MongoDB University

What’s New in MongoDB 3.2 Workshop

What’s New in MongoDB 3.2 Workshop

Release 3.2

MongoDB, Inc.

June 17, 2016

Contents

1 Introduction

Warm Up **(page 2)** Activities to get the class started

Lab: Installing and Configuring MongoDB **(page 3)** Install MongoDB and experiment with a few operations.

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 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
```

Alternatively, launch with the WiredTiger storage engine.

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

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

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

The MMAPv1 Data Directory

```
ls /data/db
```

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

The WiredTiger Data Directory

```
ls /data/db
```

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

Import Exercise Data

```
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 The 3.2 Mongo Shell CRUD API

Learning Objectives

Upon completing this module students should be able to:

- Outline mappings between the old CRUD API and the new
- Locate the MongoDB CRUD API spec in GitHub

New CRUD API

- With the release of MongoDB 3.0, the drivers were updated to support a new CRUD API.
- See the CRUD API Spec[1] for details.
- MongoDB 3.2 includes support for the new CRUD API in the mongo shell.
- They important differences are in the methods used for writes.

insert() Operations

- The new CRUD API provides two methods to replace *insert()*.
  - *insertOne()*
  - *insertMany()*
- These provide more useful return values.

[1] https://github.com/mongodb/specifications/tree/master/source/crud

insertOne()

- *insertOne()* accepts one document to insert.
- Returns a document containing the *_id* of the document inserted.
- We will look at *insertOne()* in more detail in another section.

insertMany()

- Use *insertMany()* to perform bulk inserts.
- It accepts an array of the documents to be inserted.
- It returns a document containing an array of the *_id* values for documents inserted.
- We will look at *insertMany()* in detail in the next section.

remove() Operations

- The CRUD API provides two methods to replace *remove()*
  - *deleteOne()*
  - *deleteMany()*
- *remove()* is prone to user error:
  - By default, all documents matching the filter are removed.
  - To remove just one, users must pass a second parameter.
- The new methods eliminate this problem.
- We will look at *deleteOne()* and *deleteMany()* in detail in the next section.

update() Operations

- The following methods replace *update()* in the new API.
  - *updateOne()*
  - *updateMany()*

Rationale

- The default for *update()* is to update just one document.
  - Pass an optional Boolean parameter to update many documents.
  - This is the opposite behavior of *remove()*.
  - It is a source of confusion.

Rationale, cont'd

- *update()* also makes it easy to accidentally overwrite a document.
  - If we fail to use an update operator such as *$set* or *$inc*.
  - *updateOne* and *updateMany* throw an exception on such calls.
- We will look at *updateOne()* and *updateMany()* in more detail in the next section.

Replacing Documents

- If you do want to overwrite a document, use *replaceOne()*.
- *replaceOne()* does not require the use of an update operator.
- We will look at *replaceOne()* in more detail in the next section.

findAndModify() Operations

- For some use cases it is important to return the document modified.
- In earlier versions of MongoDB, *findAndModify()* was the method of choice.
- *findAndModify()* is a complex, combining the functionality of three operations:
  - delete
  - replace
  - update (including upserts)

Simplifying *findAndModify()* Operations

- In MongoDB 3.2 the shell accommodates the functionality of *findAndModify()* with:
  - *findOneAndDelete()*
  - *findOneAndReplace()*
  - *findOneAndUpdate()*
- These methods enable you to get the value of a modified document.
- We will discuss all three methods in another section.

2.2 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
```

2.3 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

- $[2] is a positional operator that specifies an element in an array to update.
- It acts as a placeholder for the first element that matches the query document.
- `$` replaces the element in the specified position with the value given.
- Example:

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

[2]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" : "Jaws III", "category" : [ "horror" ] },
                      { $set : { "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();
```

2.4 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 Document Validation

3.1 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 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)

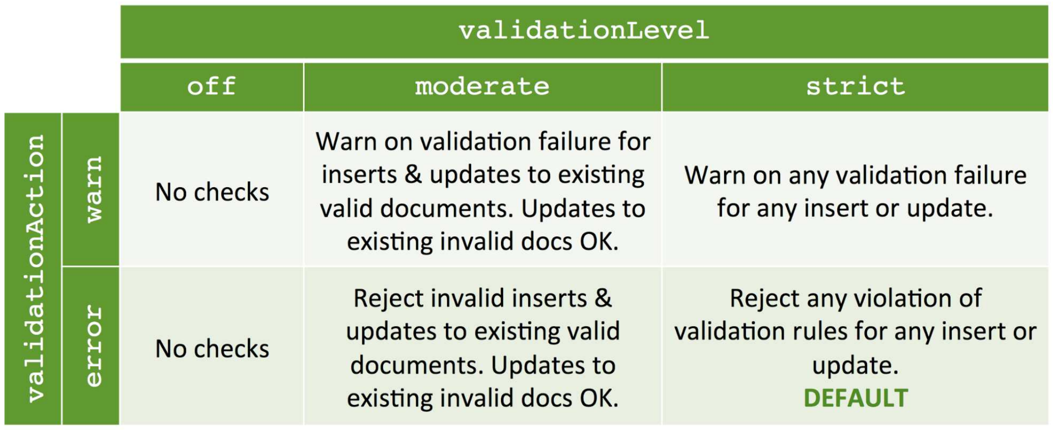
Another use case:

- 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

`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



		`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**

Quiz

- What are the uses for the two validationLevels?
- What are the uses for the two validationActions?

`validationLevel`: "strict"

- Useful when:
  - Creating a new collection
  - Validating writes to an existing collection already in compliance
  - 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

`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

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
- CRUD and aggregation methods support this option
- For deployments with access control enabled, this is subject to user roles restrictions
- See the MongoDB server documentation for details
- Security roles:
  - `dbAdmin` and `restore` can bypass validation
  - `bypassDocumentValidation` action can be set

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
- Testing shows negligible impact on performance

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?

3.2 Lab: Document Validation

Exercise: Add validator to existing collection

- Import the `posts` collection (from posts.json) and look at a few documents to understand the schema.
- Insert the following document into the `posts` collection

```
{"Hi":"I'm not really a post, am I?"}
```

- Discuss: what are some restrictions on documents that a validator could and should enforce?
- Add a validator to the `posts` collection that enforces those restrictions
- Remove the previously inserted document and try inserting it again and see what happens

Exercise: Create collection with validator

- **Create a collection `employees` with a validator that enforces the following restrictions on documents:**
  - The `name` field must exist and be a string
  - The `salary` field must exist and be between 0 and 10,000 inclusive.
  - The `email` field is optional but must be an email address in a valid format if present.
  - The `phone` field is optional but must be a phone number in a valid format if present.
  - At least one of the `email` and `phone` fields must be present.

Exercise: Create collection with validator (expected results)

```
// Valid documents
{"name":"Jane Smith", "salary":45, "email":"js@example.com"}
{"name":"Tim R. Jones", "salary":30, "phone":"234-555-6789","email":"trj@example.com"}
{"name":"Cedric E. Oxford", "salary":600, "phone":"918-555-1234"}

// Invalid documents
{"name":"Connor MacLeod", "salary":9001, "phone":"999-555-9999", "email":"thrcnbnly1"}
{"name":"Herman Hermit", "salary":9}
{"name":"Betsy Bedford", "salary":50, "phone":"", "email":"bb@example.com"}
```

Exercise: Change validator rules

- **Modify the validator for the `employees` collection to support the following additional restrictions:**
  - The `status` field must exist and must only be one of the following strings: “active”, “on_vacation”, “terminated”
  - The `locked` field must exist and be a boolean

Exercise: Change validator rules (expected results)

```
// Valid documents
{"name":"Jason Serivas", "salary":65, "email":"js@example.com", "status":"terminated", "locked":
{"name":"Logan Drizt", "salary":39, "phone":"234-555-6789","email":"ld@example.com", "status":"a
{"name":"Mann Edger", "salary":100, "phone":"918-555-1234", "status":"on_vacation", "locked":fal

// Invalid documents
{"name":"Steven Cha", "salary":15, "email":"sc@example.com", "status":"alive", "locked":false}
{"name":"Julian Barriman", "salary":15, "email":"jb@example.com", "status":"on_vacation", "locke
```

Exercise: Change validation level

Now that the `employees` validator has been updated, some of the already-inserted documents are not valid. This can be a problem when, for example, just updating an employee's salary.

- Try to update TODO
- Change the validation level of the `employees` collection to allow updates of existing invalid documents, but still enforce validation of inserted documents and existing valid documents.

Exercise: Bypass validation

In some circumstances, it may be desirable to bypass validation to insert or update documents.

- Use the `bypassDocumentValidation` option to insert the document `{"hi":"there"}` into the `employees` collection
- Use the `bypassDocumentValidation` option to give all employees a salary of 999999.

Exercise: Change validation action

In some cases, it may be desirable to simply log invalid actions, rather than prevent them.

- Change the validation action of the `employees` collection to reflect this behavior

4 Partial Indexes

4.1 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?

- Partial indexes only index the documents in a collection that match a filter expression.
- Benefits include:
  - Lower storage requirements
  - 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.
- The 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 a specified field exists.
- The functionality of sparse indexes is subsumed by partial indexes.

```
db.integers.createIndex(
    { importance : 1 },
    { partialFilterExpression : { importance : { $exists : true } } }
    )  // creates the equivalent of a sparse index
```

Quiz

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

Partial Indexes - Advantages

- Reduce storage requirements for indexes:
  - Disk
  - Memory
- Can reduce the performance impact of indexes on writes to a 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 them

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" }
```

5 Aggregation

5.1 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
- 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
  - 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
  - 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:

```
// $projecting arrays example
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" ] } } ] )
```

5.2 Lab: 3.2 Aggregation

Data Set

We have a normalized data set of digg `users` and their corresponding `stories` linked by `user._id`

```
> db.users.findOne()
{
"_id": ObjectId("575f1b138cc929dbcb20954e"),
"name": "babychen",
"registered": 1141570067,
"fullname": "Babychen Mathew",
"icon": "http://digg.com/users/babychen/l.png",
"profileviews": 24749
}

> db.stories.find({user: ObjectId("575f1b138cc929dbcb20954e")})
{
"_id": ObjectId("4ba267dc238d3ba3ca000001"),
"user": ObjectId("575f1b138cc929dbcb20954e"),
"href": "http://digg.com/people/Jedi_believer_who_refused_to_remove_hood_gets_an_apology",
"title": "'Jedi' believer who refused to remove hood gets an apology!",
...
```

Exercise: Get Stories Users

- Get all user names, story title and link that have topic **people**.
- Make sure you use only one request to the database to accomplish this task.

Exercise: Get Users Stories

- For every user, get all stories that have 100 comments or more.
- Again, perform this using only one request to the database.

Exercise: Sample Users

Get a random sample of five users, and their corresponding stories, where the user name starts with char `n`.

Exercise: Slice Users Stories

For each user that has four stories published, get the user's name and the last two stories!

Exercise: Users Stories

From all users that have more than 100000 profile views, get their third story.

6 3.2 Cluster Operations

6.1 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

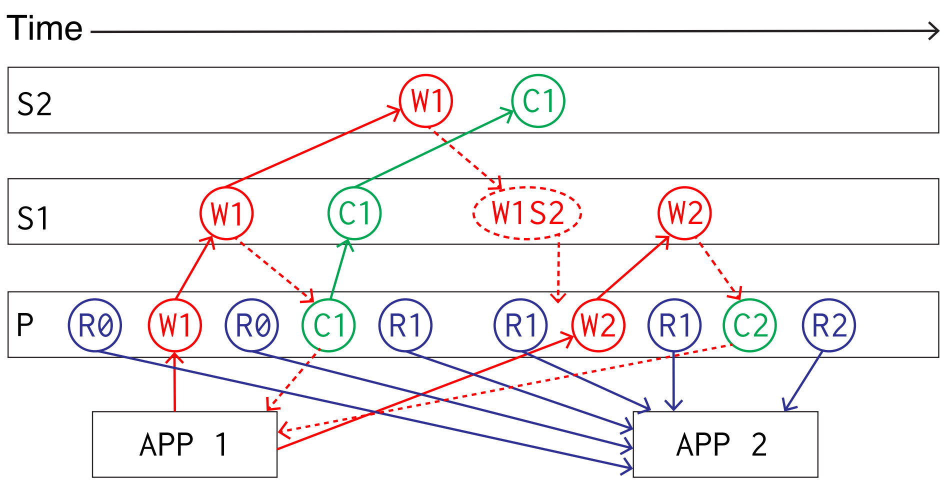
Dirty Reads and Write Concern

- 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)?

7 Security

7.1 Authorization

Learning Objectives

Upon completing this module, students should be able to:

- Outline MongoDB's authorization model
- List authorization resources
- Describe actions users can take in relation to resources
- Create roles
- Create privileges
- Outline MongoDB built-in roles
- Grant roles to users

Authorization vs Authentication

Authorization and Authentication are generally confused and misinterpreted concepts:

- Authorization defines the rules by which users can interact with a given system:
  - Which operations can they perform
  - Over which resources
- Authentication is the mechanism by which users identify and are granted access to a system:
  - Validation of credentials and identities
  - Controls access to the system and operational interfaces

Authorization Basics

- MongoDB enforces a role-based authorization model.
- A user is granted roles that determine the user's access to database resources and operations.

The model determines:

- Which roles are granted to users
- Which privileges are associated with roles
- Which actions can be performed over different resources

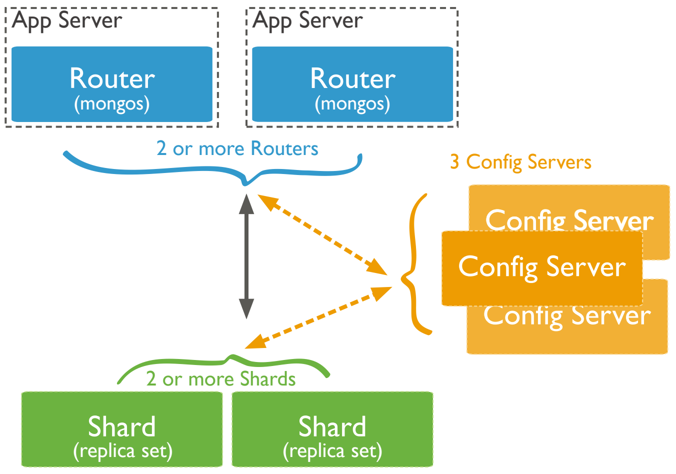
What is a resource?

- Databases?
- Collections?
- Documents?
- Users?
- Nodes?
- Shard?
- Replica Set?

Authorization Resources

- Databases
- Collections
- But that is not all. See next several slides.

Cluster Resources



Types of Actions

Given a resource, we can consider the available actions:

- Query and write actions
- Database management actions
- Deployment management actions
- Replication actions
- Sharding actions
- Server administration actions
- Diagnostic actions
- Internal actions

Specific Actions of Each Type

Query / Write	Database Mgmt	Deployment Mgmt
find	enableProfiler	planCacheRead
insert	createIndex	storageDetails
remove	createCollection	authSchemaUpgrade
update	changeOwnPassword	killop
	...	...

See the complete list of actions[3] in the MongoDB documentation.

Authorization Privileges

A privilege defines a pairing between a resource as a set of permitted actions.

Resource:

```
{"db": "yourdb", "collection": "mycollection"}
```

Action: `find`

Privilege:

```
{
  resource: {"db": "yourdb", "collection": "mycollection"},
  actions: ["find"]
}
```

[3]https://docs.mongodb.com/manual/reference/privilege-actions/

Authorization Roles

MongoDB grants access to data through a role-based authorization system:

- Built-in roles: pre-canned roles that cover the most common sets of privileges users may require
- User-defined roles: if there is a specific set of privileges not covered by the existing built-in roles you are able to create your own roles

Built-in Roles

Database Admin	Cluster Admin	All Databases
dbAdmin	clusterAdmin	readAnyDatabase
dbOwner	clusterManager	readWriteAnyDatabase
userAdmin	clusterMonitor	userAdminAnyDatabase
	hostManager	dbAdminAnyDatabase

Database User	Backup & Restore
read	backup
readWrite	restore

Superuser	Internal
root	__system

Built-in Roles

To grant roles while creating an user:

```
use admin
db.createUser(
  {
    user: "myUser",
    pwd: "$up3r$3cr7"
    roles: [
      {role: "readAnyDatabase", db: ""},
      {role: "dbOwner", db: "superdb"},
      {role: "readWrite", db: "yourdb"}
    ]
  }
)
```

Built-in Roles

To grant roles to existing user:

```
use admin
db.grantRolesToUser( {
  "reportsUser",
  [
    { role: "read", db: "accounts" }
  ]
} )
```

User-defined Roles

- If no suitable built-in role exists, we can can create a role.
- Define:
  - Role name
  - Set of privileges
  - List of inherit roles (optional)

```
use admin
db.createRole({
  role: "insertAndFindOnlyMyDB",
  privileges: [
    {resource: { db: "myDB", collection: "" }, actions: ["insert", "find"]}
  ],
  roles: []})
```

Role Privileges

To check the privileges of any particular role we can get that information using the `getRole` method:

```
db.getRole("insertAndFindOnlyMyDB", {showPrivileges: true})
```

7.2 Authentication

Learning Objectives

Upon completing this module, you should understand:

- Authentication mechanisms
- External authentication
- Native authentication
- Internal node authentication
- Configuration of authentication mechanisms

Authentication

- Authentication is concerned with:
  - Validating identities
  - Managing certificates / credentials
  - Allowing accounts to connect and perform authorized operations
- MongoDB provides native authentication and supports X509 certificates, LDAP, and Kerberos as well.

Authentication Mechanisms

MongoDB supports a number of authentication mechanisms:

- SCRAM-SHA-1 (default >= 3.0)
- MONGODB-CR (legacy)
- X509 Certificates
- LDAP (MongoDB Enterprise)
- Kerberos (MongoDB Enterprise)

Internal Authentication

For internal authentication purposes (mechanism used by replica sets and sharded clusters) MongoDB relies on:

- Keyfiles
  - Shared password file used by replica set members
  - Hexadecimal value of 6 to 1024 chars length
- X509 Certificates

Simple Authentication Configuration

To get started we just need to make sure we are launching our mongod instances with the `--auth` parameter.

```
mongod --dbpath /data/db --auth
```

For any connections to be established to this mongod instance, the system will require a username and password.

```
mongo -u user -p
Enter password:
```

7.3 Lab: Secure mongod

Premise

It is time for us to get started setting up our first MongoDB instance with authentication enabled!

Launch `mongod`

Let's start by launching a `mongod` instance:

```
mkdir /data/secure_instance_dbpath
mongod --dbpath /data/secure_instance_dbpath --port 28000
```

At this point there is nothing special about this setup. It is just an ordinary `mongod` instance ready to receive connections.

Root level user

Create a `root` level user:

```
mongo --port 28000 admin  // Puts you in the _admin_ database

use admin
db.createUser( {
  user: "maestro",
  pwd: "maestro+rules",
  customData: { information_field: "information value" },
  roles: [ {role: "root", db: "admin" } ]
} )
```

Enable Authentication

Launch `mongod` with `auth` enabled

```
mongo admin --port 28000 --eval 'db.shutdownServer()'
mongod --dbpath /data/secure_instance_dbpath --auth
```

Connect using the recently created `maestro` user.

```
mongo --port 28000 admin -u maestro -p
```

7.4 Encryption

Learning Objectives

Upon completing this module, students should understand:

- The encryption capabilities of MongoDB
- Network encryption
- Native encryption
- Third party integrations

Encryption

MongoDB offers two levels of encryption

- Transport layer
- Encryption at rest (MongoDB Enterprise >=3.2)

Network Encryption

- MongoDB enables TLS/SSL for transport layer encryption of traffic between nodes in a cluster.
- Three different network architecture options are available:
  - Encryption of application traffic connections
  - Full encryption of all connections
  - Mixed encryption between nodes

Native Encryption

MongoDB Enterprise comes with a encrypted storage engine.

- Native encryption supported by WiredTiger
- Encrypts data at rest
  - AES256-CBC: 256-bit Advanced Encryption Standard in Cipher Block Chaining mode (default)
    - symmetric key (same key to encrypt and decrypt)
  - AES256-GCM: 256-bit Advanced Encryption Standard in Galois/Counter Mode
  - FIPS is also available
- Enables integration with key management tools

Encryption and Replication

- Encryption is not part of replication:
  - Data is not natively encrypted on the wire
    - Requires transport encryption to ensure secured transmission
  - Encryption keys are not replicated
    - Each node should have their own individual keys

Third Party Integration

- Key Management Interoperability Protocol (KMIP)
  - Integrates with Vormetric Data Security Manager (DSM) and SafeNet KeySecure
- Storage Encryption
  - Linux Unified Key Setup (LUKS)
  - IBM Guardium Data Encryption
  - Vormetric Data Security Platform
    - Also enables Application Level Encryption on per-field or per-document
  - Bitlocker Drive Encryption

7.5 Lab: Secured Replica Set - KeyFile (Optional)

Premise

Security and Replication are two aspects that are often neglected during the Development phase to favor usability and faster development.

These are also important aspects to take in consideration for your Production environments, since you probably don't want to have your production environment **Unsecured** and without **High Availability**!

This lab is to get fully acquainted with all necessary steps to create a secured replica set using the `keyfile` for cluster authentication mode

Setup Secured Replica Set

A few steps are required to fully setup a secured Replica Set:

1. Instantiate one `mongod` node with no `auth` enabled
2. Create a `root` level user
3. Create a `clusterAdmin` user
4. Generate a keyfile for internal node authentication
5. Re-instantiate a `mongod` with `auth` enabled, `keyfile` defined and `replSet` name
6. Add Replica Set nodes

We will also be basing our setup using MongoDB configuration files[4]

Instantiate `mongod`

This is a rather simple operation that requires just a simple instruction:

```
$ pwd
/data
$ mkdir -p /data/secure_replset/{1,2,3}; cd secure_replset/1
```

Then go to this yaml file[5] and copy it into your clipboard

```
$ pbpaste > mongod.conf; cat mongod.conf
```

Instantiate `mongod` (cont'd)

```
systemLog:
  destination: file
  path: "/data/secure_replset/1/mongod.log"
  logAppend: true
storage:
  dbPath: "/data/secure_replset/1"
  wiredTiger:
    engineConfig:
      cacheSizeGB: 1
net:
  port: 28001
processManagement:
  fork: true
# setParameter:
#   enableLocalhostAuthBypass: false
# security:
#   keyFile: /data/secure_replset/1/mongodb-keyfile
```

[4]https://docs.mongodb.org/manual/reference/configuration-options/
[5]https://github.com/thatnerd/work-public/blob/master/mongodb_trainings/secure_replset_config.yaml

Instantiate `mongod` (cont'd)

After defining the basic configuration we just need to call `mongod` passing the configuration file.

```
mongod -f mongod.conf
```

Create root user

We start by creating our typical `root` user:

```
$ mongo admin --port 28001

> use admin
> db.createUser(
{
  user: "maestro",
  pwd: "maestro+rules",
  roles: [
    { role: "root", db: "admin" }
    ]
})
```

Create clusterAdmin user

We then need to create a clusterAdmin user to enable management of our replica set.

```
$ mongo admin --port 28001

> db.createUser(
{
  user: "pivot",
  pwd: "i+like+nodes",
  roles: [
    { role: "clusterAdmin", db: "admin" }
    ]
})
```

Generate a keyfile

For internal Replica Set authentication we need to use a keyfile.

```
openssl rand -base64 741 > /data/secure_replset/1/mongodb-keyfile
chmod 600 /data/secure_replset/1/mongodb-keyfile
```

Add keyfile to the configuration file

Now that we have the *keyfile* generated it's time to add that information to our configuration file. Just un-comment the last few lines.

```
systemLog:
  destination: file
  path: "/data/secure_replset/1/mongod.log"
  logAppend: true
storage:
  dbPath: "/data/secure_replset/1"
net:
  port: 28001
processManagement:
  fork: true
setParameter:
  enableLocalhostAuthBypass: false
security:
  keyFile: /data/secure_replset/1/mongodb-keyfile
```

Configuring Replica Set

- Now it's time to configure our Replica Set
- The desired setup for this Replica Set should be named "VAULT"
- It should consist of 3 data bearing nodes