ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΜΣ ΚΥΒΕΡΝΟΑΣΦΑΛΕΙΑ
ΚΑΙ ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ

MSc CYBERSECURITY
AND DATA SCIENCE
DEPT OF INFORMATICS
UNIVERSITY OF PIRAEUS

Διαχείριση Μεγάλων Δεδομένων

Big Data Management

Εργαστηριακή Διάλεξη MongoDB

Σταύρος Μαρούλης
stavmars@athenarc.gr

DATA STORIES

Outline

- Overview
- Installation
- MongoDB vs. PostgreSQL
- Data Modeling
- Data Types
- Database, Collection & Document Operations
- Relationships
- Indexing
- Database Replication

Overview

- Cross-platform, document-oriented database providing:
    - High performance
    - High availability
    - Easy scalability

mongoDB

- (Basic) Terminology:
    - **Database:** Group of MongoDB Collections.
    - **Collection**: Group of MongoDB Documents.
        - Equivalent of an RDBMS table.
        - Does not enforce a certain schema → Documents within a Collection can have different fields.
    - **Document**: A set of key-value pairs.
        - Dynamic schema → documents in the same collection do not need to have the same set of fields/structure
        - Common fields in a collection's documents may hold different types of data

Installation

- To install MongoDB Community Edition (CE) please follow the instructions (for the OS of your choice) at https://docs.mongodb.com/manual/installation/

- To run the latest version of MongoDB as a Docker Container:

```
docker run -d -p 27017:27017 --name MONGO_CONTAINER mongo:latest
```

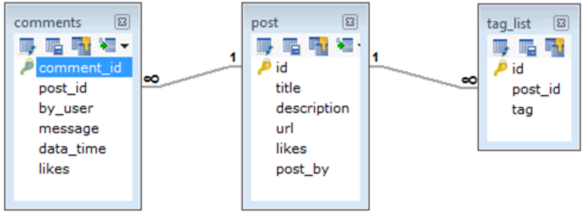
MongoDB vs. PostgreSQL

- Why use MongoDB:
    - Document-Oriented Storage
    - Data model and query language based on JSON (BSON)
    - No complex joins
    - Schema-less
    - Full Index Support
    - Replication and high availability
    - Auto-sharding
    - Ease of scale-out
    - Rich queries
    - Fast in-place updates

MongoDB vs. PostgreSQL (cont.)

PostgreSQL	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key **_id** provided by mongodb itself)

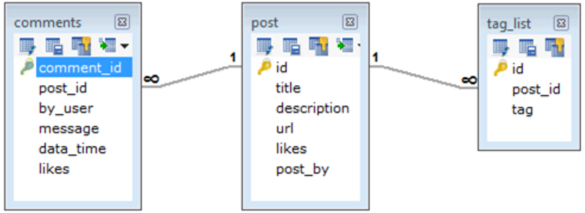
Data Modeling



Suppose we need a non-relational database design for a blog in order to chat with the readers in real-time. The database of that blog has the following requirements.

- Every post has a **unique** title, description, url, the name of its publisher and total number of likes.
- Every post can have **one or more** tags.
- Every post has **comments given by users** along with their name, message, data-time and likes.
  - On each post, there can be zero or more comments.
- In RDBMS schema, we would use three tables (**posts**, **tags** and **comments**).

Data Modeling (cont.)



- When designing a database in **NoSQL** the key challenge in data modeling is balancing:
    - The needs of the application
    - The performance characteristics of the database engine
    - The data retrieval patterns
- **Always** consider the application usage of the data (i.e. queries, updates, and processing of the data) as well as the inherent structure of the data itself.
- Back to our example:
    - Suppose we create an RDMS-like schema
        - To fetch the comments of of a post in the blog, we would need to do a join, which is not -natively- supported in MongoDB (spoilers!) → Not good as far as efficiency is concerned.
    - If we opt for a better, more unified schema, there would be no need for join queries
        - The blog would run smoothly and we would get a paycheck instead of the boot.

```
{
   _id: 001
   title: “Lab III - MongoDB”,
   description: ”Introduction to MongoDB”,
   post_by: “DataStories”,
   url: “datastories.org”,
   tags: [“MongoDB”, “Datatories”, “DBMS”, “NoSQL”],
   likes: 128,
   comments: [
      {
         by_user:'User1',
         message: 'Message1',
         data_time: '24/10/2019, 15:37',
         likes: 64
      },
      {
         by_user:'User2',
         message: 'Message2',
         data_time: '24/10/2019, 15:40',
         likes: 32
      }
   ]
}
```

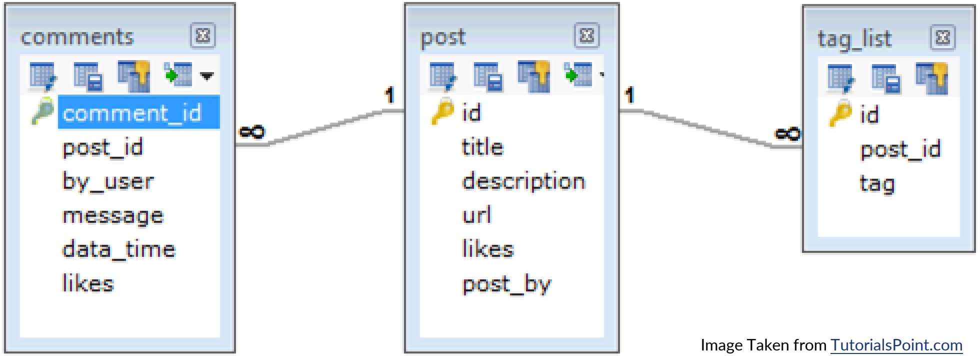


Image Taken from TutorialsPoint.com

Data Types

- String
    - String in MongoDB must be UTF-8 valid.
- Integer
    - 32 bit or 64 bit depending upon your server.
- Boolean
- Double
- Arrays
    - This type is used to store arrays or list or multiple values into one key.
- Date
    - This datatype is used to store the current date or time in UNIX time format
- ObjectId

    - 4 bytes timestamp, 3 bytes machine id, 2 bytes process id (pid), 3 bytes incrementer

MongoDB documentation [3], covers all the data types that can be used.

Database Operations

To access MongoDB, type “**mongosh**” in a Terminal

- Check your currently selected database:
  - **db**
- Check your databases list:
  - **show dbs**
  - **show databases**
- Create Database:
  - **use** <DATABASE_NAME>
- Drop Database:
  - **db.dropDatabase()**

Collection Operations

- Create a Collection:
  - **db.createCollection(name, options)**
- Drop a Collection:
  - **db.**<COLLECTION_NAME>**.drop()**
- View Collections (Within a Database):
  - **show collections**

Document Operations

- Insert Document
    - **db.**<COLLECTION_NAME>**.insertOne(document)**
    - **db.**<COLLECTION_NAME>**.insertMany(documents)**
    - **db.**<COLLECTION_NAME>**.save(document)**
- Example:

```
db.<COLLECTION_NAME>.insertOne([{
   _id: ObjectId(7df78ad8902c),
   title: 'MongoDB 101,
   description: 'Introduction to MongoDB',
   by: 'datastories',
   url: 'http://www.datastories.org',
   tags: ['mongodb', 'database', 'NoSQL', ‘Introduction’],
   likes: 10
}])
```

Document Operations (cont.)

- Query Document
  - **db.**<COLLECTION_NAME>**.find({<key>:<value>})**
- Using the above commands without any clause

→ `SELECT * FROM COLLECTION_NAME;`

- To return only one document → find().limit(1)

→ `SELECT * FROM COLLECTION_NAME LIMIT 1;`

- <value> can vary from a certain key value to a key clause in the format {<condition>:<value>}, where <condition> can be: **$lt(e), $gt(e), $ne**

Document Operations (cont.)

- To return the (famous) people that died during 2016, the SQL Query is:

```
SELECT * FROM name_basics WHERE “deathYear” = 2016;
```

- In MongoDB, the very same query can be written in the following ways:
  - **db**.name_basics.**findOne**({“deathYear”: 2016}) # For getting **only** the first occurence
  - **db**.name_basics.**find**({“deathYear”: 2016}) # For getting **all** occurences

Document Operations (cont.)

- To query documents based on the **AND** condition:
    - Use separate {<key>:<value>} pairs separated by comma (,) in find();

    - Use the **$and** clause within find():

      `$and: [{key₁: value₁}, {key₂:value₂}, …, {keyₙ:valueₙ}]`
- To query documents based on the **OR** condition:
    - Use the **$or** clause within find():

      `$or: [{key₁: value₁}, {key₂:value₂}, …, {keyₙ:valueₙ}]`

Document Operations (cont.)

- The **SQL** Query:

```
SELECT * FROM name_basics WHERE “deathYear”>2016 OR “deathYear”<1955
```

- In **MongoDB** can be written as:

```
db.name_basics.find({'$or':[ {'deathYear':{'$gt': 2016}},
                             {'deathYear':{'$lt': 1955}} ]})
```

- Likewise, the **SQL** Query:

```
SELECT * FROM name_basics WHERE “deathYear”<2016 AND “deathYear”>1955
```

- In **MongoDB** can be written as:

```
db.name_basics.find({'deathYear':{'$lt':2016, '$gt':1955}})
```

Document Operations (cont.)

- Update Document:
  - **db.**<COLLECTION_NAME>**.updateOne**(CRITERIA, UPDATED_DATA, OPTIONS)
  - **db.**<COLLECTION_NAME>**.updateMany**(CRITERIA, UPDATED_DATA, OPTIONS)
  - Example: To find the titles where the value of the field “titleType” is equal to ‘**short**’ and **set** it to ‘**shortMovie**’:

```
db.title_basics. updateMany( {titleType: ‘short'}, {"$set":{titleType : ‘shortMovie’}} )
```

- Another way to update a document → save() method:
  - **db.**<COLLECTION_NAME>.**save**({_id: ObjectId(...), NEW_DATA})
  - Example: To replace the document with _id: “**7df78ad8902c**”:

```
db.mycol.save({"_id" : ObjectId(7df78ad8902c), "title": "Mongo 102", "by": "DataStories.org"})
```

MongoDB documentation [4], covers all the operators that can be used within an update query.

Document Operations (cont.)

- Delete Document:
  - SQL Equivalent:

    ```
    DELETE * FROM table WHERE condition(s);
    ```

  - To remove one occurrence:
    - **db.**<COLLECTION_NAME>**.deleteOne**(DELETION_CRITERIA)
  - To remove all occurrences:
    - **db.**<COLLECTION_NAME>**.deleteMany**(DELETION_CRITERIA)
  - <DELETION_CRITERIA>: Can be any {<key>:<value>} clause as written in the previous slides
- Example: To remove the (famous) people that their (primary) profession is “director” or “assistant director”:
  - `db.name_basics.deleteMany({ 'primaryProfession': {$in: ['director’, 'assistantdirector’]} })`
  - `db.name_basics.deleteOne({'primaryName': 'Lauren Bacall'})`

Projection

- SQL Equivalent:

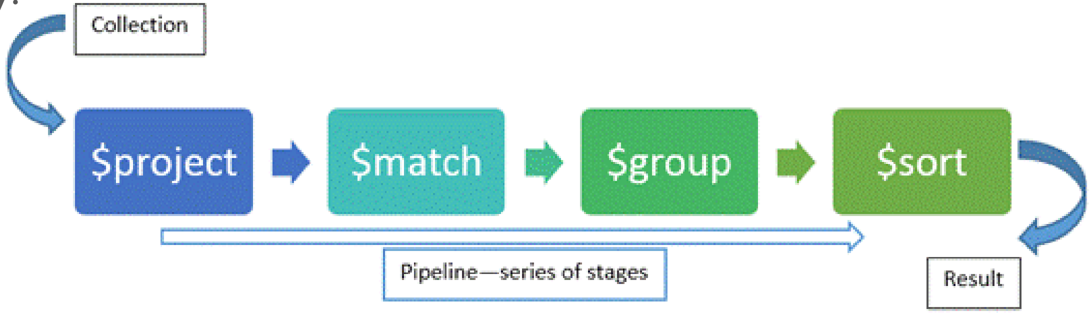
  SELECT key$_1$, key$_2$, …, key$_n$ FROM COLLECTION_NAME WHERE condition(s);

- Within **find()** method:
  - `db.COLLECTION_NAME.find({<KEY-VALUE-CLAUSES>},{<KEY`$_1$`>:1/0, …, <KEY`$_n$`>:1/0})`

- To get the names of the (famous) people who are born on 1946:
  - **SQL Query:**
    - `SELECT “primaryName” FROM name_basics WHERE “birthYear” = 1946;`
  - **MongoDB**
    - `db.name_basics.find({'birthYear':1946}, {'_id':0, 'primaryName':1})`

Aggregating Documents

- Aggregation is a “**pipeline**” and is just exactly that, being “**piped**” processes that feed input into the each stage as it goes along.
- When calling aggregate on a collection, we pass a list of operators.
  - Stages
  - Expressions
  - Accumulators
- Documents are processed through the stages in sequence, with each stage applying to each document individually.



Aggregation Pipeline. Image by Rohan Paul (medium.com)

Aggregating Documents (cont.)

- Basic Command:
    - `db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)`
- Example *(Get the actors that are over 100 years old)*:
    -
```
db.name_basics.aggregate([
        {"$addFields":{'age': {'$subtract':['$deathYear', '$birthYear']}}},
        {'$match':{'age':{'$gt':100}}},
        {'$project':{'_id':0, 'age':0}},
        {'$limit':5} ])
```

    - **SQL** Equivalent:

```
SELECT * FROM name_basics WHERE (“deathYear”-”birthYear”) > 100 LIMIT 5;
```

Aggregating Documents (cont.)

- Adding new Fields:
    - `{$addFields: {'field_1':<value_1>, …, 'field_n':<value_n>}}`
    - Adds new fields to documents and outputs documents that contain all existing fields from the input documents and newly added fields.

- Matching Documents
    - `{$match: {'field_1':<value_1>, …, 'field_n':<value_n>}}`
    - Filters the documents to pass only the ones that match the specified condition(s).

- Projecting Fields
    - `{$project: {'_id:0/1', 'field_1':1, …, 'field_n':1}}`
    - Passes along the documents with the requested fields to the next stage in the pipeline.

- Deconstructing Array Fields
    - `{$unwind: <field path>}`
    - Deconstructs an array field from each document and outputs a document for each element with the value of the array field replaced by the element.

Aggregating Documents (cont.)

- Grouping Documents

    -
      ```
      {$group:{
            _id: <expression>,                     # Group By Expression
            <field1>: { <accumulator1> : <expression1> }, …,
            <fieldn>: { <accumulatorn> : <expressionn> }
      }}
      ```
    - Groups input documents by the specified _id expression
    - The **_id** field of each output document contains the unique **group by** value.
    - The output documents can also contain computed fields that hold the values of some accumulator expression.
- The <accumulator$_i$> operator can be one of the following accumulator operators:
    - $sum, $avg, $min, $max, $push, $addToSet, $first, $last

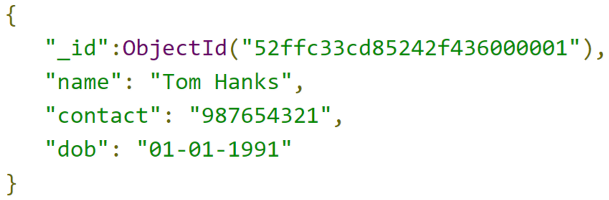
Aggregating Documents (cont.)

- Sorting Documents:
  - `{$sort: {KEY:1/-1, ...}}`
    - 1 (*resp.* -1) → ascending (*resp.* descending) order
    - **Default** behaviour (if no preference is stated) → **Ascending order**
    - SQL Equivalent: `SELECT * FROM COLLECTION_NAME ORDER BY KEY [ASC/DESC];`
- Limiting Documents:
  - `{$limit: {NUMBER}}`
  - SQL Equivalent: `SELECT * FROM COLLECTION_NAME LIMIT NUMBER;`
- Skipping Documents:
  - `{$skip: {NUMBER}}`
  - SQL Equivalent: `SELECT * FROM COLLECTION_NAME OFFSET NUMBER;`

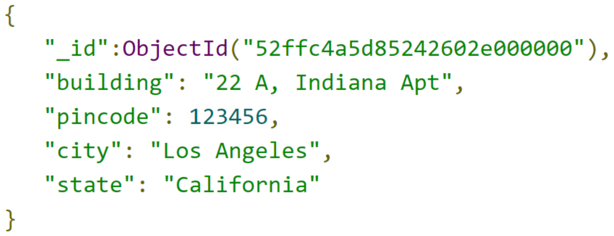
MongoDB documentation [5], covers everything you need to know about the stages, expressions and accumulators that can be used in an aggregation pipeline, while [6, 7] covers a more hands-on example regarding some of the most used operators.

Relationships

- By default, MongoDB does not have a join operator/command
- However, in order to reduce redundancy and read/write overheads we can use:
    - Embedded Relationships
    - Referenced Relationships
    - Database References
- Relationships represent how various documents are logically related to each other.
    - Can be modeled via Embedded and Referenced approaches.
    - Such relationships can be either **1:1**, **1:N**, **N:1** or **N:N**.



```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "name": "Tom Hanks",
   "contact": "987654321",
   "dob": "01-01-1991"
}
```



```
{
   "_id":ObjectId("52ffc4a5d85242602e000000"),
   "building": "22 A, Indiana Apt",
   "pincode": 123456,
   "city": "Los Angeles",
   "state": "California"
}
```

Relationships (cont.)

- In the embedded approach, we will embed the address document inside the user document.
- Pros:
  - All related data are embedded in a single document
  - Easy to retrieve and maintain.
- Cons:
  - Introduces redundancy
  - Document size growth rate increases.
  - May impact read/write performance.
- To get the address of “Tom Hanks”:
  - **db**.users.**findOne**({"name":"Tom Hanks"}, {"address":1})

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Hanks",
   "address": [
      {
         "building": "22 A, Indiana Apt",
         "pincode": 123456,
         "city": "Los Angeles",
         "state": "California"
      },
      {
         "building": "170 A, Acropolis Apt",
         "pincode": 456789,
         "city": "Chicago",
         "state": "Illinois"
      }
   ]
}
```

Relationships (cont.)

- In the referenced approach, both the user and address documents will be maintained separately but the user document will contain a field that will reference the address document's id field.
- Pros:
  - Normalized Structure
- Cons:
  - May increase query complexity.
  - Slower performance if we need to retrieve both the user and their addresses

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Hanks",
   "address_ids": [
      ObjectId("52ffc4a5d85242602e000000"),
      ObjectId("52ffc4a5d85242602e000001")
   ]
}
```

- Example: To get the address of “Tom Hanks”:
  - **var** result = **db.**users.**findOne**({"name":"Tom Hanks"}, {"address_ids":1})
  - **var** addresses = **db.**address.**find**({ "_id": {"$in":result["address_ids"]} })

Database References

- Referenced Relationships (a.k.a. Manual References) are often a quick and easy solution
- However, when a document contains references from different collections → **MongoDB DBRefs** can be of great value when it comes to usability
- Using DBRefs (the field order is essential):
    - **$ref** – This field specifies the collection of the referenced document
    - **$id** – This field specifies the _id field of the referenced document
    - **$db** – This is an optional field and contains the name of the database in which the referenced document lies

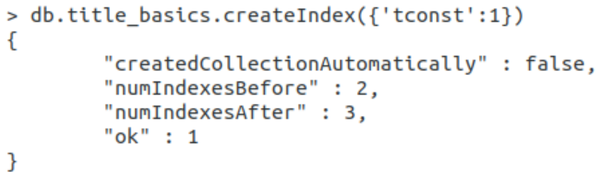
Indexing Documents

- Like traditional RDBMSs, MongoDB supports Indexes for faster queries
    - By default MongoDB indexes use a B-tree data structure
    - Hash Index is also supported, but **only** for equality queries
    - Index can be created on either one or multiple fields (compound index)
        - A compound index cannot include a hashed index component.
- SQL Equivalent:

```
CREATE INDEX IF NOT EXISTS “tconst_0” ON title_basics USING btree (“tconst” ASC|DESC);
```

MongoDB documentation [8], covers everything you need to know regarding indexing documents, while [9, 10] cover a more hands-on example regarding some of the most used index types and their performance gains.

Indexing Documents (cont.)



```
> db.title_basics.createIndex({'tconst':1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 2,
        "numIndexesAfter" : 3,
        "ok" : 1
}
```

- Basic Syntax:
    - B-Tree (default): `db.COLLECTION_NAME.createIndex({<KEY>: 1})`
        - 1/-1 → Ascending/Descending Index
    - Hash Index: `db.COLLECTION_NAME.createIndex({<KEY>: ”hashed”})`
- Index Operations:
    - View Indexes: `db.COLLECTION_NAME.getIndexes()`
    - Delete Indexes: `db.COLLECTION_NAME.dropIndex("<INDEX-NAME>")`
    - View Statistics: `db.COLLECTION_NAME.stats()`
        - `Use .indexSizes to get` **`only`** `the size of index`
- Further parameters can be specified, with few of them are:
    - name (string), unique (boolean), partialFilterExpression (document), expireAfterSeconds (integer)

Analyzing Query Performance

- The **explain** operator provides information regarding:
    - The query
    - The indexes used in a query
    - Several query-oriented statistics
- The **hint** operator instructs the query optimizer to use the specified index to run a query.
- Very useful utilities for:
    - Analyzing how well your indexes are optimized.
    - Testing performance of a query with different indexes.
- Basic syntax:
    - db.users.find({<CLAUSES>}, {<PROJECTION_OPTIONS>}) .explain(“<MODE>”)
- Modes:
    - **queryPlanner**, executionStats, allPlansExecution

Overall...

- Operational Considerations
  - Each index requires at least 8kB of data space.
  - When active, each index will consume some disk space and memory.
    - This is significant when tracked in capacity planning.
  - For a high read-to-write ratio collection, additional indexes improve performance and do not affect un-indexed read operations.
- Limitations
  - Adding an index has some negative performance impact for write operations especially for collections with the high write-to-read ratio.
    - Indexes will be expensive in that each insert must also update any index.
  - Indexes are most effective at retrieving small subsets of data and become less and less efficient as you need to get larger percentages of a collection

Database Replication

- Replication is the process of synchronizing data across multiple servers.
- Provides redundancy and increases data availability (24/7)...
  - ...with multiple copies of data on different database servers.
- Protects a database from the loss of a single server
- Allows you to recover from hardware failure and service interruptions

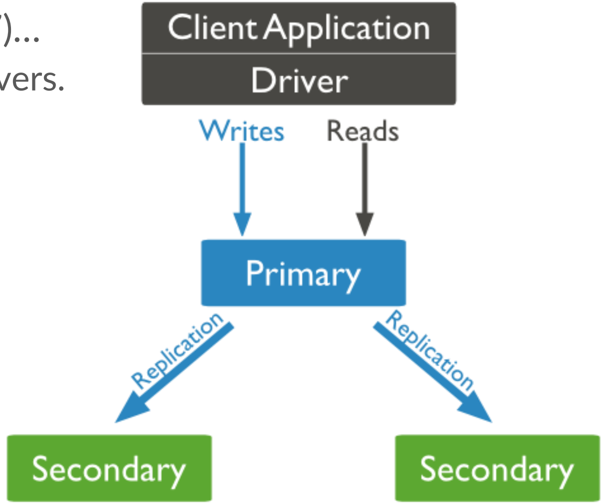


Image Taken from TutorialsPoint.com

Database Replication

- MongoDB achieves replication by the use of replica set.
- A replica set is a group of two or more nodes
- In a replica set:
  - One node is primary node; and
  - All remaining nodes are secondary.
- All data replicates from primary to secondary node.
- At the time of automatic failover or maintenance:
  - Election establishes for primary; and
  - A new primary node is elected.
- After the recovery of failed node:
  - Again joins the replica set; and
  - Works as a secondary node.

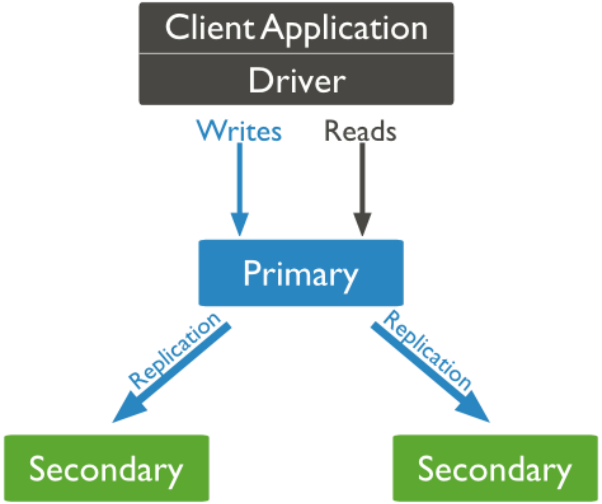


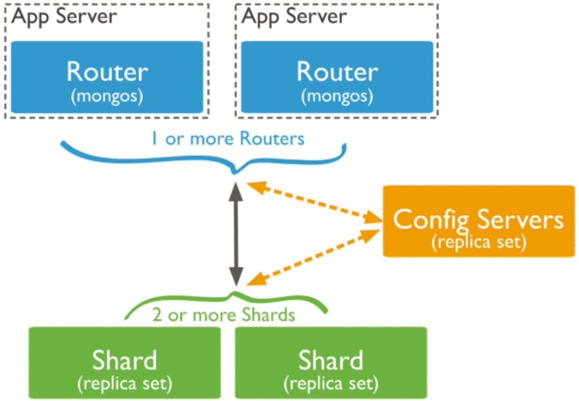
Image Taken from TutorialsPoint.com

Database Replication

- Set up a Replica Set
  - Shutdown already running MongoDB server.
  - Start the MongoDB server by specifying -- replSet option.
- Basic “--replSet” Syntax:
  - **mongod --port** "PORT" **--dbpath** "YOUR_DB_DATA_PATH" **--replSet** "REPLICA_SET_INSTANCE_NAME"
- Add Members to Replica Set:
  - **rs.add(HOST_NAME:PORT)**
  - NOTE:You can add mongod instance to replica set only when you are connected to primary node.
    - Check if you’re connected to master node → **db.isMaster()**

... is this the end?

- ... Only the beginning. We've only scratched the surface.
- MongoDB is capable of more advanced concepts [15, 16] such as:
    - Build Clusters for Database Sharding (distributing data across multiple machines)
    - ... and lots more...



Thank you for your Attention!!

References

1. A (friendly) Introduction to MongoDB, https://www.tutorialspoint.com/mongodb/index.htm
2. MongoDB Documentation, https://docs.mongodb.com/
3. MongoDB Data Types, https://docs.mongodb.com/manual/reference/bson-types/
4. MongoDB Update Operators, https://docs.mongodb.com/manual/reference/operator/update/
5. MongoDB Aggregation Pipeline, https://docs.mongodb.com/manual/core/aggregation-pipeline/
6. Aggregation in MongoDB, https://medium.com/@paulrohan/aggregation-in-mongodb-8195c8624337
7. Aggregation in MongoDB II, https://www.codeproject.com/Articles/1149682/Aggregation-in-MongoDB
8. MongoDB Indexes, https://docs.mongodb.com/manual/indexes/
9. Understanding MongoDB Indexes, https://severalnines.com/database-blog/understanding-mongodb-indexes
10. MongoDB Indexes and Performance, https://hackernoon.com/mongodb-indexes-and-performance-2e8f94b23c0a
11. MongoDB Text Indexes, https://docs.mongodb.com/manual/core/index-text/
12. Full-Text Search in MongoDB, https://code.tutsplus.com/tutorials/full-text-search-in-mongodb--cms-24835
13. How to Speed-Up MongoDB Regex Queries by a Factor of up-to 10, https://medium.com/statuscode/how-to-speed-up-mongodb-regex-queries-by-a-factor-of-up-to-10-73995435c606
14. Analyze Query Performance in MongoDB, https://docs.mongodb.com/manual/tutorial/analyze-query-plan/
15. MongoDB Advanced Concepts - Replication and Sharding, https://www.slideshare.net/knoldus/mongodb-advance-concepts-replication-and-sharding
16. MongoDB Advanced Concepts - MapReduce, https://docs.mongodb.com/manual/core/map-reduce/