MongoDB University

MongoDB Developer Training

MongoDB Developer Training

Release 3.2

MongoDB, Inc.

January 19, 2017

Contents

1 Introduction

1.1 Warm Up

Introductions

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

Getting to Know You

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

MongoDB Experience

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

10gen

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

Origin of MongoDB

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

1.2 MongoDB Overview

Learning Objectives

Upon completing this module students should understand:

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

MongoDB is a Document Database

Documents are associative arrays like:

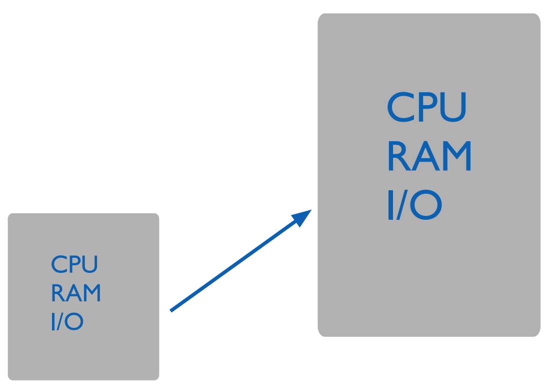
- Python dictionaries
- Ruby hashes
- PHP arrays
- JSON objects

An Example MongoDB Document

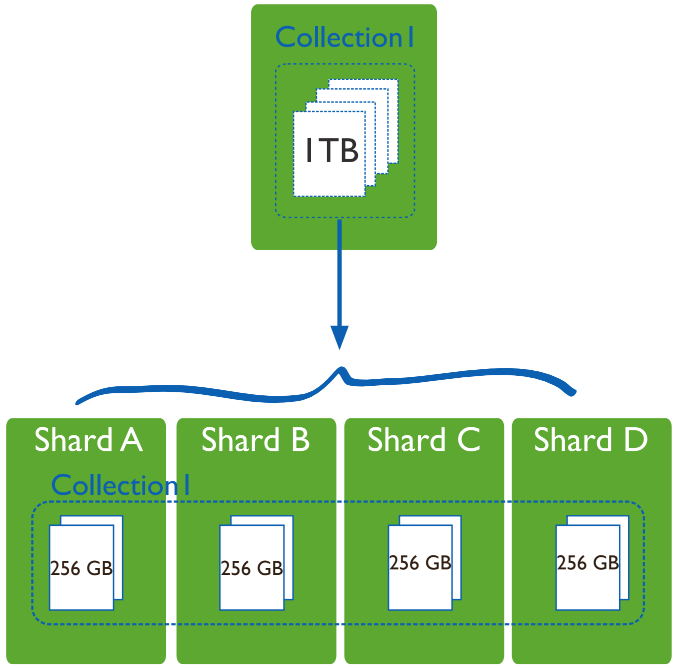
A MongoDB document expressed using JSON syntax.

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

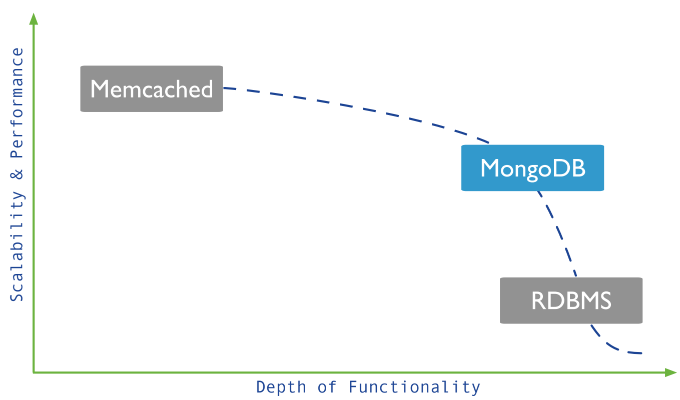
Vertical Scaling



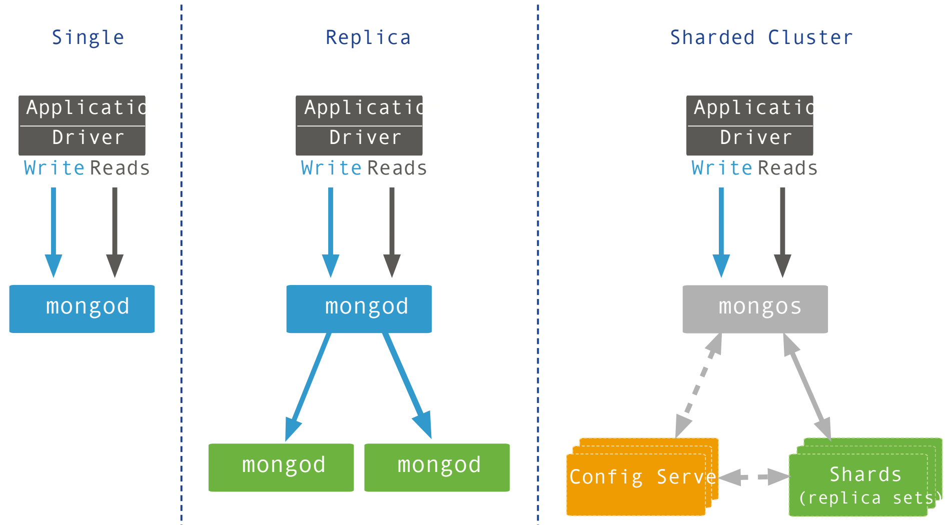
Scaling with MongoDB



Database Landscape



MongoDB Deployment Models



1.3 MongoDB Stores Documents

Learning Objectives

Upon completing this module, students should understand:

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

JSON

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

A Simple JSON Object

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

JSON Keys and Values

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

Example Field Values

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

[1] http://json.org/

BSON

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

BSON Hello World

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

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

A More Complex BSON Example

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

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

Documents, Collections, and Databases

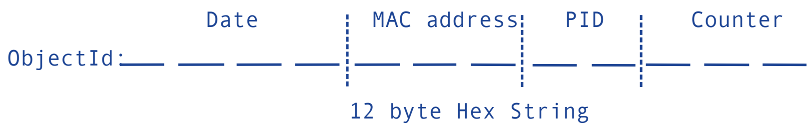
- Documents are stored in collections.
- Collections are contained in a database.
- Example:
  - Database: products
  - Collections: books, movies, music
- Each database-collection combination defines a namespace.
  - products.books
  - products.movies
  - products.music

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

The `_id` Field

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

ObjectIds



1.4 Lab: Installing and Configuring MongoDB

Learning Objectives

Upon completing this exercise students should understand:

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

Production Releases

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

- Windows
- OSX
- Linux
- Solaris

Installing MongoDB

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

Linux Setup

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

sudo mkdir -p /data/db

sudo chmod -R 744 /data/db

sudo chown -R `whoami` /data/db
```

Install on Windows

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

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

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

Create a Data Directory on Windows

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

  ```
  md \data\db
  ```

Launch a `mongod`

Explore the `mongod` command.

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

Launch a `mongod` with the MMAPv1 storage engine:

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

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

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

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

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

The MMAPv1 Data Directory

```
ls /data/db
```

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

The WiredTiger Data Directory

```
ls /data/db
```

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

Import Exercise Data

```
unzip usb_drive.zip

cd usb_drive

mongoimport -d sample -c tweets twitter.json

mongoimport -d sample -c zips zips.json

mongoimport -d sample -c grades grades.json

cd dump

mongorestore -d sample city

mongorestore -d sample digg
```

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

Launch a Mongo Shell

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

```
mongo
```

Display available commands.

```
help
```

Explore Databases

Display available databases.

```
show dbs
```

To use a particular database we can type the following.

```
use <database_name>

db
```

Exploring Collections

```
show collections
```

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

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

Admin Commands

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

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

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

2 CRUD

2.1 Creating and Deleting Documents

Learning Objectives

Upon completing this module students should understand:

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

Creating New Documents

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

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

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

Example: Inserting a Document

Experiment with the following commands.

```
use sample

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

db.movies.find()
```

Implicit `_id` Assignment

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

Example: Assigning _ids

Experiment with the following commands.

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

Inserts will fail if...

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

Example: Inserts will fail if...

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

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

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

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

`insertMany()`

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

Ordered `insertMany()`

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

Example: Ordered `insertMany()`

Experiment with the following operation.

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

Unordered `insertMany()`

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

Example: Unordered `insertMany()`

Experiment with the following insert.

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

The Shell is a JavaScript Interpreter

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

Exercise: Creating Data in the Shell

Experiment with the following commands.

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

db.stuff.find()
```

Deleting Documents

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

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

Using `deleteOne()`

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

Using `deleteMany()`

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

Example: Deleting Documents

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

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

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

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

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

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

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

Dropping a Collection

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

Example: Dropping a Collection

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

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

Dropping a Database

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

Example: Dropping a Database

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

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

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

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

2.2 Reading Documents

Learning Objectives

Upon completing this module students should understand:

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

The `find()` Method

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

Query by Example

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

Example: Querying by Example

Experiment with the following sequence of commands.

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

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

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

Querying Arrays

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

Example: Querying Arrays

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

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

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

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

Querying with Dot Notation

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

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

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

Example: Querying with Dot Notation

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

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

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

Example: Arrays and Dot Notation

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

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

Projections

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

Projection: Example (Setup)

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

Projection: Example

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

Projection Documents

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

Example: Projections

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

Cursors

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

Example: Introducing Cursors

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

it
it
```

Example: Cursor Objects in the Mongo Shell

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

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

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

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

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

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

Cursor Methods

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

Example: Using `count()`

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

// all 100
db.testcol.count()

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

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

Example: Using `sort()`

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

db.testcol.find()

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

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

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

The `skip()` Method

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

The `limit()` Method

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

The `distinct()` Method

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

Example: Using `distinct()`

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

2.3 Query Operators

Learning Objectives

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

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

Comparison Query Operators

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

Example (Setup)

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

```
    "title" : "Home Alone",
    "category" : [ "family", "comedy" ],
    "imdb_rating" : 7.4
  }
] )
```

Example: Comparison Operators

```
db.movies.find()

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

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

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

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

Logical Query Operators

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

Example: Logical Operators

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

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

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

Example: Logical Operators

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

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

Element Query Operators

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

Example: Element Operators

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

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

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

Array Query Operators

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

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

Example: Array Operators

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

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

Example: $elemMatch

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

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

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

2.4 Lab: Finding Documents

Exercise: student_id < 65

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

Exercise: Inspection Result "Fail" & "Pass"

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

Exercise: View Count > 1000

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

Exercise: Most comments

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

Exercise: Television or Videos

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

Exercise: News or Images

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

2.5 Updating Documents

Learning Objectives

Upon completing this module students should understand

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

The `replaceOne()` Method

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

First Parameter to `replaceOne()`

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

Second Parameter to `replaceOne()`

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

Example: `replaceOne()`

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

The `updateOne()` Method

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

`$set` and `$unset`

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

Example (Setup)

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

Example: `$set` and `$unset`

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

Update Operators

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

Example: Update Operators

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

The `updateMany()` Method

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

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

Example: `updateMany()`

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

Array Element Updates by Index

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

Example: Update Array Elements by Index

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

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

Array Operators

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

Example: Array Operators

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

The Positional $ Operator

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

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

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

Example: The Positional $ Operator

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

Upserts

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

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

Upsert Mechanics

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

Example: Upserts

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

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

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

save()

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

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

Example: save()

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

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

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

Be careful with save()

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

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

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

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

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

doc.imdb_rating = 7.4


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

`findOneAndUpdate()` and `findOneAndReplace()`

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

`findOneAndUpdate()` and `findOneAndReplace()` Options

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

Example: `findOneAndUpdate()`

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

`findOneAndDelete()`

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

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

2.6 Lab: Updating Documents

Exercise: Pass Inspections

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

Exercise: Set `fine` value

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

Exercise: Increase `fine` in ROSEDALE

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

Exercise: Give a pass to "MONGODB"

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

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

Exercise: Updating Array Elements

Insert a document representing product metrics for a backpack:

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

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

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

3 Indexes

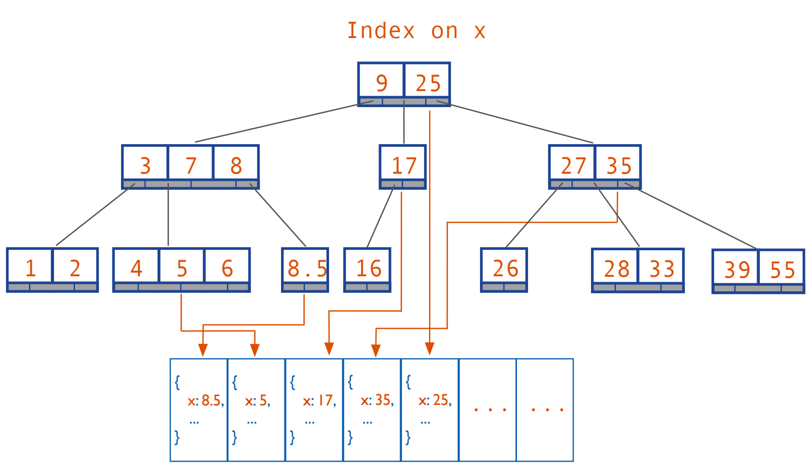
3.1 Index Fundamentals

Learning Objectives

Upon completing this module students should understand:

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

Why Indexes?



Types of Indexes

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

Exercise: Using `explain()`

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

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

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

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

Results of `explain()`

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

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

Results of `explain()` - Continued

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

`explain()` Verbosity Can Be Adjusted

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

`explain("executionStats")`

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

Now we have query statistics:

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

`explain("executionStats")` - Continued

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

`explain("executionStats")` Output

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

Other Operations

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

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

`db.<COLLECTION>.explain()`

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

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

equivalent to

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

also equivalent to

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

Using `explain()` for Write Operations

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

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

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

Single-Field Indexes

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

Creating an Index

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

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

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

Listing Indexes

List indexes for a collection:

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

List index keys:

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

Indexes and Read/Write Performance

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

Index Limitations

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

Use Indexes with Care

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

Additional Index Options

- Sparse
- Unique
- Background

Sparse Indexes in MongoDB

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

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

Defining Unique Indexes

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

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

Building Indexes in the Background

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

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

3.2 Compound Indexes

Learning Objectives

Upon completing this module students should understand:

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

Introduction to Compound Indexes

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

The Order of Fields Matters

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

- Equality tests, e.g.,

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

- Range queries, e.g.,

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

- Sorting, e.g.,

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

Designing Compound Indexes

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

Example: A Simple Message Board

Requirements:

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

Load the Data

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

Start with a Simple Index

Start by building an index on { timestamp : 1 }

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

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

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

Analysis:

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

Query Adding `username`

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

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

`totalKeysExamined` > `n`.

Include `username` in Our Index

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

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

`totalKeysExamined` > `n`

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

A Different Compound Index

Drop the index and build a new one with user.

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

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

`totalKeysExamined` is 2. `n` is 2.

`totalKeysExamined == n`

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

Let Selectivity Drive Field Order

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

Adding in the Sort

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

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

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

In-Memory Sorts

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

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

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

Avoiding an In-Memory Sort

Rebuild the index as follows.

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

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

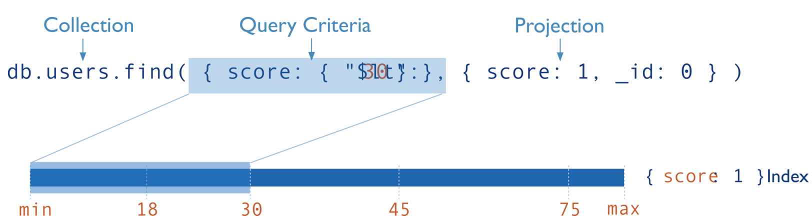
No need for `stage : SORT`

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

General Rules of Thumb

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

Covered Queries



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

Exercise: Covered Queries

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

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

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

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

3.3 Lab: Optimizing an Index

Exercise: What Index Do We Need?

Run the the following Javascript file from the handouts.

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

In the shell that launches execute the following method

```
performance.init()
```

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

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

Exercise: Avoiding an In-Memory Sort

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

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

Exercise: Avoiding an In-Memory Sort, 2

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

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

3.4 Multikey Indexes

Learning Objectives

Upon completing this module, students should understand:

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

Introduction to Multikey Indexes

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

Example: Array of Numbers

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

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

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

Exercise: Array of Documents, Part 1

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

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

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

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

Exercise: Array of Documents, Part 2

For each of the three queries below:

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

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

Exercise: Array of Arrays, Part 1

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

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

Exercise: Array of Arrays, Part 2

For each of the queries below:

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

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

How Multikey Indexes Work

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

Multikey Indexes and Sorting

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

Exercise: Multikey Indexes and Sorting

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

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

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

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

Limitations on Multikey Indexes

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

Example: Multikey Indexes on Multiple Fields

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

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

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

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

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

3.5 Hashed Indexes

Learning Objectives

Upon completing this module, students should understand:

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

What is a Hashed Index?

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

Why Hashed Indexes?

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

Limitations

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

Floating Point Numbers

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

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

Creating a Hashed Index

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

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

3.6 Geospatial Indexes

Learning Objectives

Upon completing this module, students should understand:

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

Introduction to Geospatial Indexes

We can use geospatial indexes to quickly determine geometric relationships:

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

Easiest to Start with 2 Dimensions

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

Location Field

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

Find Nearby Documents

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

Flat vs. Spherical Indexes

There are two types of geospatial indexes:

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

Flat Geospatial Index

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

Spherical Geospatial Index

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

Creating a 2d Index

Creating a 2d index:

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

Possible options key-value pairs:

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

Exercise: Creating a 2d Index

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

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

Inserting Documents with a 2d Index

There are two accepted formats:

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

Exercise: Inserting Documents with 2d Fields

- Insert 2 documents into the ‘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.

Querying Documents Using a 2d Index

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

Example: Find Based on 2d Coords

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

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

Creating a 2dsphere Index

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

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

The geoJSON Specification

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

geoJSON Considerations

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

Simple Types of 2dsphere Objects

Point: A single point on the globe

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

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

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

Polygons

Simple Polygon:

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

Polygon with One Hole:

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

Other Types of 2dsphere Objects

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

Exercise: Inserting geoJSON Objects (1)

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

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

Exercise: Inserting geoJSON Objects (2)

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

Exercise: Inserting geoJSON Objects (3)

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

Exercise: Creating a 2dsphere Index

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

Querying 2dsphere Objects

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

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

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

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

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

3.7 TTL Indexes

Learning Objectives

Upon completing this module students should understand:

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

TTL Index Basics

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

Creating a TTL Index

Create with:

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

Exercise: Creating a TTL Index

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

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

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

Exercise: Check the Collection

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

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

3.8 Text Indexes

Learning Objectives

Upon completing this module, students should understand:

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

What is a Text Index?

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

Creating a Text Index

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

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

Exercise: Creating a Text Index

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

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

Creating a Text Index with Weighted Fields

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

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

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

Creating a Text Index with Weighted Fields

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

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

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

Text Indexes are Similar to Multikey Indexes

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

Exercise: Inserting Texts

Let’s add some documents to our montyPython collection.

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

Querying a Text Index

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

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

Exercise: Querying a Text Index

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

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

Exercise: Querying Using Two Words

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

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

Search for a Phrase

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

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

Text Search Score

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

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

3.9 Lab: Finding and Addressing Slow Operations

Exercise: Determine Indexes Needed

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

3.10 Lab: Using `explain()`

Exercise: explain("executionStats")

Drop all indexes from previous exercises:

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

Create an index for the "active" field:

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

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

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

4 Drivers

4.1 Introduction to MongoDB Drivers

Learning Objectives

Upon completing this module, students should understand:

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

MongoDB Supported Drivers

- C[6]
- C++[7]
- C#[8]
- Java[9]
- Node.js[10]
- Perl[11]
- PHP[12]
- Python[13]
- Ruby[14]
- Scala[15]

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

MongoDB Community Supported Drivers

35+ different drivers for MongoDB:

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

Driver Specs

To ensure drivers have a consistent functionality, series of publicly available specification documents[16] for:

- Authentication[17]
- CRUD operations[18]
- Index management[19]
- SDAM[20]
- Server Selection[21]
- Etc.

Driver Settings (Per Operation)

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

Driver Settings (Per Connection)

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

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

Insert a Document with the Java Driver

Connect to a MongoDB instance on localhost:

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

Access the test database:

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

Insert a myDocument Document into the test.blog collection:

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

Insert a Document with the Python Driver

Connect to a MongoDB instance on localhost:

```
client = MongoClient()
```

Access the test database:

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

Insert a myDocument Document into the test.blog collection:

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

Insert a Document with the C++ Driver

Connect to the "test" database on localhost:

```
mongocxx::instance inst{};
mongocxx::client conn{};

auto db = conn["test"];
```

Insert a myDocument Document into the test.blog collection:

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

4.2 Lab: Driver Tutorial (Optional)

Tutorial

Complete the Python driver tutorial[22] for a concise introduction to:

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

[22] http://api.mongodb.org/python/current/tutorial.html

5 Aggregation

5.1 Aggregation Tutorial

Learning Objectives

Upon completing this module students should understand:

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

Aggregation Basics

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

The Aggregation Pipeline

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

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

Aggregation Stages

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

The Match Stage

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

The Project Stage

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

A Twitter Dataset

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

Tweets Data Model

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

Analyzing Tweets

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

Friends and Followers

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

Exercise: Friends and Followers

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

Exercise: $match and $project

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

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

The Group Stage

- For those coming from the relational world, `$group` is similar to the SQL `GROUP BY` statement.
- $group operations require that we specify which field to group on.
- Documents with the same identifier will be aggregated together.
- With $group, we aggregate values using accumulators[23].

Tweet Source

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

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

Exercise: Tweet Source

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

Group Aggregation Accumulators

Accumulators available in the group stage:

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

Rank Users by Number of Tweets

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

Exercise: Adding List of Tweets

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

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

The Unwind Stage

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

Example: User Mentions in a Tweet

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

Using $unwind

Who includes the most user mentions in their tweets?

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

Data Processing Pipelines

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

Most Unique User Mentions

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

Same Operator ($group), Multiple Stages

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

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

The Sort Stage

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

The Skip Stage

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

The Limit Stage

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

The Out Stage

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

5.2 Optimizing Aggregation

Learning Objectives

Upon completing this module students should understand:

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

Aggregation Options

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

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

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

Aggregation Limits

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

Limits Prior to MongoDB 2.6

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

Optimization: Reducing Documents in the Pipeline

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

Optimization: Sorting

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

Automatic Optimizations

MongoDB will perform some optimizations automatically. For example:

- If a `$project` stage is used late in the pipeline it may be used to eliminate those fields earlier if possible.
- A `$sort` followed by a `$match` will be executed as a `$match` followed by a `$sort` to reduce the number of documents to be sorted.
- A `$skip` followed by a `$limit` will be executed as a `$limit` followed by a `$skip`, with the $limit parameter increased by the $skip amount to allow $sort + $limit coalescence.
- See: Aggregation Pipeline Optimation[24]

5.3 Lab: Aggregation Framework

Exercise: Working with Array Fields

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

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

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

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

Exercise: Repeated Aggregation Stages

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

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

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

Please note:

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

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

Exercise: Projection

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

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

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

Exercise: Descriptive Statistics

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

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

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

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

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

6 Introduction to Schema Design

6.1 Schema Design Core Concepts

Learning Objectives

Upon completing this module, students should understand:

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

What is a schema?

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

Example: Normalized Data Model

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

Example: Denormalized Version

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

Schema Design in MongoDB

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

Three Considerations

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

Case Study

- A Library Web Application
- Different schemas are possible.

Author Schema

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

User Schema

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

Book Schema

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

Example Documents: Author

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

Example Documents: User

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

Example Documents: Book

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

Embedded Documents

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

Example: Embedded Documents

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

Embedded Documents: Pros and Cons

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

Linked Documents

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

Example: Linked Documents

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

Linked Documents: Pros and Cons

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

Arrays

- Array of scalars
- Array of documents

Array of Scalars

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

Array of Documents

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

Exercise: Users and Book Reviews

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

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

Solution A: Users and Book Reviews

Reviews may be queried by user or book

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

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

Solution B: Users and Book Reviews

Optimized to retrieve reviews by user

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

Solution C: Users and Book Reviews

Optimized to retrieve reviews by book

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

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

Store Binary Files in MongoDB with GridFS

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

How GridFS Works

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

Schema Design Use Cases with GridsFS

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

6.2 Schema Evolution

Learning Objectives

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

- Development Phase
- Production Phase
- Iterative Modifications

Development Phase

Support basic CRUD functionality:

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

Development Phase: Known Query Patterns

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

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

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

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

Production Phase

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

Production Phase: Read Patterns

List books by author last name

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

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

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

Addressing List Books by Last Name

"Cache" the author name in an embedded document.

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

Queries are now one step

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

Production Phase: Write Patterns

Users can review a book.

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

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

Caveats:

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

Exercise: Recent Reviews

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

Solution: Recent Reviews, Schema

Store users' reviews in monthly buckets.

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

Solution: Recent Reviews, Update

Adding a new review to the appropriate bucket

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

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

Solution: Recent Reviews, Read

Display the 10 most recent reviews by a user

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

num = 0;

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

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

Solution: Recent Reviews, Delete

Deleting a review

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

6.3 Common Schema Design Patterns

Learning Objectives

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

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

One-to-One Relationship

Let's pretend that authors only write one book.

One-to-One: Linking

Either side, or both, can track the relationship.

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

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

One-to-One: Embedding

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

One-to-Many Relationship

In reality, authors may write multiple books.

One-to-Many: Array of IDs

The "one" side tracks the relationship.

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

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

One-to-Many: Single Field with ID

The "many" side tracks the relationship.

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

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

One-to-Many: Array of Documents

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

Many-to-Many Relationship

Some books may also have co-authors.

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

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

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

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

Query for all books by a given author.

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

Query for all authors of a given book.

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

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

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

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

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

Query for all books by a given author.

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

Query for all authors of a given book.

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

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

Tree Structures

E.g., modeling a subject hierarchy.

Allow users to browse by subject

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

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

Alternative: Parents and Ancestors

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

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

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

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

Find Sub-Categories

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

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

Summary

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

7 Replica Sets

7.1 Introduction to Replica Sets

Learning Objectives

Upon completing this module, students should understand:

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

Use Cases for Replication

- High Availability
- Disaster Recovery
- Functional Segregation

High Availability (HA)

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

Disaster Recovery (DR)

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

Functional Segregation

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

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

Large Replica Sets

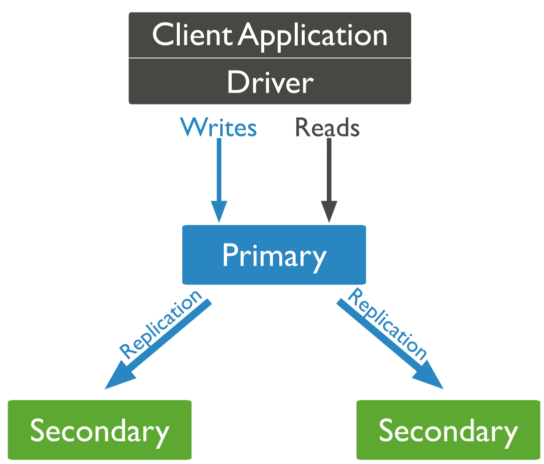
Functional segregation can be further exploited by using large replica sets.

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

Replication is Not Designed for Scaling

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

Replica Sets



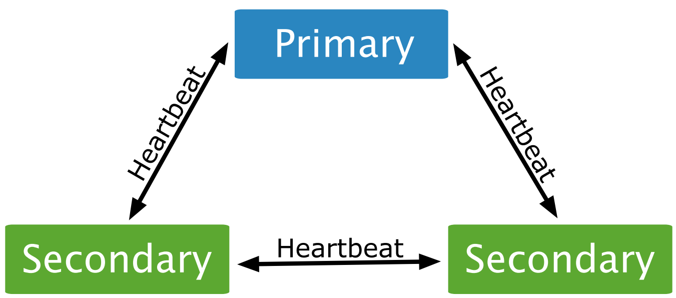
Primary Server

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

Secondaries

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

Heartbeats



The Oplog

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

7.2 Write Concern

Learning Objectives

Upon completing this module students should understand:

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

What happens to the write?

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

Answer

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

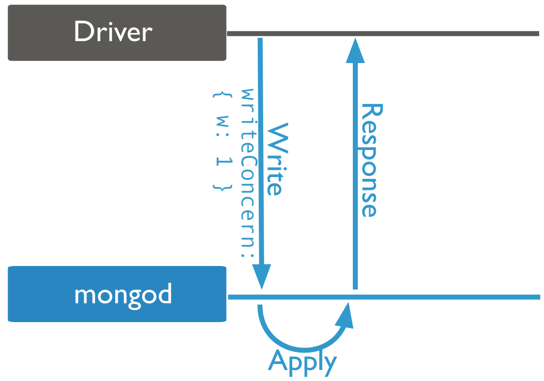
Balancing Durability with Performance

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

Defining Write Concern

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

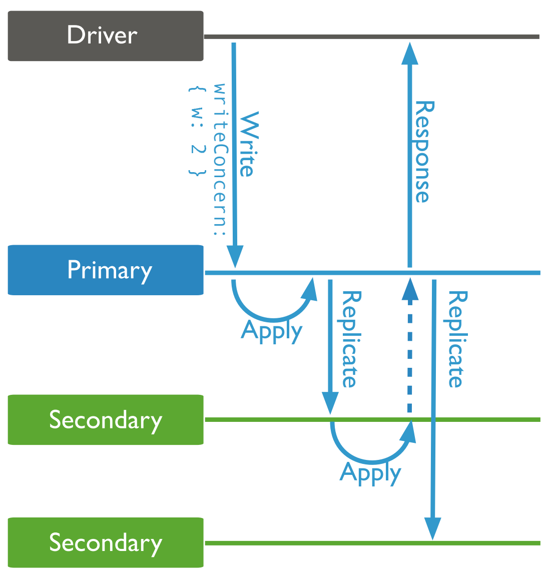
Write Concern: `{ w : 1 }`



Example: `{ w : 1 }`

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

Write Concern: `{ w : 2 }`



Example: `{ w : 2 }`

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

Other Write Concerns

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

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

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

Example: `{ w : "majority" }`

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

Quiz: Which write concern?

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

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

Further Reading

See Write Concern Reference[25] for more details on write concern configurations, including setting timeouts and identifying specific replica set members that must acknowledge writes (i.e. tag sets[26]).

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

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

7.3 Read Preference

What is Read Preference?

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

Use Cases

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

Not for Scaling

- In general, do *not* read from secondaries to provide extra capacity for reads.
- Sharding[27] increases read and write capacity by distributing operations across a group of machines.
- Sharding is a better strategy for adding capacity.

Read Preference Modes

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

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

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

Tag Sets

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

For example, in the mongo shell:

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

8 Sharding

8.1 Introduction to Sharding

Learning Objectives

Upon completing this module, students should understand:

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

Contrast with Replication

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

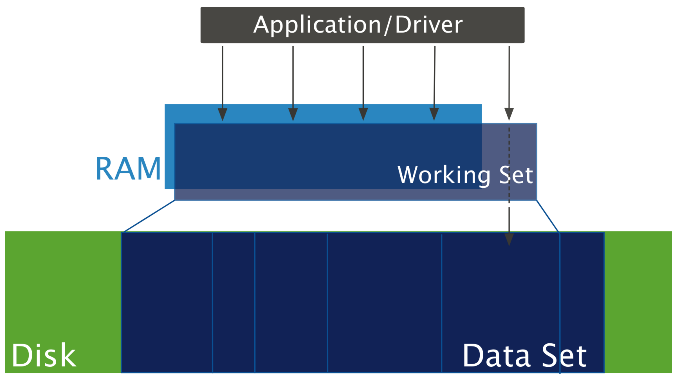
Sharding is Concerned with Scale

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

Vertical Scaling

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

The Working Set



Limitations of Vertical Scaling

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

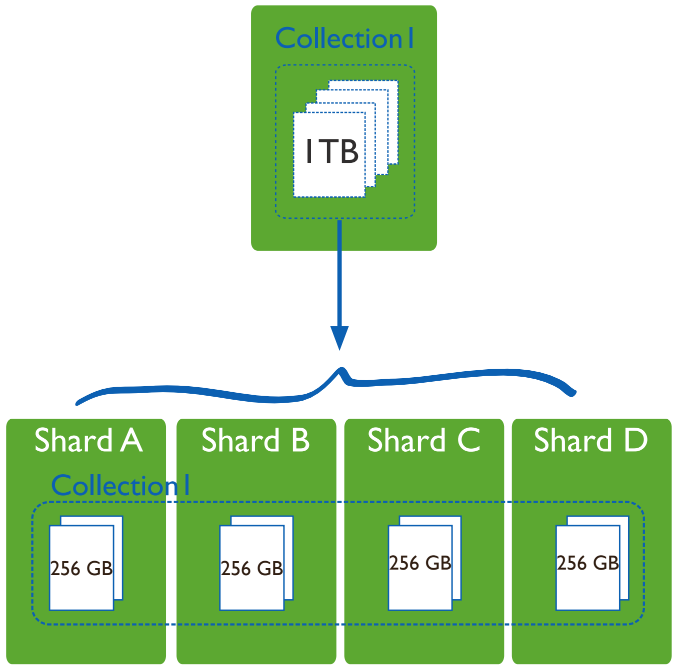
Sharding Overview

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

When to Shard

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

Dividing Up Your Dataset



Sharding Concepts

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

- Shard Keys
- Chunks

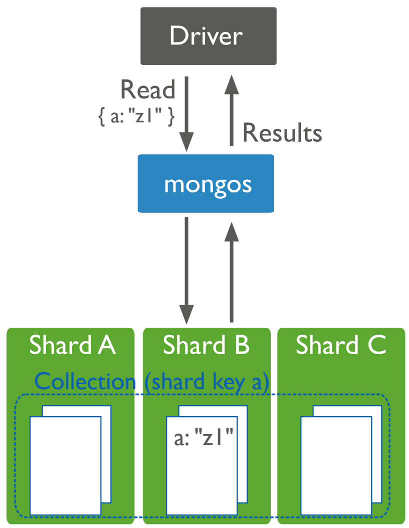
Shard Key

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

Shard Key Ranges

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

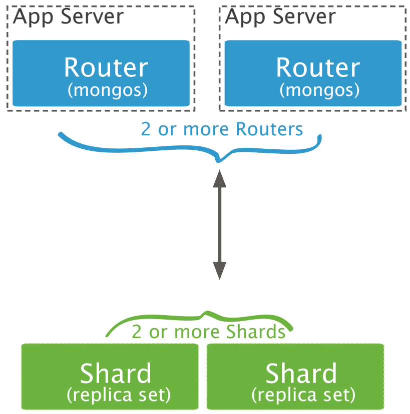
Targeted Query Using Shard Key



Chunks

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

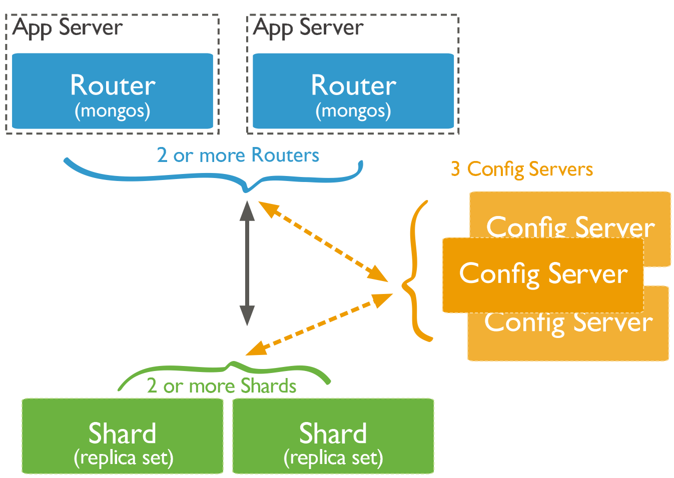
Sharded Cluster Architecture



Mongos

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

Config Servers



Config Server Hardware Requirements

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

Possible Imbalance?

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

Balancing Shards

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

With a Good Shard Key

You might easily see that:

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

With a Bad Shard Key

You might see that:

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

Choosing a Shard Key

Generally, you want a shard key:

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

More Specifically

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

Cardinality

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

Non-Monotonic

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

Shards Should be Replica Sets

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

9 New in 3.2

9.1 Aggregation in MongoDB 3.2

Learning Objectives

Upon completing this module, students will be able to:

- List and use the new aggregation stages in MongoDB 3.2
  - `$sample`
  - `$indexStats`
  - `$lookup`
- Use the new or revised operators in MongoDB 3.2

Sample Dataset

Mongoimport the `companies.json` file:

```
mongoimport -d training -c companies --drop companies.json
```

- You now have a dataset of companies on your server.
- We will use these for our examples.

New Pipeline Operators

- `$sample` used to pull in a random set of documents
- `$indexStats` shows how many hits the indexes get since the server process started
- `$lookup` enables you to do a left outer join across two collections

Introduction to `$sample`

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

Example: `$sample`

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

Introduction to `$indexStats`

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

Example: `$indexStats`

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

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

Introduction to `$lookup`

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

Introduction to `$lookup` (continued)

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

Example: Using `$lookup`

Create a separate collection for `$lookup`

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

Example: Using `$lookup` (Continued)

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

Reviewing the Output

- All companies matching the filter are included.
- Even if there is no corresponding comment (as for IBM)
- Note that the comment documents joined to each match are included in their entirety.
- Does not include documents in commentOnEmployees that don't match.

New Aggregation Functionality

3.2 introduced several new operators and expanded the functionality of a few operators:

- New accumulators for `$group`
- New arithmetic operators
- New array operators
- General enhancements

New Accumulators

- Used in the `$group` stage
- `$stdDevSamp` - sample standard deviation
- `$stdDevPop` - population standard deviation

```
db.companies.aggregate( [
{ $match : { number_of_employees: { $lt: 1000, $gte: 100 } } },
{ $group : {
    _id : null,
    mean_employees: { $avg : "$number_of_employees" },
    std_num_employees : { $stdDevPop: "$number_of_employees" } } }
] )
```

New Arithmetic Operators

- `$sqrt`: Calculate a square root
- `$abs`: Calculate the absolute value
- `$log`: Calculate the logarithm in a specified base
- `$log10`: Log base 10
- `$ln`: Natural logarithm

New Arithmetic Operators (Continued)

- `$pow`: Raise a number to an exponent
- `$exp`: Raise e to a power
- `$trunc`: Truncate a number to its integer
- `$ceil`: Round up to an integer
- `$floor`: Round down to an integer

Example: `$trunc`

```
db.companies.aggregate( [
  { $match : { number_of_employees: { $gte: 100, $lt: 1000 } } },
  { $group : { _id : null,
               mean_employees: { $avg: "$number_of_employees" } } },
  { $project : { _id: 0,
                 truncated_mean_employees: { $trunc : "$mean_employees" } } }
   ] )
```

New Array Operators

- `$slice`: returns a portion of an array
- `$arrayElemAt`: Returns an element at the index
- `$concatArrays`: Concatenates two or more arrays
- `$isArray`: Determines if the operand is an array or not
- `$filter`: Selects a subset of the array, based on the filter

Example: `$filter`

```
db.companies.aggregate( [
  { $match : { "funding_rounds.round_code": "e" } },
  { $project : {
      _id: 0, name: 1,
      series_e_funding: {
        $filter: {
          input: "$funding_rounds",
          as: "series_e_funding",
          cond: { $eq : [ "$$series_e_funding.round_code", "e" ] } } } }
  }, {
    $project : {
      name: 1,
      "series_e_funding.raised_amount": 1,
      "series_e_funding.raised_currency_code": 1,
      "series_e_funding.year": 1 }
  } ] )
```

Changes to `$unwind` Behavior

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

`$unwind` with a Document Operand

`$unwind` also supports this form:

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

Document Operand Semantics

- `path` – field path to an array field
- `includeArrayIndex` – the name of a new field to hold the array index of the element (optional)
- `preserveNullAndEmptyArrays` – output a document only if `true` and the path is null, missing, or an empty array

Using Accumulators with `$project`

For array fields, the following accumulators can be used in the `$project` stage starting in 3.2:

- `$avg`: Averages over values
- `$sum`: Sums the values
- `$min`: Finds the minimum value
- `$max`: finds the maximum value
- `$stdDevPop`, `$stdDevSamp`

Example: `$project` Accumulators

```
db.foo.drop()
db.foo.insertMany( [ { numbers: [ 1, 2, 3, 4, 5 ] },
                     { numbers: [ 6, 7, 8, 9, 10 ] } ] )
db.foo.aggregate( [
  {
    $project: {
      _id: 0,
      avg: { $avg: "$numbers" },
      sum: { $sum: "$numbers" },
      min: { $min: "$numbers" },
      max: { $max: "$numbers" },
      stDevSamp: { $stdDevSamp: "$numbers" } }
  } ] )
db.foo.find( {}, { _id: 0 } )  // these are the original documents
```

`$project` ing Arrays

You can now use `$project` to create arrays from existing non-array fields:

```
db.plants.drop()
db.plants.insertMany( [
  { _id: "yellow plants", fruit: "banana", vegetable: "squash" },
  { _id: "red plants", fruit: "strawberry", vegetable: "radish" } ] )
db.plants.aggregate( [
  { $project: { plant_list: [ "$fruit", "$vegetable" ] } } ] )
```

9.2 New Cluster Operations in MongoDB 3.2

Learning Objectives

Upon completing this module, students will be able to:

- Distinguish stale from dirty reads
- Use read concern in MongoDB 3.2
- Describe how read concern prevents dirty reads
- List the features of Replication Protocol 1
- List the benefits of using config servers as replica sets (CSRS)

Background: Stale Reads

- Reads that do not reflect the most recent writes are stale
- These can occur when reading from secondaries
- Systems with stale reads are “eventually consistent”
- Reading from the primary minimizes odds of stale reads
  - They can still occur in rare cases

Stale Reads on a Primary

- In unusual circumstances, two members may simultaneously believe that they are the primary
  - One can acknowledge `{ w : "majority" }` writes
    - This is the true primary
  - The other was a primary
    - But a new one has been elected
- In this state, the other primary will serve stale reads

Background: Dirty Reads

- Dirty reads are not stale reads
- Dirty reads occur when you see a view of the data
  - ... but that view *may* not persist
  - ... even in the history (i.e., oplog)
- Occur when data is read that has not been committed to a majority of the replica set
  - Because that data *could* get rolled back

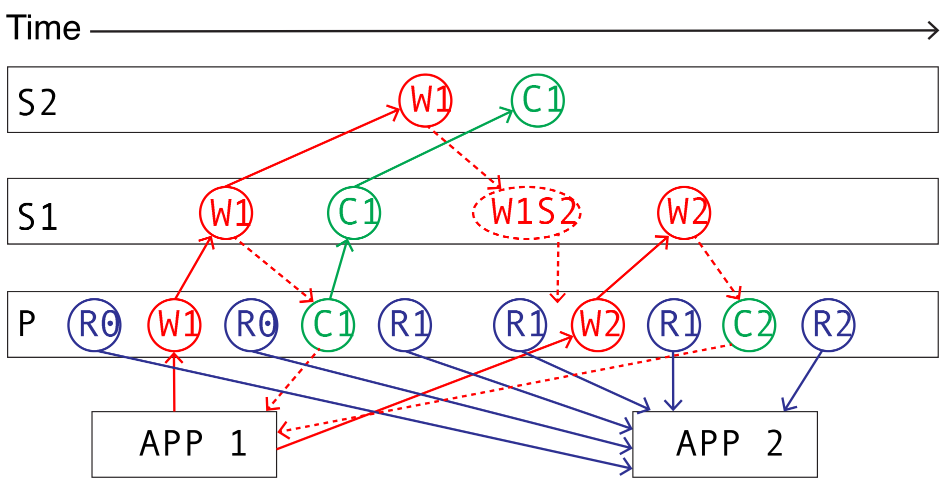
Dirty Reads and Write Concern

- Write concern alone can not prevent dirty reads
  - Data on the primary may be vulnerable to rollback
- Read concern was implemented to allow developers the option of preventing dirty reads

Introduction to Read Concern

- Two settings
  - "local": read the most recent data on the server
    * This is the historical behavior.
    * Exposes the application to dirty reads
  - "majority": data updates only when majority acknowledged
    * A version of the data is retained pre-acknowledgment
    * Writes get committed after a majority has them
      · Committed first on the primary
      · When a majority acknowledges the write

Example: Read Concern Level Majority



Quiz

What is the difference between a dirty read and a stale read?

Read Concern and Read Preference

- Read preference determines the server you read from
  - Primary, secondary, etc.
- Read concern determines the view of the data you see, and does not update its data the moment writes are received

Read Concern and Read Preference: Secondary

- The primary has the most current view of the data
  - Secondaries learn which writes are committed from the primary
- Data on secondaries might be behind the primary
  - But never ahead of the primary

Using Read Concern

- To use read concern, you must:
  - Use WiredTiger on all members
  - Launch all mongods in the set with
    - `--enableMajorityReadConcern`
  - Specify the read concern level to the driver
- You should:
  - Use write concern `{ w : "majority" }`
  - Otherwise, an application may not see its own writes

Example: Using Read Concern

- First, launch a replica set
  - Use `--enableMajorityReadConcern`
- A script is in the *shell_scripts* directory of the USB drive.

```
./launch_replset_for_majority_read_concern.sh
```

Example: Using Read Concern (Continued)

```
#!/usr/bin/env bash
echo 'db.testCollection.drop();' | mongo --port 27017 readConcernTest; wait
echo 'db.testCollection.insertOne({message: "probably on a secondary." } );' |
     mongo --port 27017 readConcernTest; wait
echo 'db.fsyncLock()' | mongo --port 27018; wait
echo 'db.fsyncLock()' | mongo --port 27019; wait
echo 'db.testCollection.insertOne( { message : "Only on primary." } );' |
     mongo --port 27017 readConcernTest; wait
echo 'db.testCollection.find().readConcern("majority");' |
     mongo --port 27017 readConcernTest; wait
echo 'db.testCollection.find(); // read concern "local"' |
```

Quiz

What must you do in order to make the database return documents that have been replicated to a majority of the replica set members?

Replication Protocol Version 1

- MongoDB 3.2 introduced a new replication protocol.
  - Replication protocol version 1 is the new protocol.
  - Replication protocol version 0 was used in earlier versions of MongoDB.
- With version 1, secondaries now write to disk before acknowledging writes.
- `{ w : "majority" }` now implies `{ j : true }`
- Set the replication protocol version using the `protocolVersion` parameter in your replica set configuration.
- Version 1 is the default in MongoDB >=3.2.

Replication Protocol Version 1 (continued)

- Also adds `electionTimeoutMillis` as an option
  - For secondaries: How long to wait before calling for an election
  - For primaries: How long to wait before stepping down
    - After losing contact with the majority
    - This applies to the primary only
- Required for read concern level "majority"

CSRS: Config Servers as Replica Sets

- With MongoDB 3.2, config servers can be replica sets
  - Subject to all standard rules of a replica set
  - Using read concern level "majority"
- Your config server replica set needs a primary
  - Without a primary, the config metadata can't change
    - No chunk splits, no chunk migrations
    - This will last until a new primary is elected

CSRS: Advantages

- Provides the same availability guarantees as your data
- Provides the same durability guarantees as your data
- You can tune the size of the replica set
  - Not restricted to 3 servers
  - Suitable for large deployments across data centers

Quiz

What are the advantages of replication protocol 1?

Quiz

What are the advantages of config servers as replica sets (CSRS)?

9.3 Document Validation

Learning Objectives

Upon completing this module, students should be able to:

- Define the different types of document validation
- Distinguish use cases for document validation
- Create, discover, and bypass document validation in a collection
- List the restrictions on document validation

Introduction

- Prevents or warns when the following occurs:
  - Inserts/updates that result in documents that don’t match a schema
- Prevents or warns when inserts/updates do not match schema constraints
- Can be implemented for a new or existing collection
- Can be bypassed, if necessary

Example

```
db.createCollection( "products",
   {
      validator: {
          price : { $exists : true }
      },
      validationAction: "error"
   }
)
```

Why Document Validation?

Consider the following use case:

- Several applications write to your data store
- Individual applications may validate their data
- You need to ensure validation across all clients

Why Document Validation? (Continued)

Another use case:

- You have changed your schema in order to improve performance
- You want to ensure that any write will also map the old schema to the new schema
- Document validation is a simple way of enforcing the new schema after migrating
  - You will still want to enforce this with the application
  - Document validation gives you another layer of protection

Anti-Patterns

- Using document validation at the database level without writing it into your application
  - This would result in unexpected behavior in your application
- Allowing uncaught exceptions from the DB to leak into the end user's view
  - Catch it and give them a message they can parse

`validationAction` and `validationLevel`

- Two settings control how document validation functions
- `validationLevel` – determines how strictly MongoDB applies validation rules
- `validationAction` – determines whether MongoDB should error or warn on invalid documents

Details

orphan

		validationLevel		
		off	moderate	strict
validationAction	warn	No checks	Warn on validation failure for inserts & updates to existing valid documents. Updates to existing invalid docs OK.	Warn on any validation failure for any insert or update.
	error	No checks	Reject invalid inserts & updates to existing valid documents. Updates to existing invalid docs OK.	Reject any violation of validation rules for any insert or update. **DEFAULT**

`validationLevel`: "strict"

- Useful when:
  - Creating a new collection
  - Validating writes to an existing collection already in compliance
  - Insert only workloads
  - Changing schema and updates should map documents to the new schema
- This will impose validation on update even to invalid documents

`validationLevel`: "moderate"

- Useful when:
  - Changing a schema and you have not migrated fully
  - Changing schema but the application can't map the old schema to the new in just one update
  - Changing a schema for new documents but leaving old documents with the old schema

`validationAction`: "error"

- Useful when:
  - Your application will no longer support valid documents
  - Not all applications can be trusted to write valid documents
  - Invalid documents create regulatory compliance problems

`validationAction`: "warn"

- Useful when:
  - You need to receive all writes
  - Your application can handle multiple versions of the schema
  - Tracking schema-related issues is important
    * For example, if you think your application is probably inserting compliant documents, but you want to be sure

Creating a Collection with Document Validation

```
db.createCollection( "products",
   {
      validator: {
          price: { $exists: true }
      },
      validationAction: "error"
   }
)
```

Seeing the Results of Validation

To see what the validation rules are for all collections in a database:

```
db.getCollectionInfos()
```

And you can see the results when you try to insert:

```
db.products.insertOne( { price: 25, currency: "USD" } )
```

Adding Validation to an Existing Collection

```
db.products.drop()
db.products.insertOne( { name: "watch", price: 10000, currency: "USD" } )
db.products.insertOne( { name: "happiness" } )
db.runCommand( {
   collMod: "products",
   validator: {
       price: { $exists: true }
   },
   validationAction: "error",
   validationLevel: "moderate"
} )
db.products.updateOne( { name : "happiness" }, { $set : { note: "Priceless." } } )
db.products.updateOne( { name : "watch" }, { $unset : { price : 1 } } )
db.products.insertOne( { name : "inner peace" } )
```

Bypassing Document Validation

- You can bypass document validation using the `bypassDocumentValidation` option
  - On a per-operation basis
  - Might be useful when:
    * Restoring a backup
    * Re-inserting an accidentally deleted document
- For deployments with access control enabled, this is subject to user roles restrictions
- See the MongoDB server documentation for details

Limits of Document Validation

- Document validation is not permitted for the following databases:
  - admin
  - local
  - config
- You cannot specify a validator for `system.*` collections

Document Validation and Performance

- Validation adds an expression-matching evaluation to every insert and update
- Performance load depends on the complexity of the validation document
  - Many workloads will see negligible differences

Quiz

What are the validation levels available and what are the differences?

Quiz

What command do you use to determine what the validation rule is for the *things* collection?

Quiz

On which three databases is document validation not permitted?

9.4 Partial Indexes

Learning Objectives

Upon completing this module, students should be able to:

- Outline how partial indexes work
- Distinguish partial indexes from sparse indexes
- List and describe the use cases for partial indexes
- Create and use partial indexes

What are Partial Indexes?

- Indexes with keys only for the documents in a collection that match a filter expression.
- Relative to standard indexes, benefits include:
  - Lower storage requirements
    - On disk
    - In memory
  - Reduced performance costs for index maintenance as writes occur

Creating Partial Indexes

- Create a partial index by:
  - Calling `db.collection.createIndex()`
  - Passing the `partialFilterExpression` option
- You can specify a `partialFilterExpression` on any MongoDB index type.
- Filter does not need to be on indexed fields, but it can be.

Example: Creating Partial Indexes

- Consider the following schema:

  ```
  { "_id" : 7, "integer" : 7, "importance" : "high" }
  ```

- Create a partial index on the "integer" field
- Create it only where "importance" is "high"

Example: Creating Partial Indexes (Continued)

```
db.integers.createIndex(
  { integer : 1 },
  { partialFilterExpression : { importance : "high" },
  name : "high_importance_integers" } )
```

Filter Conditions

- As the value for `partialFilterExpression`, specify a document that defines the filter.
- The following types of expressions are supported.
- Use these in combinations that are appropriate for your use case.
- Your filter may stipulate conditions on multiple fields.
  - equality expressions
  - `$exists`: true expression
  - `$gt`, `$gte`, `$lt`, `$lte` expressions
  - `$type` expressions
  - `$and` operator at the top-level only

Partial Indexes vs. Sparse Indexes

- Both sparse indexes and partial indexes include only a subset of documents in a collection.
- Sparse indexes reference only documents for which at least one of the indexed fields exist.
- Partial indexes provide a richer way of specifying what documents to index than does sparse indexes.

```
db.integers.createIndex(
    { importance : 1 },
    { partialFilterExpression : { importance : { $exists : true } } }
    )  // similar to a sparse index
```

Quiz

Which documents in a collection will be referenced by a partial index on that collection?

Identifying Partial Indexes

```
> db.integers.getIndexes()
[
...,
  {
    "v" : 1,
    "key" : {
      "integer" : 1
    },
    "name" : "high_importance_integers",
    "ns" : "test.integers",
    "partialFilterExpression" : {
      "importance" : "high"
    }
  },
  ...
]
```

Partial Indexes Considerations

- Not used when:
  - The indexed field is not in the query
  - A query goes outside of the filter range, even if no documents are out of range
- You can `.explain()` queries to check index usage

Quiz

Consider the following partial index. Note the `partialFilterExpression` in particular:

```
{
  "v" : 1,
  "key" : {
    "score" : 1,
    "student_id" : 1
  },
  "name" : "score_1_student_id_1",
  "ns" : "test.scores",
  "partialFilterExpression" : {
    "score" : {
      "$gte" : 0.65
    },
    "subject_name" : "history"
  }
}
```

Quiz (Continued)

Which of the following documents are indexed?

```
{ "_id" : 1, "student_id" : 2, "score" : 0.84, "subject_name" : "history" }
{ "_id" : 2, "student_id" : 3, "score" : 0.57, "subject_name" : "history" }
{ "_id" : 3, "student_id" : 4, "score" : 0.56, "subject_name" : "physics" }
{ "_id" : 4, "student_id" : 4, "score" : 0.75, "subject_name" : "physics" }
{ "_id" : 5, "student_id" : 3, "score" : 0.89, "subject_name" : "history" }
```

10 Application Engineering

10.1 MongoMart Introduction

What is MongoMart

MongoMart is an on-line store for buying MongoDB merchandise. We'll use this application to learn more about interacting with MongoDB through the driver.

MongoMart Demo of Fully Implemented Version

- View Items
- View Items by Category
- Text Search
- View Item Details
- Shopping Cart

View Items

- http://localhost:8080
- Pagination and page numbers
- Click on a category

View Items by Category

- http://localhost:8080/?category=Apparel
- Pagination and page numbers
- "All" is listed as a category, to return to all items listing

Text Search

- http://localhost:8080/search?query=shirt
- Search for any word or phrase in item title, description or slogan
- Pagination

View Item Details

- http://localhost:8080/item?id=1
- Star rating based on reviews
- Add a review
- Related items
- Add item to cart

Shopping Cart

- http://localhost:8080/cart
- Adding an item multiple times increments quantity by 1
- Change quantity of any item
- Changing quantity to 0 removes item

10.2 Java Driver Labs (MongoMart)

Introduction

- In this lab, we'll set up and optimize an application called MongoMart. MongoMart is an on-line store for buying MongoDB merchandise.

Lab: Setup and Connect to the Database

- Import the "item" collection to a standalone MongoDB server (without replication) as noted in the README.md file of the /data directory of MongoMart
- Become familiar with the structure of the Java application in /java/src/main/java/mongomart/
- Modify the MongoMart.java class to properly connect to your local database instance

Lab: Populate All Necessary Database Queries

- After running the MongoMart.java class, navigate to "localhost:8080" to view the application
- Initially, all data is static and the application does not query the database
- Modify the ItemDao.java and CartDao.java classes to ensure all information comes from the database (do not modify the method return types or parameters)

Lab: Use a Local Replica Set with a Write Concern

- It is important to use replication for production MongoDB instances, however, Lab 1 advised us to use a standalone server.
- Convert your local standalone mongod instance to a three node replica set named "shard1"
- Modify MongoMart's MongoDB connection string to include at least two nodes from the replica set
- Modify your application's write concern to MAJORITY for all writes to the "cart" collection, any writes to the "item" collection should continue using the default write concern of W:1

10.3 Python Driver Labs (MongoMart)

Introduction

- In this lab, we'll set up and optimize an application called MongoMart.
- MongoMart is an on-line store for buying MongoDB merchandise.

Lab: Setup and Connect to the Database

- Import the "item" collection to a standalone MongoDB server (without replication) as noted in the README.md file of the /data directory of MongoMart
- Become familiar with the structure of the Python application in /
- Start the application by running "python mongomart.py", stop it by using ctrl-c
- Modify the mongomart.py file to properly connect to your local database instance

Lab: Populate All Necessary Database Queries

- After running "python mongomart.py", navigate to "localhost:8080" to view the application
- Initially, all data is static and the application does not query the database
- Modify the itemDao.py and cartDao.py files to ensure all information comes from the database (you will not need to modify parameters for each DAO method)

Lab: Use a Local Replica Set with a Write Concern

- It is important to use replication for production MongoDB instances, however, Lab 1 advised us to use a standalone server.
- Convert your local standalone mongod instance to a three node replica set named “rs0”
- Modify MongoMart’s MongoDB connection string to include at least two nodes from the replica set
- Modify your application’s write concern to MAJORITY for all writes to the database

11 MongoDB Cloud & Ops Manager

11.1 MongoDB Cloud & Ops Manager

Learning Objectives

Upon completing this module students should understand:

- Features of Cloud & Ops Manager
- Available deployment options
- The components of Cloud & Ops Manager

Cloud and Ops Manager

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

- Monitoring
- Automation
- Backups

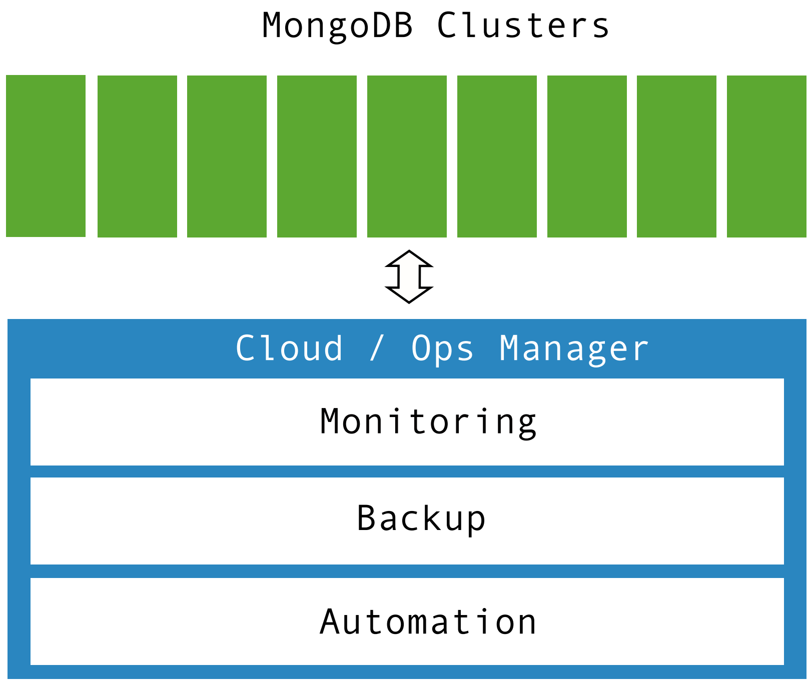
Deployment Options

- Cloud Manager: Hosted, https://www.mongodb.com/cloud
- Ops Manager: On-premises

Architecture

Cloud Manager

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



Ops Manager

On-premises, with additional features for:

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

Cloud & Ops Manager Use Cases

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

Creating a Cloud Manager Account

Free account at https://www.mongodb.com/cloud

11.2 Automation

Learning Objectives

Upon completing this module students should understand:

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

What is Automation?

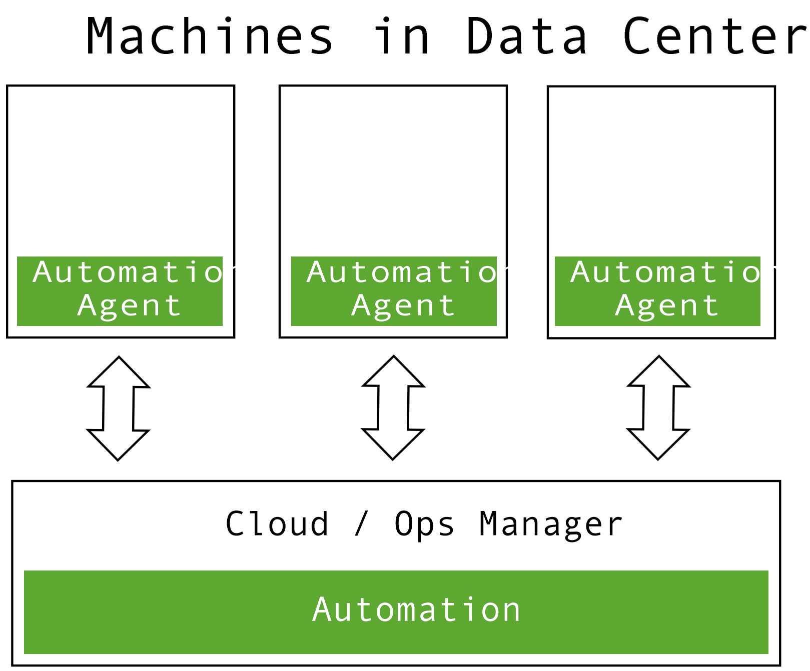
Fully managed MongoDB deployment on your own servers:

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

How Does Automation Work?

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

Automation Agents



Sample Use Case

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

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

Upgrades Using Automation

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

Automation: Behind the Scenes

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

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

Configuration File

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

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

Automation Goal State

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

Demo

- The instructor will demonstrate using Automation to set up a small cluster locally.
- Reference documentation:
- The Automation Agent[28] - The Automation API[29] - Configuring the Automation Agent[30]

11.3 Lab: Cluster Automation

Learning Objectives

Upon completing this exercise students should understand:

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

Exercise #1

Create a cluster using Cloud Manager automation with the following topology:

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

Exercise #2

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

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

[28] https://docs.cloud.mongodb.com/tutorial/nav/automation-agent/
[29] https://docs.cloud.mongodb.com/api/
[30] https://docs.cloud.mongodb.com/reference/automation-agent/