MongoDB 3.x

ACID

- **Atomicity**
  - "**all or nothing**": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged
- **Consistency**
  - Ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules (constraints, cascades, triggers etc).

ACID

- **Isolation**
  - Ensures that the concurrent execution of transactions results in a system state that could have been obtained if transactions are executed serially
- **Durability**
  - Means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors

CAP Theorem

- **Consistency**

A service that is consistent should follow the rule of ordering for updates that spread across all replicas in a cluster - “what you write is what you read”, regardless of location.

- **Availability**

A service should be available. There should be a guarantee that every request receives a response about whether it was successful or failed.

- **Partition Tolerance**

The system continues to operate despite arbitrary message loss or failure of part of the system.

CAP Theorem continued…

Though its desirable to have Consistency, High-Availability and Partition-tolerance in every system, unfortunately no system can achieve all three at the same time.

BASE Model

- **Basic Availability:**

Data will be available even in case of multiple failures. This is possible by spreading the data across many storage systems with a high degree of replication.

- **Soft state:**

The state of the system could change over time, so even during times without input there may be changes going on due to 'eventual consistency,' thus the state of the system is always 'soft.'

- **Eventual consistency:**

The system will eventually become consistent once it stops receiving input. The data will propagate to everywhere it should sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one.

Eventual Consistency

- Given a sufficiently long period of time over which no changes are sent, all updates can be expected to **propagate eventually** through the system and all the replicas will be consistent

- Conflict resolution:
  - **Read repair:** The correction is done when a read finds an inconsistency. This slows down the read operation.
  - **Write repair:** The correction takes place during a write operation, if an inconsistency has been found, slowing down the write operation.
  - **Asynchronous repair:** The correction is not part of a read or write operation.

ACID vs. BASE

Sr.No.	ACID (used in RDBMS)	BASE (used in NoSQL)
1.	Strong consistency	Weak consistency (Stale data OK)
2.	Isolation	Last write wins
3.	Transaction	Program managed
4.	Robust database	Simple database
5.	Consistent & Partion-tolerant	Available & Partition-tolerant

RDBMS vs NoSQL

Sr.No.	RDBMS	NoSQL
1.	Handles Limited Data Volumes	Handles Huge Data Volumes
2.	Vertically scaled (Scale-in)	Horizontally scaled (Scale-out)
3.	SQL is used as query language	No declarative query language
4.	Predefined Schema	Schema less
5.	Supports relational data and its relationships are stored in separate tables	Supports unstructured and unpredictable data
6.	Based on ACID model	Based on BASE model
7.	Transaction Management is strong	Transaction Management is weak

NoSQL introduction

- NoSQL is a non-relational database management system, different from traditional RDBMS in some significant ways
- **Carlo** Strozzi used the term NoSQL in 1998 to name his lightweight, open-source relational database that did not expose the standard SQL interface

NoSQL introduction

- In 2009, **Eric** Evans reused the term to refer databases which are non-relational, distributed, and does not conform to ACID
  - The NoSQL term should be used as in the Not-Only-SQL and not as No to SQL or Never SQL

When to use NoSQL

1. You need to handle extremely large data sets.
2. You need extremely fast in-memory data.
3. You need Schema less & de-normalized database.
4. You want to handle database in Object oriented fashion.

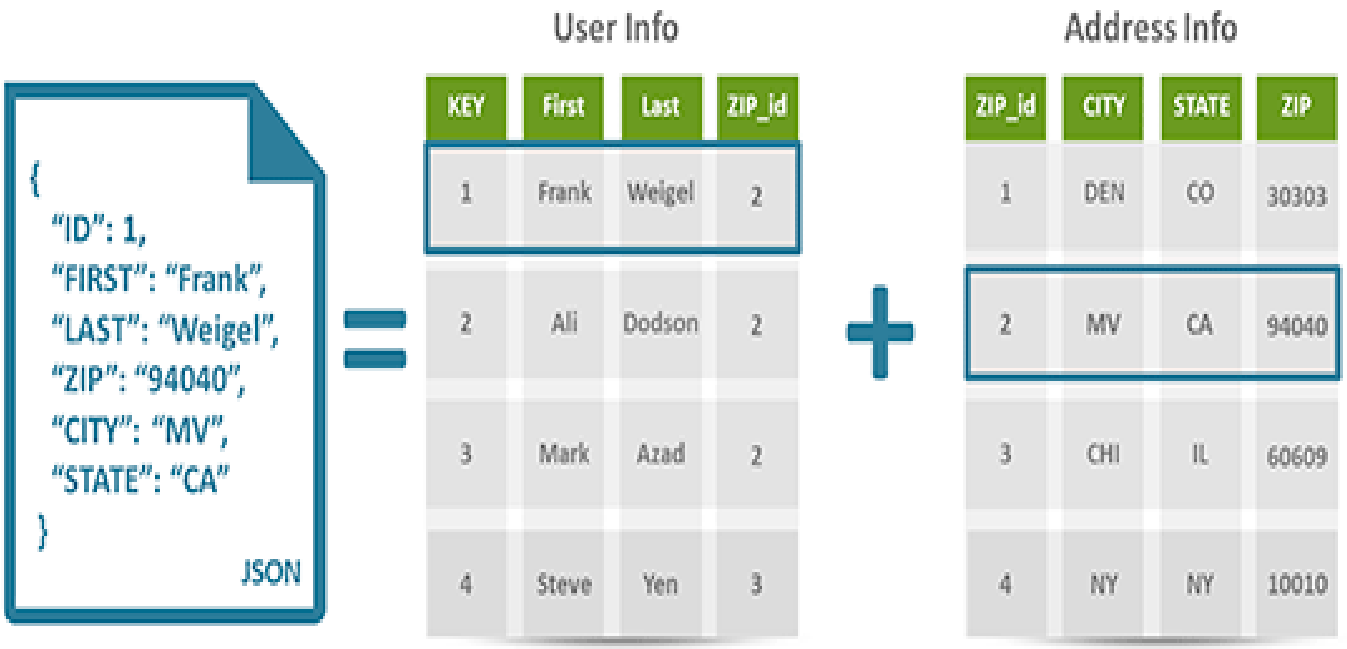
NoSQL (Not only SQL)

- **Document DBs**
  - MongoDB, CouchDB, …
- **Graph DBs**
  - Neo4j, FlockDB…
- **Column oriented DBs**
  - HBase, Cassandra, BigTable…
- **Key-Value DBs**
  - Memcache, MemcacheDB, **Redis**, Voldemort, Dynamo…

Document based DBs

The data which is a collection of key value pairs is compressed as a document store quite similar to a key-value store. However, the only difference is that the values stored (referred as "documents") provide some structure and encoding of the managed data. XML, JSON, BSON (Binary JSON) are some common standard encodings.

Document based DBs continue…



Graph based DBs

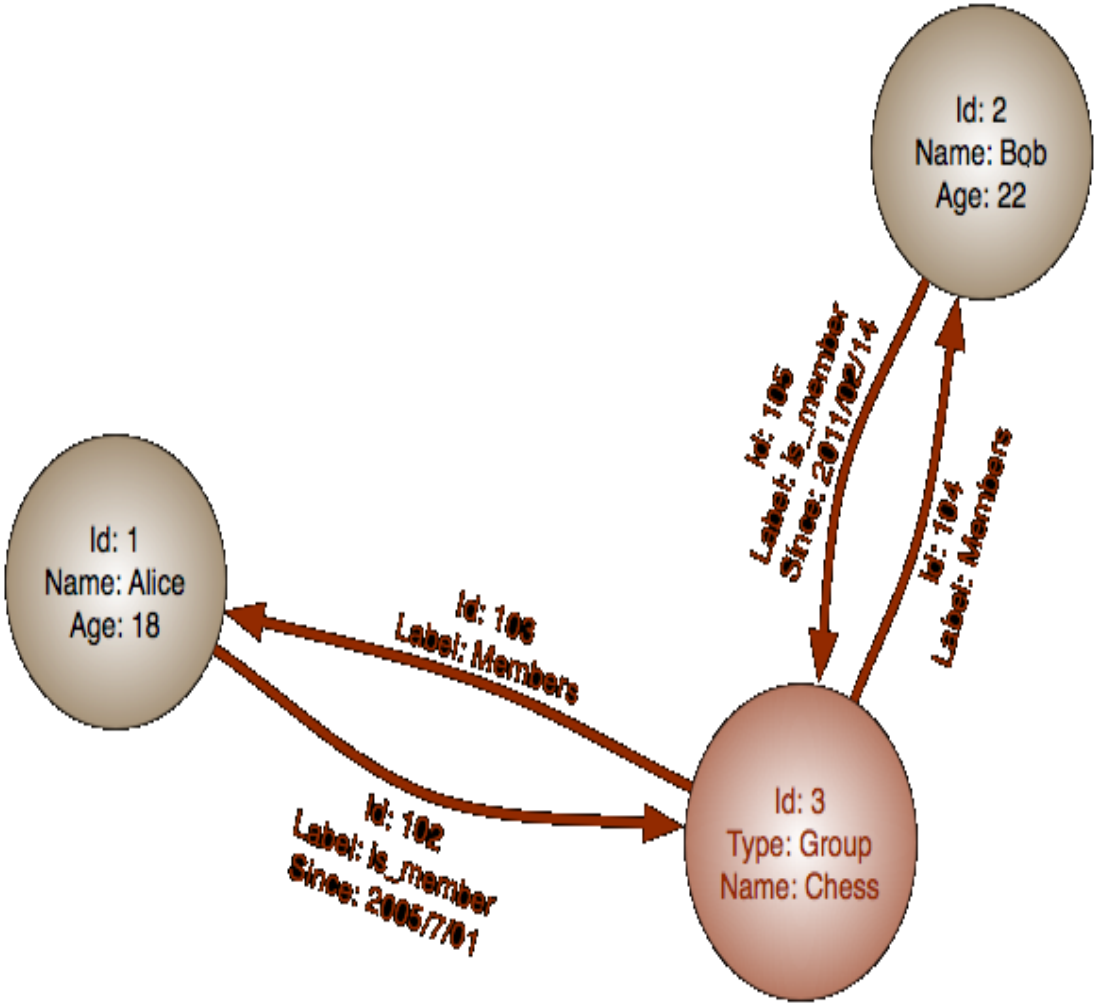
In Graph based databases, data is stored using flexible graph based representation. It uses the following terms:

Node: Nodes represent entities such as people, businesses, accounts, or any other item you might want to keep track of.

Properties: Properties are pertinent information that relate to nodes. For instance, if "Account" were one of the nodes, one might have it tied to properties such as "accno", "acc_name".

Edges: Edges are the lines that connect nodes to nodes or nodes to properties and they represent the relationship between the two.

Graph based DBs continue…



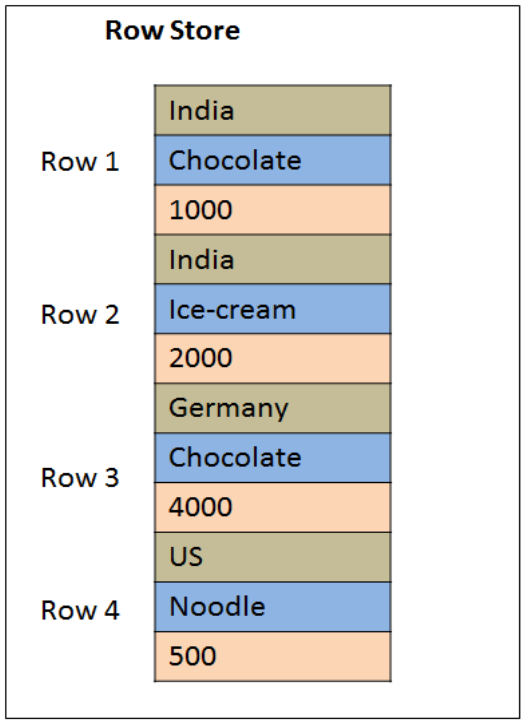
Column based DBs

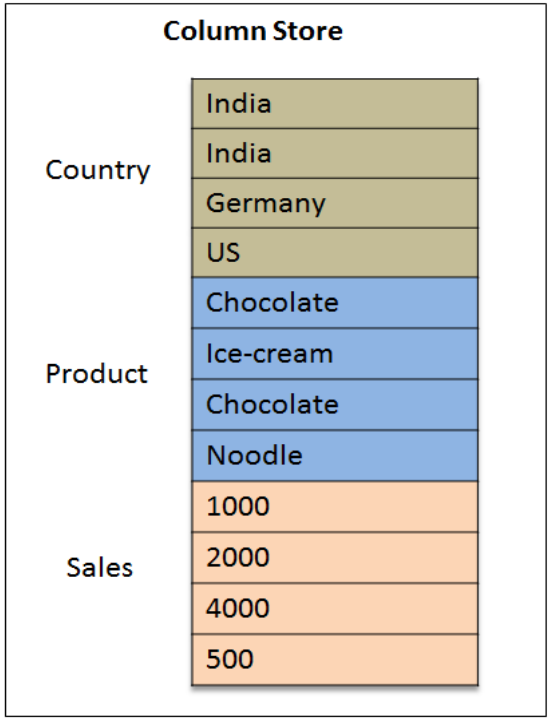
In column-oriented NoSQL database, data is stored in cells grouped in columns of data rather than as rows of data. Columns are logically grouped into column families. Column families can contain a virtually unlimited number of columns that can be created at runtime or the definition of the schema. Read and write is done using columns rather than rows.

Column based DBs continue..

Table

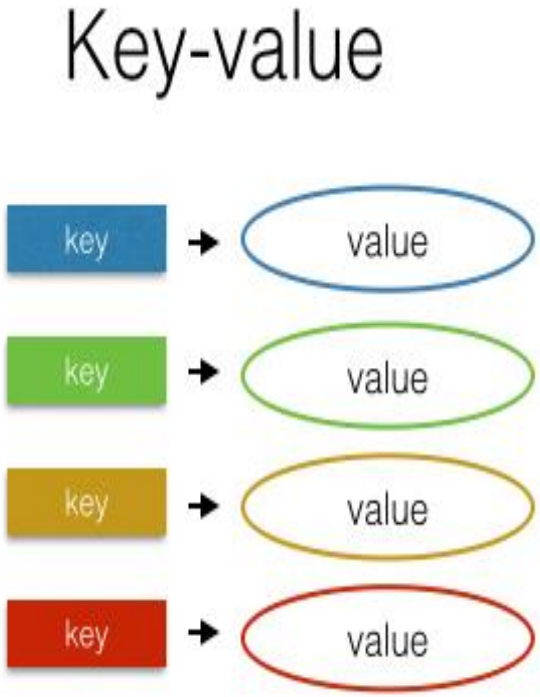
	Country	Product	Sales
Row 1	India	Chocolate	1000
Row 2	India	Ice-cream	2000
Row 3	Germany	Chocolate	4000
Row 4	US	Noodle	500





Key-value based DBs

The key-value NoSQL database uses a hash table in which there exists a unique key and a pointer to a particular item of data.



Introduction to MongoDB

- MongoDB was developed in 2007 by a New York based organization '10gen' which is now called as MongoDB Inc.
- The word Mongo is derived from the word Humongous means very large.
- MongoDB is developed using C++ & JavaScript programming languages.
- Download MongoDB & refer its documentation at https://www.mongodb.org/
- Mongo works on default port 27017.

Advantages of MongoDB

- **Simplicity**

  Mongo adopts storage in JSON format makes it simple database rather than adding complexities that come with relational databases.

- **Data Replication and Reliability**

  MongoDB allow users to replicate data on multiple mirrored servers which ensures data reliability. In case a server crashes, its mirror is still available and database processing remains unaltered.

- **NoSQL Queries**

  Mongo JSON Based document oriented queries are extremely fast as compared to traditional sql queries.

- **Schema Free Migrations**

  In MongoDB, schema is defined by the code. Hence, in case of database migrations, no schema compatibility issue arises.

- **Open Source**

  MongoDB is a database server that is open source and customizable according to the requirements of the organization.

MongoDB installation

- Download MongoDB from https://www.mongodb.org/downloads
- Install the .msi file. Note that Mongo goes not support windows XP.
- Create a directory c:\data\db since MongoDB's default data directory is located there.
- You may override the default directory using following command:

  %MONGO_HOME%\bin>mongod.exe –dbpath c:\mongodb\db
- In order to start mongodb, use the command mongod.exe.
- In order to fire queries, use the command mongo.exe
- More info at https://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/

MongoDB GUI clients

- MongoVUE (http://www.mongovue.com/)
- Robomongo (http://robomongo.org/)
- RockMongo (http://rockmongo.com/)

MongoDB Terminologies

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded documents
Primary key	Primary key supplied by MongoDB itself

MongoDB datatypes

Double

String

Object

Array

Binary Data

ObjectId

Boolean

Date

Null

Integer

Timestamp

Mongo queries

1. Create new database: >use xordb
2. Find the current database: >db
3. List down all databases: >show dbs
4. Delete database: >use xordb THEN >db.dropDatabase()
5. Create collection: >db.createCollection("orders")

Mongo queries continued…

List all collection from database: >show collections

Drop collection: >db.COLLECTION_NAME.drop()

Insert document: >db.COLLECTION_NAME.insert({
name: 'Chairs', quantity: 35, price: 5000})

Display all documents within collection:
>db.COLLECTION_NAME.find().pretty()

Mongo queries continued…

Find documents based upon filter criteria:

```
>db.Orders.find({name: 'Bag Purchase'}).pretty()

>db.Orders.find({price: {$lt: 60000}}).pretty()

>db.Orders.find({$or: [ { price: {$lt: 60000}}, {name: 'Bag Purchase'} ] }).pretty()
```

Update document:

```
>db.Orders.update({name: 'Car Purchase'}, {$set: {name: 'Car sale' }})
```

Mongo queries continued…

Save document:

```
>db.Orders.save({
        "_id": ObjectId("565bbac2c95843b6ef06c00a"),
"name": "Handbag purchase", "price": 2300
        })
```

Delete document:

```
>db.Orders.remove({ name: "Handbag purchase" })

>db.Orders.remove({ name: "Handbag purchase" }, 1)

>db.Orders.remove()
```

Mongo queries continued…

Projection in document:

```
>db.Orders.find({name: 'Laptop Purchase'}, {price: 1}).pretty()
```

Limiting documents:

```
>db.Orders.find().limit(2).pretty()
```

Skipping documents:

```
>db.Orders.find().limit(1).skip(1).pretty()
```

Sorting documents:

```
>db.Orders.find().sort({price: 1}).pretty()
```

Capped collections

- db.createCollection("logs",{capped:true,size:10000,max:1000})
- Capped collections are fixed-size circular collections.
- Capped collection will start deleting the oldest document in the collection without providing any explicit command.
- Capped collection follows FIFO(First in first out) model.
- The aggregation pipeline operator $out cannot write results to a capped collection.
- If the size field is less than or equal to 4096, then the collection will have a cap of 4096 bytes.
- You cannot shard a capped collection.

Capped collections continue…

- We cannot delete a document from capped collection.

  db.logs.remove({status: 'DONE') // Error
- Convert existing collection to capped:

  db.runCommand({"convertToCapped":"logs",size:10000})
- Find out whether a collection is capped or not:

  db.logs.isCapped()
- Performing find() operation on capped collection gives results ordering same as the insertion order. However, if we wish to reverse it, for example I wish to find out last 5 logs from 'logs' collection then use:

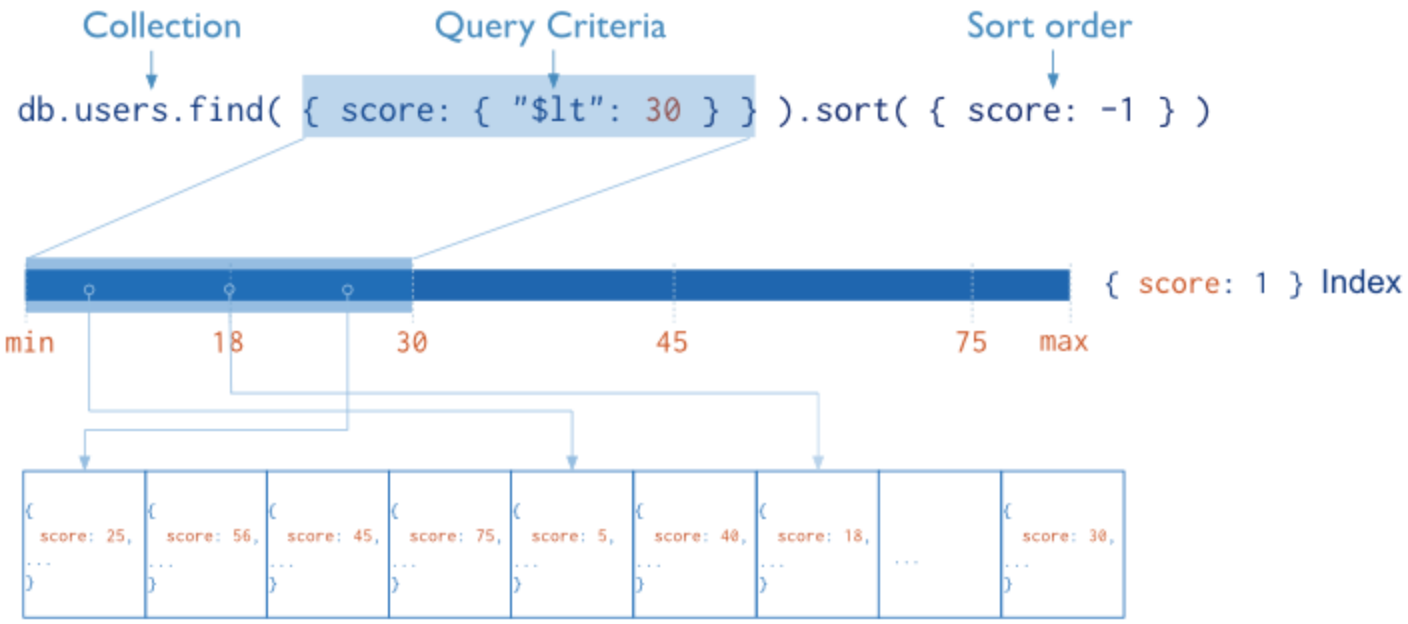
  db.logs.find().sort({ $natural: -1 })

Mongo shell methods

Mongo provides extensive support of built-in methods that helps to write complex query. You can find the complete list of methods at:

https://docs.mongodb.org/manual/reference/method/

Mongo Indexing



- Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form.
- The index stores the value of a specific field or set of fields, ordered by the value of the field.

Why Indexing?

- Using indexing, mongo need not scan every document in a collection.
- Indexing improves query execution performance.

Index types

- **_id (default)**

  All MongoDB collections have an index on the _id field that exists by default.

- **Single Field**

  db.users.createIndex({ "name" : 1 })

- **Compound Index**

  db.users.createIndex({ "name" : 1, “marks “: -1})

- **Multikey Index**

  Multikey indexes are applicable for the fields that hold an array value.

  db.users.createIndex({ users.phones: 1})

Index types continue…

- **Geospatial Index**

  To execute queries on geospatial co-ordinate data, mongo provides geospatial index.

- **Text Indexes**

  MongoDB provides text indexes to support text search of string content in documents of a collection.

  db.feedback.createIndex({ comments: "text" })

- **Hashed Indexes**

  Hashed index type indexes the hash of the value of a field. It is used in hash based sharding.

Key points about indexes

- You can drop a mongo index using following query:
  - db.users.dropIndex({ “name": 1 })
- You can drop all the indexes on your collection using:
  - db.users.dropIndexes();
- You can list down all indexes applied on any collection using:
  - db.getCollection('Orders').getIndexes()
- In order to improve the query performance, no of indexes on a collection needs to be optimized. It means, no index or many indexes on collection, both will increase the query execution time.
- To analyze the query execution, use .explain().

Covered queries

Indexes improve the efficiency of read operations by reducing the amount of data that query operations need to process. Hence it is recommended to index the fields which are mentioned in your query.

A query is called as ‘Covered Query’ if:

- all the fields in the query are part of an index, and
- all the fields returned in the results are in the same index.

For example:

db.users.createIndex({ "name" : 1, “marks “: -1})

db.users.find({marks: 75}, {name: 1})

MongoDB Text Search

- Using "text indexes", we can achieve text search in mongo documents. For example:
  - db.logs.createIndex({ description: "text" } //text index on description field
  - db.logs.createIndex({ "$**": "text" } //wildcard index: text index on all string based fields within "logs" collection
- "text indexes" can include any field whose value is string or array of string elements.
- We can use $text operator to search text through mongo query.
- $text operator searches the whole word only.

MongoDB Text Search

- Using $text operator inside find() method:
  - db.logs.find({$text: {$search: "apple"} })
- Using $text operator in aggregation pipeline:
  - db.logs.aggregate([{$match: {$text: {$search: "apple"} } }])
- In order to search the string anywhere within a field:
  - db.logs.find({"desc": /appple/ })

Mongo Regular Expression ($regex)

- $regex operator provides regular expression capabilities for pattern matching strings in queries. The basic syntax is:
  - { <field>: { $regex: 'pattern', $options: '<options>' } } OR
  - { <field>: { $regex: /pattern/<options> } }
  - { name: { $regex: /acme.*corp/, $options: "si" } }
  - The character '^' means start & '$' means end.

Option	Description
i	Case insensitive pattern match
m	Indicates multiline match
x	Extended capability to ignore all white spaces in $regex pattern.
s	Allows dot character(.) to match all characters including newline characters.

Regular expression examples

- Find documents where country starts with ‘Al’ in case insensitive manner:
  - db.worldinfo.find({ country: { $regex: /^Al/i } })
- Find the multi line feedbacks starting with word ‘You’:
  - db.feedback.find({ description: { $regex: /^You/, $options: “m” } })
- Find feedbacks starting with word ‘You’ & ignore white spaces, comments (#) & \n:
  - db.feedback.find({ description: { $regex: /^You/, $options: “x” } })
- Match all characters including new line as well as case insensitive search.
  - db.feedback.find({ description: { $regex: /m.*line/, $options: “si” } })

MongoDB Aggregation Framework

Sometimes Mongo developer wants to analyze and crunch it in interesting ways. Here we can use aggregation facility provided by Mongo.

Aggregation groups values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.

Aggregation Framework

The aggregation framework lets you transform and combine documents in a collection. Basically, you build a pipeline that processes a stream of documents through several building blocks: filtering, projecting, grouping, sorting, limiting, and skipping.

Take an example of collection ‘Magazine’ having following fields:

```
{
   "_id" : ObjectId("565c22ba95c0b732f6f6eede"),
   "name" : "India Today",
   "price" : 150,
   "author" : “Ivan"
}
```

Query example

List down the first 3 authors based upon number of magazines they have published. In order to write Mongo query for this, lets first categorize the steps:

- Project the authors out of each magazine document.
- Group the authors by name, counting the number of occurrences.
- Sort the authors by the occurrence count, descending.
- Limit results to the first three.

Query example continued…

- In order to project the author from magazine, we use $project operator:

  ```
  {"$project" : {"author" : 1}}
  ```

- Grouping by author name & count number of authors within collection, we use $group operator:

  ```
  {"$group" : {"_id" : "$author", "count" : {"$sum" : 1}}}
  ```

- Sorting based number of upon magazines published by an author, we use $sort:

  ```
  {"$sort" : {"count" : -1}}
  ```

- Show only limited number of records, use $limit:

  ```
  {"$limit" : 3}
  ```

Query example continued…

Final query:

```
db.magazine.aggregate(
        {"$project" : {"author" : 1}},
        {"$group" : {"_id" : "$author", "count" : {"$sum"
: 1}}},
        {"$sort" : {"count" : -1}},
        {"$limit" : 2}
)
```

Pipeline operations

The aggregation framework extensively uses pipeline operations. Here one operator generate document & it becomes input to another operator & so on.. Finally the last operator returns the result to the client.

The operators can be combined in any order & repeated many times as necessary.

Pipeline operators

$project

```
{ "$project" : {"author" : "$author"} }
```

$group

```
{ "$group": { "_id": "$author", "count": {"$sum": 1} } }
```

$sort

```
{ "$sort": {count: -1 } }
```

$limit

```
{ "$limit": 2 }
```

$skip

```
{ $skip : 5 }
```

$match

```
{ $match : { author : “Ivan" } }
```

$unwind

```
{ $unwind : "$prices" }
```

$group operators

$sum

"$group" : { "totalRevenue" : {"**$sum**" : "$revenue"} }

$avg

"$group" : {“averageRevenue" : {"**$avg**" : "$revenue"}}

$max, $min

"$group" : { “costlyBook”: {“**$max**”: “$bookPrice”} }

$first, $last

$group: { _id: "$item", firstSalesDate: { **$first**: "$date" } } }

$addToSet

$group: { _id: { day: { $dayOfYear: "$date"}, itemsSold: { **$addToSet**: "$item" } } }

$push

$group: { _id: { day: { $dayOfYear: "$date"}, itemsSold: { **$push**: { item: "$item", quantity: "$quantity" } } } }

Pipeline expressions

Pipeline expressions allow us to perform more powerful operations in aggregation framework like manipulating numeric values, playing with date fields, performing operations on strings, adding various logical conditions etc.

Pipeline expressions are divided into following types:

- Mathematical expressions
- Date expressions
- String expressions
- Logical expressions

Mathematical expressions

Mathematical expressions let you to manipulate numeric values.

$add

```
{ $project: { item: 1, total: { $add: [ "$price", "$fee" ] } } }
```

$subtract

```
{ $project: { item: 1, dateDifference: { $subtract: [ "$date", 5 * 60 * 1000 ] } } }
```

$multiply

```
{ $project: { date: 1, item: 1, total: { $multiply: [ "$price", "$quantity" ] } } }
```

$divide

```
{ $project: { name: 1, workdays: { $divide: [ "$hours", 8 ] } } }
```

$mod

```
{ $project: { remainder: { $mod: [ "$hours", "$tasks" ] } } }
```

Date expressions

Using date expressions, we can perform several operations on date field. For example, in order to find month for a date:

```
month: { $month: "$date" }
```

$dayOfYear

$dayOfMonth

$dayOfWeek

$year

$month

$week

$hour

$minute

$second

$milisecond

$dateToString

String expressions

Spring expressions can be used to perform basic operations on string.

$substr

```
{"$substr" : ["$firstName", 0, 1]}
```

$concat

```
{“$concat”: [ “$firstName”, “$lastName” ] }
```

$toLower

```
{“$toLower”: “$firstName” }
```

$toUpper

```
{“$toUpper”: “$firstName” }
```

Logical expressions

Logical expressions are used to perform conditional operations.

$cmp

isHighQuantity: { **$cmp**: ["$qty", 250] }

$strcasecmp

isNameMatching: { $strcasecmp: ["$firstName", “TOM"] }

$eq, $ne, $gt, $gte, $lt, $lte

isQuantity250: { **$eq**: ["$qty", 250] }

$and, $or

result: { **$or**: [{ $gt: ["$qty", 250] }, { $lt: ["$qty", 200] }] }

$not

result: { **$not**: [{ $gt: ["$qty", 250] }] }

$cond

discount: { **$cond**: { **if**: { $gte: ["$qty", 250] }, then: 30, **else**: 20 } }

$ifNull

description: { **$ifNull**: ["$description", "Unspecified"] }

Useful links

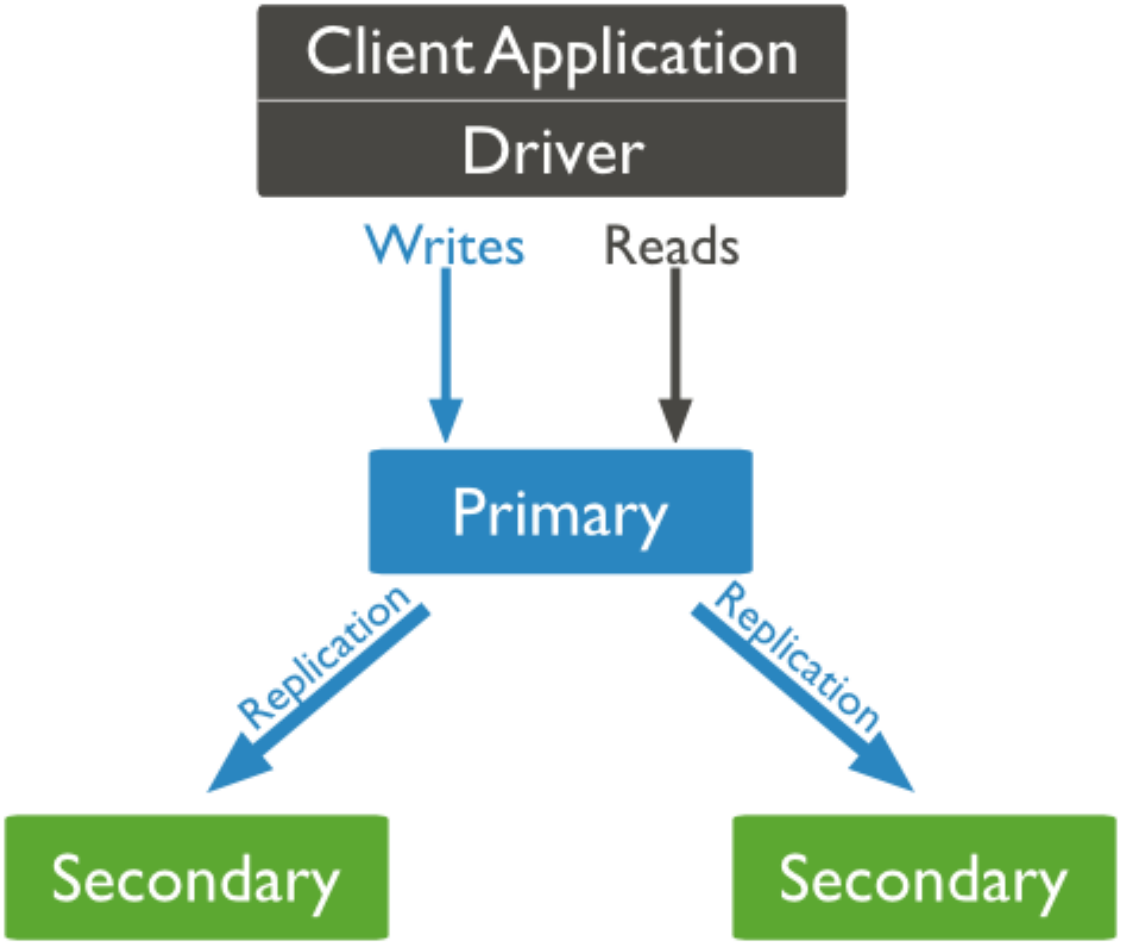
SQL to aggregation mapping chart:

https://docs.mongodb.org/manual/reference/sql-aggregation-comparison/

Aggregation Pipeline Operators

https://docs.mongodb.org/manual/reference/operator/aggregation/

MongoDB Replication



MongoDB replication continued..

- What is replication?
  - Replication is the process of synchronizing data across multiple mongo servers.

- Why replication?
  - High availability of data
  - Data safety
  - Disaster recovery
  - No system downtime for maintenance
  - Efficient reading due to extra copies of db servers

Replication concepts

- The heart of replication is replica set. A replica set is a group of ***mongod*** instances that host the same data set. These mongod instances are also called as nodes.
- Every replica set should have 2 or more nodes.
- Every replica set has one & only one primary node & remaining secondary nodes.
- Primary node can be used to read & write the data.
- Secondary node is used to only read the data.
- In case of primary node failure, election is conducted & any one secondary node becomes primary node.

Setting up Replica set

- Create 3 directories for mongo data: c:\data\node1, c:\data\node2, c:\data\node3.
- Start 3 mongo instances with different port & different database path:
  - start mongod --replSet myRS --logpath "1.log" --dbpath c:\data\node1 --port 27017
  - start mongod --replSet myRS --logpath "2.log" --dbpath c:\data\node2 --port 27018
  - start mongod --replSet myRS --logpath "3.log" --dbpath c:\data\node3 --port 27019
- Connect to mongo default port no instance.
  - mongo --port 27017

Setting up Replica set continued…

- Specify the replica set configuration:

```
rsconf = {
        _id: "myRS",
        members: [ { _id: 0, host: "Anand-KU:27017" } ]
       }
```

- Initiate the replica set configuration:
  - rs.initiate(rsconf)
- Configure the replica set:
  - rs.conf()
- Add two secondary nodes in replica set:
  - rs.add("Anand-KU:27018")
  - rs.add("Anand-KU:27019")

Testing replica set

- Connect to default mongo i.e. 27017
- Insert a document in ‘contact’ collection:
  - db.contact.insert({name: “Tom", phone: 11111})
- Connect to mongo at 27018
- Enable reading from secondary node:
  - db.getMongo().setReadPref('secondary')
- Read all documents within ‘contact’ collection:
  - db.contact.find().pretty()
  - You should get the document inserted inside 27017 mongo instance. This proves that mongo data is getting replicated across all the nodes. You can also try for 27019 port.

MongoDB data backup

The **mongodump** tool reads data from a MongoDB database and creates high fidelity BSON files. To run mongodump, make sure mongod is running.

- To backup complete data of default mongod instance:
  >mongodump
- To backup complete data of specific mongod instance:
  >mongodump --host comp-12 --port 27018
- To backup complete data from specified db path to specified backup directory:
  >mongodump --dbpath c:\data\db --out c:\data\backup
- To backup specified collection of specified database:
  >mongodump --collection accounts --db test
- To find more details of mongodump command:
  >mongodump --help

Restoring mongo dump

mongorestore command is used to restore backup data.

- To restore data from ‘dump’ directory to data\db:

  >mongostore

- To restore data from specified host & port:

  >mongorestore --host COMP-12 --port 27018

  OR

  >mongorestore --host COMP-12:27018

- To restore data from accounts collection belonging to test db:

  >mongorestore --db test dump\test\accounts.bson --collection accounts

Mongo atomicity & transactions

- Write operation is always atomic on a single document.
- This atomicity is also applicable on multiple embedded documents within a single document.
- However $isolated operator usage achieves multi-document level write operation in atomic fashion.

```
db.foo.update( { status : "A" , $isolated : 1 } )
```

- $isolated operator ensures that no client sees the changes until write operation on multiple documents is completed.
- $isolated is not "all-or-nothing" atomicity.
- $isolated operator does not work on sharded clusters.

Mongo GridFS

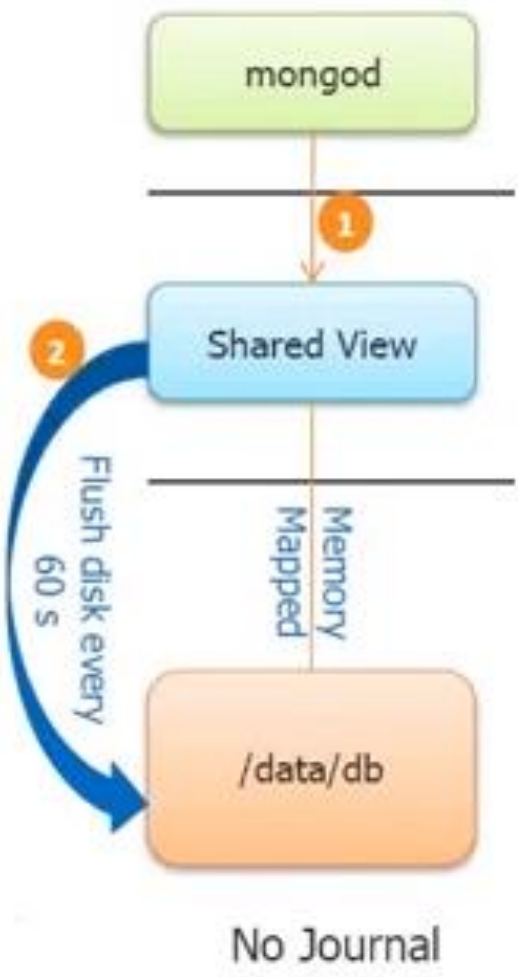
- Maximum size of any Mongo document can be 16 MB.
- If we wish to add a file into mongo document having length more than 16 MB then we need GridFS.
- GridFS can be used where you wish to read small part of large file.
- GridFS is a specification for storing and retrieving files that exceed the BSON-document size limit of 16MB.
- Instead of storing a file in a single document, GridFS divides a file into parts, or chunks, and stores each chunk as a separate document.
- Default chunk size is 255 KB.

Mongo GridFS query

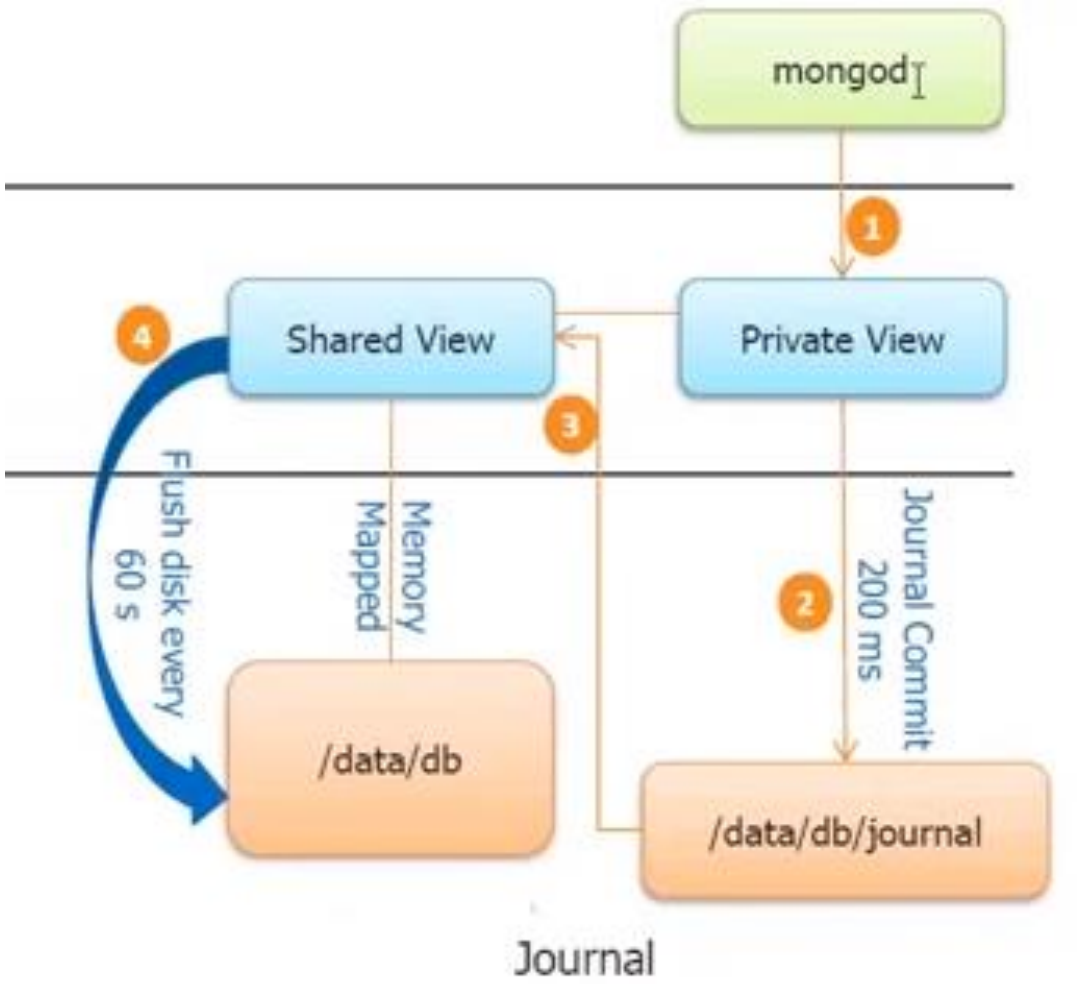
- GridFS has two collections for file storage:
  - fs.chunks (Stores file chunks)
  - fs.files (Stores file's metadata)
- Adding a file into mongo GridFS:
  - >mongofiles.exe -d testdb put order_details.docx
- To see your file call the find() function on fs.files collection:
  - >db.fs.files.find()
- In order to see the chunks stored in fs.chunks:
  - db.fs.chunks.find({files_id:ObjectId('639a811bf8b4aa4d33fdf33f')})
  - // where '639a811bf8b4aa4d33fdf33f' is a file_id referred from fs.files collection.

Mongo Journaling

- Journaling is a methodology of ‘write ahead logging’ to on-disk journal files.
- Any data write without journaling goes to ‘shared view’ which is flushed to persistent file after every 60 sec.
- If there is any failure going to happen during 60 sec then shared view data will not be written into file. This is against principle of data durability.



Mongo with Journaling support



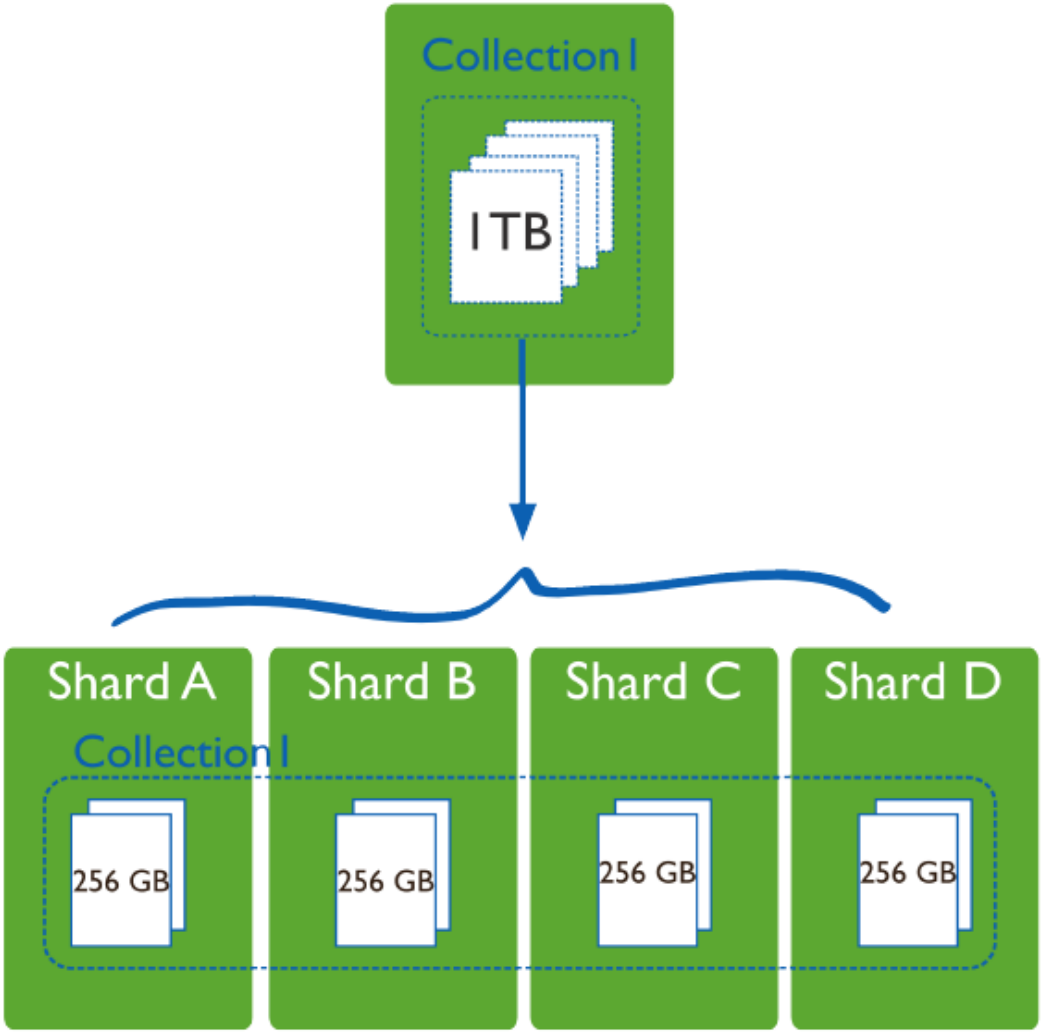
Mongo with Journaling support continue…

- Journaling is by default enabled in MongoDB 3.0
- When we write data using mongo query, the ‘Private view’ registers the query & write it into journal files in every 200 ms. Note that ‘Private view’ does not hold the data.
- Now the query is executed & data is written into ‘Shared view’ which flushes the data into file in every 60 sec.
- Suppose data is written into ‘Shared view’ but before flushing data into file there is a failure. In such case when we start the mongo db, it looks into journal files & executes the queries which are not flushed into file by ‘Shared view’.

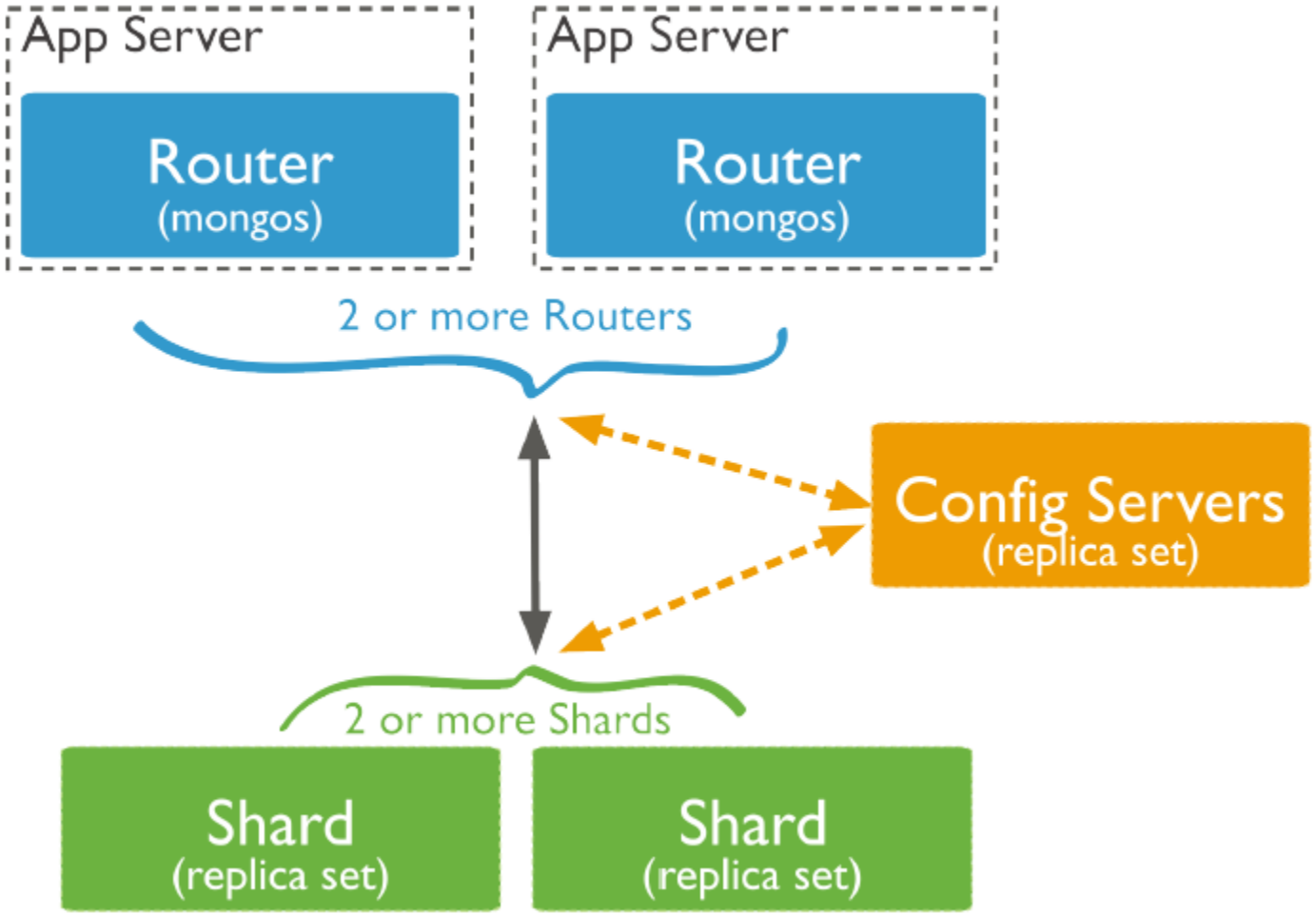
Mongo Sharding

- RDBMS uses vertical scaling where as NoSQL uses horizontal scaling. Horizontal scaling is nothing by sharding.
- Sharding is a method for storing data across multiple machines.
- Sharding, or horizontal scaling, divides the data set and distributes the data over multiple servers, or shards.
- Database systems with large data sets and high throughput applications use sharding.

Data distribution in sharding



Sharding components



Sharding components continue…

- **Shard:**

  Shard stores the data. Each shard is a replica set.

- **Query routers or mongos:**

  Query router acts as an interface between client & appropriate shard or shards. A client sends requests to mongos, which then routes the operations to the shards and returns the results to the clients.

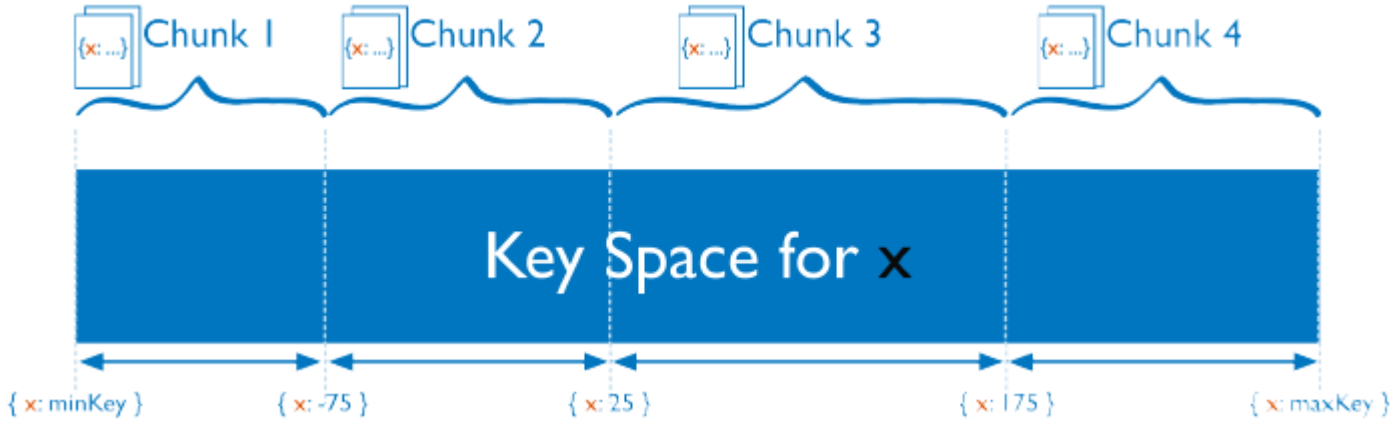
- **Config servers:**

  Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards.

Shard keys

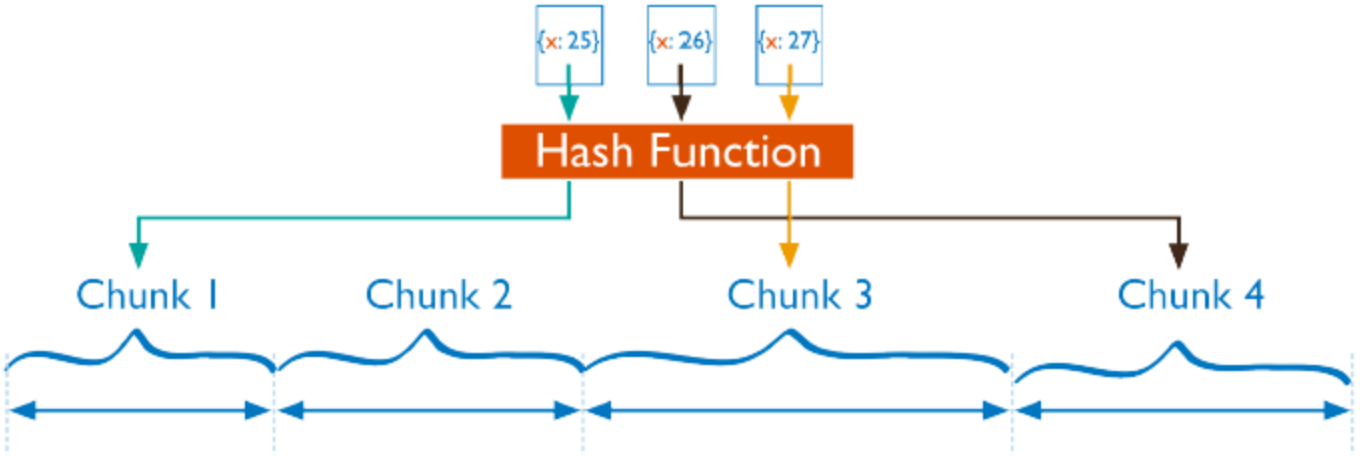
- To shard a collection, we need to select a shard key from collection fields.
- A shard key is either an indexed field or an indexed compound field that exists in every document in the collection.
- MongoDB divides the shard key values into chunks and distributes the chunks evenly across the shards.
- MongoDB's division of shard keys into chunks is based upon 2 partitioning principles:
  - Range based partitioning
  - Hash based partitioning

Range based partitioning



- In range-based sharding, MongoDB divides the data set into ranges determined by the shard key values to provide range based partitioning.
- Documents with "close" shard key values are likely to be in the same chunk, and therefore on the same shard.
- Range based partitioning may lead to uneven distribution of data.
- Range based partitioning is efficient in case of range based query execution.

Hash based partitioning

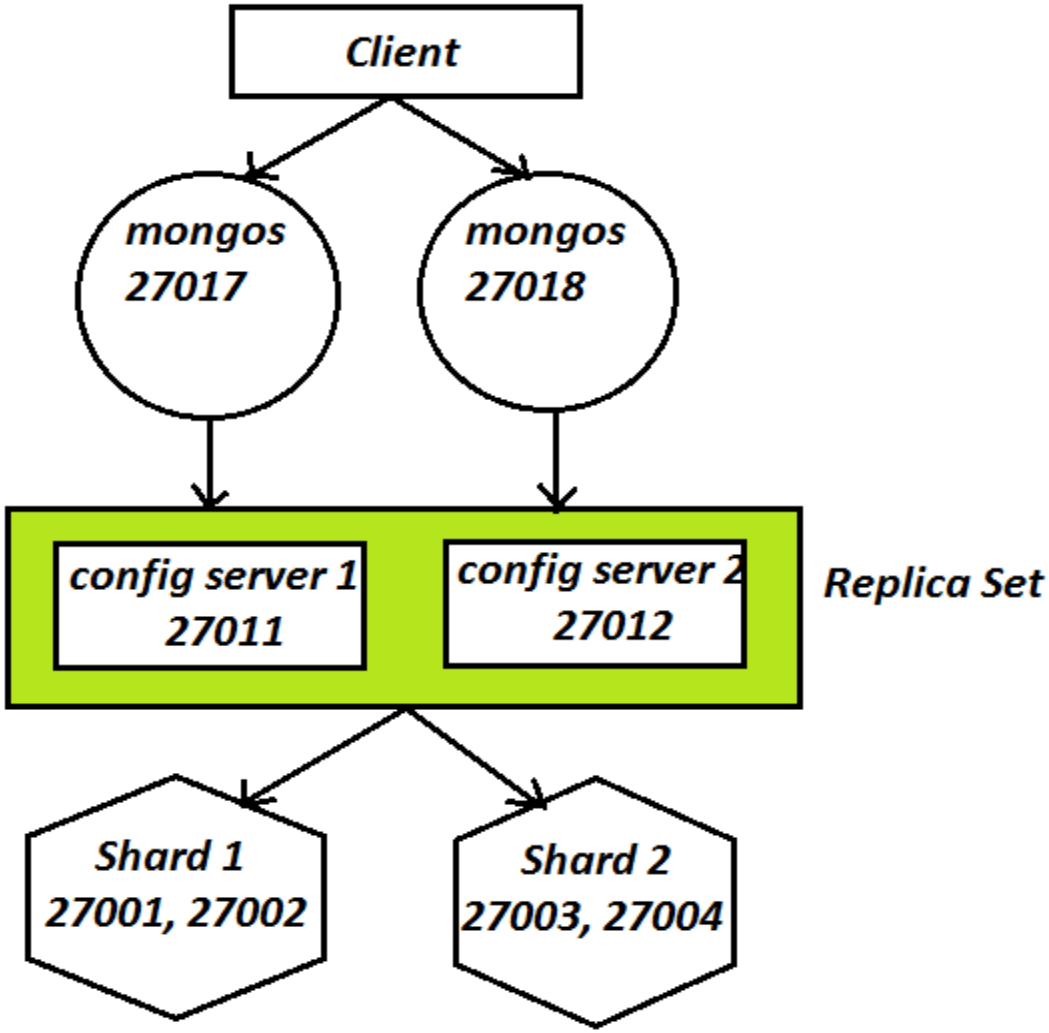


- In hash based partitioning, MongoDB computes a hash of a field's value, and then uses these hashes to create chunks.
- With hash based partitioning, two documents with "close" shard key values are unlikely to be part of the same chunk.
- We can specify hash based sharding for collection using the following command:
  - sh.shardCollection("xordb.users", { username: "hashed" })

Tag aware sharding

- We can achieve customized data distribution using ‘Tag aware sharding’.
- ‘Tag aware sharding’ allows MongoDB administrators to direct the balancing policy.
- Tags are the primary mechanism to control the behavior of the balancer and the distribution of chunks across shards.

Applying sharding



Steps for applying sharding

- Configure multiple shards i.e. replica sets.
- Configure replica set for config server.
- Configure query router or mongos.
- Add shard information to mongos.
  - ***sh.addShard("RS_1/COMP12:27001")***
- Enable sharding on database.
  - ***sh.enableSharding("xordb")***
- Enable sharding on collection.
  - ***sh.shardCollection("xordb.users", {_id: 1}, true)***
  - ***sh.shardCollection("xordb.contacts", {fname: "hashed"})***
- Try to insert multiple documents into the collection & notice the chunks distribution using sh.status() method.

Questions