
mongoDB

What is NoSQL

NoSQL is a non-relational DMS, that does not require a fixed schema, avoids joins, and is easy to scale.

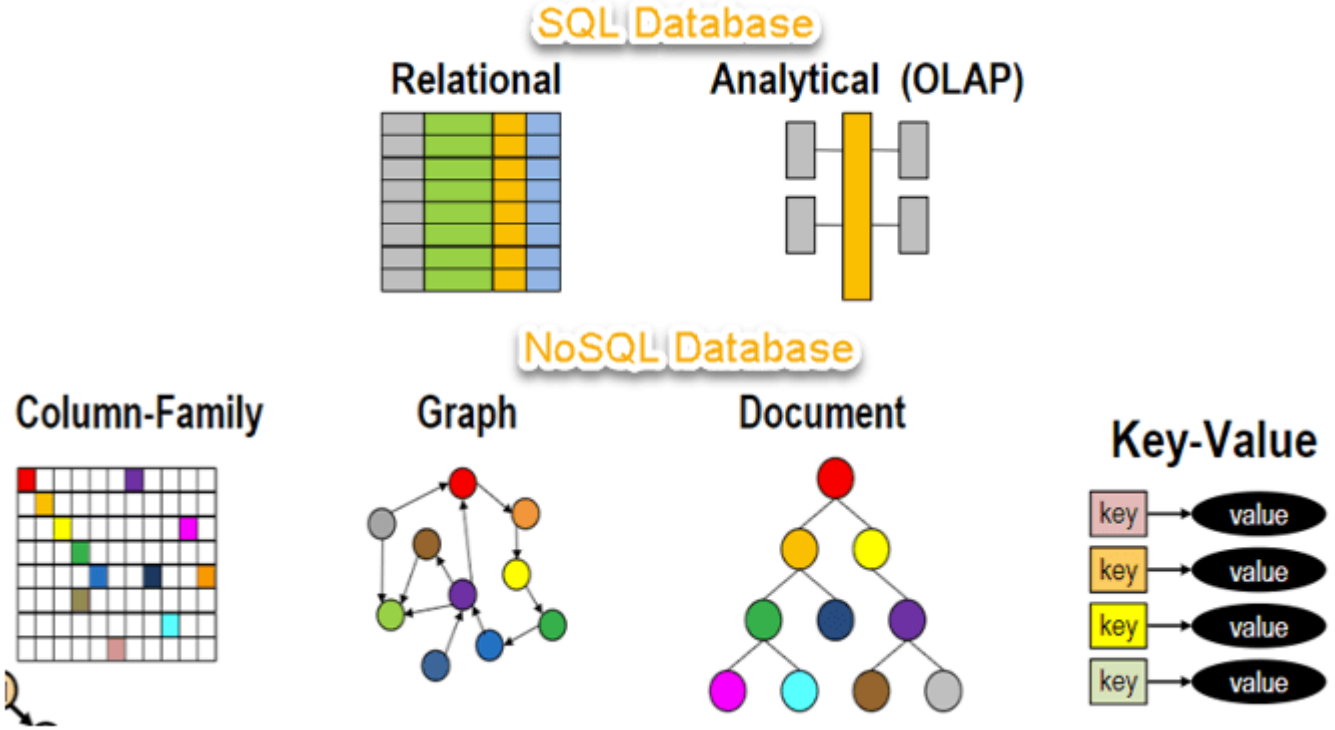
The purpose of using a NoSQL database is for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps.
For example, companies like Twitter, Facebook, Google collect terabytes of user data every single day.

NoSQL database stands for "Not Only SQL" or "Not SQL." Though a better term would NoREL NoSQL caught on. Carl Strozz introduced the NoSQL concept in 1998.

Traditional RDBMS uses SQL syntax to store and retrieve data for further insights.

NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data.

What is NoSQL

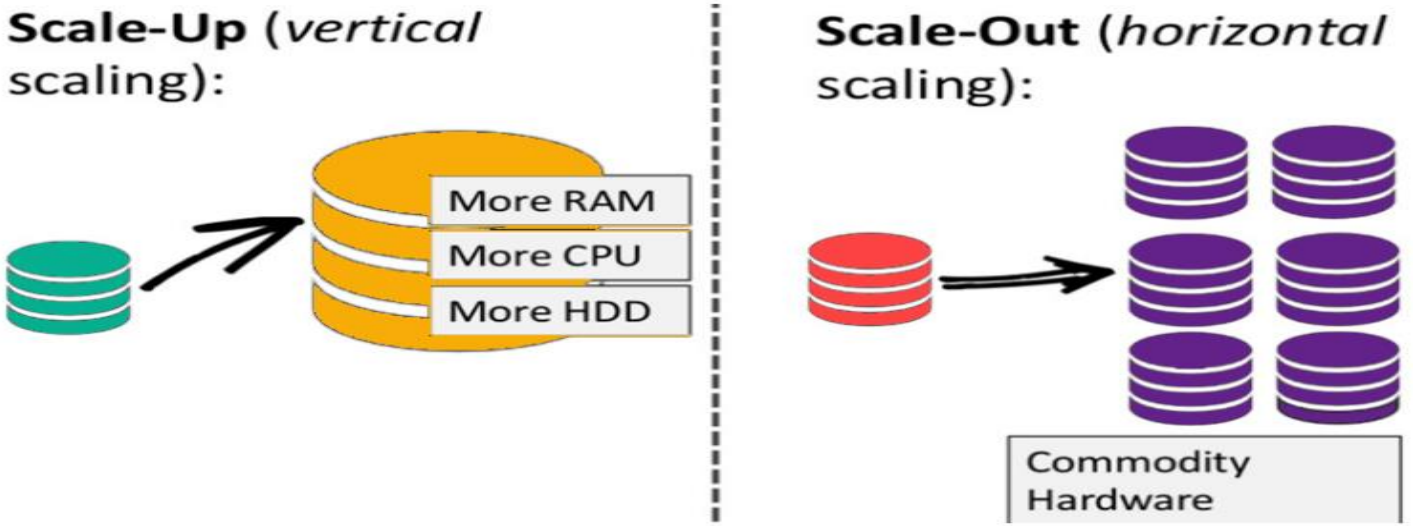


mongoDB

Why NoSQL

The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.

To resolve this problem, we could "scale up" our systems by upgrading our existing hardware. This process is expensive.
The alternative for this issue is to distribute database load on multiple hosts whenever the load increases. This method is known as "scaling out."



mongoDB

Why NoSQL

A Big Data project is normally typified by:

- High data velocity: lots of data coming in very quickly, possibly from different locations.
- Data variety: storage of data that is structured, semi-structured and unstructured.
- Data volume: data that involves many terabytes or petabytes in size.
- Data complexity: data that is stored and managed in different locations or data centers.

mongoDB

Why NoSQL

Scenarios where NoSQL SHOULD be used:

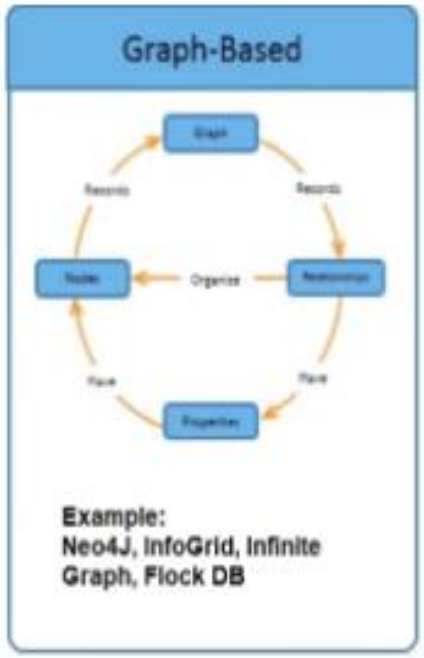
- Your relational database will not scale to your traffic at an acceptable cost.
- In a NoSQL database, there is no fixed schema and no joins. NoSQL can take advantage of “scaling out”. Scaling out refers to spreading the load over many commodity systems.
- You have local data transactions which do not have to be very durable. e.g. “liking” items on websites.

Scenarios where NoSQL SHOULD NOT be used:

- Normally an interface is provided for storing your data. Do not try to use a complicated query in that interface.
- The developer should always keep in mind that NoSQL database is not built on tables and usually doesn’t use structured query language.
- If consistency is mandatory and there will be no drastic changes in terms of the data volume.

Find more cases, self study and make a report

NoSQL Types



mongoDB

NoSQL Types

Key Value Pair Based

Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load.

Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.

For example, a key-value pair may contain a key like "Website" associated with a value like "Guru99".

Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key value stores help the developer to store schema-less data. They work best for shopping cart contents.

Redis, Dynamo, Riak are some examples of key-value store DataBases. They are all based on Amazon's Dynamo paper.

NoSQL Types

Column-based

Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately. Values of single column databases are stored contiguously.

ColumnFamily			
Row Key	Column Name		
	Key	Key	Key
	Value	Value	Value
	Column Name		
	Key	Key	Key
	Value	Value	Value

They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.

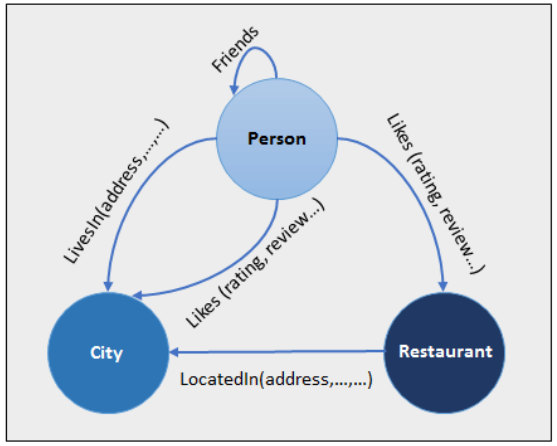
Column-based NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs,

HBase, Cassandra, HBase, Hypertable are examples of column based database.

NoSQL Types

Graph-Based

A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.



Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.

Graph base database mostly used for social networks, logistics, spatial data.

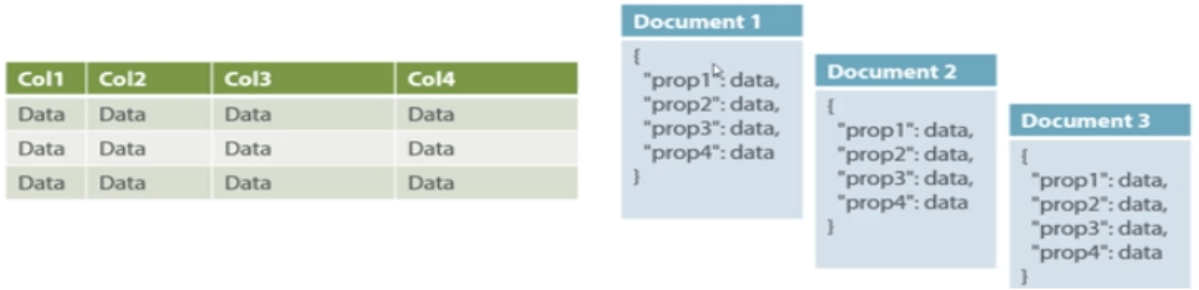
Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

mongoDB

NoSQL Types

Document-Oriented:

Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.



Relational Vs. Document

In this diagram on your left you can see we have rows and columns, and in the right, we have a document database which has a similar structure to JSON. Now for the relational database, you have to know what columns you have and so on. However, for a document database, you have data store like JSON object. You do not require to define which make it flexible.

The document type is mostly used for CMS systems, blogging platforms, real-time analytics & e-commerce applications. It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.
Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated DBMS systems.

mongoDB

RDBMS vs NoSQL

RDBMS

- Structured and organized data
- Structured query language (SQL)
- Data and its relationships are stored in separate tables.
- Data Manipulation Language, Data Definition Language
- Tight Consistency

NoSQL

- Stands for Not Only SQL
- No declarative query language
- No predefined schema
- Key-Value pair storage, Column Store, Document Store, Graph databases
- Eventual consistency rather ACID property
- Unstructured and unpredictable data
- CAP Theorem
- Prioritizes high performance, high availability and scalability
- BASE Transaction

What is the CAP Theorem?

CAP theorem is also called brewer's theorem. It states that is impossible for a distributed data store to offer more than two out of three guarantees

- Consistency
- Availability
- Partition Tolerance

Consistency:
The data should remain consistent even after the execution of an operation. This means once data is written, any future read request should contain that data. For example, after updating the order status, all the clients should be able to see the same data.

Availability:
The database should always be available and responsive. It should not have any downtime.

Partition Tolerance:
Partition Tolerance means that the system should continue to function even if the communication among the servers is not stable. For example, the servers can be partitioned into multiple groups which may not communicate with each other. Here, if part of the database is unavailable, other parts are always unaffected.

What is the CAP Theorem?

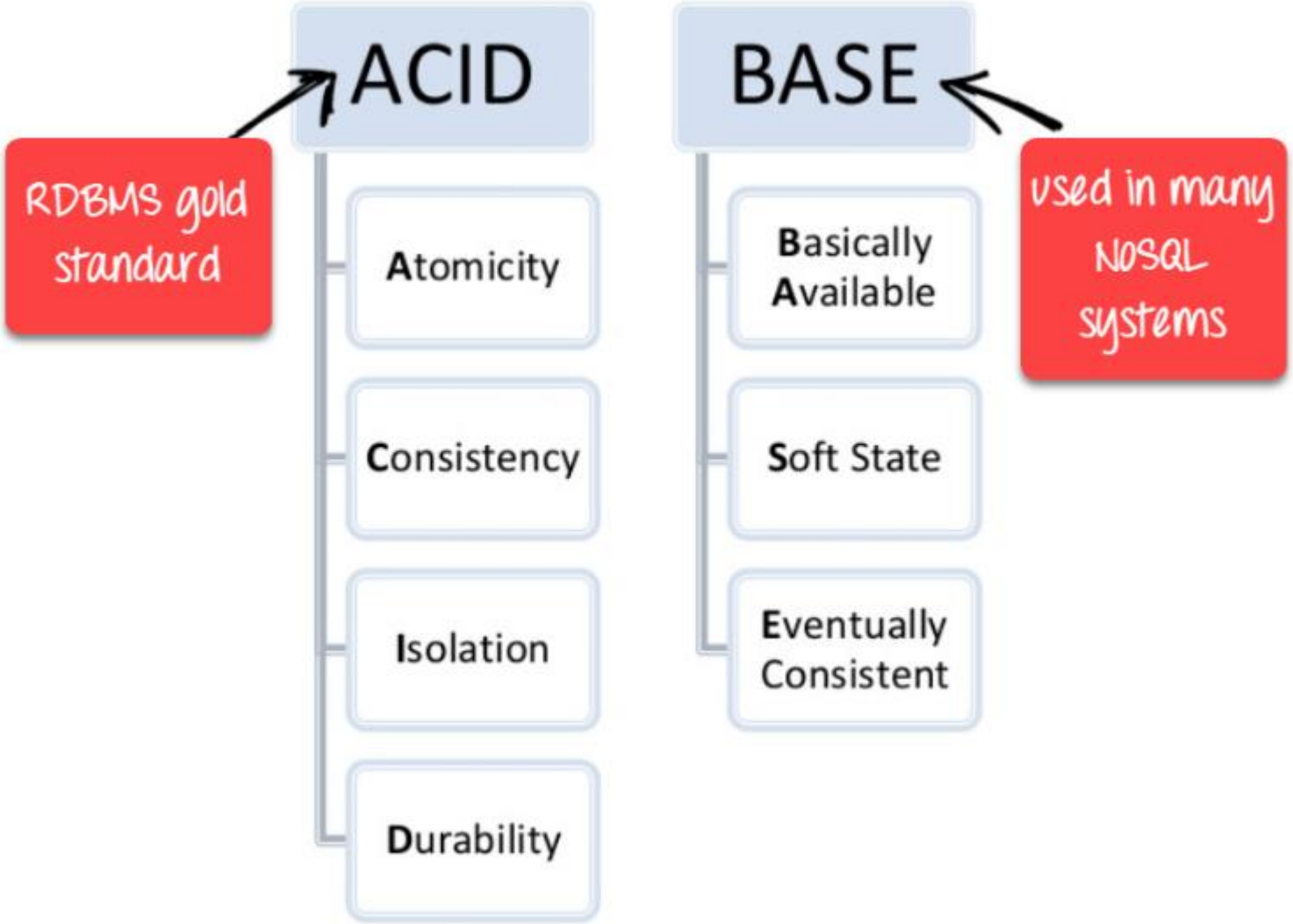
Eventual Consistency
The term "eventual consistency" means to have copies of data on multiple machines to get high availability and scalability. Thus, changes made to any data item on one machine has to be propagated to other replicas.

Data replication may not be instantaneous as some copies will be updated immediately while others in due course of time. These copies may be mutually, but in due course of time, they become consistent. Hence, the name eventual consistency.

BASE: Basically Available, Soft state, Eventual consistency

- Basically, available means DB is available all the time as per CAP theorem
- Soft state means even without an input; the system state may change
- Eventual consistency means that the system will become consistent over time

What is the CAP Theorem?



mongoDB

Advantages of NoSQL

- Can be used as Primary or Analytic Data Source
- Big Data Capability
- No Single Point of Failure
- Easy Replication
- No Need for Separate Caching Layer
- It provides fast performance and horizontal scalability.
- Can handle structured, semi-structured, and unstructured data with equal effect
- Object-oriented programming which is easy to use and flexible
- NoSQL databases don't need a dedicated high-performance server
- Support Key Developer Languages and Platforms
- Simple to implement than using RDBMS
- It can serve as the primary data source for online applications.
- Handles big data which manages data velocity, variety, volume, and complexity
- Excels at distributed database and multi-data center operations
- Eliminates the need for a specific caching layer to store data
- Offers a flexible schema design which can easily be altered without downtime or service

Disadvantages of NoSQL

- No standardization rules
- Limited query capabilities
- RDBMS databases and tools are comparatively mature
- It does not offer any traditional database capabilities, like consistency when multiple transactions are performed simultaneously.
- When the volume of data increases it is difficult to maintain unique values as keys become difficult
- Doesn't work as well with relational data
- The learning curve is stiff for new developers
- Open source options so not so popular for enterprises.

BREAK TIME

mongoDB

What is MongoDB?

MongoDB is a document-oriented NoSQL database used for high volume data storage. MongoDB is a database which came into light around the mid-2000s. It falls under the category of a NoSQL database.

MongoDB is a document-oriented database which stores data in JSON-like documents with dynamic schema. It means you can store your records without worrying about the data structure such as the number of fields or types of fields to store values. MongoDB documents are similar to JSON objects.

mongoDB

What is MongoDB?

- MongoDB stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time
- The document model maps to the objects in your application code, making data easy to work with
- Ad hoc queries, indexing, and real time aggregation provide powerful ways to access and analyze your data

Who are Using MongoDB?

- Castlight Health
- IBM
- Citrix
- Twitter
- T-Mobile
- Zendesk
- HTC
- InVision
- Intercom etc.
- Sony
- BrightRoll
- Foursquare

Why MongoDB?

- **It is Open Source :)**

- **High Write Load :**

MongoDB by default prefers high insert rate over transaction safety. If you need to load tons of data lines with a low business value for each one.

- **High Availability in an Unreliable Environment :**

Setting replicaSet (set of servers that act as Master-Slaves) is easy and fast. Moreover, recovery from a node (or a data center) failure is instant, safe and automatic.

- **You need to Grow Big:**

Databases scaling is hard (a single MySQL table performance will degrade when crossing the 5-10GB per table).

Why MongoDB?

- **Location Based:**

MongoDB has built in spacial functions, so finding relevant data from specific locations is fast and accurate.

- **Data Set is Going to be Big (from 1GB) and Schema is Not Stable:**

Adding new columns to RDBMS can lock the entire database in some database, or create a major load and performance degradation in other.

- **You Don't have a DBA:**

If you don't have a DBA, and you don't want to normalize your data and do joins, you should consider MongoDB.

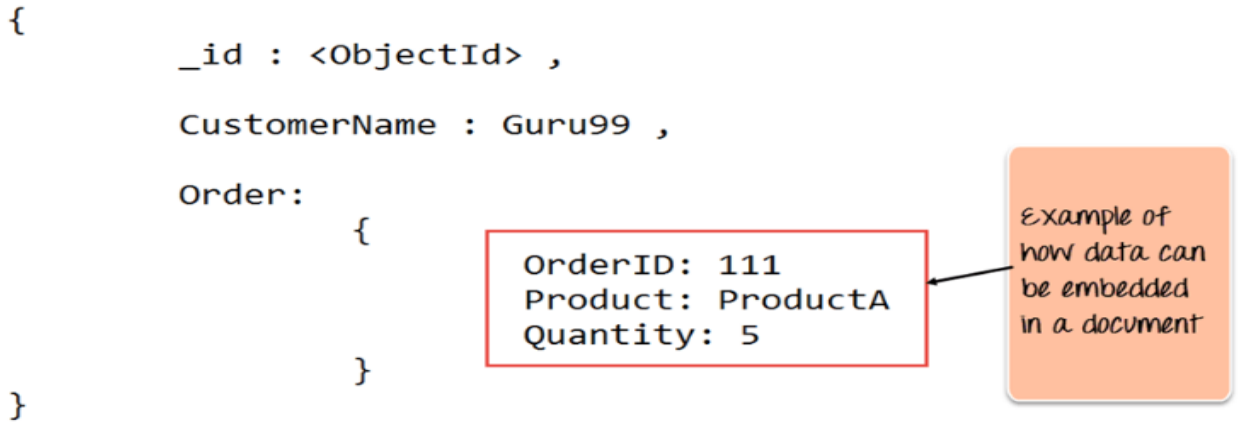
Note: If you are expecting to go big, please notice that you will need to follow some best practices to avoid pitfalls.

https://www.mongodb.com/use-cases

Working with MongoDB

MongoDB Example
The below example shows how a document can be modeled in MongoDB.

1- The _id field is added by MongoDB to uniquely identify the document in the collection.
2- What you can note is that the Order Data (OrderID, Product, and Quantity) which in RDBMS will normally be stored in a separate table, while in MongoDB it is actually stored as an embedded document in the collection itself. This is one of the key differences in how data is modeled in MongoDB.



mongoDB

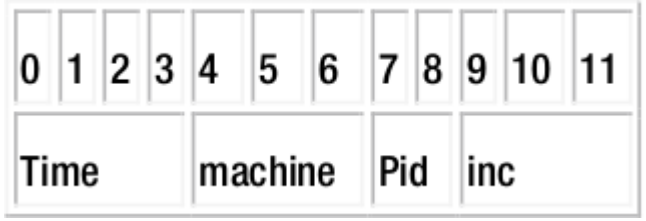
MongoDB Architecture

1. _id –

- This is a field required in every MongoDB document.
- The _id field represents a unique value in the MongoDB document.
- The _id field is like the document's primary key.
- If you create a new document without an _id field, MongoDB will automatically create the field.

_Id	CustomerID	CustomerName	OrderID
563479cc8a8a4246bd27d784	11	Guru99	111
563479cc7a8a4246bd47d784	22	Trevor Smith	222
563479cc9a8a4246bd57d784	33	Nicole	333

ObjectId values consist of 12 bytes, where the first four bytes are a timestamp that reflect the ObjectId's creation. Specifically:



- a 4-byte value representing the seconds since the Unix epoch,
- a 5-byte random value, and
- a 3-byte counter, starting with a random value.

mongoDB

MongoDB Architecture

2. Collection – This is a grouping of MongoDB documents. A collection is the equivalent of a table which is created in any other RDMS such as Oracle or MS SQL. A collection exists within a single database. As seen from the introduction collections don't enforce any sort of structure.

3. Cursor – This is a pointer to the result set of a query. Clients can iterate through a cursor to retrieve results.

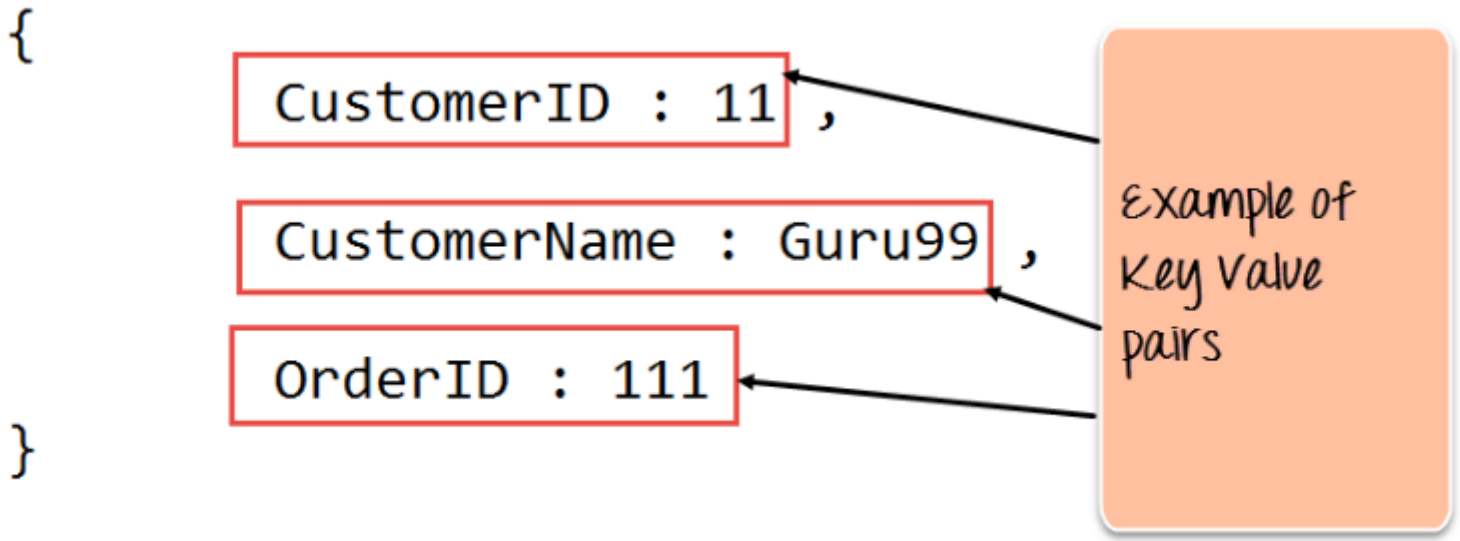
4. Database – This is a container for collections like in RDMS wherein it is a container for tables. Each database gets its own set of files on the file system. A MongoDB server can store multiple databases.

5. Document - A record in a MongoDB collection is basically called a document. The document, in turn, will consist of field name and values.

6. Field - A name-value pair in a document. A document has zero or more fields. Fields are analogous to columns in relational databases.

MongoDB Architecture

The following diagram shows an example of Fields with Key value pairs. So in the example below CustomerID and 11 is one of the key value pair's defined in the document.



mongoDB

MongoDB Architecture

7. JSON – JavaScript Object Notation (JSON) is an open, human and machine-readable standard that facilitates data interchange, and along with XML is the main format for data interchange used on the modern web. JSON supports all the basic data types you'd expect: numbers, strings, and boolean values, as well as arrays and hashes.
Mongo Basics

Binary JSON (BSON) MongoDB represents JSON documents in binary-encoded format called BSON behind the scenes. BSON extends the JSON model to provide additional data types and to be efficient for encoding and decoding within different languages.

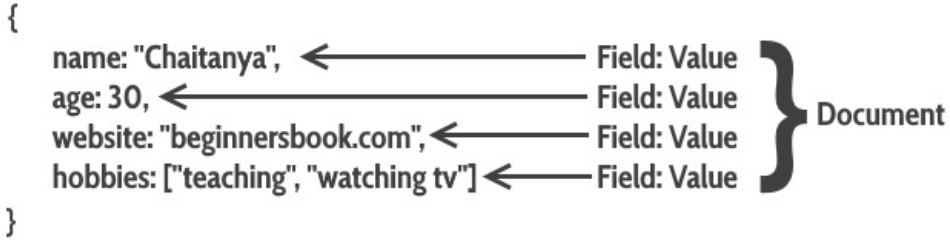
Data Types

Type	Number	Alias	Notes
Double	1	“double”	
String	2	“string”	
Object	3	“object”	
Array	4	“array”	
Binary Data	5	“binData”	
Undefined	6	“undefined”	Deprecated
ObjectId	7	“objectId”	
Boolean	8	“bool”	
Date	9	“date”	
Null	10	“null”	
Regular Expression	11	“regex”	
DBPointer	12	“dbPointer”	Deprecated
JavaScript	13	“javascript”	
Symbol	14	“symbol”	Deprecated
Javascript (with scope)	15	“javascriptWithScope”	
32-bit integer	16	“int”	
Timestamp	17	“timestamp”	
64-bit Integer	18	“long”	
Decimal128	19	“decimal	New in Version 3.4
Min Key	-1	“minKey”	
Max Key	127	“maxKey”	

mongoDB

SQL to MongoDB Mapping

Collections in MongoDB is equivalent to the tables in RDBMS.
Documents in MongoDB is equivalent to the rows in RDBMS.
Fields in MongoDB is equivalent to the columns in RDBMS.



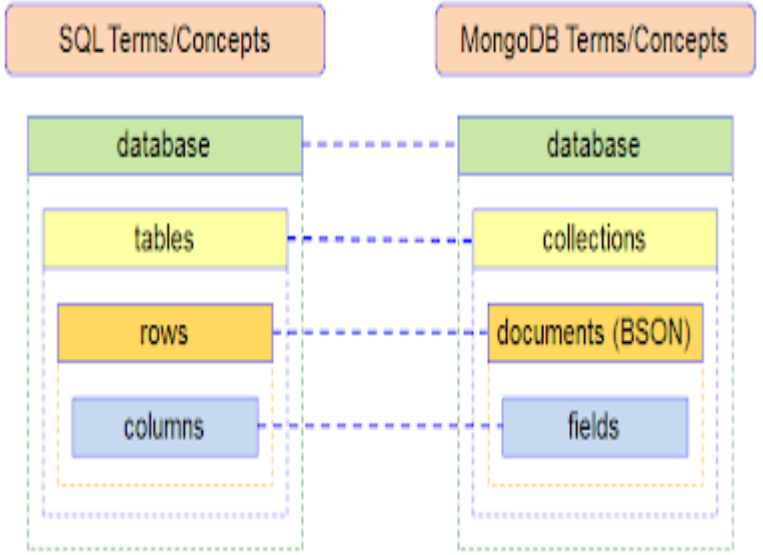


Table vs Collection

Relational Database

Student_Id	Student_Name	Age	College
1001	Chaitanya	30	Beginnersbook
1002	Steve	29	Beginnersbook
1003	Negan	28	Beginnersbook

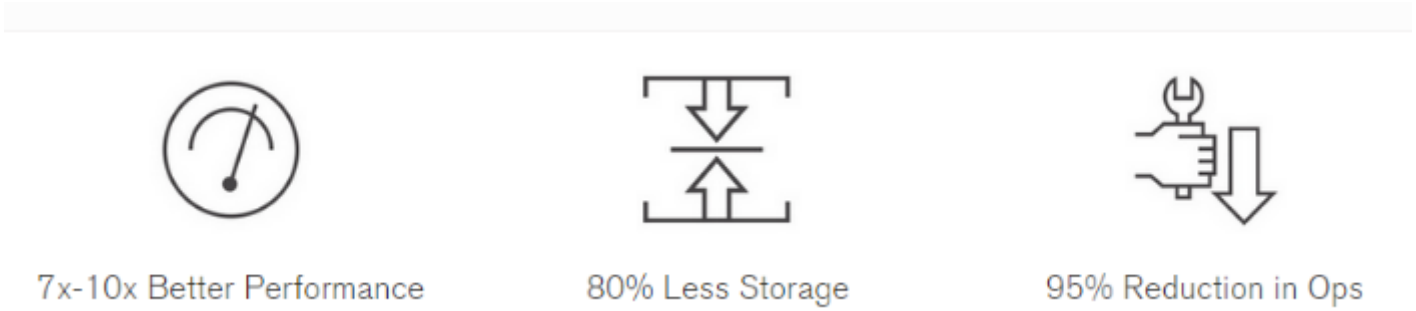
MongoDB

```
{
 "_id": ObjectId("....."),
 "Student_Id": 1001,
 "Student_Name": "Chaitanya",
 "Age": 30,
 "College": "Beginnersbook"
}
{
 "_id": ObjectId("....."),
 "Student_Id": 1002,
 "Student_Name": "Steve",
 "Age": 29,
 "College": "Beginnersbook"
}
{
 "_id": ObjectId("....."),
 "Student_Id": 1003,
 "Student_Name": "Negan",
 "Age": 28,
 "College": "Beginnersbook"
}
```

mongoDB

How MongoDB Stores Data?

- **RDMS** stores data in tables format and uses structured query language (SQL) to query database.
- **RDBMS** has pre-defined database schema based on the requirements and a set of rules to define the relationships between fields in tables.



- **MongoDB** stores data in documents in-spite of tables. You can change the structure of records (which is called as documents in MongoDB) simply by adding new fields or deleting existing ones.
- **MongoDB** helps you to represent hierarchical relationships, to store arrays, and other more complex structures easily. MongoDB provides high performance, high availability, easy scalability and out-of-the-box replication and auto-sharding.

mongoDB

Terminology

RDBMS		MongoDB
Database	→	Database
Table	→	Collection
Index	→	Index
Row	→	Document
Join	→	Embedding & Linking

mongoDB

Platform And Language Support

- C
- C++
- C# and .NET
- Java
- Node.js
- Perl
- PHP, PHP Libraries, Frameworks, and Tools.
- Python
- Ruby
- Mongoid (Ruby ODM)

MongoDB Advantages

- MongoDB is a Schema less document type database.
- MongoDB support field, range based query, regular expression or regex etc for searching the data from the stored data.
- MongoDB is very easy to scale up or down.
- MongoDB basically uses internal memory for storing the working temporary datasets for which it is much faster.
- MongoDB support primary and secondary index on any fields.
- MongoDB supports replication of database.
- We can perform load balancing in the MongoDB by using Sharding. It scales the database horizontally by using Sharding.
- MongoDB can be used as a file storage system which is known as a GridFS.
- MongoDB provides the different ways to perform aggregation operations on the data like aggregation pipeline, map reduce or single objective aggregation commands.
- MongoDB can store any type of file which can be any size without effecting our stack
- MongoDB basically use JavaScript objects in place of procedure.

MongoDB Limitations

- Since MongoDB is not as strong ACID (Atomic, Consistency, Isolation & Durability) as compare to the most RDBMS systems.
- It can’t handle complex transactions
- In MongoDB, there is not provision for Stored Procedure or functions or trigger so there are no chances to implement any business logic in the database level which can be done in any RBMS systems.

BREAK TIME

Getting Start

Installation and configuration

1- Reload local package database

```
sudo apt-get update
```

2- Install the MongoDB packages

```
sudo apt-get install mongodb
```

3- MongoDB service

```
sudo service mongodb start
```

Follow Installation Ref: https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/

Or direct download link: https://www.mongodb.com/download-center/community

MongoDB Tools And Packages

bsondump Reads contents of BSON-formatted rollback files.

mongo The database shell.

mongod The core database server.

mongodump Database backup utility.

mongoexport Export utility (JSON, CSV, TSV), not reliable for backup.

mongofiles Manipulates files in GridFS objects.

mongoimport Import utility (JSON, CSV, TSV), not reliable for recoveries.

For more Tools and Packages visite the following link:
https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/#mongodb-packages

Mongo Shell

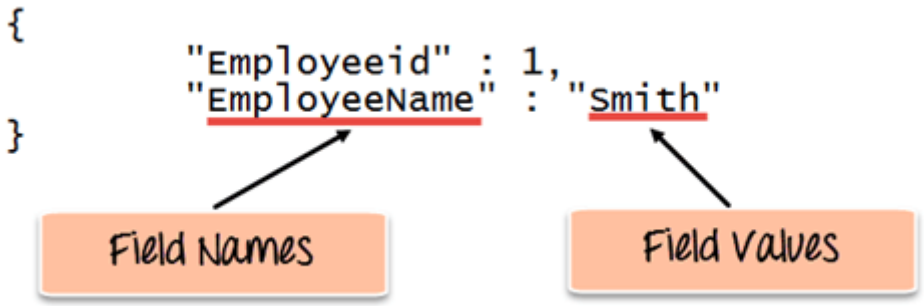
The mongo shell is an interactive JavaScript interface to MongoDB.
You can use the mongo shell to query and update data as well as perform administrative operations.

Start MongoDB Shell
mongo

Database

In **MongoDB**

- The First basic step is to have a database and collection in place.
- The database is used to store all of the collections
- The collection in turn is used to store all of the documents.
- The documents in turn will contain the relevant Field Name and Field values.



The Field Names of the document are "Employeeid" and "EmployeeName" and the Field values are "1" and "Smith' respectively. A bunch of documents would then make up a collection in MongoDB.

mongoDB

Create Database

Show databases

`show dbs` Print a list of all databases on the server.

Create database

`use <db>` create a database in MongoDB. If the database does not exist a new one will be created.

Show current database

```
>db
```

Insert document

```
> db.user.insert({name: "Ankit", age
```

Drop Database

1- list all data base

```
show dbs
```

2- get current database

```
db
```

3- select your database

```
Use <db name>
```

4- drop current database

```
>db.dropDatabase()
```

Create Collection

- **Creating a Collection/Table using insert()**

The easiest way to create a collection is to insert a record (which is nothing but a document consisting of Field names and Values) into a collection. If the collection does not exist a new one will be created.

```
db.Employee.insert
(
        {
                "Employeeid" : 1,
                "EmployeeName" : "Martin"
        }
)
```

Create Collection

- **Creating a Collection/Table using createCollection()**

Because MongoDB creates a collection implicitly when the collection is first referenced in a command, this method is used primarily for creating new collections that use specific options.

For example, you use db.createCollection() to create a capped collection, or to create a new collection that uses document validation.

The db.createCollection() method has the following **parameters**:

Parameter	Type	Description
name	string	The name of the collection to create. See Naming Restrictions.
options	document	Optional. Configuration options for creating a capped collection, for preallocating space in a new collection, or for creating a view.

mongoDB

Create Collection

- **Creating a Collection/Table using createCollection()**

The options document contains the following **fields**:

Field	Type	Description
capped	boolean	Optional. To create a capped collection, Specify true. If you specify true, the size parameter has to be specified.
autoindexid	boolean	Optional. The default value is false. Disables automatic creation of an index on the _id field.
size	number	Optional. Specifies the maximum size for capped collection. MongoDB removes the older documents once the max size limit is reached.
max	number	Optional. The maximum no. of documents allowed in a capped collection. Size has precedence over this.
validator	document	Optional. Allows the user to specify validation rules for the collection
Validation level	string	Optional. Determines the strictness of validation. Use "off", "strict" and "moderate".

For more fields visit the following link:
https://docs.mongodb.com/manual/reference/method/db.createCollection/?fbclid=IwAR2xbNmgagAdJvvxbCRTOpWStr_aaaNVxyHvwTcFxZ1j4u17MDrotgMkXGo

mongoDB

Drop Collection

```
db.collection_name.drop()
```

WRAP UP

Database Operations (DDL)

Show databases

```
show dbs
```

Create a database

```
use <db name>
```

Show current database

```
db
```

Collections Operations (DDL)

Create a collection

```
db.createCollection('name');
```

Create a collection while inserting (CRUD operation)

```
db.myCollection.insert({ key: "value" });
```

Show Collections

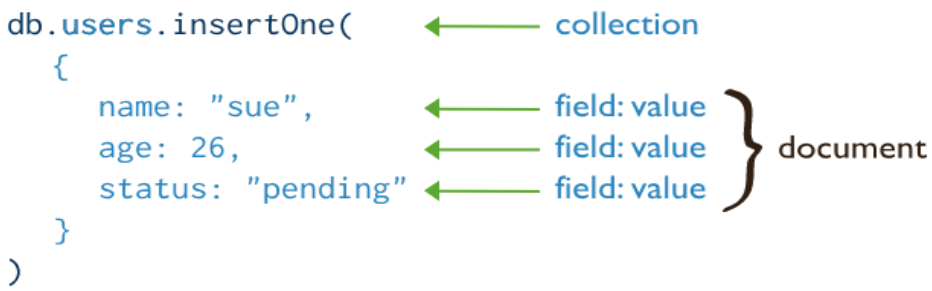
```
show collections
```

CRUD Operations (DML)

The "insert" command can also be used to insert multiple documents into a collection at one time. The below code example can be used to insert multiple documents at a time.

Create Operations

Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.



Insert Documents

Insert a single document into collection

db.collection.insertOne({item: “value”}); or db.collection.insert({ item: “value” });

Insert multiple documents

db.collection.insertMany([{item: “value” },{item:”value2”}, {item:”value3”}]);

mongoDB

CRUD Operations (DML)

Javascript variable could be used as following example:

Step 1) Create a **JavaScript** variable called myEmployee to hold the array of documents
Step 2) Add the required documents with the Field Name and values to the variable
Step 3) Use the insert command to insert the array of documents into the collection

```
var myEmployee=
                [

                        {
                                "Employeeid" : 1,
                                "EmployeeName" : "Smith"
                        },
                        {
                                "Employeeid"   : 2,
                                "EmployeeName" : "Mohan"
                        },
                        {
                                "Employeeid"   : 3,
                                "EmployeeName" : "Joe"
                        },

                ];

                db.Employee.insert(myEmployee);
```

CRUD Operations (DML)

Read Operations

The method of fetching or getting data from a MongoDB database is carried out by using queries.

While performing a query operation, one can also use criteria's or conditions which can be used to retrieve specific data from the database.

MongoDB provides a function called **db.collection.find ()** which is used for retrieval of documents from a MongoDB database.

```
db.users.find(                          <-- collection
   { age: { $gt: 18 } },                <-- query criteria
   { name: 1, address: 1 }              <-- projection
).limit(5)                              <-- cursor modifier
```

mongoDB

CRUD Operations (DML)

Query Documents

Select all documents in collections

```
db.collection.find({});
```

Select with Where equality

```
db.collection.find({ item: “value” });
```

Query using Query Operations

```
db.collection.find({status:{ $in:[ “a”, “b”] }});
```

And Condition

```
db.collection.find({status: "A",qty:{$lt: 30 }})
```

OR Condition

```
db.collection.find({$or:[{status: "A" },{ qty: { $lt: 30}}]})
```

CRUD Operations (DML)

Update Operations

Update operations modify existing **documents** in a **collection.**

```
db.Employee.update(

{"Employeeid" : 1},

{$set: { "EmployeeName" : "NewMartin"}});

---------------------

db.examples.updateOne({ item: "papers" },{$set: { "size.uom": "cm", status: "P" },$currentDate: {
lastModified: true }})

----------------------

db.users.updateMany( {age: {$lt :18}}, {$set: { status:"reject"}} )
```

CRUD Operations (DML)

Delete Operations

Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:

```
db.examples.deleteOne( { status: "D" } )

---------------------------

db.user.deletemany(

{ status:"reject"}

)
```

SQl MongoDB

Simple Querying

Query collection using find()

E.g. >db.user.find(); or >db.user.find().pretty();

SELECT * FROM user;

Query collection with condition using find()

db.user.find({“age":“18"}).pretty();

SELECT * FROM user WHERE age=“18";

Fetch selective field from collection based on a criteria
db.user.find({“age":“18"},{"user_id" : 1}).pretty();

SQL:

SELECT user_id FROM user WHERE age=“18";

mongoDB

SQl MongoDB

Simple Querying

Fetch more than one fields from collection based on a criteria

>db.user.find({“age":“18"},{"user_id" : 1,"password":1,"date_of_birth":1}).pretty();

SQL:

SELECT user_id,password,date_of_birth

FROM user

WHERE age=“18";

mongoDB

Sort

One can specify the order of documents to be returned based on ascending or descending order of any key in the collection.

sorting one column in descending order

db.Employee.find().sort({Employeeid:-1}).forEach(printjson)

SELECT * FROM Employee ORDER BY Employeeid DESC;

Sorting one column in ascending order

db.Employee.find().sort({Employeeid:1}).forEach(printjson)

SELECT * FROM Employee ORDER BY Employeeid

Sorting on more than one column Employeeid ascending and password descending

>db. Employee.find().sort({" Employeeid":1,"password":-1})

More Query

skip() and limit()

This modifier is used to limit the number of documents which are returned in the result set for a query.

```
db.Employee.find().limit(2).forEach(printjson);
```

Sometimes it is required to return a certain number of results after a certain number of documents. The skip() can do this job.

```
db.Employee.find().skip(2).forEach(printjson);
```

Query Modifiers

Query modifiers determine the way that queries will be executed.

Name	Description
`$comment`	Adds a comment to the query to identify queries in the database profiler output.
`$explain`	Forces MongoDB to report on query execution plans. See `explain()`.
`$hint`	Forces MongoDB to use a specific index. See `hint()`
`$max`	Specifies an *exclusive* upper limit for the index to use in a query. See `max()`.
`$maxTimeMS`	Specifies a cumulative time limit in milliseconds for processing operations on a cursor. See `maxTimeMS()`.
`$min`	Specifies an *inclusive* lower limit for the index to use in a query. See `min()`.
`$orderby`	Returns a cursor with documents sorted according to a sort specification. See `sort()`.
`$query`	Wraps a query document.
`$returnKey`	Forces the cursor to only return fields included in the index.
`$showDiskLoc`	Modifies the documents returned to include references to the on-disk location of each

Query language

The **$query** operator forces MongoDB to interpret an expression as a query.

The **$query** can be evaluated as the WHERE clause of SQL and $orderby sorts the results as specified order.

we want to fetch documents from the collection "user" which contains the value of "date_of birth " is "16/10/2010" and the value of “age" is “20" and sort the fetched results in descending order,

db.user.find({"date_of_birth" : "16/10/2010",“birth":“20"}).sort({"profession":-1}).pretty();
Using The Query Language

db.user.find({$query : {"date_of_birth" : "16/10/2010",“age":“20"}, $orderby : {"profession":-1}}).pretty();

For more Query modifiers:

https://docs.mongodb.com/manual/reference/operator/query-modifier/

mongoDB

MongoDB Dot Notation

Self-study and make a Report

Data Modelling

Data in MongoDB is schema-less, so no need of defining a structure for the data before insertion.

Since, MongoDB is a document based database, any document within the same collection is not mandatory to have same set of fields or structure. This helps in easily mapping the documents with the entity objects.

the documents in a collection of MongoDB will always share the same data structure(recommended for best performance, not mandatory).

The vital factor or challenge in modelling the data in a given database is load balancing and hence ensuring the performance aspect effectively. It is a mandate while modelling the data to consider the complete usage of data (CRUD operations) along with how the data will be inherited as well.

Data Modelling

Flexible Schema

- The documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.

- To change the structure of the documents in a collection, such as add new fields, remove existing fields, or change the field values to a new type, update the documents to the new structure

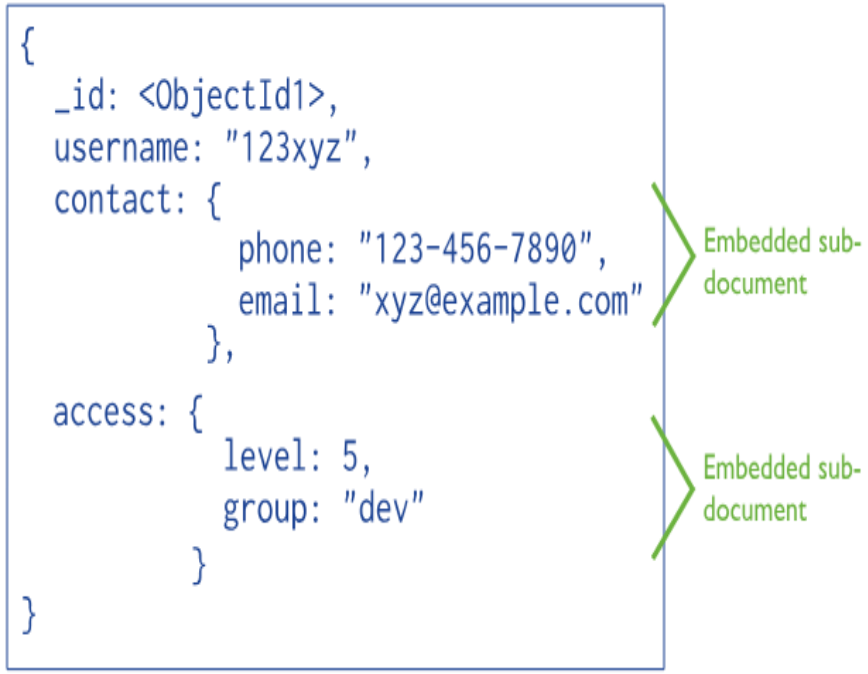
Document Structure

There can be two ways to establish relationships between the data in MongoDB:

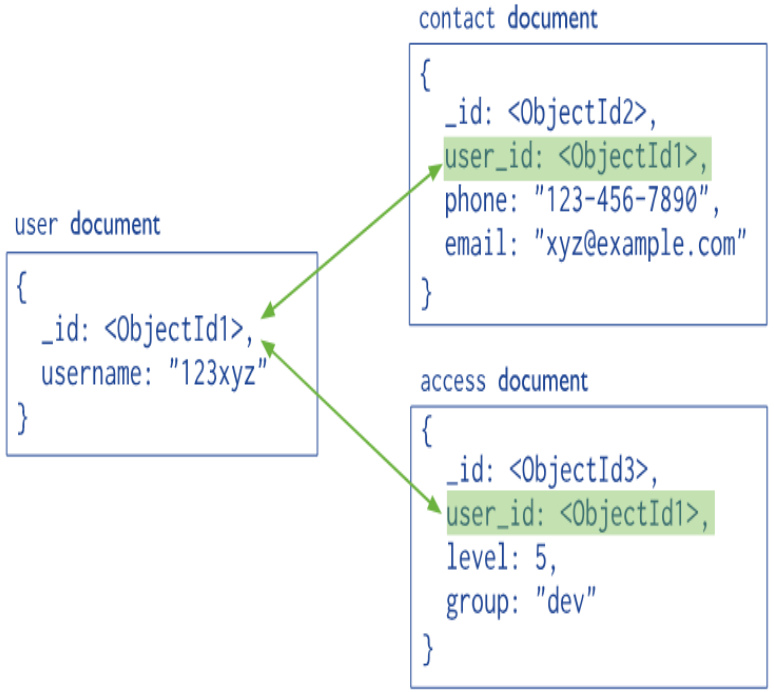
- Referenced Documents
- Embedded Documents

Data Modelling

Embedded documents capture relationships between data by storing related data in a single document structure



References store the relationships between data by including links or references from one document to another.



mongoDB

Data Modelling

Atomicity

Single Document

A write operation is atomic on the level of a single document, even if the operation modifies multiple embedded documents within a single document
A denormalized data model facilitates atomic operations.
Modifying multiple documents, the modification of each document is atomic, but the operation as a whole is not atomic.

Multi-Document Transactions

Starting in version 4.0 multi-document transactions for replica sets.

Schema validation

- **Schema validation** allows you to define the specific structure of documents in each collection.
- If anyone tries to insert some documents which don't match with the defined schema
- MongoDB can reject this kind of operation or give warnings according to the type of validation action.
- MongoDB provides two ways to validate your schema, Document validation, and JSON schema validation.

Schema validation

Document validation

```
{
   "name": "Alex",
   "email": "alex@gmail.com",
   "mobile": "123-456-7890"
}
```

- name, email fields are mandatory
- mobile numbers should follow specific structure: xxx-xxx-xxxx

To add this validation, we can use the “validator” construct while creating a new collection.

```
db.createCollection("users", {
  validator: {
      $and: [
         {
            "name": {$type: "string", $exists: true}
         },
         {
            "mobile": {$type: "string", $regex: /^[0-9]{3}-[0-9]{3}-[0-9]{4}$/}
         },
         {
            "email": {$type: "string", $exists: true}
         }
      ]
   }
})
```

mongoDB

Schema validation

Document validation

```
db.users.insert({
   "name": "akash"
})

WriteResult({
   "nInserted" : 0,
   "writeError" : {
      "code" : 121,
      "errmsg" : "Document failed validation"
   }
})
----------------------------
db.users.insert({
   "name": "akash",
   "email": "akash@gmail.com",
   "mobile": "123-456-7890"
})

WriteResult({ "nInserted" : 1 })
```

Schema validation

jsonSchema Validation

```
db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: [ "name", "year"],
      properties: {
        name: {
          bsonType: "string",
          description: "must be a string and is required"
        }
      }
    }
  }
})
```

Designing Schema

- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Relationships in MongoDB

Model One-to-One Relationships with Embedded Documents

```
// patron document
{
  _id: "joe",
  name: "Joe Bookreader"
}

// address document
{
  patron_id: "joe", // reference to patron document
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}
```

```
{
  _id: "joe",
  name: "Joe Bookreader",
  address: {
          street: "123 Fake Street",
          city: "Faketon",
          state: "MA",
          zip: "12345"
         }
}
```

Relationships in MongoDB

Model One-to-Many Relationships with Embedded Documents

```
// patron document
{
  _id: "joe",
  name: "Joe Bookreader"
}

// address documents
{
  patron_id: "joe", // reference to patron document
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}

{
  patron_id: "joe",
  street: "1 Some Other Street",
  city: "Boston",
  state: "MA",
  zip: "12345"
}
```

```
{
  "_id": "joe",
  "name": "Joe Bookreader",
  "addresses": [
        {
          "street": "123 Fake Street",
          "city": "Faketon",
          "state": "MA",
          "zip": "12345"
        },
        {
          "street": "1 Some Other
Street",
          "city": "Boston",
          "state": "MA",
          "zip": "12345"
        }
      ]
 }
```

Relationships in MongoDB

Model One-to-Many Relationships with Document References

```
_id: "oreilly",
  name: "O'Reilly Media",
  founded: 1980,
  location: "CA"
}

{
  _id: 123456789,
  title: "MongoDB: The Definitive Guide",
  author: [ "Kristina Chodorow", "Mike Dirolf" ],
  published_date: ISODate("2010-09-24"),
  pages: 216,
  language: "English",
  publisher_id: "oreilly"
}

{
  _id: 234567890,
  title: "50 Tips and Tricks for MongoDB Developer",
  author: "Kristina Chodorow",
  published_date: ISODate("2011-05-06"),
  pages: 68,
  language: "English",
  publisher_id: "oreilly"
```

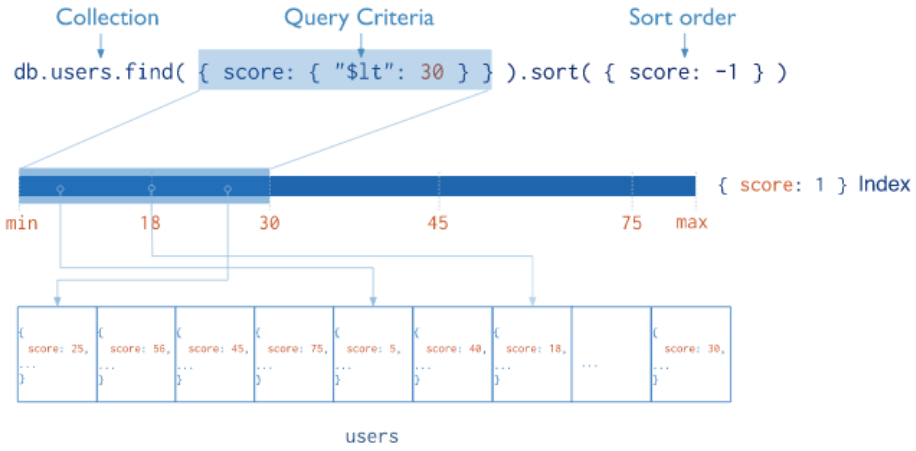
Indexes

Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form.

Stores the value of a specific field or set of fields, ordered by the value of the field.

The ordering of the index entries supports efficient equality matches and range-based query operations.

MongoDB can return sorted results by using the ordering in the index.



mongoDB

Indexes

Default _id Index

MongoDB creates a **unique index** on the **_id** field during the creation of a collection.

The **_id index** prevents clients from inserting two documents with the same value for the _id field.
You cannot drop this index on the _id field.

Create an Index

db.collection.createIndex(<key and index type specification>, <options>)

mongoDB

Indexes - Index Types

Single Field index

MongoDB provides complete support for indexes on any field in a collection of documents.

By default, all collections have an index on the _id field, and applications and users may add additional indexes to support important queries and operations.

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
  "location": { state: "NY", city: "New York" }
}
```

```
db.records.createIndex( { score: 1 } )
```

Indexes - Index Types

Compound Indexes

MongoDB supports compound indexes, where a single index structure holds references to multiple fields within a collection's documents.

```
db.collection.createIndex( { <field1>: <type>, <field2>: <type2>, ... } )
```

```
{
 "_id": ObjectId(...),
 "item": "Banana",
 "category": ["food", "produce", "grocery"],
 "location": "4th Street Store",
 "stock": 4,
 "type": "cases"
}
```

Hashed Indexes

MongoDB supports hash-based sharding and provides hashed indexes. These indexes are the hashes of the field value.

mongoDB

Indexes - Index Types

Multikey Indexes
To index a field that holds an array value, MongoDB creates an index key for each element in the array.
These multikey indexes support efficient queries against array fields.
Multikey indexes can be constructed over arrays that hold both scalar values (e.g. strings, numbers) and nested documents.

```
db.coll.createIndex( { <field>: < 1 or -1 > } )
```

Geospatial Index
To query geospatial data, MongoDB supports two types of indexes – 2d indexes and 2dsphere indexes.

Text Indexes
MongoDB provides a text index type that supports searching for string content in a collection. These text indexes do not store language-specific stop words (e.g. “the”, “a”, “or”) and stem the words in a collection to only store root words.

```
db.reviews.createIndex( { comments: "text" } )
```

mongoDB

Indexes - Index Properties

Unique Indexes

The unique property for an index causes MongoDB to reject duplicate values for the indexed field

```
db.records.createIndex({name:1},{unique:true})
```

Partial Indexes

Partial Indexes only index the documents that match the filter criteria. If we are creating an index with some conditions applied then it is a partial index

```
db.records.createIndex({city:1},{partialFilterExpression:{name:"os39"}})
```

Sparse Indexes

The sparse property ensures that the index only contains entries for documents with the indexed field. The index will skip the documents without the indexed field.

```
db.reco.createIndex({name:1},{unique:true,sparese:true})
```

TTL Indexes (Total Time To Live)

special indexes in MongoDB. These indexes are used to auto-delete documents from a collection after the specified time duration. The option that we use is expireAfterSecods to provide the expiration time. This property is ideal for certain types of information like machine-generated data, logs and session information that only need to be there for a finite amount of time in a database.

```
db.reco.createIndex({name:1},{expireAfterSecods:90})
```

mongoDB

Indexes - Indexing Limitations

Range Limitations

A collection cannot have indexed more than 64.
The name of the index can contain only 164 characters.
A compound index can have 31 fields indexed at max.

RAM Usage

The total size of indexes must not exceed the RAM size as the indexes are stored in the RAM. If this limit exceeds, it will cause deletion of some indexes and deteriorate the performance.

Query Limitations

Queries should not use expressions like $nin, $not, etc.Queries should not use arithmetic operators like $mod etc.

A Query should not use $where clause.

Indexed Key Limits

MongoDB will not create an index if the value of existing index field exceeds the index key limit.

Aggregation

Aggregation operations

- process data records and return computed results.
- group values from multiple documents together
- can perform a variety of operations on the grouped data to return a single result.

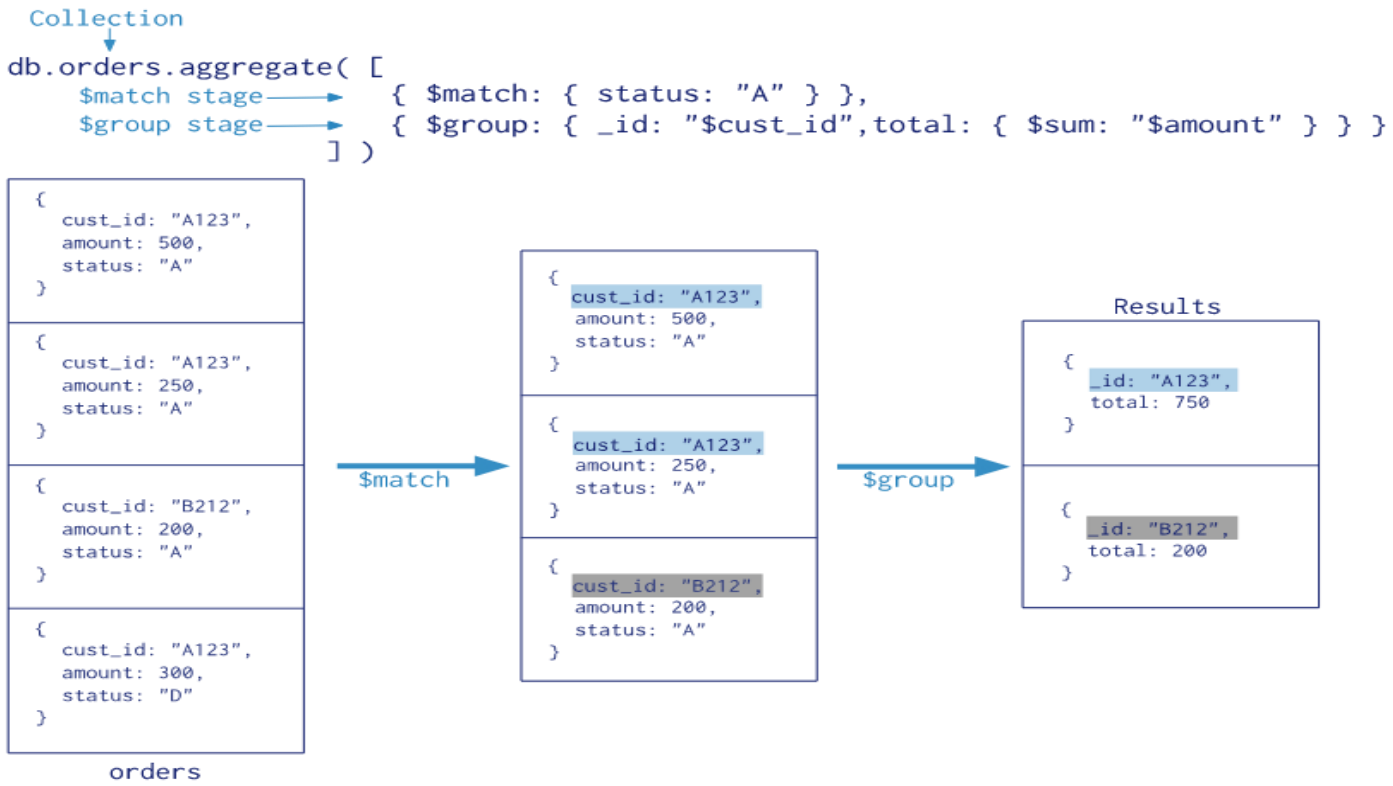
MongoDB provides three ways to **perform aggregation:**

- The aggregation pipeline
- The map-reduce function
- Single purpose aggregation methods

mongoDB

Aggregation - Aggregation pipeline

Multiple documents enter the pipeline and then these documents are being transformed into aggregated results.
The operations being performed during pipeline include filter and document transformation in which they operate like queries and modify the form of output document respectively.



mongoDB

Aggregation - Aggregation pipeline

Stages of MongoDB Aggregation Pipeline

$project

- used to select certain fields from a collection.
- Add, remove or reshape a key are ablicabl.

```
db.books.aggregate( [ { $project : { title : 1 , author : 1 } } ] )
```

$match

- used in filtering operation
- reduce the number of documents that are given as input to the next stage.

```
db.articles.aggregate([ { $match : { author : "dave" } } ]);
```

$group

- groups all documents based on some keys.
- The output documents also contain computed fields that hold the values of some accumulator expression grouped

$min, $max, $sum, $avg

```
db.sales.aggregate( [ { $group : { _id : "$item" } } ] )
```

mongoDB

Aggregation - Aggregation pipeline

Stages of MongoDB Aggregation Pipeline

$sort

- set the sort order to 1 or -1 to specify an ascending or descending sort

```
db.users.aggregate([{ $sort : { age : -1, posts: 1 }
```

$skip & $limit

- Using skip to skip forward in the list of all documents for the given limit.
- Using limit to limit the number of documents we want to look at by specifying the limit as per our convenience.

$first & $last

- used to get the first and last values in each group of documents.

$unwind

- used for all documents, that are using arrays.

mongoDB

Aggregation - Aggregation pipeline

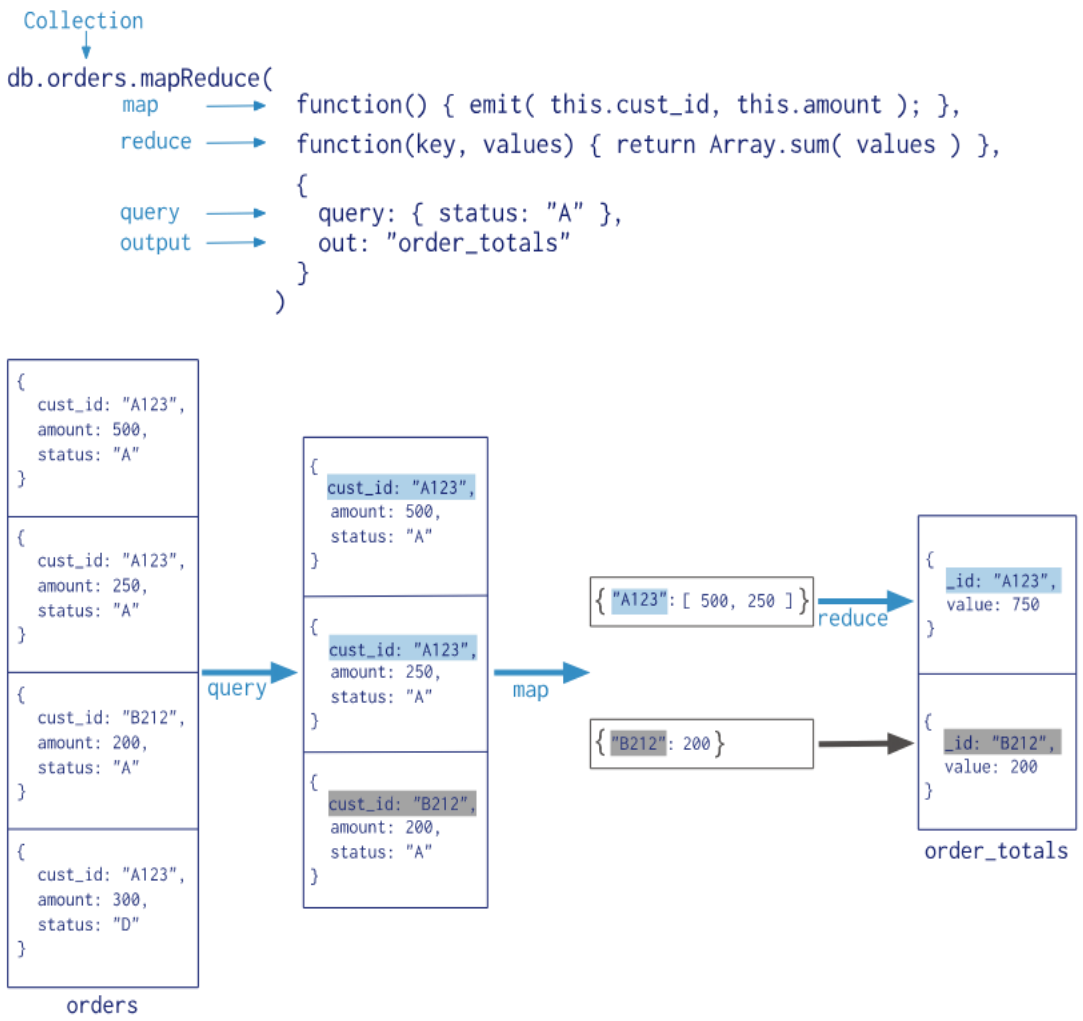
Mapping To SQL

SQL command	Aggregation framework operator
SELECT	$project
	$group functions: $sum, $min, $avg, etc.
FROM	db.collectionName.aggregate(...)
JOIN	$unwind
WHERE	$match
GROUP BY	$group
HAVING	$match

Aggregation - Map-Reduce

map-reduce operations have two phases:
1- A map stage that processes each document and emits one or more objects for each input document
2- Reduce phase that combines the output of the map operation.

- Optionally, map-reduce can have a finalizestage to make final modifications to the result.
- Map-reduce uses custom JavaScript functions to perform the map and reduce operations including the optional finalize operation.
- Although custom JavaScript provide great flexibility compared to the aggregation pipeline but still map-reduce is less efficient and more complex than the aggregation pipeline.
- Map-reduce can operate on a sharded collection. Map-reduce operations can also output to a sharded collection.



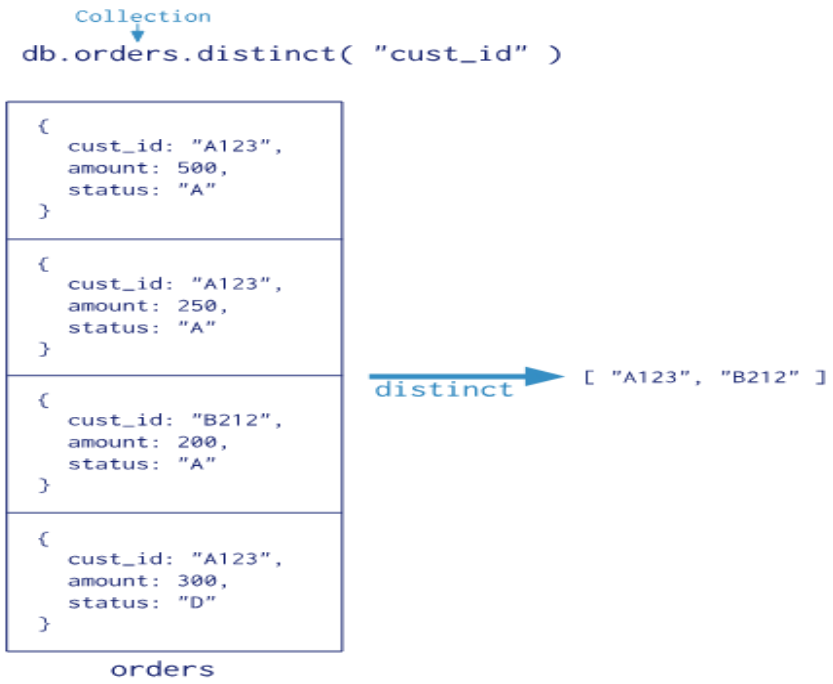
mongoDB

Aggregation - Single Purpose Aggregation Operations

MongoDB also provides

- db.collection.estimatedDocumentCount()
- db.collection.count()
- db.collection.distinct().

All of these operations aggregate documents from a single collection. While these operations provide simple access to common aggregation processes, they lack the flexibility



mongoDB

Aggregation

- Arithmetic Expression Operators
- Array Expression Operators
- Boolean Expression Operators
- Comparison Expression Operators
- Conditional Expression Operators
- Date Expression Operators
- Literal Expression Operator
- Object Expression Operators
- Set Expression Operators
- String Expression Operators
- Text Expression Operator
- Trigonometry Expression Operators
- Type Expression Operators
- Accumulators (`$group`)
- Accumulators (in Other Stages)
- Variable Expression Operators

mongoDB

Aggregation

`$arrayElemAt`	**Returns the element at the specified array index.**
`$arrayToObject`	Converts an array of key value pairs to a document.
`$concatArrays`	Concatenates arrays to return the concatenated array.
`$filter`	Selects a subset of the array to return an array with only the elements that match the filter condition.
`$in`	Returns a boolean indicating whether a specified value is in an array.
`$indexOfArray`	Searches an array for an occurence of a specified value and returns the array index of the first occurence. If the substring is not found, returns `-1`.
`$isArray`	Determines if the operand is an array. Returns a boolean.
`$map`	Applies a subexpression to each element of an array and returns the array of resulting values in order. Accepts named parameters.
`$objectToArray`	Converts a document to an array of documents representing key-value pairs.
`$range`	Outputs an array containing a sequence of integers according to user-defined inputs.

mongoDB

Aggregation

If we want to fetch documents from the collection "userinfo" which contains the value of "age" is either 19 or 22

mongodb command can be used :

```
db.user.find({"age" :{$in :[19,22]}}).pretty();
```

Sql:

```
SELECT * FROM user WHERE age IN(19,22);
```

```
db.user.find({ "sex" : "Male" , $or : [ { "age" : 17 } , { "date_of_join" : "17/10/2009" } ]
} ).pretty();
```

```
FROM user WHERE sex="Male" AND (age=17 or date_of_join="17/10/2009");
```

Aggregation

$gt

- **true** when the first value is greater than the second value.
- **false** when the first value is less than or equivalent to the second value.

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
---------------------------------------------
db.inventory.aggregate(
  [
    {
      $project:
        {
          item: 1,
          qty: 1,
          qtyGt250: { $gt: [ "$qty", 250 ] },
          _id: 0
        }
    }
  ]
)
```

Aggregation

$type operator in mongodb matches values based on their BSON type.
$exist operator checks whether the particular field exist or not.

Fetch documents containing 'null'

```
db.testtable.find( { "interest" : null } ).pretty();
```

Fetch documents with $type Operator (10 in type table is refer to null)

```
db.testtable.find( { "interest" : { $type : 10 } } ).pretty();
```

fetch documents from the collection "testtable" which does not contain the "interest" column

```
db.testtable.find( { "interest" : { $exists : false } } ).pretty();
```

Replication

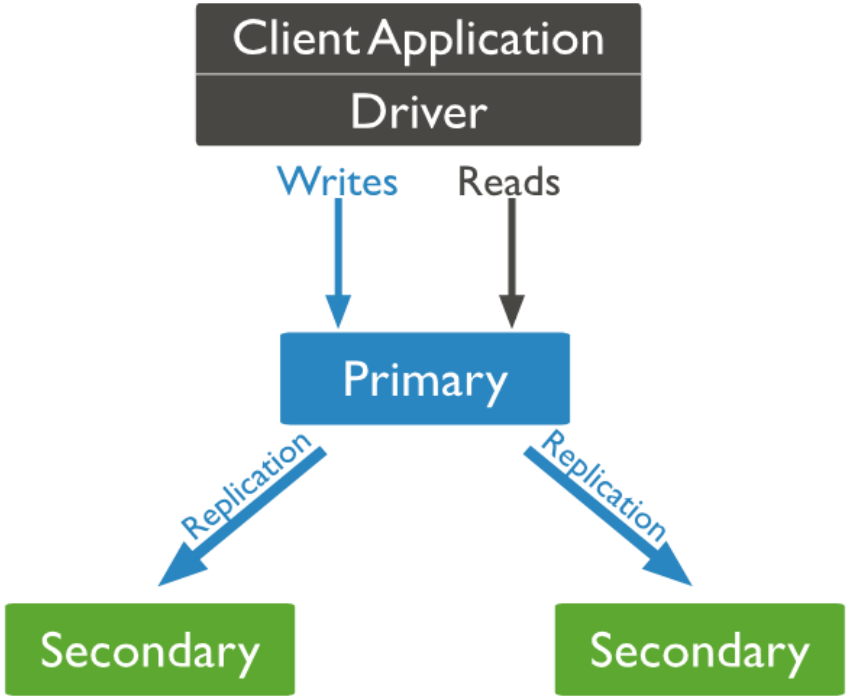
A replica set in MongoDB

- Is a group of mongod processes that maintain the same data set.
- provide redundancy and high availability, and are the basis for all production deployments.
- Increase read capacity
- increases data availability, With multiple copies of data on different database servers
- replication provides a level of fault tolerance against the loss of a single database server.

Replication - Replication Members

Primary

- The only member in the replica set that receives write operations.
- MongoDB applies write operations on the primary and then records the operations on the primary's oplog.
- Secondary members replicate this log and apply the operations to their data sets.
- The replica set can have at most one primary. If the current primary becomes unavailable, an election determines the new primary.

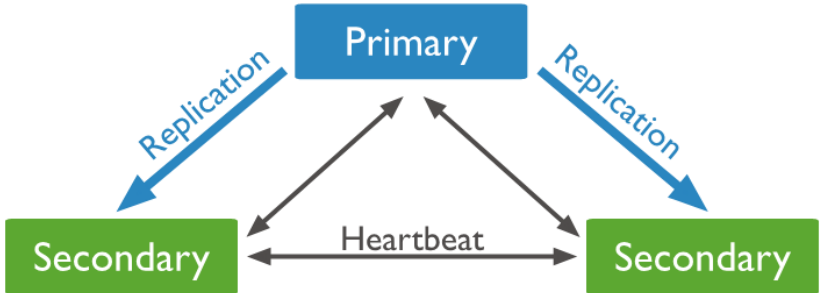


mongoDB

Replication - Replication Members

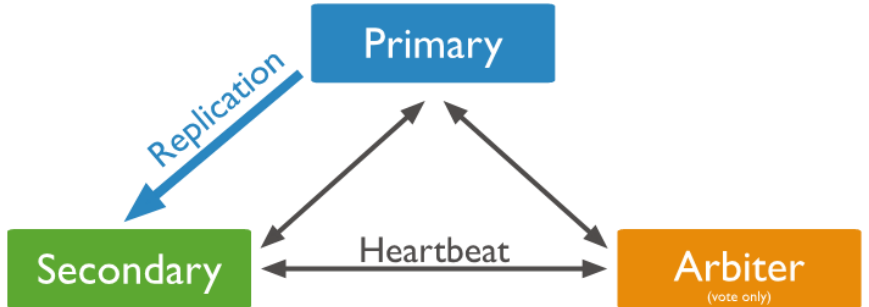
secondary members.

- A secondary maintains a copy of the primary's data set.
- To replicate data, a secondary applies operations from the primary's oplog to its own data set in an asynchronous process.
- A replica set can have one or more secondaries.
- clients cannot write data to secondaries, clients can read data from secondary members.
- A secondary can become a primary. If the current primary becomes unavailable, the replica set holds an election to choose which of the secondaries becomes the new primary.



Arbiter

- Does not have a copy of data set and cannot become a primary. However, an arbiter participates in elections for primary.
- An arbiter will always be an arbiter whereas a primary may step down and become a secondary and a secondary may become the primary during an election.



mongoDB

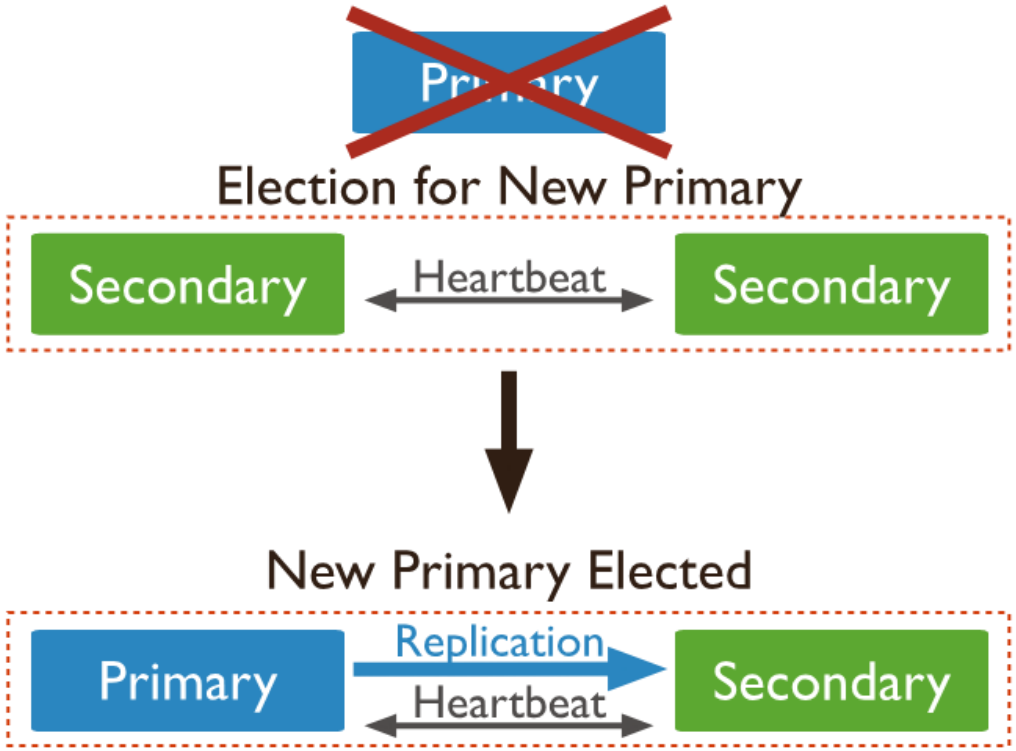
Replication - Replication Members

Automatic Failover

If primary go offline more than the configured electionTimeoutMillis period (10 seconds by default).

An eligible secondary calls for an election to nominate itself as the new primary

The median time before a cluster elects a new primary should not typically exceed 12 seconds.



mongoDB

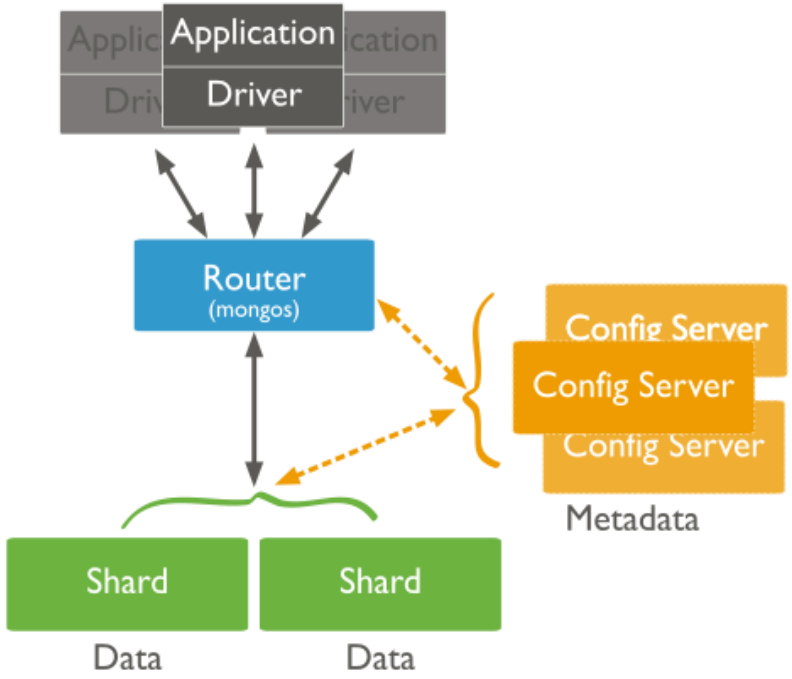
Replication - Setup Replica Set

- Shutdown already running MongoDB server.
- Ensure that all mongod.exe instances which will be added to the replica set are installed on different servers.
- Ensure that all mongo.exe instances can connect to each other.
  mongo –host ServerB –port 27017
- Start the first mongod.exe instance with the replSet option.
  mongo –replSet "Replica1"
- initiate the replica set by rs.initiate ()
- Verify the replica set by rs.conf() to ensure the replica set up properly
- To check the status of replica set issue the command rs.status().
- Add members, start mongod instances on multiple machines.
  rs.add(HOST_NAME:PORT)
  rs.add("mongod1.net:27017")

Sharding

MongoDB supports horizontal scaling through sharding.

- Sharding is a Method for distributing data across multiple machines.
- Splitting data across multiple horizontal partitions.



mongoDB

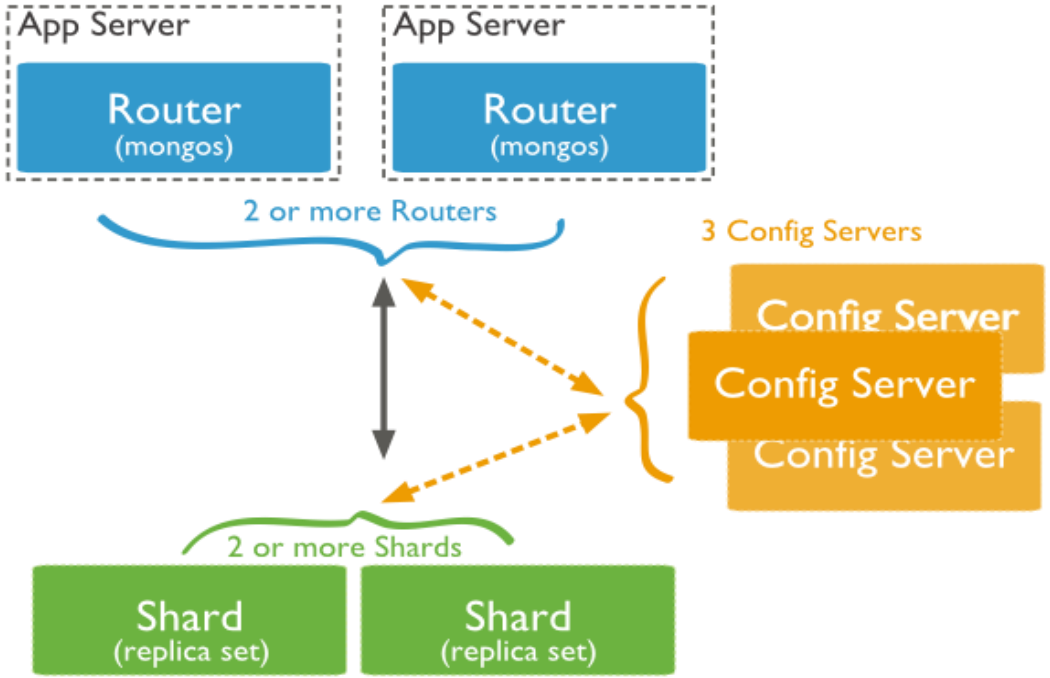
Sharded cluster Component

A MongoDB sharded cluster consists of the following components:

shard: Each shard contains a subset of the sharded data. Each shard can be deployed as a replica set.

mongos: The mongos acts as a query router, providing an interface between client applications and the sharded cluster.

config servers: Config servers store metadata and configuration settings for the cluster. As of MongoDB 3.4, config servers must be deployed as a replica set (CSRS).

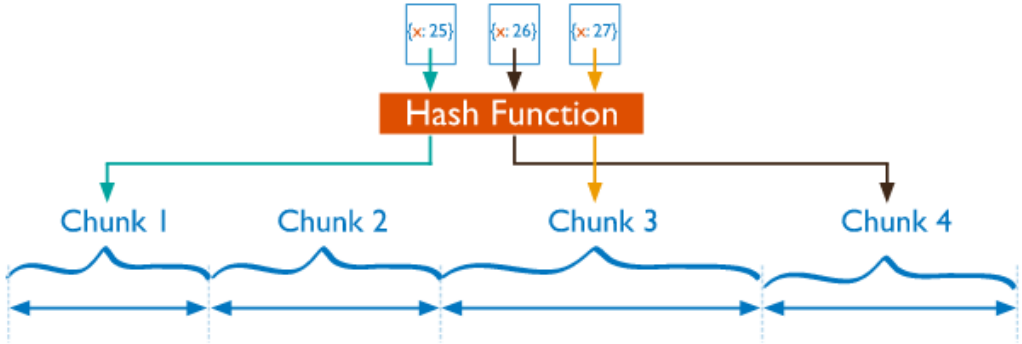


mongoDB

Sharding Strategy

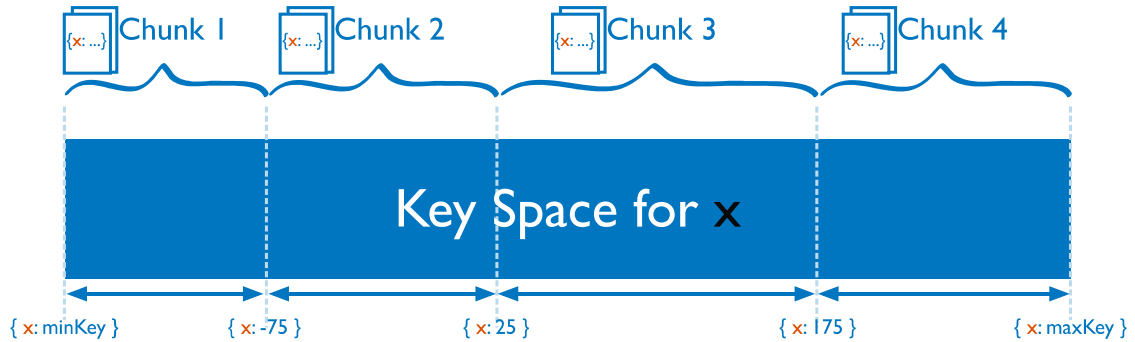
Hashed Sharding

involves computing a hash of the shard key field's value. Each chunk is then assigned a range based on the hashed shard key values.



Ranged Sharding

Ranged sharding involves dividing data into ranges based on the shard key values.
Each chunk is then assigned a range based on the shard key values.



mongoDB

Set Shard in MongoDB

- Create a database for config server.

  mkdir /data/exampledb
- Start the MongoDB instance in configuration mode.

  mongod –exampledb ExamplesD: 27019
- Start the mongos instance by specifying the configuration server.

  mongos –exampledb ExamplesD: 27019
- From mongo shell connect to mongo's instance

  mongo –host ServerD –port 27017
- If we have two servers named S1 and S2, which are to be clustered then use the following command.

  sh.addShard("S1:27017")

  sh.addShard("S2:27017")
- Enable sharding for the database.

  sh.enableSharding(Studentdb)
- Enable sharding for collection

  sh.shardCollection("db.Student" , { "Studentid" : 1 , "StudentName" : 1})

Create User & Add Role in MongoDB

Create Administrator User

```
db.createUser(
{           user: “user99",
            pwd: "password",

            roles:[{role: "userAdminAnyDatabase" , db:"admin"}]})
```

Create User for Single Database

```
db.createUser(
{
            user: "Employeeadmin",

            pwd: "password",

            roles:[{role: "userAdmin" , db:"Employee"}]})
```

Managing users

```
db.createUser(
{
            user: "Mohan",

            pwd: "password",

            roles:[
            {
                        role: "read" , db:"Marketing"},

                        role: "readWrite" , db:"Sales"}
            }
                         ]
})
```

MongoDB Backup and Restore Data

creating a backup mongodump command

mongodump --dbpath DB_PATH --out BACKUP_DIRECTORY

mongodump --dbpath /data/db/ --out /data/backup/

mongorestore

mongorestore <options> <directory or file to restore>

MongoDB Export Data & Import Data

MongoDB Export Data

mongoexport --host <host_name> --username <user_name> --password <password> --db <database_name> --collection <collection_name> --out <output_file>

--host is an optional parameter that specifies the remote server Mongo database instance.
--username and --password are the optional parameters that specify the authentication details of a user.
--db specifies the database name.
--collection specifies the collection name.
--out specifies the path of the output file. If this is not specified, the result is displayed on the console.

MongoDB Import Data

mongoimport --host <host_name> --username <user_name> --password <password> --db <database_name> --collection <collection_name> --file <input_file>

--host is an optional parameter that specifies the remote server Mongo database instance.
--username and --password are the optional parameters that specify the authentication details of a user.
--db specifies the database name.
--collection specifies the collection name.
--file specifies the path of the input file. If this is not specified, the standard input (i.e. stdin) is used

mongoimport --db warehouse --collection umongo --file <file_path>\warehouse_umongo_production_bkp_feb28.json --jsonArray

mongoDB