ave",
                       floor : 7 } },
      {
          type : "apartment",
          location : { streetNumber : 335,
                       street : "43rd Street",
                       number : 745 } } ] )

db.buildings.find( { "location.street" : "7th Ave" } ) // Two matches
```

Notes:

Exercise: Arrays and Dot Notation

Experiment with the following commands.

```
db.things.insert( [
    { type : "fruit",
      examples : [ { type : "banana", color : "yellow" },
                   { type : "apple", color : "red" },
                   { type : "mango", color : "red" } ] },
    { type : "cars",
      examples : [ { model : "Camaro", color : "red" },
                   { model : "Pinto", color : "yellow" },
                   { model : "Tacoma", color : "blue" } ] },
    { type : "planets",
      examples : [ { name : "Mars", color : "red" },
                   { name : "Venus", color : "blue" },
                   { name : "Earth", color : "blue" } ] } ] )

 db.things.find( { "examples.color" : "blue" } ) // two documents
```

Notes:

Cursors

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

Notes:

Exercise: Introducing Cursors

Experiment with the following commands.

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

it
it
```

Notes:

Exercise: Cursor Objects in the Mongo Shell

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

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

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

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

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

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

Notes:

Cursor Methods

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

Notes:

Exercise: Using `count()`

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

// all 100
db.testcol.count()

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

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

Notes:

Exercise: Using `sort()`

Experiment with the following sort commands.

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

db.testcol.find()

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

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

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

Notes:

The `skip()` Method

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

Notes:

The `limit()` Method

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

Notes:

The `distinct()` Method

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

Notes:

Exercise: Using `distinct()`

Experiment with the following commands and note what `distinct()` returns.

```
db.testcol.drop()
db.testcol.insert( [ { a : 2 , b : 3 },
                     { a : 2 },
                     { a : "hello" },
                     { a : "hello" },
                     { a : { hello : "world" } } ] )
db.testcol.distinct( "a" )
```

Notes:

2.3 Query Operators

Learning Objectives

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

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

Notes:

Comparison Query Operators

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

Notes:

Exercise: Comparison Operators

Experiment with the following.

```
db.testcol.drop()
for (i=1;i<=5;i++) { db.testcol.insert( { a : i } ) };
db.testcol.insert( { } )                          // No "a" field
db.testcol.find()

db.testcol.find( { a : { $gte : 2 } } )

db.testcol.find( { a : { $ne : 2 } } )

db.testcol.find( { a : { $in : [ 3, 2 ] } } )

db.testcol.find( { a : { $nin : [ 3, 2 ] } } )
```

Notes:

Logical Query Operators

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

Notes:

Exercise: Logical Operators (Setup)

Create a collection we can experiment with.

```
db.testcol.drop()
for (i=1; i<=3; i++) {
    for (j=1; j<=3; j++) {
        db.testcol.insert( { a : i, b : j } )
    }
};
db.testcol.insert( { b : 10 } )  // No "a" field
db.testcol.find()
```

Notes:

Exercise: Logical Operators

Experiment with the following.

```
db.testcol.find( { $or : [ { a : 1 }, { b : 2 } ] } )

db.testcol.find( { a : { $not : { $gt : 3 } } } )

db.testcol.find( { $nor : [ { a : 3 } , { b : 3 } ] } )

db.testcol.find( { b : { $gt : 2 , $lte : 10 } } )  // and is implicit

db.testcol.find( { $and : [ { $or : [ { a : 1 }, { a : 2 } ] },
                            { $or : [ { b : 2 }, { b : 3 } ] } ] } )
```

Notes:

Element Query Operators

- `$exists`: Select documents based on the existence of a particular field.
- `$type`: Select documents based on their type.
- See BSON types[3] for reference on types.

Notes:

Exercise: Element Operators

Experiment with the following.

```
db.testcol.drop()
// by default, the mongo shell treats numbers as floating-point values
db.testcol.insert( [ { a : 1 }, { b : 1 }, { a : NumberInt(2) },
                     { b : "b" } ] )

db.testcol.find( { a : { $exists : true } } )

// type 1 is Double
db.testcol.find( { b : { $type : 1 } } )

// type 2 is String
db.testcol.find( { b : { $type : 2 } } )

// type 16 is 32-bit integer
// use NumberInt(), NumberLong() to handle integers in the mongo shell
db.testcol.find( { a : { $type : 16 } } )
```

Notes:

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

Array Query Operators

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

Notes:

Exercise: Array Operators

Experiment with the following.

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

db.testcol.find( { a : { $all : [ 1, 2 ] } } )

db.testcol.find( { a : { $size : 3 } } )

// at least one element must match both conditions
db.testcol.find( { a : { $elemMatch : { $gte : 2, $lte : 4 } } } )

// at least one element must match either condition
// does not need to be the same element
db.testcol.find( { a : { $gte : 2, $lte : 4 } } )
```

Notes:

2.4 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.

Notes:

The `update()` Method

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

Notes:

Parameters to `update()`

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

Notes:

`$set` and `$unset`

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

Notes:

Exercise: `$set` and `$unset`

Experiment with the following. Do a `find()` after each update to view the results.

```
db.testcol.drop()
for (i=1; i<=5; i++) { db.testcol.insert( { _id : i, a : i, b : i } ) }

db.testcol.update( { _id : 3 }, { $set : { a : 6 } } )

db.testcol.update( { _id : 5 } , { $set : { c : 5 } } )

db.testcol.update( { _id : 5 } , { $set : { c : 7 , a : 7 } } )

db.testcol.update( { _id : 5 } , { d : 4 } )

db.testcol.update( { _id : 4 } , { $unset :  { a : 1 } } )
```

Notes:

Update Operators

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

Notes:

Exercise: Update Operators

Experiment with the following update operators.

```
db.testcol.drop()
for (i=1; i<=5; i++) { db.testcol.insert( { _id : i, a : i, b : i } ) }
db.testcol.find()

db.testcol.update( { _id : 2 }, { $inc : { a : -3 } } )

db.testcol.update( { _id : 1 }, { $inc : { q : 1 } } )

db.testcol.update( { _id : 3 }, { $mul : { a : 4 } } )

db.testcol.update( { _id : 4 }, { $rename : { a : "xyz" } } )

db.testcol.update( { _id : 5 }, { $min : { a : 3 } } )

db.testcol.update( { _id : 1 },
                   { $currentDate : { c : { $type : "timestamp" } } } )
```

Notes:

`update()` Defaults to one Document

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

Notes:

Updating Multiple Documents

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

Notes:

Exercise: Multi-Update

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

```
db.testcol.drop()
for (i=1; i<=5; i++) { db.testcol.insert( { _id : i, a : i } ) }

db.testcol.update( { _id : { $lt : 5 } }, { $set : { a : 6 } } )

db.testcol.update( { _id : { $lt : 5 } }, { $set : { a : 7 } },
                   { multi : true } )
```

Notes:

Array Operators

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

Notes:

Exercise: Array Operators

Experiment with the following updates.

```
db.testcol.drop()
for (i=1; i<=5; i++) { db.testcol.insert(
    { _id : i, a : i, b: [ 1, 2, 3, 3, "NA", 6 ] } ) }

db.testcol.update( { _id : 1 }, { $push : { b : 3 } } )
db.testcol.update( { _id : 2 }, { $pushAll : { b : [ 1, 2, 3 ] } } )

db.testcol.update( { _id : 2 }, { $pop : { b : "" } } )

db.testcol.update( { _id : 3 }, { $pull : { b : 3 } } )
db.testcol.update( { _id : 4 },
                   { $pullAll : { b : [ 1, 2, "NA", 4 ] } } )

db.testcol.update( { _id : 5 }, { $addToSet : { b : 2 } } )
db.testcol.update( { _id : 5 }, { $addToSet : { b : 4 } } )
```

Notes:

The Positional $ Operator

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

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

Notes:

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

Exercise: The Positional $ Operator

Experiment with the following commands.

```
db.testcol.drop()
for (i=1; i<=5; i++) { db.testcol.insert(
    { _id: i, a: i, b: [ 1, 2, 3, 3, "NA", 6 ] } ) }
db.testcol.find()

db.testcol.update( { b: "NA" }, { $set: { "b.$" : 11 } },
                   { multi: true } )
db.testcol.find()
```

Notes:

Upserts

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

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

Notes:

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.

Notes:

Exercise: Upserts

Experiment with the following upserts.

```
db.testcol.drop()
for (i=1; i<=5; i++) {
    db.testcol.insert( { _id: i, a: i, b: i } ) }
db.testcol.find()

db.testcol.update( { a: 4 }, { $inc: { b : 3 } }, { upsert: true } )

db.testcol.update( { a: 12 }, { $inc: { b : 3 } }, { upsert: true } )

db.testcol.update( { _id: 6, a: 6 }, { c: 155 }, { upsert: true } )
```

Notes:

`save()`

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

  ```
  db.collection.save( document )
  ```

Notes:

Exercise: `save()`

```
db.testcol.drop()
for (i=1; i<=5; i++) { db.testcol.save( { _id : i, a : i, b : i } ) }
db.testcol.find()

// Look at the code for save. Note that it involves an upsert.
db.testcol.save

// new document, _id created
db.testcol.save( { a : 3 } )

db.testcol.save( { _id : 6, a : 6 } )            // new document

db.testcol.save( { _id : 3, a : 12, b : 12  } )  // update!!
```

Notes:

Be Careful with `save()`

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

```
db.testcol.drop()
db.testcol.insert( { _id : 2, a : 2, b : 2 } )

db.testcol.find( { _id : 2 } )
doc = db.testcol.findOne( { _id: 2 } )

db.testcol.update( { _id: 2 }, { $inc: { b : 1 } } )
db.testcol.find()

doc.c = 11
doc

db.testcol.save(doc)  // just lost our incrementing of b.
db.testcol.find()
```

Notes:

3 Indexes

Index Fundamentals **(page 55)** An introduction to MongoDB indexes.

Compound Indexes **(page 59)** Indexes on two or more fields.

Multikey Indexes **(page 65)** Indexes on array fields.

Hashed Indexes **(page 70)** Hashed Indexes.

Geospatial Indexes **(page 72)** Geospatial indexes: both those on legacy coordinate pairs and those supporting queries that calculate geometries on an earth-like sphere.

TTL Indexes **(page 80)** Time-To-Live Indexes.

Text Indexes **(page 82)** Free text indexes on string fields.

3.1 Index Fundamentals

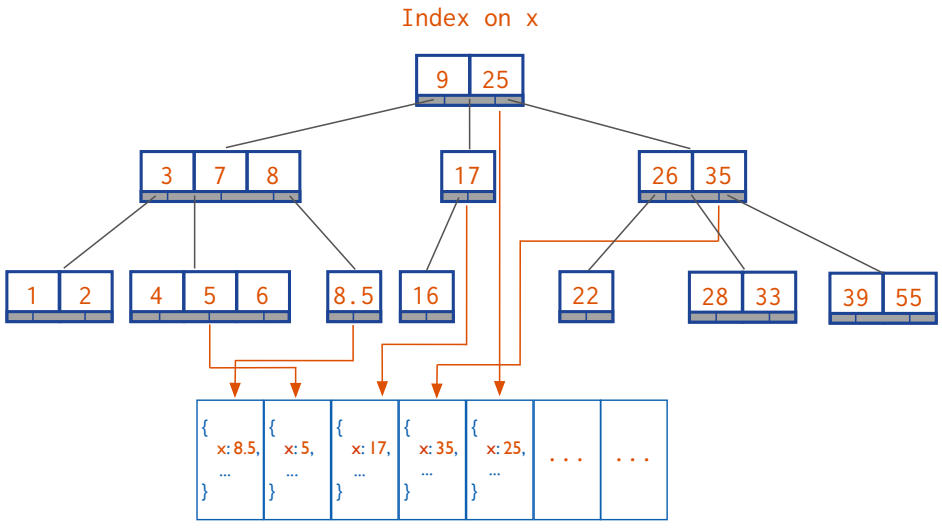
Learning Objectives

Upon completing this module students should understand:

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

Notes:

Why Indexes?



Notes:

Types of Indexes

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

Notes:

Exercise: Using `explain()`

- Let's explore what MongoDB does for the following query by using `explain()`.
- We are projecting only `user.name` so that the results are easy to read.

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

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

Notes:

Results of `explain()`

You will see results similar to the following.

```
{
    "cursor" : "BasicCursor",
    "isMultiKey" : false,
    "n" : 8,
    "nscannedObjects" : 51428,
    "nscanned" : 51428,
    "nscannedObjectsAllPlans" : 51428,
    "nscannedAllPlans" : 51428,
    "scanAndOrder" : false,
    "indexOnly" : false,
    "nYields" : 401,
    "nChunkSkips" : 0,
    "millis" : 161,
    "server" : "new-host-3.home:27017",
    "filterSet" : false
}
```

Notes:

Understanding `explain()` Output

- `n` displays the number of documents that match the query.
- `nscannedObjects` displays the number of documents the retrieval engine considered during the query.
- `nscanned` displays how many documents in an existing index were scanned.
- An `nscanned` value much higher than `nreturned` indicates we need a different index.
- Given `nscannedObjects`, this query will benefit from an index.

Notes:

Single-Field Indexes

- Based on a single field of the documents in a collection
- The field may be a top-level field
- You may also create an index on fields in embedded documents

Notes:

Creating an Index

- The following creates a single-field index on `user.followers_count`.
- `explain()` indicated there will be a substantial performance improvement in handling this type of query.

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

Notes:

Indexes and Read/Write Performance

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

Notes:

Index Limitations

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

Notes:

Use Indexes with Care

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

Notes:

3.2 Compound Indexes

Learning Objectives

Upon completing this module students should understand:

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

Notes:

Introduction to Compound Indexes

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

Notes:

The Order of Fields Matters

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

- Equality tests, e.g.,

```
db.example.find( { a : 15, b : 17 } )
```

- Range queries, e.g.,

```
db.example.find( { a : 15, b : { $lt : 85 } } )
```

- Sorting, e.g.,

```
db.example.find( { a : 15, b : 17 } ).sort( { b : -1 } )
```

Notes:

Designing Compound Indexes

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

Notes:

Example: A Simple Message Board

Requirements:

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

Notes:

Load the Data

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

Notes:

Start with a Simple Index

Start by building an index on { timestamp : 1 }

```
db.messages.ensureIndex( { 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()
```

Notes:

Query Adding `username`

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

Notes:

Include `username` in Our Index

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

Notes:

`ncanned > n`

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

Notes:

A Different Compound Index

Drop the index and build a new one with user.

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

Notes:

`nscanned == n == 2`

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

Notes:

Let Selectivity Drive Field Order

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

Notes:

Adding in the Sort

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

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

Notes:

In-Memory Sorts

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

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

Notes:

Avoiding an In-Memory Sort

Rebuild the index as follows.

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

Notes:

General Rules of Thumb

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

Notes:

3.3 Multikey Indexes

Learning Objectives

Upon completing this module, students should understand:

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

Notes:

Introduction to Multikey Indexes

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

Notes:

Example: Array of Numbers

```
db.testcol.drop()
db.testcol.ensureIndex( { x : 1 } )
a = [ { x : [ 1, 2, 3 ], y : [ "a", "b" ] },
      { x : [ 3, 4 ], y : [ "a", "b" ] },
      { x : [ 4, 5 ], y : [ "a", "b" ] },
      { x : 3, y : [ "a", "b" ] }, { x : 4, y : [ "a", "b" ] } ]
db.testcol.insert( a )
db.testcol.find( { x : 3 } )
db.testcol.find( { x : 3 } ).explain()
db.testcol.find( { "x.2" : 3 } )
db.testcol.find( { "x.2" : 3 } ).explain()
```

Notes:

Exercise: Array of Documents, Part 1

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

```
db.testcol.drop()
b = [ { x : [ { name : "Alice", number : 1 }, { name : "Bob", number : 2 },
              { name : "Cherry", number : 3 } ] },
      { x : [ { name : "Cherry", number : 3 },
              { name : "Dan", number : 4 } ] },
      { x : [ { name : "Dan", number : 4 },
              { name : "Erica", number : 5 } ] },
      { x : { name : "Cherry", number : 3 } },
            { name : "Dan", number : 4 } ]
db.testcol.insert(b)
db.testcol.ensureIndex( { x : 1 } )
db.testcol.find()
```

Notes:

Exercise: Array of Documents, Part 2

For each of the three queries below:

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

```
db.testcol.find( { x : { name : "Cherry", number : 3 } } )
db.testcol.find( { x : { number : 3 } } )
db.testcol.find( { "x.number" : 3 } )
```

Notes:

Exercise: Array of Arrays, Part 1

Add some documents and create an index:

```
db.testcol.drop()
db.testcol.ensureIndex( { x : 1 } )
c = [ { x : [ [ 1, 2 ], [ 2, 3 ], [ 3, 4] ] },
      { x : [ [ 3, 4 ], [ 4, 5 ] ] },
      { x : [ [ 4, 5 ], [ 5, 6 ] ] },
      { x : [ 3, 4 ] },
      { x : [ 4, 5 ] } ]
db.testcol.insert(c)
db.testcol.find()
```

Notes:

Exercise: Array of Arrays, Part 2

For each of the queries below:

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

```
db.testcol.find( { x : [ 3, 4 ] } )
db.testcol.find( { x : 3 } )
db.testcol.find( { "x.1" : [ 4, 5 ] } )
db.testcol.find( { "x.1" : 4 } )
```

Notes:

How Multikey Indexes Work

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

Notes:

Multikey Indexes and Sorting

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

Notes:

Exercise: Multikey Indexes and Sorting

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

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

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

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

Notes:

Limitations on Multikey Indexes

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

Notes:

Example: Multikey Indexes on Multiple Fields

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

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

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

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

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

Notes:

3.4 Hashed Indexes

Learning Objectives

Upon completing this module, students should understand:

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

Notes:

What is a Hashed Index?

- Hashed indexes are based on field values like any other index.
- The difference is that the values are hashed and it is the hashed value that is indexed.
- The hashing function collapses sub-documents and computes the hash for the entire value.
- MongoDB can use the hashed index to support equality queries.
- Hashed indexes do not support multi-key indexes, i.e. indexes on array fields.
- Nor do they support range queries.

Notes:

Why Hashed Indexes?

- In MongoDB, the primary use for hashed indexes is to support sharding a collection using a hashed shard key.
- In some cases, the field we would like to use to shard data would make it difficult to scale using sharding.
- Using a hashed shard key to shard a collection ensures an even distribution of data and overcomes this problem.
- See Shard a Collection Using a Hashed Shard Key[5] for more details.
- We discuss sharding in detail in another module.

Notes:

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

Limitations

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

Notes:

Floating Point Numbers

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

Notes:

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.ensureIndex( { a: "hashed" } )
```

Notes:

3.5 Geospatial Indexes

Learning Objectives

Upon completing this module, students should understand:

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

Notes:

Introduction to Geospatial Indexes

We can use geospatial indexes to quickly determine geometric relationships:

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

Notes:

Easiest to Start with 2 Dimensions

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

Notes:

Location Field

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

Notes:

Find Nearby Documents

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

Notes:

Flat vs. Spherical Indexes

There are two types of geospatial indexes:

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

Notes:

Flat Geospatial Index

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

Notes:

Spherical Geospatial Index

- Knows about 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.
- Uses geoJSON objects (Points, LineString, and Polygons).
- Coordinate pairs are converted into geoJSON Points.

Notes:

Creating a 2d Index

Creating a 2d index:

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

Possible options key-value pairs:

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

Notes:

Exercise: Creating a 2d Index

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

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

Notes:

Inserting Documents with a 2d Index

There are two accepted formats:

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

Notes:

Exercise: Inserting Documents with 2d Fields

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

Notes:

Querying Documents Using a 2d Index

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

Notes:

Example: Find Based on 2d Coords

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

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

Notes:

Creating a 2dsphere Index

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

```
db.collection.ensureIndex( { <location field> : "2dsphere" } )
```

Notes:

The geoJSON Specification

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

Notes:

geoJSON Considerations

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

Notes:

Simple Types of 2dsphere Objects

Point: A single point on the globe

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

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

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

Notes:

Polygons

Simple Polygon:

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

Polygon with One Hole:

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

Notes:

Other Types of 2dsphere Objects

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

Notes:

Exercise: Inserting geoJSON Objects (1)

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

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

Notes:

Exercise: Inserting geoJSON Objects (2)

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

Notes:

Exercise: Inserting geoJSON Objects (3)

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

Notes:

Exercise: Creating a 2dsphere Index

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

Notes:

Querying 2dsphere Objects

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

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

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

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

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

Notes:

3.6 TTL Indexes

Learning Objectives

Upon completing this module students should understand:

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

Notes:

TTL Index Basics

- TTL is short for "Time To Live".
- This must index a field of type "Date" (including ISODate) or "Timestamp"
- Any Date field older than expireAfterSeconds will get deleted at some point

Notes:

Creating a TTL Index

Create with:

```
db.collection.ensureIndex( { field_name : 1 },
                           { expireAfterSeconds : some_number } )
```

Notes:

Exercise: Creating a TTL Index

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

```
db.testcol.drop()
db.testcol.ensureIndex( { a : 1 }, { expireAfterSeconds : 30 } )

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

Notes:

Exercise: Check the Collection

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

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

Notes:

3.7 Text Indexes

Learning Objectives

Upon completing this module, students should understand:

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

Notes:

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”)

Notes:

Creating a Text Index

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

```
db.collection.ensureIndex( { <field name> : "text" } )
```

Notes:

Exercise: Creating a Text Index

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

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

Notes:

Text Indexes are Similar to Multikey Indexes

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

Notes:

Exercise: Inserting Texts

Let's add some documents to our montyPython collection.

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

Notes:

Querying a Text Index

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

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

Notes:

Exercise: Querying a Text Index

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

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

Notes:

Exercise: Querying Using Two Words

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

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

Notes:

Search for a Phrase

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

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

Notes:

Text Search Score

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

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

Notes:

4 Aggregation

Aggregation Tutorial **(page 86)** An introduction to the the aggregation framework, pipeline concept, and stages.

Optimizing Aggregation **(page 98)** Resource management in the aggregation pipeline.

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

Notes:

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.

Notes:

The Aggregation Pipeline

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

```
db.collection.aggregate( [ { stage1 }, { stage2 }, ... ],
                         { options } )
```

Notes:

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)

Notes:

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.

Notes:

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 } } } ] )
```

Notes:

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.

Notes:

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 } } ] )
```

Notes:

Exercise: Renaming fields with $project

Use the $project operator to rename a field

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

Notes:

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[6] in another lesson.

Notes:

[6] 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.

Notes:

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,
        ...
    },
}
```

Notes:

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

Notes:

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.

Notes:

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 } ] )
```

Notes:

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:

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

Notes:

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.

Notes:

Group using $avg

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

Notes:

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 }
] )
```

Notes:

Group Aggregation Operators

The complete list of operators available in the group stage:

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

Notes:

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.

Notes:

Process

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

Notes:

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 } }
] )
```

Notes:

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.

Notes:

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.

Notes:

Example: User Mentions in a Tweet

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

Notes:

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 })
```

Notes:

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.

Notes:

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.

Notes:

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 }
] )
```

Notes:

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 } }])

Notes:

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 }, ...])

Notes:

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 }, ...])

Notes:

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” }

Notes:

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.

Notes:

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.

Notes:

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.

Notes:

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.

Notes:

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.

Notes:

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.

Notes:

Automatic Optimizations

MongoDB will perform some optimizations automatically. For example:

- If a `$project` stage is used late in the pipeline it may be used to eliminate those fields earlier if possible.
- A `$sort` followed by a `$match` will be executed as a `$match` followed by a `$sort` to reduce the number of documents to be sorted.
- A `$skip` followed by a `$limit` will be executed as a `$limit` followed by a `$skip`, with the $limit parameter increased by the $skip amount to allow $sort + $limit coalescence.
- See: Aggregation Pipeline Optimation[7]

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

Notes:

5 Schema Design

Schema Design Core Concepts **(page 102)** An introduction to schema design in MongoDB.

Schema Evolution **(page 109)** Considerations for evolving a MongoDB schema design over an application's lifetime.

Common Schema Design Patterns **(page 114)** Common design patterns for representing 1-1, 1-M, and M-M relationships and tree structures in MongoDB.

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

```
{   _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: 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

```
{   _id: 1,
    title: "The Great Gatsby",
    slug: "9781857150193-the-great-gatsby",
    author: 1,
    available: true,
    isbn: "9781857150193",
    pages: 176,
    publisher: { publisher_name: "Everyman's Library",
                 date: ISODate("1991-09-19T00:00:00Z"),
                 publisher_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..." } ]
}
```

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

```
{   _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..." } ]
}
```

Array of Documents

```
{   _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..." } ]
}
```

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") } ]
}
```

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

Notes:

Development Phase

Support basic CRUD functionality:

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

Notes:

Development Phase: Known Query Patterns

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

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

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

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

Notes:

Production Phase

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

Notes:

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 }});
```

Notes:

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 })
```

Notes:

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

Notes:

Exercise: Recent Reviews

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

Notes:

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")
        }
    ]
}
```

Notes:

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 }}
);
```

Notes:

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() &amp;amp;&amp;amp; num &amp;lt; 10) {
    doc = cursor.next();

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

Notes:

Solution: Recent Reviews, Delete

Deleting a review

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

Notes:

5.3 Common Schema Design Patterns

Learning Objectives

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

- 1-1 Relationships
- 1-M Relationships
- M-M Relationships
- Tree Structures

Notes:

1-1 Relationship

Let's pretend that authors only write one book.

Notes:

1-1: 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,
}
```

Notes:

1-1: Embedding

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

Notes:

1-M Relationship

In reality, authors may write multiple books.

Notes:

1-M: 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]
}
```

Notes:

1-M: 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...
}
```

Notes:

1-M: 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...
}
```

Notes:

M-M Relationship

Some books may also have co-authors.

Notes:

M-M: 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]
}
```

Notes:

M-M: 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 });
```

Notes:

M-M: 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"
}
```

Notes:

M-M: 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 }});
```

Notes:

Tree Structures

E.g., modeling a subject hierarchy.

Notes:

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

Notes:

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"
}
```

Notes:

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"
}
```

Notes:

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.

Notes:

6 Replica Sets

Introduction to Replica Sets **(page 121)** An introduction to replication and replica sets.

Elections in Replica Sets **(page 125)** The process of electing a new primary (automated failover) in replica sets.

Replica Set Roles and Configuration **(page 131)** Configuring replica set members for common use cases.

The Oplog: Statement Based Replication **(page 133)** The process of replicating data from one node of a replica set to another.

Write Concern **(page 139)** Balancing performance and durability of writes.

Read Preference **(page 144)** Configuring clients to read from specific members of a replica set.

Exercise: Setting up a Replica Set **(page 146)** Launching members, configuring, and initiating 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

Notes:

Use Cases for Replication

- High Availability
- Disaster Recovery
- Functional Segregation

Notes:

High Availability (HA)

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

Notes:

Disaster Recovery (DR)

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

Notes:

Functional Segregation

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

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

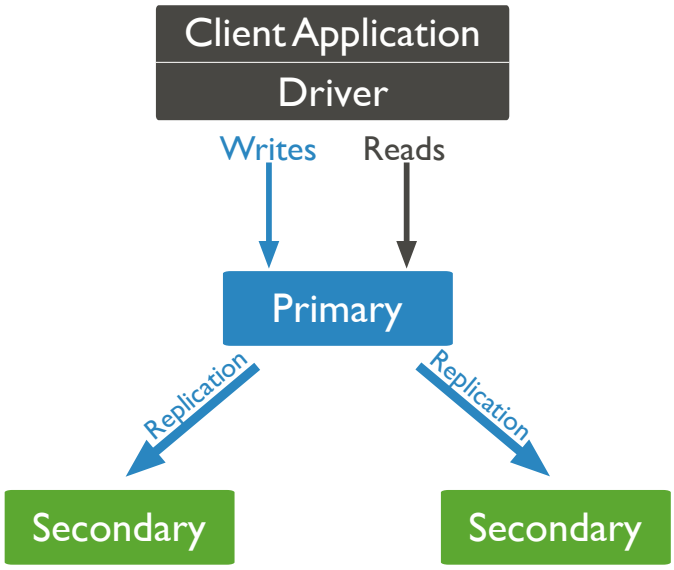
Notes:

Replication is Not Designed for Scaling

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

Notes:

Replica Sets



Notes:

Primary Server

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

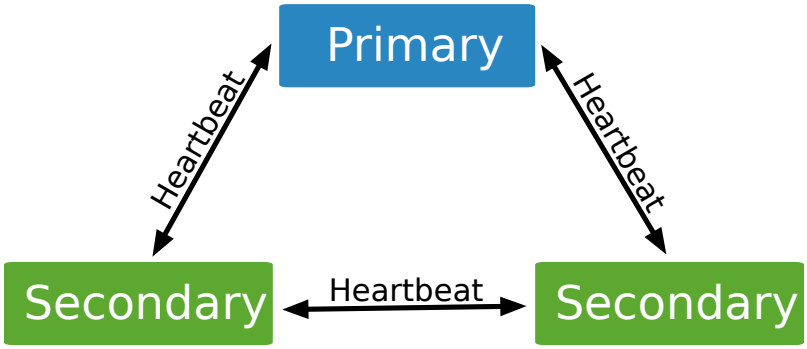
Notes:

Secondaries

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

Notes:

Heartbeats



Notes:

The Oplog

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

Notes:

6.2 Elections in Replica Sets

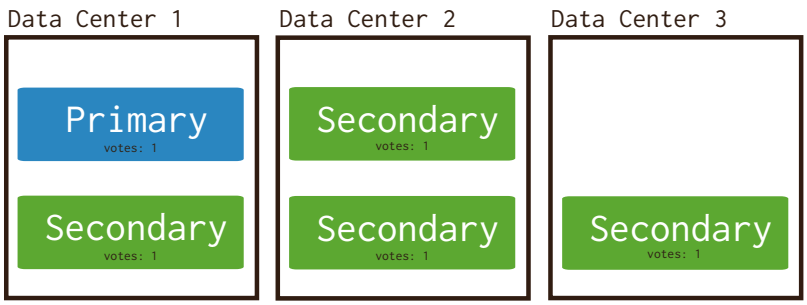
Learning Objectives

Upon completing this module students should understand:

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

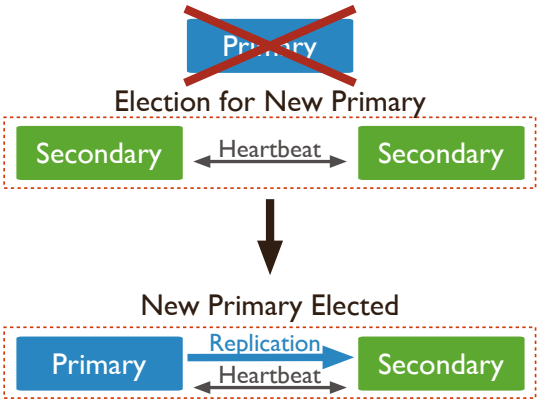
Notes:

Members and Votes



Notes:

Calling Elections



Notes:

Selecting a New Primary

Three factors are important in the selection of a primary:

- Priority
- Optime
- Connections

Notes:

Priority

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

Notes:

Optime

- Optime: Operation time, which is the timestamp of the last operation the member applied from the oplog.
- To be elected primary, a member must have the most recent optime.
- Only optimes of visible members are compared.

Notes:

Connections

- Must be able to connect to a majority of the members in the replica set.
- Majority refers to the total number of votes.
- Not the total number of members.

Notes:

When will a primary step down?

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

Notes:

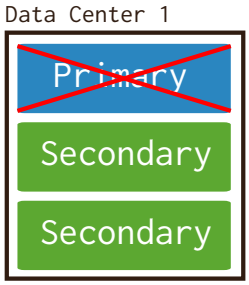
Exercise: Elections in Failover Scenarios

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

Notes:

Scenario A: 3 Data Nodes in 1 DC

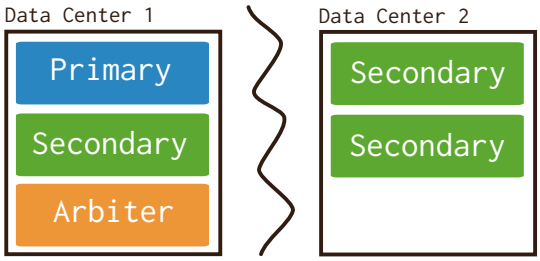
Which secondary will become the new primary?



Notes:

Scenario B: 3 Data Nodes in 2 DCs

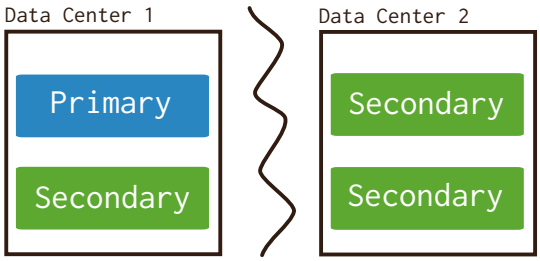
Which member will become primary following this type of network partition?



Notes:

Scenario C: 4 Data Nodes in 2 DCs

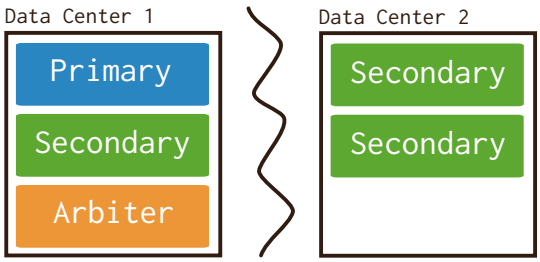
What happens following this network partition?



Notes:

Scenario D: 5 Nodes in 2 DCs

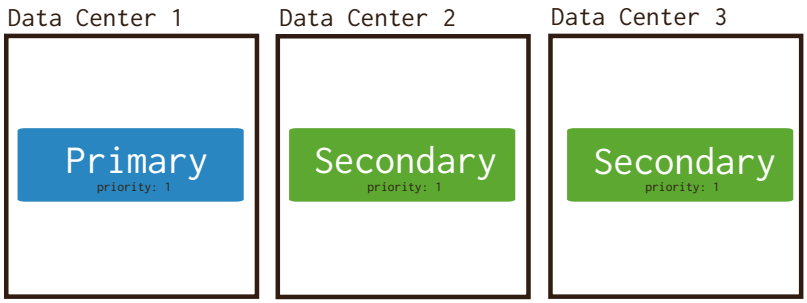
The following is similar to Scenario C, but with the addition of an arbiter in Data Center 1. What happens here?



Notes:

Scenario E: 3 Data Nodes in 3 DCs

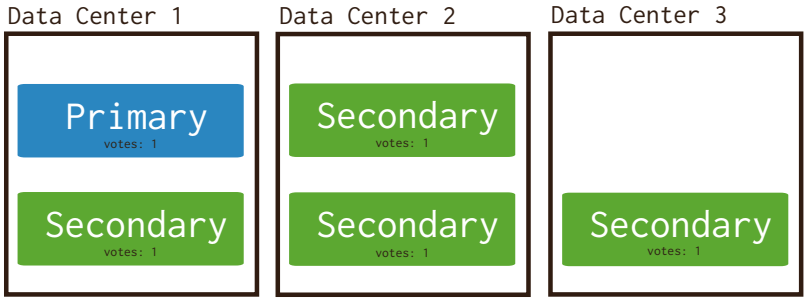
- What happens here if any one of the nodes/DCs fail?
- What about recovery time?



Notes:

Scenario F: 5 data nodes in 3 DCs

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



Notes:

6.3 Replica Set Roles and Configuration

Learning Objectives

Upon completing this module students should understand:

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

Notes:

Example: A Five-Member Replica Set Configuration

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

Notes:

Configuration

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

Notes:

Principal Data Center

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

Notes:

Data Center 2

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

Notes:

What about `dc1-3` and `dc2-2`?

```
// Both are hidden.
// Clients will not distribute reads to hidden members.
// We use hidden members for dedicated tasks.
{ _id : 3, host : "dc1-3.example.net:27017", hidden : true },
{ _id : 4, host : "dc2-2.example.net:27017", hidden : true,
  slaveDelay: 7200 }
```

Notes:

What about `dc2-2`?

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

Notes:

6.4 The Oplog: Statement Based Replication

Learning Objectives

Upon completing this module students should understand:

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

Notes:

Binary Replication

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

Notes:

Tradeoffs

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

Notes:

Statement-Based Replication

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

Notes:

Example

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

```
db.foo.remove({ age : 30 })
```

This will be represented in the oplog with records such as the following:

```
{ "ts" : Timestamp(1407159845, 5), "h" : NumberLong("-704612487691926908"),
  "v" : 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 65 } }
{ "ts" : Timestamp(1407159845, 1), "h" : NumberLong("6014126345225019794"),
  "v" : 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 333 } }
{ "ts" : Timestamp(1407159845, 4), "h" : NumberLong("8178791764238465439"),
  "v" : 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 447 } }
{ "ts" : Timestamp(1407159845, 3), "h" : NumberLong("-1707391001705528381"),
  "v" : 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 1033 } }
{ "ts" : Timestamp(1407159845, 2), "h" : NumberLong("-6814297392442406598"),
  "v" : 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 9971 } }
```

Notes:

Replication Based on the Oplog

- One statement per document affected by each write: insert, update, or delete.
- Provides a level of abstraction that enables independence among the members of a replica set:
  - With regard to MongoDB version.
  - In terms of how data is stored on disk.
  - Freedom to do maintenance without the need to bring the entire set down.

Notes:

Create a Replica Set

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

```
mongo --nodb
```

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

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

Notes:

`ReplSetTest`

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

Notes:

Start the Replica Set

Start the mongod processes for this replica set.

```
replicaSet.startSet()
```

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

```
replicaSet.initiate()
```

Notes:

Status Check

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

Notes:

Connect to the Primary

Open a new shell, connecting to the primary.

```
mongo --port 31000
```

Notes:

Create some Inventory Data

Use the `store` database:

```
use store
```

Add the following inventory:

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

Notes:

Perform an Update

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

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

Notes:

View the Oplog

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

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

Notes:

Operations in the Oplog are Idempotent

- Each operation in the oplog is idempotent.
- Whether applied once or multiple times it produces the same result.
- Necessary if you want to be able to copy data while simultaneously accepting writes.

Notes:

The Oplog Window

- Oplogs are capped collections.
- Capped collections are fixed-size.
- They guarantee preservation of insertion order.
- They support high-throughput operations.
- Like circular buffers, once a collection fills its allocated space:
  - It makes room for new documents.
  - By overwriting the oldest documents in the collection.

Notes:

Sizing the Oplog

- The oplog should be sized to account for latency among members.
- The default size oplog is usually sufficient.
- But you want to make sure that your oplog is large enough:
  - So that the oplog window is large enough to support replication
  - To give you a large enough history for any diagnostics you might wish to run.

Notes:

6.5 Write Concern

Learning Objectives

Upon completing this module students should understand:

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

Notes:

What happens to the write?

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

Notes:

Answer

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

Notes:

Balancing Durability with Performance

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

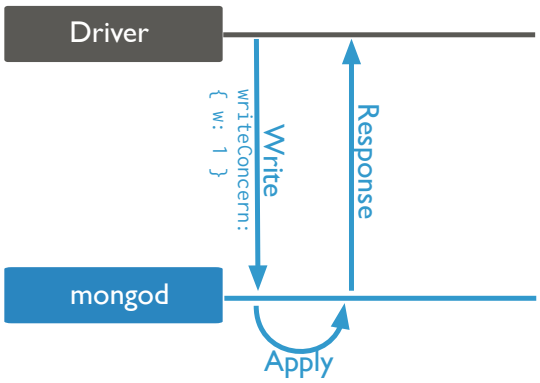
Notes:

Defining Write Concern

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

Notes:

Write Concern: `{ w : 1 }`



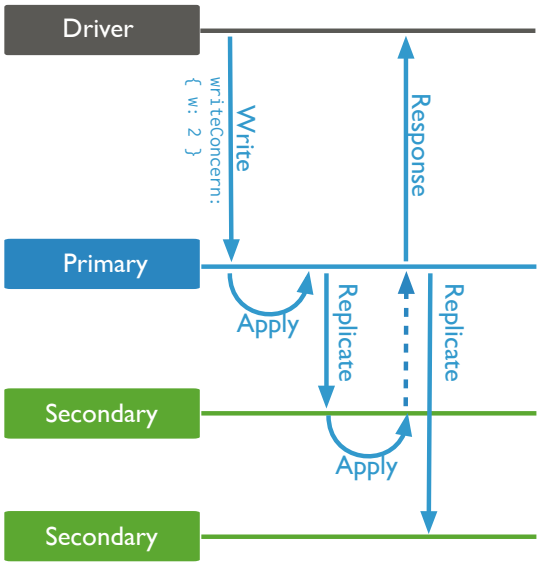
Notes:

Example: { w : 1 }

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

Notes:

Write Concern: { w : 2 }



Notes:

Example: { w : 2 }

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

Notes:

Other Write Concerns

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

Notes:

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

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

Notes:

Example: `{ w : "majority" }`

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

Notes:

Quiz: Which write concern?

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

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

Notes:

Further Reading

See Write Concern Reference[8] for more details on write concern configurations, including setting timeouts and identifying specific replica set members that must acknowledge writes (i.e. tag sets[9]).

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

6.6 Read Preference

What is Read Preference?

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

Notes:

Use Cases

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

Notes:

Not for Scaling

- In general, do *not* read from secondaries to provide extra capacity for reads.
- Sharding[10] increases read and write capacity by distributing operations across a group of machines.
- Sharding is a better strategy for adding capacity.

Notes:

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

Read Preference Modes

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

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

Notes:

Tag Sets

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

For example, in the mongo shell:

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

Notes:

6.7 Exercise: Setting up a Replica Set

Overview

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

Notes:

Create Data Directories

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

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

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

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

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

Notes:

Launch Each Member

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

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

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

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

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

Notes:

Status

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

Notes:

Connect to a MongoDB Instance

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

```
mongo --port 27017
```

Notes:

Configure the Replica Set

Note the port number of the primary from the output of `rs.status()`, which we'll need for the next step.

```
var config = {
  _id: "mySet",
  members: [
    { _id: 0, host: "localhost:27017" },
    { _id: 1, host: "localhost:27018" },
    { _id: 2, host: "localhost:27019" }
  ]
}
rs.initiate(config)

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

exit    # or Ctrl-d
```

Notes:

Write to the Primary

Connect to the primary with mongo shell by issuing a command such as the following:

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

Now insert a simple test document via mongo shell. Once the insert succeeds, exit the mongo shell.

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

Notes:

Read from a Secondary

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

```
mongo --port <NON_PRIMARY_PORT>   # e.g. 27018
```

Read from the secondary

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

Notes:

Review the Oplog

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

Notes:

Changing Replica Set Configuration

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

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

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

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

Notes:

Verifying Configuration Change

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

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

Verify that the replica set configuration is now as expected:

```
rs.conf()
```

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

```
rs.status()
```

Notes:

Further Reading

- Replica Configuration[11]
- Replica States[12]

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

7 Sharding

Introduction to Sharding **(page 150)** An introduction to sharding.

Balancing Shards **(page 160)** Chunks, the balancer, and their role in a sharded cluster.

Shard Tags **(page 163)** How tag-based sharding works.

Exercise: Setting Up a Sharded Cluster **(page 165)** Deploying a sharded cluster.

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

Notes:

Contrast with Replication

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

Notes:

Sharding is Concerned with Scale

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

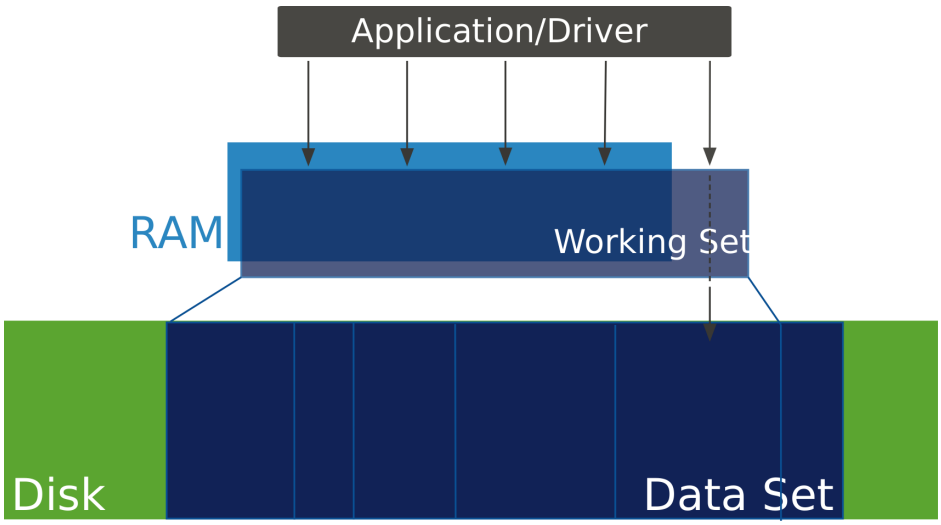
Notes:

Vertical Scaling

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

Notes:

The Working Set



Notes:

Limitations of Vertical Scaling

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

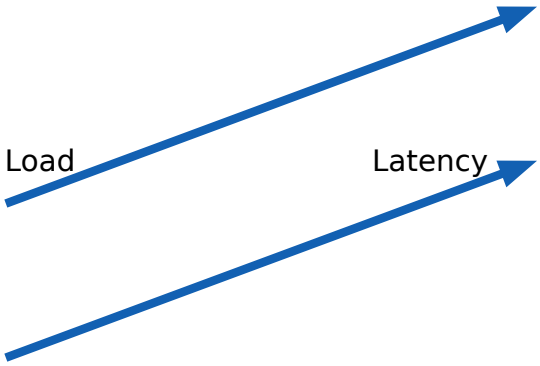
Notes:

Sharding Overview

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

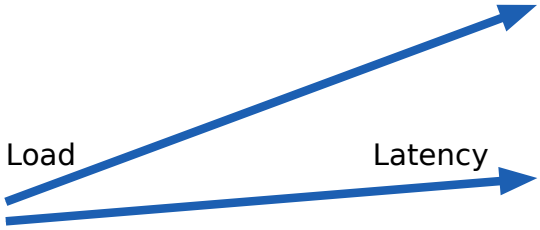
Notes:

A Model that Does Not Scale



Notes:

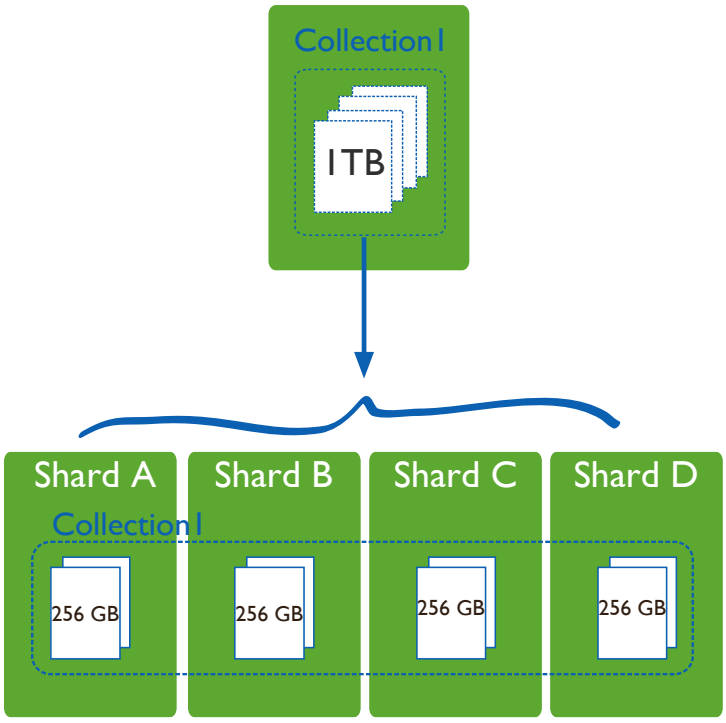
A Scalable Model



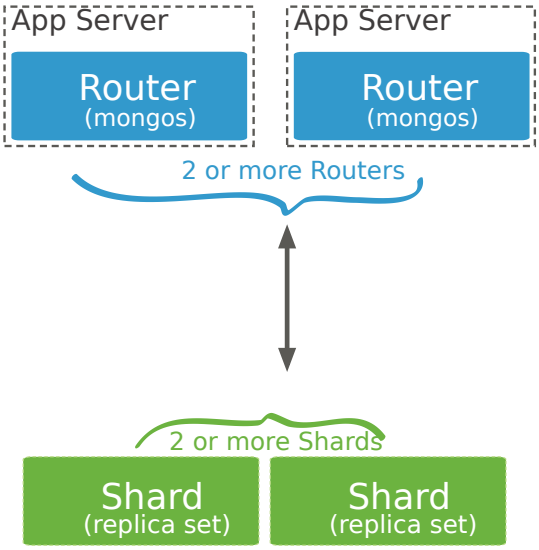
Notes:

Sharding Basics

Notes:



Sharded Cluster Architecture



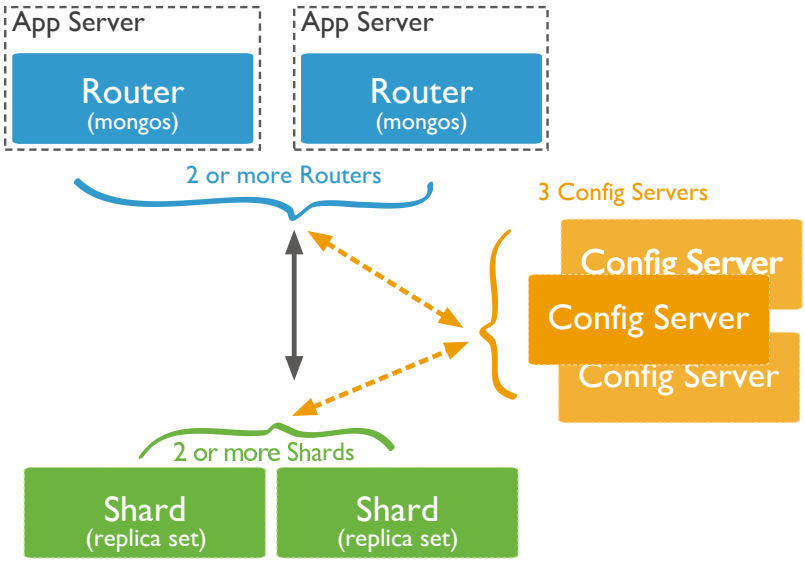
Notes:

Mongos

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

Notes:

Config Servers



Notes:

Config Server Hardware Requirements

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

Notes:

When to Shard

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

Notes:

Possible Imbalance?

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

Notes:

Balancing Shards

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

Notes:

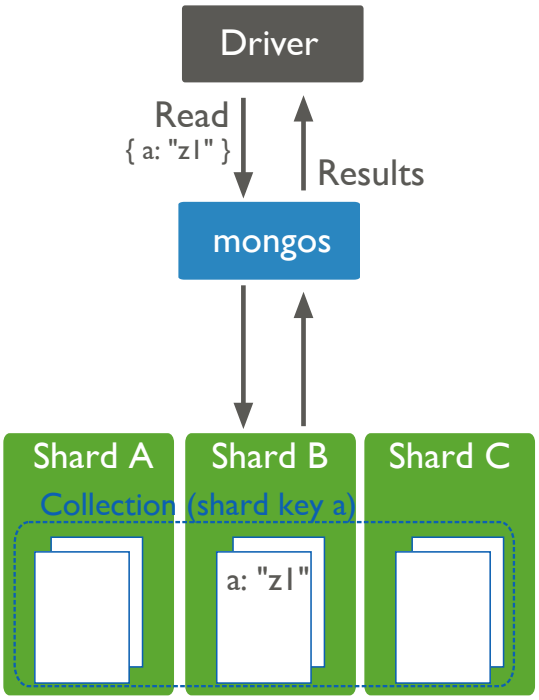
What is a Shard Key?

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

Notes:

Targeted Query Using Shard Key

Notes:



With a Good Shard Key

You might easily see that:

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

Notes:

With a Bad Shard Key

You might see that:

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

Notes:

Choosing a Shard Key

Generally, you want a shard key:

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

Notes:

More Specifically

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

Notes:

Cardinality

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

Notes:

Non-Monotonic

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

Notes:

Shards Should be Replica Sets

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

Notes:

7.2 Balancing Shards

Learning Objectives

Upon completing this module students should understand:

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

Notes:

Chunks and the Balancer

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

Notes:

Chunks in a Newly Sharded Collection

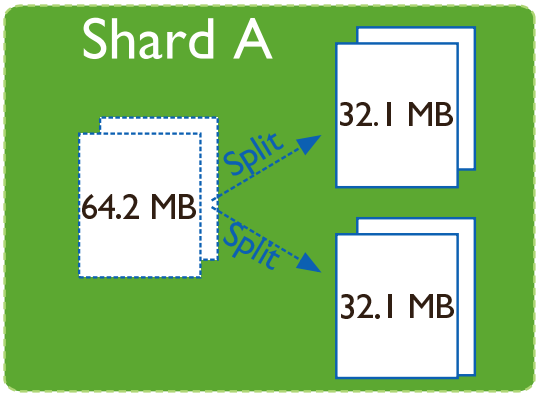
- The range of a chunk is defined by the shard key values of the documents the chunk contains.
- When a collection is sharded it starts with just one chunk.
- The first chunk for a collection will have the range:

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

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

Notes:

Chunk Splits



Notes:

Pre-Splitting Chunks

- You may pre-split data before loading data into a sharded cluster.
- Pre-splitting is useful if:
  - You plan to do a large data import early on
  - You expect a heavy initial server load and want to ensure writes are distributed.

Notes:

Start of a Balancing Round

- A balancing round may be initiated by any mongos in the cluster.
- This happens when the difference in the number of chunks between two shards becomes to large.
- Specifically, the difference between the shard with the most chunks and the shard with the fewest.
- A balancing round starts when the imbalance reaches:
  - 2 when the cluster has < 20 chunks
  - 4 when the cluster has 20-79 chunks
  - 8 when the cluster has 80+ chunks

Notes:

Balancing is Resource Intensive

- Chunk migration requires copying all the data in the chunk from one shard to another.
- MonogDB can migrate only a single chunk at a time.
- MongoDB creates splits only after an insert operation.
- For these reasons, it is possible to define a balancing window to ensure the balancer will only run during scheduled times.

Notes:

Chunk Migration Steps

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

Notes:

Concluding a Balancing Round

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

Notes:

7.3 Shard Tags

Learing Objectives

Upon completing this module students should understand:

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

Notes:

Tags - Overview

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

Notes:

Example: DateTime

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

Notes:

Example: Location

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

Notes:

Example: Premium Tier

- You have customers who want to pay for a "premium" tier.
- The shard key permits you to distinguish one customer's documents from all others.
- Tag the document ranges for each customer so that their documents will be located on shards of the appropriate tier.
- Shards tagged as premium tier run on high performance servers.
- Other shards run on commodity hardware.
- See Manage Shard Tags[13]

Notes:

Tags - Caveats

- Because tagged chunks will only be on certain servers, if you tag more than those servers can handle, you'll have a problem.
  - You're not only worrying about your overall server load, you're worrying about server load for each of your tags.
- Your chunks will evenly distribute themselves across the available chunks. You cannot control things more fine grained than your tags.

Notes:

7.4 Exercise: Setting Up a Sharded Cluster

Learning Objectives

Upon completing this module students should understand:

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

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

Notes:

Our Sharded Cluster

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

Notes:

Sharded Cluster Configuration

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

Notes:

Build Our Data Directories

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

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

On Windows, run the following commands instead:

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

```
md c:\data\cluster\shard2\m0 c:\data\cluster\shard2\m1 c:\data\cluster\shard2\arb
md c:\data\cluster\s0 c:\data\cluster\s1
```

Notes:

Initiate a Replica Set

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

mongod --replSet shard1 --smallfiles --nojournal --noprealloc \
       --dbpath ~/data/cluster/shard0/m1 --logpath ~/data/cluster/shard0/m1/mongod.log \
       --fork --port 27108

mongod --replSet shard0 --smallfiles --nojournal --noprealloc \
       --dbpath ~/data/cluster/shard0/arb --logpath ~/data/cluster/shard0/arb/mongod.log \
       --fork --port 27109

echo "cfg = {'_id': 'shard0', 'version': 1,
             'members': [{'_id' : 0, 'host' : 'localhost:27107'},
                         {'_id': 1, 'host': 'localhost:27108' },
                         {'_id': 2, 'host': 'localhost:27109', 'arbiterOnly': true}]};
     rs.initiate(cfg);" | mongo --port 27107
```

Notes:

Spin Up a Second Replica Set

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

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

mongod --replSet shard1 --smallfiles --nojournal --noprealloc \
       --dbpath ~/data/cluster/shard1/arb --logpath ~/data/cluster/shard1/arb/mongod.log \
       --fork --port 27119

echo "cfg = {'_id': 'shard1', 'version': 1,
             'members': [{'_id': 0, 'host': 'localhost:27117'},
                         {'_id': 1, 'host': 'localhost:27118'},
                         {'_id': 2, 'host': 'localhost:27119', 'arbiterOnly': true}]};
     rs.initiate(cfg);" | mongo --port 27117
```

Notes:

A Third Replica Set

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

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

mongod --replSet shard2 --smallfiles --nojournal --noprealloc \
       --dbpath ~/data/cluster/shard2/arb --logpath ~/data/cluster/shard2/arb/mongod.log \
       --fork --port 27129

echo "cfg = {'_id': 'shard2', 'version': 1,
             'members': [{'_id': 0, 'host': 'localhost:27127'},
                         {'_id': 1, 'host': 'localhost:27128'},
                         {'_id': 2, 'host': 'localhost:27129', 'arbiterOnly': true}]};
     rs.initiate(cfg);" | mongo --port 27127
```

Notes:

Status Check

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

Notes:

Launch Config Servers

```
mongod --smallfiles --nojournal --noprealloc \
       --dbpath $CONFIG/c0 --logpath ~/data/cluster/config/c0/mongod.log \
       --fork --port 27227 --configsvr

mongod --smallfiles --nojournal --noprealloc \
       --dbpath $CONFIG/c1 --logpath ~/data/cluster/config/c1/mongod.log \
       --fork --port 27228 --configsvr

mongod --smallfiles --nojournal --noprealloc \
       --dbpath $CONFIG/c2 --logpath ~/data/cluster/config/c2/mongod.log \
       --fork --port 27229 --configsvr
```

Notes:

Launch the Mongos Processes

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

```
mongos --logpath ~/data/cluster/s0/mongos.log --fork --port 27017 \
       --configdb localhost:27227,localhost:27228,localhost:27229

mongos --logpath ~/data/cluster/s1/mongos.log --fork --port 27018 \
       --configdb localhost:27227,localhost:27228,localhost:27229
```

Notes:

Add All Shards

```
echo 'sh.addShard( "shard0/localhost:27107" ); sh.addShard("shard1/localhost:27117" );
     sh.addShard( "shard2/localhost:27127" ); sh.status()' | mongo
```

Note: Instead of doing this through a bash (or other) shell command, you may prefer to launch a mongo shell and issue each command individually.

Notes:

Enable Sharding and Shard a Collection

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

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

echo 'for (i=0; i<10000; i++) { docArr = []; for (j=0; j<1000; j++) {
     docArr.push( { a : i, b : j, c : "Filler String 000000000000000000000
     000000000000000000000000000000000000000000000000000000000000000000000
     000000000000000" } )  }; db.testcol.insert(docArr) }' | mongo
```

Notes:

Observe What Happens

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

```
sh.status()
```

Notes:

8 Security

Security **(page 171)** An overview of security options for MongoDB.

8.1 Security

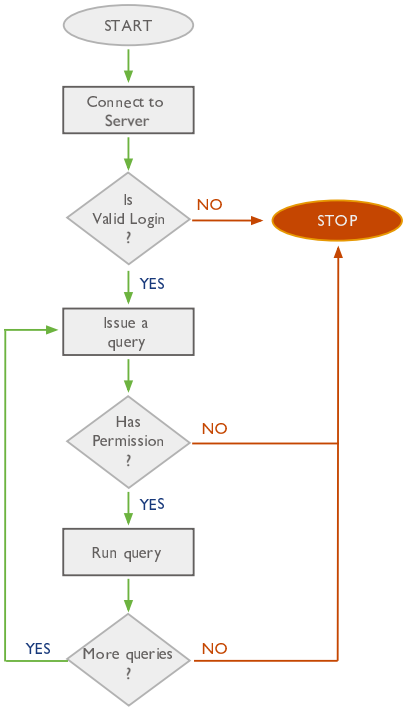
Learning Objectives

Upon completing this module students should understand:

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

Notes:

Overview



Notes:

Authentication Options

- MONGODB-CR Authentication (username & password)
- x.509 Authentication (using x.509 Certificates)
- Kerberos (through an Enterprise subscription)
- LDAP

Notes:

Authorization via MongoDB

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

Notes:

Network Exposure Options

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

Notes:

Encryption (SSL)

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

Notes:

Native MongoDB Auth

- Uses the Challenge/Response mechanism
- Sometimes called MongoDB-CR
- Start a mongod instance with `--auth` to enable this feature
- You can initially login using localhost
  - Called the “localhost exception”.
  - Stops working when you create a user.

Notes:

Exercise: Create an Admin User, Part 1

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

Notes:

Exercise: Create an Admin User, Part 2

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

Notes:

Using MongoDB Roles

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

Notes:

Exercise: Creating a `readWrite` User, Part 1

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

Notes:

Exercise: Creating a `readWrite` User, Part 2

Log in with the user you just created.

Notes:

MongoDB Custom User Roles

- You can create custom user roles in MongoDB.
- You do this by modifying the system.roles collection.
- You can also inherit privileges from other roles into a given role.
- You won't remember how to do this, so if you need it, consult the docs[14].

Notes:

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

9 Reporting Tools and Diagnostics

Performance Troubleshooting **(page 176)** An introduction to reporting and diagnostic tools for MongoDB.

9.1 Performance Troubleshooting

Learning Objectives

Upon completing this module students should understand basic performance troubleshooting techniques and tools including:

- `mongostat`
- `mongotop`
- `db.setProfilingLevel()`
- `db.currentOp()`
- `db.collection.stats()`
- `db.serverStatus()`

Notes:

`mongostat` and `mongotop`

- `mongostat` samples a server every second.
  - See current ops, pagefaults, network traffic, etc.
  - Does not give a view into historic performance; use MMS for that.
- `mongotop` looks at the time spent on reads/writes in each collection.

Notes:

Exercise: `mongostat` (setup)

In one window, perform the following commands.

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    arr = [];
    for (j=1; j<=1000; j++) {
        doc = { _id : (1000 * (i-1) + j), a : i, b : j, c : (1000 * (i-1)+ j)};
        arr.push(doc)
    };
    db.testcol.insert(arr);
    var x = db.testcol.find( { b : 255 } );
    x.next();
    var x = db.testcol.find( { _id : 1000 * (i-1) + 255 } );
    x.next();
    var x = "asdf";
    db.testcol.update( { a : i, b : 255 }, { $set : { d : x.pad(1000) } });
    print(i)
}
```

Notes:

Exercise: `mongostat` (run)

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

Notes:

Exercise: `mongostat` (create index)

- In a third window, create an index when you see things slowing down:

```
db.testcol.ensureIndex( { a : 1, b : 1 } )
```

- Look at mongostat.
- Notice that things are going significantly faster.
- Then, let's drop that and build another index.

```
db.testcol.dropIndexes()
db.testcol.ensureIndex( { b : 1, a : 1 } )
```

Notes:

Exercise: `mongotop`

Perform the following then, in another window, run `mongotop`.

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    arr = [];
    for (j=1; j<=1000; j++) {
        doc = {_id: (1000*(i-1)+j), a: i, b: j, c: (1000*(i-1)+j)};
        arr.push(doc)
    };
    db.testcol.insert(arr);
    var x = db.testcol.find( {b: 255} ); x.next();
    var x = db.testcol.find( {_id: 1000*(i-1)+255} ); x.next();
    var x = "asdf";
    db.testcol.update( {a: i, b: 255}, {$set: {d: x.pad(1000)}});
    print(i)
}
```

Notes:

db.currentOp()

- currentOp is a tool that asks what the db is doing at the moment.
- currentOp is useful for finding long-running processes.
- Fields of interest:
  - microsecs_running
  - op
  - query
  - lock
  - waitingForLock

Notes:

Exercise: db.currentOp()

Do the following then, connect with a separate shell, and repeatedly run db.currentOp().

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    arr = [];
    for (j=1; j<=1000; j++) {
        doc = {_id: (1000*(i-1)+j), a: i, b: j, c: (1000*(i-1)+j)};
        arr.push(doc)
    };
    db.testcol.insert(arr);
    var x = db.testcol.find( {b: 255} ); x.next();
    var x = db.testcol.find( {_id: 1000*(i-1)+255 }); x.next();
    var x = "asdf";
    db.testcol.update( {a: i, b: 255}, {$set: {d: x.pad(1000)}});
    print(i)
}
```

Notes:

db.collection.stats()

- Used to view the current stats for a collection.
- Everything is in bytes; use the multiplier parameter to view in KB, MB, etc
- You can also use db.stats() to do this at scope of the entire database

Notes:

Exercise: Using Collection Stats

Look at the output of the following:

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

Notes:

The Profiler

- Off by default.
- To reset, `db.setProfilerLevel(0)`
- At setting 1, it captures "slow" queries.
- You may define what "slow" is.
- Default is 100ms: `db.setProfilerLevel(1)`
- E.g., to capture 20 ms: `db.setProfilerLevel(1, 20)`

Notes:

The Profiler (continued)

- If the profiler level is 2, it captures all queries.
  - This will severely impact performance.
  - Turns all reads into writes.
- Always turn the profiler off when done (set level to 0)
- Creates `db.system.profile` collection

Notes:

Exercise: Exploring the Profiler

Perform the following, then look in your db.system.profile.

```
db.setProfilingLevel(0)
db.testcol.drop()
db.system.profile.drop()
db.setProfilingLevel(2)
db.testcol.insert( { a : 1 } )
db.testcol.find()
var x = "asdf"
db.testcol.insert( { a : x.pad(10000000) } )  // ~10 MB
db.setProfilingLevel(0)
db.system.profile.find().pretty()
```

Notes:

`db.serverStatus()`

- Takes a snapshot of server status.
- By taking diffs, you can see system trends.
- Most of the data that MMS gets is from here.

Notes:

Exercise: Using `db.serverStatus()`

- Open up two windows. In the first, type:

```
db.testcol.drop()
var x = "asdf"
for (i=0; i<=10000000; i++) {
    db.testcol.insert( { a : x.pad(100000) } )
}
```

- In the second window, type periodically:

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

Notes:

Analyzing profiler data

- Enable the profiler at default settings.
- Run for 5 seconds.
- Slow operations are captured.
- The issue is there is not a proper index on the message field.
- Allow class to discover this as the data is examined.
- You will see how fast documents are getting inserted.
- It will be slow b/c the documents are big.

Notes:

Performance Improvement Techniques

- Appropriate write concerns
- Bulk operations
- Good schema design
- Good Shard Key choice
- Good indexes

Notes:

Performance Tips: Write Concern

- Increasing the write concern increases data safety.
- This will have an impact on performance, however.
- This is especially true when there are network issues.

Notes:

Bulk Operations

- Using bulk operations can improve performance, especially when using write concern greater than 1.
- These enable the server to bulk write and bulk acknowledge.
- Can be done with both inserts and updates.

Notes:

Exercise: Comparing bulk inserts with `mongostat`

Let's spin up a 3-member replica set:

```
mkdir -p /data/replset/{1,2,3}
mongod --logpath /data/replset/1/mongod.log \
--dbpath /data/replset/1 --replSet mySet --port 27017 --fork
mongod --logpath /data/replset/2/mongod.log \
--dbpath /data/replset/2 --replSet mySet --port 27018 --fork
mongod --logpath /data/replset/3/mongod.log \
--dbpath /data/replset/3 --replSet mySet --port 27019 --fork

echo "conf = {_id: 'mySet', members: [{_id: 0, host: 'localhost:27017'}, {_id: 1, host: 'localhost:2
     {_id: 2, host: 'localhost:27019'}]}; rs.initiate(conf)" | mongo
```

Notes:

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

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

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    for (j=1; j<=1000; j++) {
        db.testcol.insert( { _id : (1000 * (i-1) + j),
                             a : i, b : j, c : (1000 * (i-1)+ j) },
                           { writeConcern : { w : 1 } } );
    };
    print(i);
}
```

Run `mongostat` and see how fast that happens.

Notes:

Bulk inserts with `{w: 3}`

Increase the write concern to 3 (safer but slower):

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    for (j=1; j<=1000; j++) {
        db.testcol.insert(
            { _id: (1000 * (i-1) + j), a: i, b: j, c: (1000 * (i-1)+ j)},
            { writeConcern: { w: 3 } }
        );
    };
    print(i);
}
```

Again, run mongostat.

Notes:

`mongostat`, bulk inserts with `{w: 3}`

- Finally, let's use bulk inserts to our advantage:
- Note that writeConcern is still { w: 3 }

```
db.testcol.drop()
for (i=1; i<=10000; i++) {
    arr = []
    for (j=1; j<=1000; j++) {
        arr.push(
            { _id: (1000 * (i-1) + j), a: i, b: j, c: (1000 * (i-1)+ j) }
        );
    };
    db.testcol.insert( arr, { writeConcern : { w : 3 } } );
    print(i);
}
```

Notes:

Schema Design

- The structure of documents affects performance.
- Optimize for your application's read/write patterns.
- We want as few requests to the database as possible to perform a given application task.
- See the data modeling section for more information.

Notes:

Shard Key Considerations

- Choose a shard key that distributes load across your cluster.
- Create a shard key such that only a small number of documents will have the same value.
- Create a shard key that has a high degree of randomness.
- Your shard key should enable a mongos to target a single shard for a given query.

Notes:

Indexes and Performance

- Reads and writes that don't use an index will cripple performance.
- In compound indexes, order matters:
  - Sort on a field that comes before any range used in the index.
  - You can't skip fields; they must be used in order.
  - Revisit the indexing section for more detail.

Notes:

10 Backup and Recovery

Backup and Recovery **(page 187)** An overview of backup options for MongoDB.

10.1 Backup and Recovery

Disasters Do Happen

Notes:

Human Disasters

Notes:

Terminology: RPO vs. RTO

- **Recovery Point Objective (RPO)**: How much data can you afford to lose?
- **Recovery Time Objective (RTO)**: How long can you afford to be off-line?

Notes:

Terminology: DR vs. HA

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

Notes:

Quiz

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

Notes:

Backup Options

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

Notes:

Document Level Backups - `mongodump`

- Dumps collection to BSON files
- Mirrors your structure
- Can be run live or in offline mode
- Does not include indexes (rebuilt during restore)
- `--dbpath` for direct file access
- `--oplog` to record oplog while backing up
- `--query/filter` selective dump

Notes:

`mongodump`

```
$ mongodump --help
Export MongoDB data to BSON files.

options:
  --help                  produce help message
  -v [ --verbose ]        be more verbose (include multiple times for
                          more verbosity e.g. -vvvvv)
  --version               print the program's version and exit
  -h [ --host ] arg       mongo host to connect to ( /s1,s2 for
  --port arg              server port. Can also use --host hostname
  -u [ --username ] arg   username
  -p [ --password ] arg   password
  --dbpath arg            directly access mongod database files in path
  -d [ --db ] arg         database to use
  -c [ --collection ] arg collection to use (some commands)
  -o [ --out ] arg (=dump)output directory or "-" for stdout
  -q [ --query ] arg      json query
  --oplog                 Use oplog for point-in-time snapshotting
```

Notes:

File System Level

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

Notes:

Ensure Consistency

Flush RAM to disk and stop accepting writes:

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

Notes:

File System Backups: Pros and Cons

- Entire database
- Backup files will be large
- Fastest way to create a backup
- Fastest way to restore a backup

Notes:

Document Level - mongorestore

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

Notes:

File System Restores

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

Notes:

Backup Sharded Cluster

1. Stop Balancer (and wait) or no balancing window
2. Stop one config server (data R/O)
3. Backup Data (shards, config)
4. Restart config server
5. Resume Balancer

Notes:

Restore Sharded Cluster

1. Dissimilar # shards to restore to
2. Different shard keys?
3. Selective restores
4. Consolidate shards
5. Changing addresses of config/shards

Notes:

Tips and Tricks

- `mongodump/mongorestore`
  - `--oplog[Replay]`
  - `--objcheck/--repair`
  - `--dbpath`
  - `--query/--filter`
- `bsondump`
  - inspect data at console
- LVM snapshot time/space tradeoff
  - Multi-EBS (RAID) backup
  - clean up snapshots

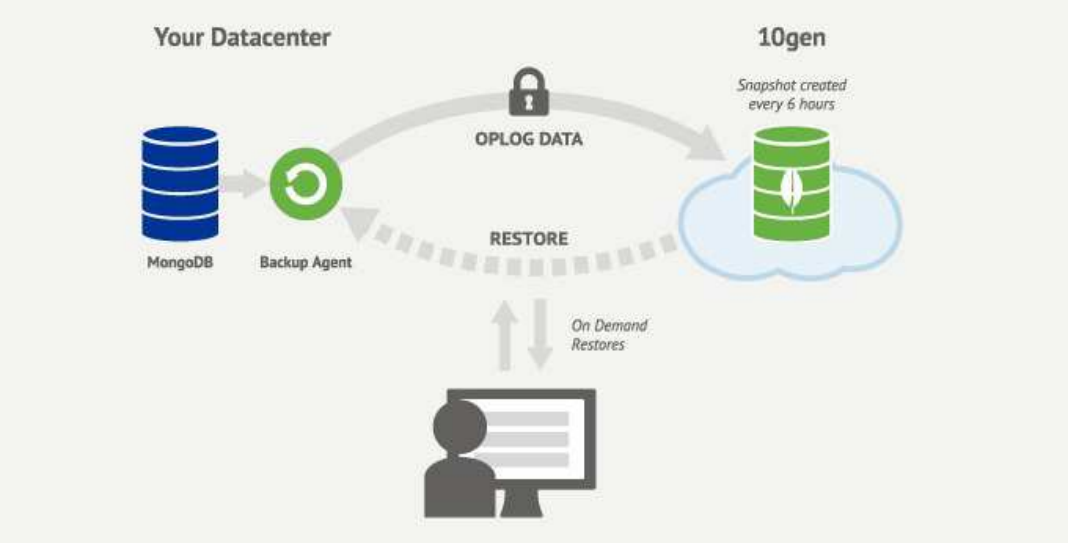
Notes:

Backup Options

- You can do it yourself as outlined in this section so far
- Or have the people who created MongoDB run your backups

Notes:

MMS Backup



Notes:

Sharded Clusters

- Balancer paused every 6 hours
- A no-op token is inserted across all shards, mongos instances, and config servers
- Oplog applied to replica sets until point in which token was inserted
- Provides a consistent state of database across shards

Notes:

Under the Hood

- From the initial sync, we rebuild your data in our datacenters and take a snapshot
- We take snapshots every 6 hours
- Oplog is stored for 48 hours

Notes:

Key Benefits

- Point in time backups
- Easy to restore
- Unlimited resources
- Fully managed

Notes:

Point in Time Backups

- Oplog stored for 48 hours
- Restore your replica set to any point-in-time in the last 48 hours by creating a custom snapshot

Notes:

Easy to Restore

- Pull from custom URL
- Push via scp

Notes:

Unlimited Restores

- Confidence in your restore process
- Build development, QA, analytics environments without impacting production

Notes:

Fully Managed

- Created by the engineers that build MongoDB
- No need to write or maintain custom backup scripts

Notes:

Getting Started

- Go to https://mms.mongodb.com and sign up

Notes: