# MongoDB Developer Training

***Release 2.6***

**MongoDB, Inc.**

November 25, 2014

## Contents

# 1 Introduction

## 1.1 Warm Up

### Introductions

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

---

**Note:**

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

---

### Getting to Know You

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

---

**Note:**

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

---

## MongoDB Experience

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

## 10gen

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

## Origin of MongoDB

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

## 1.2 MongoDB Overview

## Learning Objectives

Upon completing this module students should understand:

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

## MongoDB is a Document Database

Documents are associative arrays like:

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

## An Example MongoDB Document

A MongoDB document expressed using JSON syntax.

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

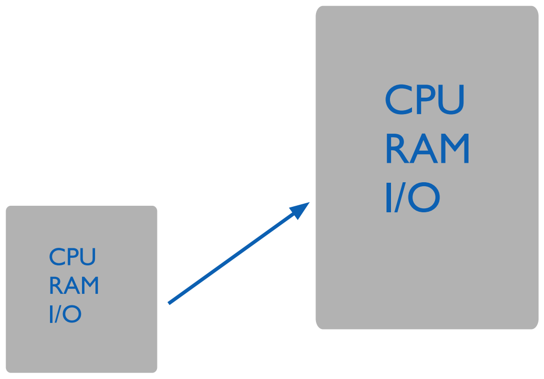
---

**Note:**

- Where relational databases store rows, MongoDB stores documents.
- Documents are hierarchical data structures.
- This is a fundamental departure from relational databases where rows are flat.

---

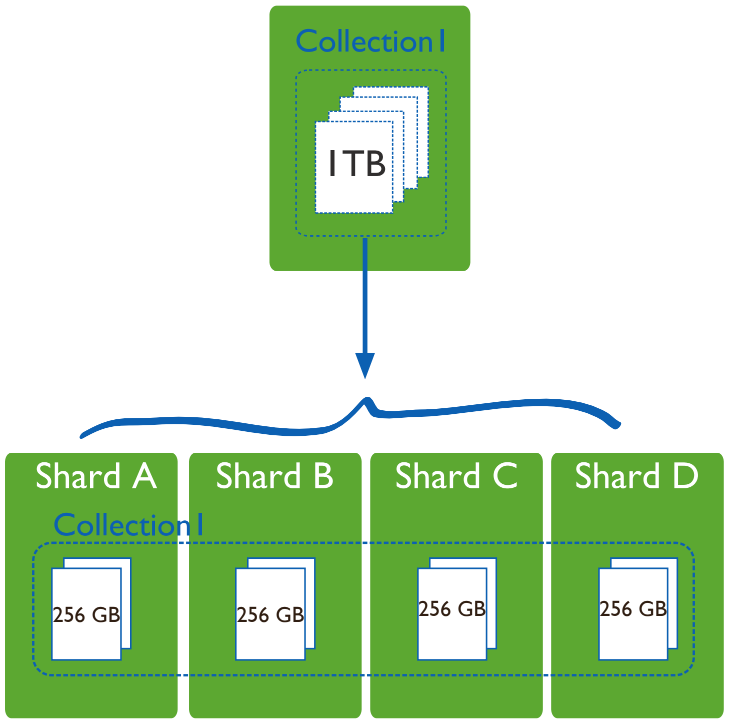
## Vertical Scaling



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

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

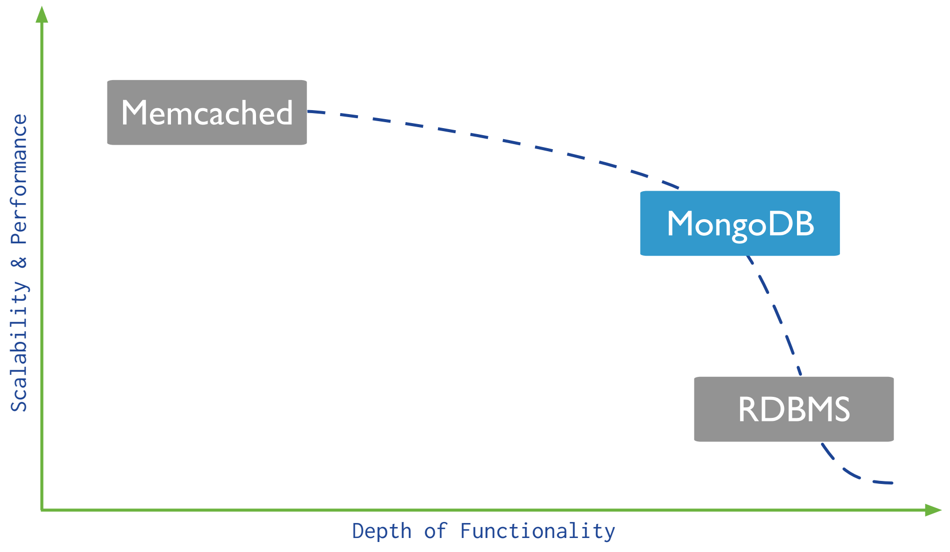
## Scaling with MongoDB



**Note:**

- MongoDB is designed to be horizontally scalable (linear).
- MongoDB scales by enabling you to shard your data.
- When you need more performance, you just buy another machine and add it to your cluster.
- MongoDB is highly performant on commodity hardware.

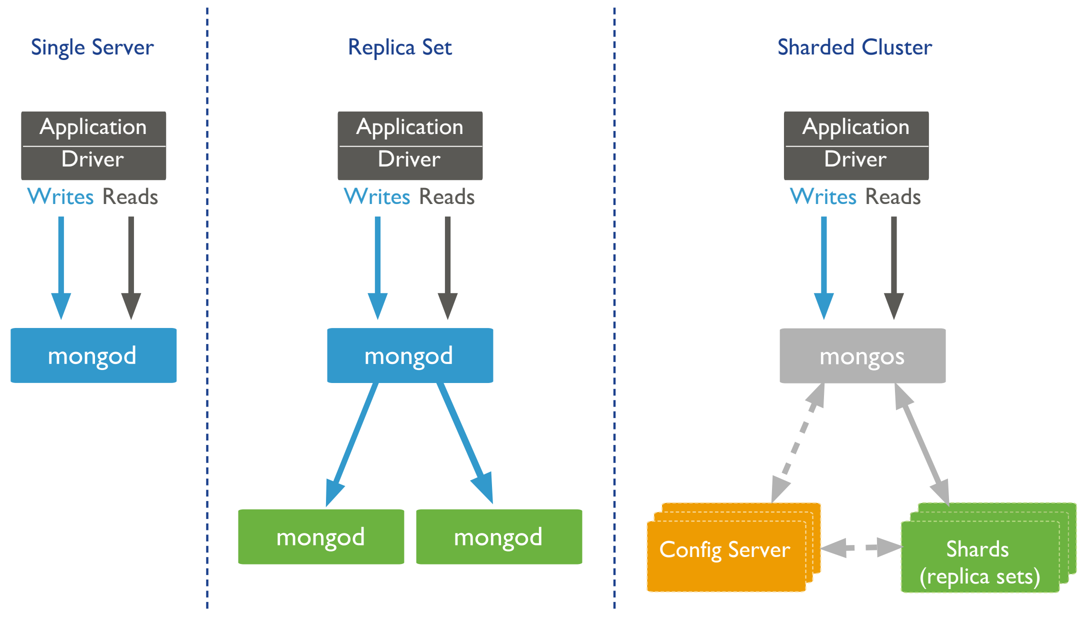
## Database Landscape



**Note:**

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

## MongoDB Deployment Models



**Note:**

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

## 1.3 MongoDB Stores Documents

### Learning Objectives

Upon completing this module, students should understand:

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

### JSON

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

### A Simple JSON Object

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

### JSON Keys and Values

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

[1] http://json.org/

## Example Field Values

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

## 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].

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

## BSON Hello World

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

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

**Note:**

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

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

## 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"
```

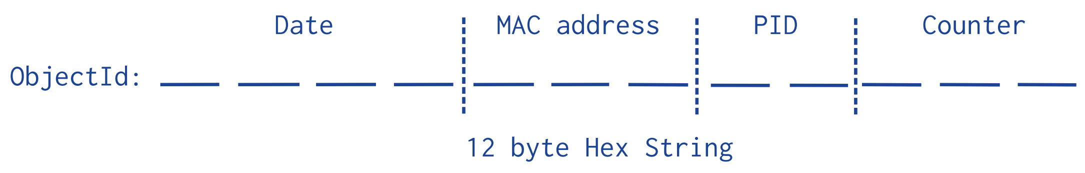
## Documents, Collections, and Databases

- 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

## The `_id` Field

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

## ObjectIds



---

**Note:**

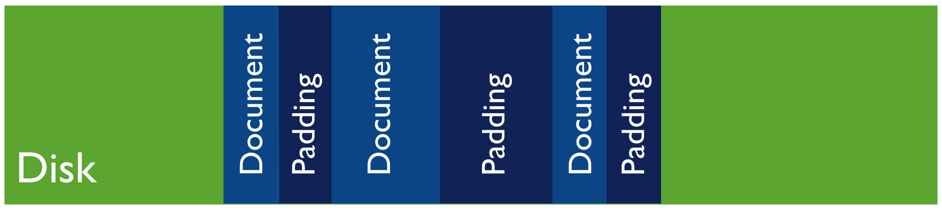
- An ObjectId is a 12-byte value.
- The first 4 bytes are a datetime reflecting when the ObjectID was created.
- The next 3 bytes are the MAC address of the server.
- Then a 2-byte process ID
- Finally, 3 bytes that are monotonically increasing for each new ObjectId created within a collection.

---

## Storing BSON Documents

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

## Padding Factor



---

**Note:**

- Padding provides room for documents to grow into.
- As documents in a collection grow and need to be moved, MongoDB will begin to add padding.
- With padding, documents will not be as likely to move after update operations.
- The `padding factor` is a multiplier that defaults to 1 (no padding).

- At a padding factor of 2, the document will be inserted at twice its actual size.
- This setting is not tunable; it is updated automatically.

---

### `usePowerOf2Sizes`

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

---

**Note:**

- Power of two sizes makes it easier for MongoDB to find new document to fill fragmented space.
- Power of two sizing was introduced in MongoDB 2.4, but must be enabled using the colMod operation.
- If a collection is read only, users should disable `usePowerof2Sizes` in MongoDB 2.6 and above.

---

## 1.4 Exercise: Installing MongoDB

### Learning Objectives

Upon completing this exercise students should understand:

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

## Production Releases

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

- Windows
- OSX
- Linux
- Solaris

## Installing MongoDB

- Visit http://www.mongodb.org/downloads
- Download and install the appropriate package for your machine.
- Even-numbered builds are production releases, e.g., 2.4.x, 2.6.x.
- Odd-numbers indicate development releases, e.g., 2.5.x, 2.7.x.
- 32-bit versions should NOT be used in production (limited to 2GB of data). They are acceptable for training classes.

## Setup

```
PATH=$PATH:path_to_mongodb/bin
```

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

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

**Note:**

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

## Launch a `mongod`

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

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

**Note:**

- Open a command shell and explore the mongod command.
- Then run mongodb.

**Note:**

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

## Import Exercise Data

```
cd usb_drive
```

```
unzip sampledata.zip
```

```
cd sampledata
```

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

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

```
cd dump
```

```
mongorestore -d sample training
```

```
mongorestore -d sample digg
```

**Note:**

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

## Launch a Mongo Shell

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

```
mongo
```

Display available commands.

```
help
```

## Explore Databases

Display available databases.

```
show dbs
```

To use a particular database we can type the following.

```
use <database_name>
```

```
db
```

**Note:**

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

## Exploring Collections

```
show collections
```

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

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

**Note:**

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

## Admin Commands

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

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

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

## The MongoDB Data Directory

```
ls /data/db
```

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

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

- sample.0 is 64 MB
- sample.1 is 128 MB
- sample.2 is 256 MB, etc.
- This continues until sample.5, which is 2 GB
- All subsequent data files are also 2 GB.

# 2 CRUD

*Creating and Deleting Documents* **(page 24)** Inserting documents into collections, deleting documents, and dropping collections.

*Reading Documents* **(page 30)** The find() command, query documents, dot notation, and cursors.

*Query Operators* **(page 36)** MongoDB query operators including: comparison, logical, element, and array operators.

*Updating Documents* **(page 39)** Using `update()` and associated operators to mutate existing documents.

## 2.1 Creating and Deleting Documents

### Learning Objectives

Upon completing this module students should understand:

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

### Creating New Documents

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

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

### Exercise: Inserting a Document

Experiment with the following commands.

```
use sample

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

db.hellos.find()
```

---

**Note:**

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

---

## Implicit `_id` Assignment

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

## Exercise: Assigning _ids

Experiment with the following commands.

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

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

---

**Note:**

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

---

## Inserts will fail if...

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

## Exercise: Inserts will fail if...

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

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

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

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

---

**Note:**

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

  ```
  db.hellos.insert( { _id : [ 1, 2, 3 ] } )
  ```

- The second insert with `_id : 3` will fail because there is already a document with `_id` of 3 in the collection.
- The following will fail because it is a malformed document (i.e. no field name, just a value).

```
db.hellos.insert( { "hello" } )
```

---

## Bulk Inserts

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

---

**Note:**

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

---

## Ordered Bulk Insert

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

## Exercise: Ordered Bulk Insert

Experiment with the following bulk insert.

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

db.things.find()
```

---

**Note:**

- This example has a duplicate key error.
- Only the first 2 documents will be inserted.

---

## Unordered Bulk Insert

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

## Exercise: Unordered Bulk Insert

Experiment with the following bulk insert.

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

## The Shell is a JavaScript Interpreter

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

## Exercise: Creating Data in the Shell

Experiment with the following commands.

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

db.stuff.find()
```

## Deleting Documents

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

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

## Using `remove()`

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

## Exercise: Removing Documents

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

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

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

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

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

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

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

## Dropping a Collection

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

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

## Exercise: Dropping a Collection

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

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

## Dropping a Database

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

## Exercise: Dropping a Database

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

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

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

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

## 2.2 Reading Documents

## Learning Objectives

Upon completing this module students should understand:

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

## 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.

## Exercise: Querying by Example

Experiment with the following sequence of commands.

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

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

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

**Note:** Matching Rules:

- Any field specified in the query must be in each document returned.
- Values for returned documents must match the conditions specified in the query document.

- If multiple fields are specified, all must be present in each document returned.
- Think of it as a logical AND for all fields.

## Querying Arrays

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

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

## Exercise: Querying Arrays

Experiment with the following sequence of commands.

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

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

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

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

## Querying with Dot Notation

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

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

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

## Exercise: Querying with Dot Notation

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

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

## Exercise: Arrays and Dot Notation

Experiment with the following commands.

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

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

**Note:**

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

## 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.

## Exercise: Introducing Cursors

Experiment with the following commands.

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

it
it
```

**Note:**

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

## Exercise: Cursor Objects in the Mongo Shell

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

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

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

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

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

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

## 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.

## Exercise: Using `count()`

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

// all 100
db.testcol.count()

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

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

**Note:**

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

## Exercise: Using `sort()`

Experiment with the following sort commands.

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

db.testcol.find()

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

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

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

**Note:**

- Sort can be executed on a cursor until the point where the first document is actually read.

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

## The `skip()` Method

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

## The `limit()` Method

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

## The `distinct()` Method

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

## Exercise: Using `distinct()`

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

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

## 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

### Exercise: Comparison Operators

Experiment with the following.

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

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

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

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

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

## 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.

## Exercise: Logical Operators (Setup)

Create a collection we can experiment with.

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

## Exercise: Logical Operators

Experiment with the following.

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

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

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

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

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

---

**Note:**

- `db.testcol.find( { a :  { $not :  { $gt :  3 } } } )`
  - Different from `db.testcol.find( { a :  { $lte :  3 } } )`
  - Returns all documents where `a` is less than or equal to 3
  - Also returns documents for which `a` does not exist
- Without the use of `$and` in the last query above:
  - The second `$or` would replace the first.
  - The second `$or` would be the only condition evaluated in the query.

---

## 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.

## Exercise: Element Operators

Experiment with the following.

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

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

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

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

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

## 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

## Exercise: Array Operators

Experiment with the following.

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

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

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

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

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

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

---

**Note:**

- Comparing the last two queries demonstrates `$elemMatch`.
- For the query using `$elemMatch` at least one element must match both conditions.
- For the last query, there must be at least one element that matches each of the conditions. One element can match the `$gte` condition and another element can match the `$lte` condition.

---

## 2.4 Updating Documents

## Learning Objectives

Upon completing this module students should understand

- The `update()` method
- The required parameters for `update()`
- Field update operators
- Array update operators
- The concept of an upsert and use cases.

## The `update()` Method

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

## Parameters to `update()`

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

## `$set` and `$unset`

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

## Exercise: `$set` and `$unset`

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

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

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

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

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

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

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

**Note:**

- db.testcol.update( { _id : 3 }, { $set : { a : 6 } } ) just updates the `a` field.
- db.testcol.update( { _id : 5 }, { $set : { c : 5 } } ) adds a `c` field to the matching document.
- db.testcol.update( { _id : 5 }, { $set : { c : 7 , a : 7 } } ) modifies only the `c` and `a` fields.
- db.testcol.update( { _id : 5 }, { d : 4 } ) is the type of update that is probably a mistake.
  - It will replace the document with _id : 5 in its entirety.
  - The new document will be, { d : 4 }.
- db.testcol.update( { _id : 4 }, { $unset : a } ) removes the `a` field.

## Update Operators

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

## Exercise: Update Operators

Experiment with the following update operators.

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

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

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

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

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

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

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

## `update()` Defaults to one Document

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

## Updating Multiple Documents

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

## Exercise: Multi-Update

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

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

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

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

---

**Note:**

- `db.testcol.update( { _id : { $lt : 5 } }, { $set : { a : 6 } } )` only updates one document.
- `db.testcol.update( { _id : { $lt : 5 } }, { $set : { a : 7 } }, { multi : true } )` updates four documents.

---

## Array Operators

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

---

**Note:**

- These operators may be applied to array fields.

---

## Exercise: Array Operators

Experiment with the following updates.

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

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

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

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

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

**Note:**

- If you have time you might want to show the students:

  ```
  db.testcol.update( { _id : 1 }, { $push : { b : [ 3, 12 ] } } )
  ```

- In `{ $pop : { b : "" } }`, the value passed for `b` is just a placeholder. Most values will work fine.
- In `{ $pullAll : { b : [ 1, 2, "NA", 4 ] } }` the value `4` is not present in the array found in `b`. This has no effect on the update.

## The Positional `$` Operator

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

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

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

## Exercise: The Positional $ Operator

Experiment with the following commands.

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

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

## Upserts

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

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

## Upsert Mechanics

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

## Exercise: Upserts

Experiment with the following upserts.

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

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

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

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

**Note:**

```
// updates the document with a: 4 by incrementing b
db.testcol.update( { a: 4 }, { $inc: { b : 3 } }, { upsert: true } )

// 1) creates a new document, 2) assigns an _id, 3) sets a to 12,
// 4) performs the update
db.testcol.update( { a: 12 }, { $inc: { b : 3 } }, { upsert: true } )

// 1) creates a new document, 2) sets _id: 6 and a: 6,
// 3) update deletes a and sets c to 155.
db.testcol.update( { _id: 6, a: 6 }, { c: 155 }, { upsert: true } )
```

## save()

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

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

## Exercise: save()

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

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

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

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

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

## Be Careful with save()

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

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

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

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

doc.c = 11
doc
```

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

# 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

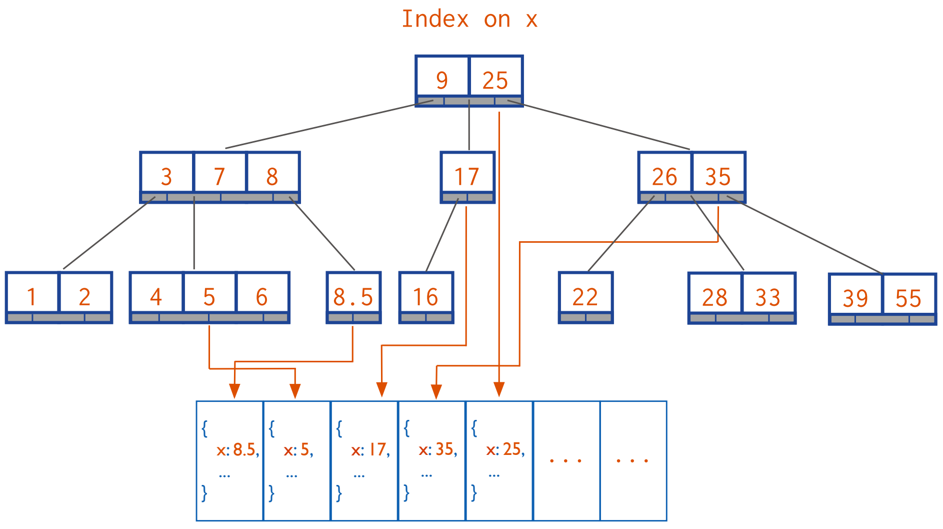
---

**Note:**

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

---

### Why Indexes?



---

**Note:**

- Without an index, in order to find all documents matching a query, MongoDB must scan every document in the collection.

- This is murder for read performance, and often write performance, too.
- If all your documents do not fit into memory, the system will page data in and out in order to scan the entire collection.
- An index enables MongoDB to locate exactly which documents match the query and where they are located on disk.
- MongoDB indexes are based on B-trees.

## Types of Indexes

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

**Note:**

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

## Exercise: Using `explain()`

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

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

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

**Note:**

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

## Results of `explain()`

You will see results similar to the following.

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

## Understanding `explain()` Output

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

## Single-Field Indexes

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

## Creating an Index

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

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

**Note:**

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

## Indexes and Read/Write Performance

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

## Index Limitations

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

## Use Indexes with Care

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

---

**Note:**

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

---

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

- Range queries, e.g.,

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

- Sorting, e.g.,

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

---

**Note:**

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

---

## Designing Compound Indexes

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

## Example: A Simple Message Board

Requirements:

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

## Load the Data

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

## Start with a Simple Index

Start by building an index on { timestamp : 1 }

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

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

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

**Note:**

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

## Query Adding `username`

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

**Note:**

- Let's add the `user` field to our query.
- Now `nscanned` > `n`.

## Include `username` in Our Index

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

**Note:**

- `nscanned` is still greater than n.
- Why?

## `ncanned` > n

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

**Note:**

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

## A Different Compound Index

Drop the index and build a new one with user.

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

**Note:**

- Now `nscanned` is 2 and `n` is 2.

## `nscanned` == `n` == 2

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

**Note:**

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

## Let Selectivity Drive Field Order

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

## Adding in the Sort

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

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

**Note:**

- Note that the `scanAndOrder` field is set to true.
- This means that MongoDB had to perform a sort in memory.
- In memory sorts for queries that retrieve large numbers of documents can degrade performance significantly.
- Especially, if they are used frequently.

## In-Memory Sorts

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

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

**Note:**

- The explain plan remains unchanged.
- The field being sorted on comes after the range fields.
- The index does not store entries in order by rating.
- To have the index structured this way we need to specify the field for sorting (rating) before the range field (timestamp).
- 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.ensureIndex( { username : 1, rating : 1, timestamp : 1 },
                         { name : "myindex" } );
db.messages.find( {
                    timestamp : { $gte : 2, $lte : 4 },
                    username : "anonymous"
                  } ).sort( { rating : -1 } ).explain();
```

**Note:**

- We have a tradeoff between `nscanned` and `scanAndOrder`
- Now `scanAndOrder` is set to `false`.
- However, `nscanned` is 3 and and `n` is 2.
- This is the best we can do in this case and in this situation is fine.
- However, if `nscanned` is much larger than `n`, this might not be the best index.
- You will have to evaluate your use case to determine how to make this tradeoff.

## General Rules of Thumb

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

## 3.3 Multikey Indexes

## Learning Objectives

Upon completing this module, students should understand:

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

## Introduction to Multikey Indexes

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

## Example: Array of Numbers

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

---

**Note:**

```
// Used the index
db.testcol.find( { x : 3 } ).explain()

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

---

## Exercise: Array of Documents, Part 1

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

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

---

**Note:**

- In this collection there are four documents.
- In each document is an array, x, containing subdocuments.
- Each subdocument has a name and number in it
- Always the same number for each name.

---

## Exercise: Array of Documents, Part 2

For each of the three queries below:

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

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

---

**Note:**

```
// 3 documents
db.testcol.find( { x : { name : "Cherry", number : 3 } } )
// Used the multi-key index. We pass a complete document for ``x``.
db.testcol.find( { x : { name : "Cherry", number : 3 } } ).explain()
```

```
// 3 documents
db.testcol.find( { "x.number" : 3 } )
// Does not use the multi-key index.
// ``x.number`` is only part of the document that is indexed.
db.testcol.find( { "x.number" : 3 } ).explain()

// We would need to add an index such as this.
db.testcol.ensureIndex( { "x.number" : 1 } )
db.testcol.find( { "x.number" : 3 } ).explain()
```

---

## Exercise: Array of Arrays, Part 1

Add some documents and create an index:

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

## Exercise: Array of Arrays, Part 2

For each of the queries below:

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

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

---

**Note:**

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

// One document found, where the element of x is just a number.
db.testcol.find( { x : 3 } )
// Used the index
db.testcol.find( { x : 3 } ).explain()

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

---

## 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 does 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.ensureIndex( { x : 1 } )

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

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

**Note:**

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

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

## Limitations on Multikey Indexes

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

## Example: Multikey Indexes on Multiple Fields

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

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

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

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

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

## 3.4 Hashed Indexes

## Learning Objectives

Upon completing this module, students should understand:

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

## What is a Hashed Index?

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

## 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.

**Note:**

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

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

## 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 $2^{53}$.

## Creating a Hashed Index

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

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

# 4 Aggregation

## 4.1 Aggregation Tutorial

### Learning Objectives

Upon completing this module students should understand:

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

### Aggregation Basics

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

### The Aggregation Pipeline

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

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

## Aggregation Stages

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

## The Match Stage

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

## Exercise: The Match Stage

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

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

// 2 docs are output from the aggregation pipeline
db.testcol.aggregate( [ { $match : { a : { $lte : 2 } } } ] )
```

## The Project Stage

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

## Exercise: Selecting fields with $project

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

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

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

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

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

---

**Note:**

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

---

## Exercise: Renaming fields with $project

Use the $project operator to rename a field

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

## Exercise: Shaping documents with $project

Experiment with the following projections.

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

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

More about $divide[6] in another lesson.

[6]http://docs.mongodb.org/manual/reference/operator/aggregation/divide/

## A Twitter Dataset

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

## Tweets Data Model

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

## Analyzing Tweets

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

**Note:**

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

## Friends and Followers

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

## Exercise: Friends and Followers

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

**Note:**

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

## Exercise: $match and $project

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

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

---

**Note:**

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

---

## The Group Stage

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

## Group using $avg

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

## Group using $push

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

```
db.tweets.aggregate( [
   { "$group" : { "_id" : "$user.screen_name",
                  "tweet_texts" : { "$push" : "$text" },
                  "count" : { "$sum" : 1 } } },
   { "$sort" : { "count" : -1 } },
   { "$limit" : 5 }
] )
```

## Group Aggregation Operators

The complete list of operators available in the group stage:

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

## Rank Users by Number of Tweets

- One common task is to rank users based on some metric.
- Let’s look at who tweets the most.
- We will use the aggregation framework to do this.

## Process

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

## Exercise: Ranking Users by Number of Tweets

Try this aggregation pipeline for yourself.

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

**Note:**

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

## Exercise: Tweet Source

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

**Note:**

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

## The Unwind Stage

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

## Example: User Mentions in a Tweet

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

## Using $unwind

Who includes the most user mentions in their tweets?

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

**Note:**

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

## Data Processing Pipelines

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

## Most Unique User Mentions

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

## Same Operator ($group), Multiple Stages

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

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

**Note:**

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

## The Sort Stage

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

## The Skip Stage

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

## The Limit Stage

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

## The Out Stage

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

## 4.2 Optimizing Aggregation

### Learning Objectives

Upon completing this module students should understand:

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

### Aggregation Options

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

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

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

### Aggregation Limits

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

## Limits Prior to MongoDB 2.6

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

## Optimization: Reducing Documents in the Pipeline

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

## Optimization: Sorting

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

## Automatic Optimizations

MongoDB will perform some optimizations automatically. For example:

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

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

# 5 Schema Design

## 5.1 Schema Design Core Concepts

### Learning Objectives

Upon completing this module, students should understand:

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

### What is a schema?

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

### Example: Normalized Data Model

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

## Example: Denormalized Version

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

## Schema Design in MongoDB

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

## Three Considerations

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

## Case Study

- A Library Web Application
- Different schemas are possible.

## Author Schema

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

## User Schema

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

## Book Schema

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

## Example Documents: Author

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

## Example Documents: User

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

## Example Documents: Book

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

## Embedded Documents

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

## Example: Embedded Documents

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

## Embedded Documents: Pros and Cons

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

## Linked Documents

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

## Example: Linked Documents

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

## Linked Documents: Pros and Cons

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

## Arrays

- Array of scalars
- Array of documents

## Array of Scalars

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

## Array of Documents

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

## Exercise: Users and Book Reviews

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

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

## Solution A: Users and Book Reviews

Reviews may be queried by user or book

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

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

## Solution B: Users and Book Reviews

Optimized to retrieve reviews by user

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

## Solution C: Users and Book Reviews

Optimized to retrieve reviews by book

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

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

## 5.2 Schema Evolution

## Learning Objectives

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

- Development Phase
- Production Phase
- Iterative Modifications

## 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.ensureIndex({ "lastName": 1 })

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

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

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

## Production Phase

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

## Production Phase: Read Patterns

List books by author last name

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

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

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

## Addressing List Books by Last Name

"Cache" the author name in an embedded document.

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

Queries are now one step

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

## Production Phase: Write Patterns

Users can review a book.

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

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

Caveats:

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

## Exercise: Recent Reviews

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

## Solution: Recent Reviews, Schema

Store users' reviews in monthly buckets.

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

## Solution: Recent Reviews, Update

Adding a new review to the appropriate bucket

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

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

## Solution: Recent Reviews, Read

Display the 10 most recent reviews by a user

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

num = 0;

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

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

## Solution: Recent Reviews, Delete

Deleting a review

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

## 5.3 Common Schema Design Patterns

## Learning Objectives

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

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

## 1-1 Relationship

Let's pretend that authors only write one book.

## 1-1: Linking

Either side, or both, can track the relationship.

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

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

## 1-1: Embedding

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

## 1-M Relationship

In reality, authors may write multiple books.

## 1-M: Array of IDs

The "one" side tracks the relationship.

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

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

## 1-M: Single Field with ID

The "many" side tracks the relationship.

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

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

## 1-M: Array of Documents

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

## M-M Relationship

Some books may also have co-authors.

## M-M: Array of IDs on Both Sides

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

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

## M-M: Array of IDs on Both Sides

Query for all books by a given author.

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

Query for all authors of a given book

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

## M-M: Array of IDs on One Side

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

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

{
    _id: 5,
    firstName: "Unknown",
    lastName: "Co-author"
}
```

## M-M: Array of IDs on One Side

Query for all books by a given author

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

Query for all authors of a given book

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

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

## Tree Structures

E.g., modeling a subject hierarchy.

## Allow users to browse by subject

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

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

## Alternative: Parents and Ancestors

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

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

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

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

## Find Sub-Categories

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

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

## Summary

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

# 6 Replica Sets

## 6.1 Introduction to Replica Sets

### Learning Objectives

Upon completing this module, students should understand:

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

### Use Cases for Replication

- High Availability
- Disaster Recovery
- Functional Segregation

### High Availability (HA)

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

---

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

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

---

## Disaster Recovery (DR)

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

## Functional Segregation

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

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

---

**Note:**

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

---

## Replication is Not Designed for Scaling

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

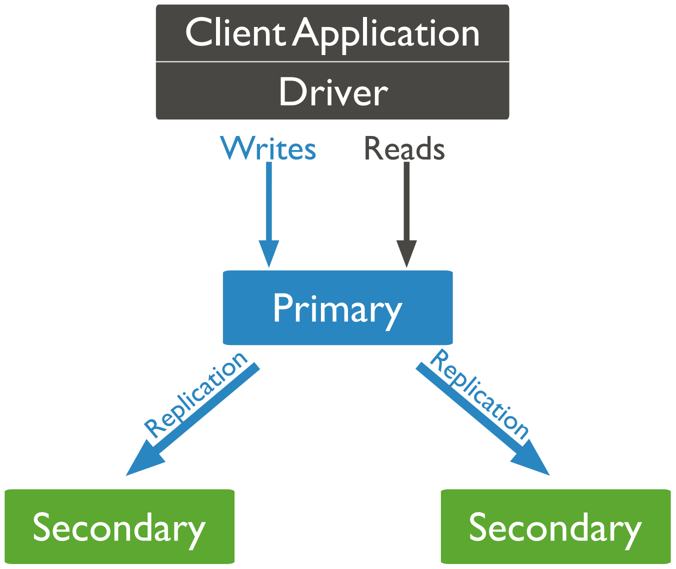
---

**Note:**

- Eventual consistency: This is often tricky to handle as most developers are used to the strong consistency that most databases (and MongoDB) have by default. It also raises the question of how stale the data can be, and what to do when it crosses the threshold (e.g. fall back to reading from the primary).
- Potential system overload: For example, consider a 3 data node replica set using secondaries to scale reads. Each node is serving reads at 70% of its capacity, which is a reasonable utilization rate. What happens if one of

the secondaries fail or is intentionally taken down for maintenance (e.g. upgrading the OS or MongoDB)? Even if the load splits evenly between the 2 remaining nodes, they will be at `70+(70/2)= 105%` capacity.

## Replica Sets



**Note:**

- MongoDB implements replication in the form of replica sets. Don't use the term master-slave as that is what we had before replica sets. It still exists for some corner cases (e.g. > 12 replicas) but should otherwise be avoided.
- A replica set consists of one or more `mongod` servers. Maximum 12 nodes in total and up to 7 with votes.
- There is at most one `mongod` that is "primary" at any one time (though there are edge cases/bugs when there is more than one).
- There are usually two or more other `mongod` instances that are secondaries.
- Secondaries may become primary if there is a failover event of some kind.
- Failover is automatic when correctly configured and a majority of nodes remain.
- The secondaries elect a new primary automatically. A primary may also voluntarily step down, like when it can no longer reach the majority of nodes to avoid a potential split brain scenario.

## Primary Server

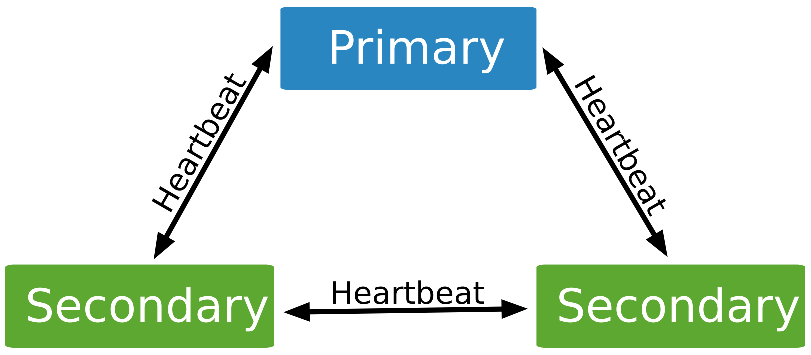
- Clients send writes the primary only.
- MongoDB, Inc. maintains client drivers in many programming languages like Java, C#, Python, Ruby, and PHP.
- MongoDB drivers are replica set aware.

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

## Secondaries

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

## Heartbeats



**Note:**

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

### The Oplog

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

## 6.2 Write Concern

### Learning Objectives

Upon completing this module students should understand:

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

### What happens to the write?

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

### Answer

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

## Balancing Durability with Performance

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

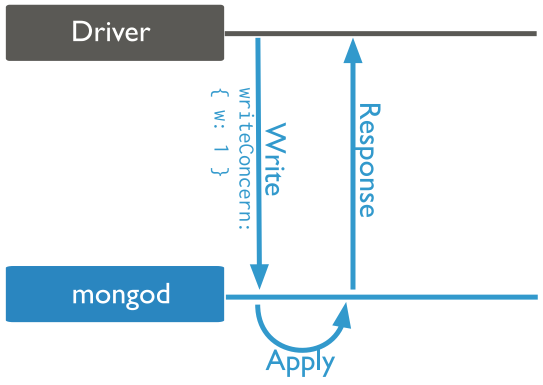
**Note:**

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

## Defining Write Concern

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

## Write Concern: `{ w : 1 }`



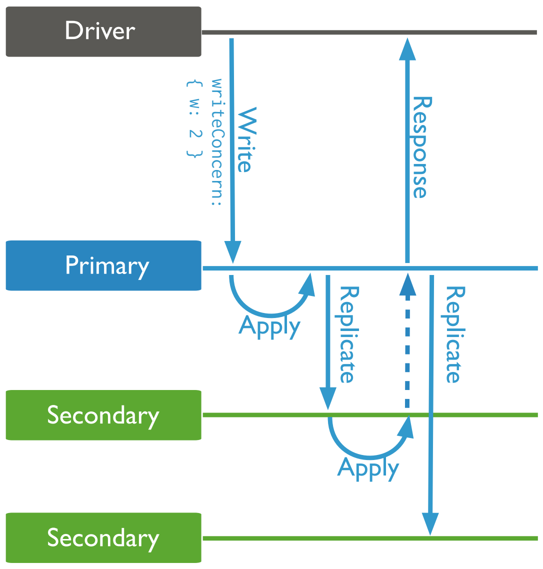
**Note:**

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

## Example: `{ w : 1 }`

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

## Write Concern: `{ w : 2 }`



**Note:**

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

## Example: `{ w : 2 }`

```
db.customer.update({ user : "mary_p" },
                   { $push : { shoppingCart:
                     { _id : 335443, name : "Brew-a-cup",
                       price : 45.79 } } },
                   { 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 members.
- Avoids hard coding assumptions about the size of your replica set into your application.
- Using majority trades off performance for durability.
- It is suitable for critical writes and to avoid rollbacks.

## Example: `{ w : "majority" }`

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

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

## Further Reading

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

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

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

## 6.3 Read Preference

### What is Read Preference?

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

### Use Cases

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

**Note:**

- If you have application servers in multiple data centers, you may consider having a geographically distributed replica set[10] and using a read preference of `nearest`.
- This allows the client to read from the lowest-latency members.
- Use `primaryPreferred` if you want an application to read from the primary under normal circumstances, but to allow possibly stale reads from secondaries during failover.

### Not for Scaling

- In general, do *not* read from secondaries to provide extra capacity for reads.
- Sharding[11] increases read and write capacity by distributing operations across a group of machines.
- Sharding is a better strategy for adding capacity.

[10]http://docs.mongodb.org/manual/core/replica-set-geographical-distribution
[11]http://docs.mongodb.org/manual/sharding

## Read Preference Modes

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

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

## Tag Sets

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

For example, in the mongo shell:

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

# 7 Sharding

## 7.1 Introduction to Sharding

### Learning Objectives

Upon completing this module, students should understand:

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

### Contrast with Replication

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

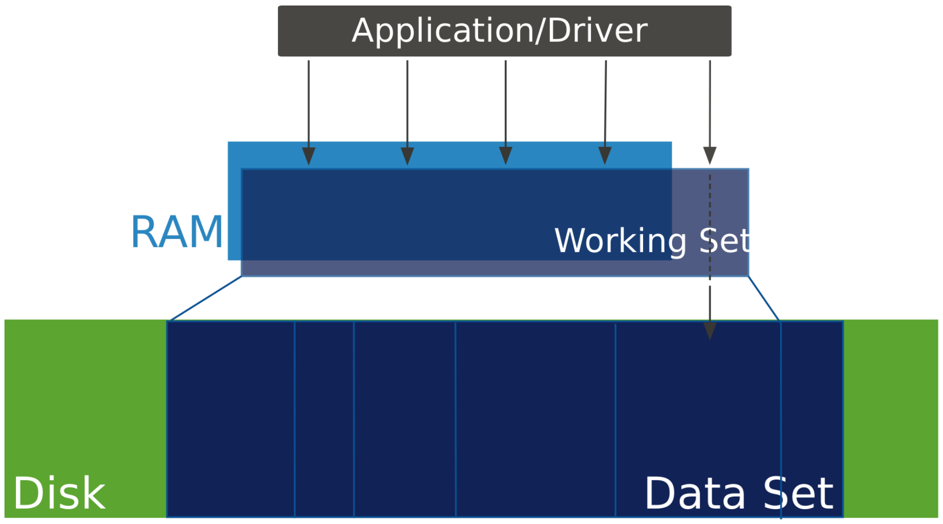
### Sharding is Concerned with Scale

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

## Vertical Scaling

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

## The Working Set



**Note:**

- The working set contains the documents and indexes that are currently being used by an application.
- It is best if the working set fits in RAM.
- In this figure there are many more documented in use than can fit in RAM.
- This would result in page faults affecting performance.
- More RAM would cover a larger area of the database.

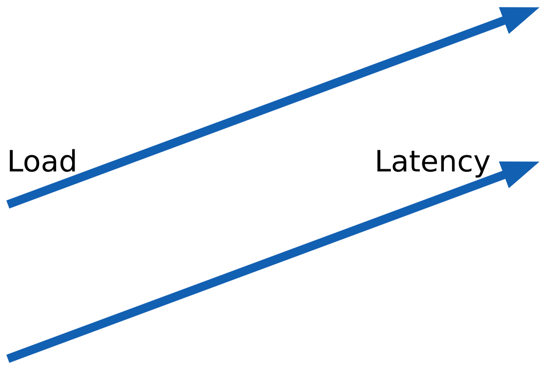
## Limitations of Vertical Scaling

- 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.

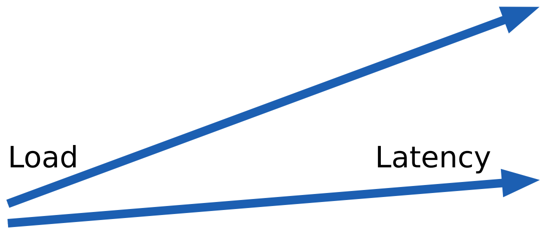
## A Model that Does Not Scale



**Note:**

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

## A Scalable Model

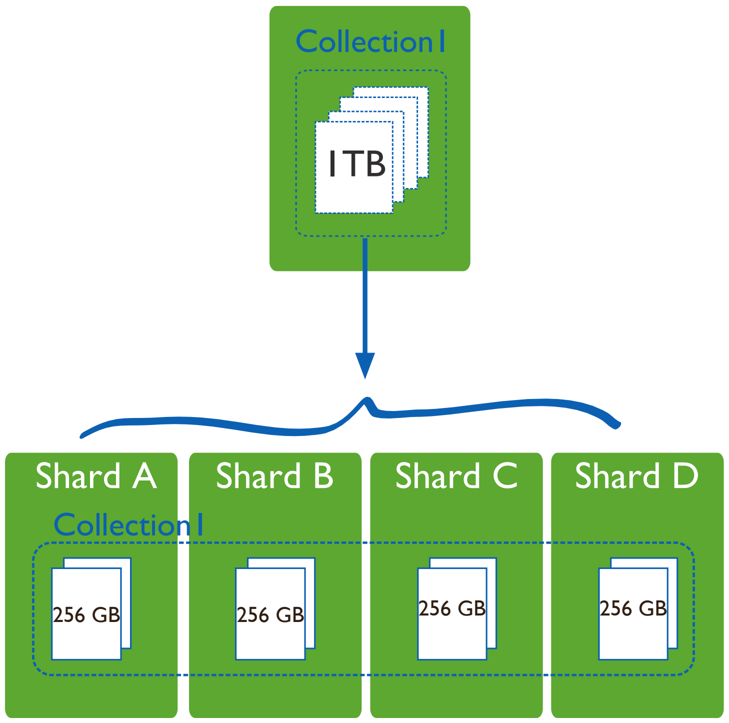


**Note:**

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

- MongoDB scales vertically and horizontally (i.e., spreading out the load across several boxes)

## Sharding Basics



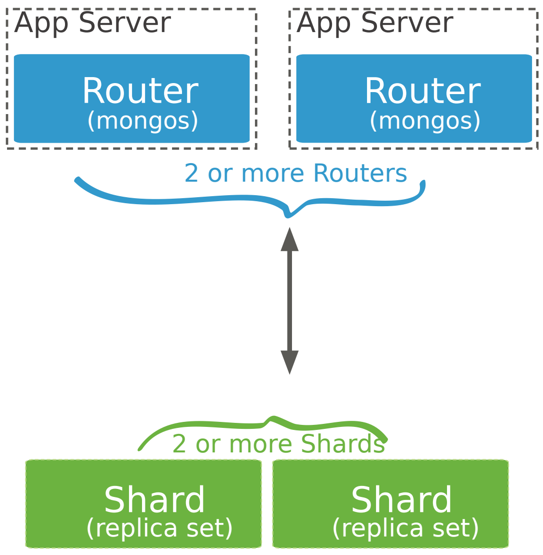
**Note:**

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

## Sharded Cluster Architecture

**Note:**

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



## Mongos

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

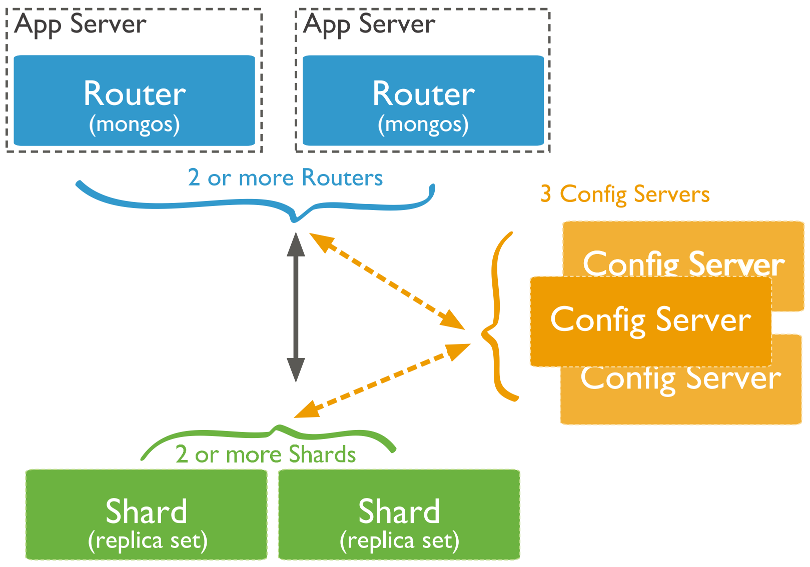
**Note:**

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

## Config Servers

**Note:**

- The previous diagram was incomplete; it was missing config servers.
- Use three config servers in production.
- These hold only metadata about the sharded collections.
  - Where your mongos servers are
  - Any hosts that are not currently available
  - What collections you have
  - How your collections are partitioned across the cluster
- Mongos processes use them to retrieve the state of the cluster.



- You can access cluster metadata from a mongos by looking at the `config` db.

## Config Server Hardware Requirements

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

## When to Shard

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

## Possible Imbalance?

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

## Balancing Shards

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

## What is a Shard Key?

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

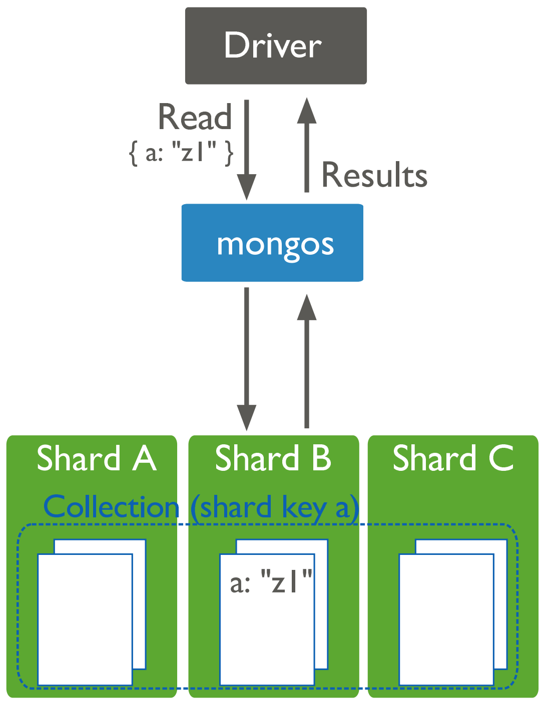
---

**Note:**

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

---

## Targeted Query Using Shard Key



## With a Good Shard Key

You might easily see that:

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

## With a Bad Shard Key

You might see that:

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

## Choosing a Shard Key

Generally, you want a shard key:

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

## More Specifically

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

## Cardinality

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

## Non-Monotonic

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

**Note:**

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

## Shards Should be Replica Sets

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

# 8 Supplementary Material (Time Permitting)

*Geospatial Indexes* **(page 118)** Geospatial indexes: both those on legacy coordinate pairs and those supporting queries that calculate geometries on an earth-like sphere.

*TTL Indexes* **(page 126)** Time-To-Live Indexes.

*Text Indexes* **(page 127)** Free text indexes on string fields.

## 8.1 Geospatial Indexes

### Learning Objectives

Upon completing this module, students should understand:

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

### Introduction to Geospatial Indexes

We can use geospatial indexes to quickly determine geometric relationships:

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

### Easiest to Start with 2 Dimensions

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

---

**Note:**

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

---

## Location Field

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

---

**Note:**

- Draw several Xs to represent other documents.

---

## Find Nearby Documents

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

## Flat vs. Spherical Indexes

There are two types of geospatial indexes:

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

## Flat Geospatial Index

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

## Spherical Geospatial Index

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

## Creating a 2d Index

Creating a 2d index:

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

Possible options key-value pairs:

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

## Exercise: Creating a 2d Index

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

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

**Note:** Answer:

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

## Inserting Documents with a 2d Index

There are two accepted formats:

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

## Exercise: Inserting Documents with 2d Fields

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

---

**Note:** Answer:

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

---

## Querying Documents Using a 2d Index

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

## Example: Find Based on 2d Coords

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

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

## Creating a 2dsphere Index

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

```
db.collection.ensureIndex( { <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.
- Includes Point, LineString, Polygon, and combinations of these.

## geoJSON Considerations

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

**Note:**

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

## Simple Types of 2dsphere Objects

**Point**: A single point on the globe

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

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

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

**Note:**

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

## Polygons

Simple Polygon:

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

Polygon with One Hole:

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

## Other Types of 2dsphere Objects

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

## Exercise: Inserting geoJSON Objects (1)

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

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

---

**Note:**

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

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

---

## Exercise: Inserting geoJSON Objects (2)

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

---

**Note:**

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

---

## Exercise: Inserting geoJSON Objects (3)

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

---

**Note:**

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

---

## Exercise: Creating a 2dsphere Index

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

---

**Note:**

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

db.flights.ensureIndex( { destination : "2dsphere" } )

db.flights.getIndexes() // see the indexes.
```

---

### Querying 2dsphere Objects

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

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

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

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

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

## 8.2 TTL Indexes

### Learning Objectives

Upon completing this module students should understand:

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

### TTL Index Basics

- TTL is short for "Time To Live".
- This must index 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.ensureIndex( { 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.testcol.drop()
db.testcol.ensureIndex( { a : 1 }, { expireAfterSeconds : 30 } )

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

## Exercise: Check the Collection

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

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

## 8.3 Text Indexes

## Learning Objectives

Upon completing this module, students should understand:

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

## What is a Text Index?

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

## Creating a Text Index

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

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

## Exercise: Creating a Text Index

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

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

## 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 "coffee cake":

  ```
  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" } }
  ) )
  ```