

Introduction to NoSQL Databases

NoSQL is a type of database management system (DBMS) that is designed to handle and store large volumes of unstructured and semi-structured data. Unlike traditional relational databases that use tables with pre-defined schemas to store data, NoSQL databases use flexible data models that can adapt to changes in data structures and are capable of scaling horizontally to handle growing amounts of data.

The term NoSQL originally referred to “non-SQL” or “non-relational” databases, but the term has since evolved to mean “not only SQL,” as NoSQL databases have expanded to include a wide range of different database architectures and data models.

NoSQL databases are generally classified into four main categories:

1. **Document databases:** These databases store data as semi-structured documents, such as JSON or XML, and can be queried using document-oriented query languages.
2. **Key-value stores:** These databases store data as key-value pairs, and are optimized for simple and fast read/write operations.
3. **Column-family stores:** These databases store data as column families, which are sets of columns that are treated as a single entity. They are optimized for fast and efficient querying of large amounts of data.
4. **Graph databases:** These databases store data as nodes and edges, and are designed to handle complex relationships between data.

Types of NoSQL database: Types of NoSQL databases and the name of the database system that falls in that category are:

1. **Graph Databases:** Examples – Amazon Neptune, Neo4j
2. **Key value store:** Examples – Memcached, Redis, Coherence
3. **Column:** Examples – Hbase, Big Table, Accumulo
4. **Document-based:** Examples – MongoDB, CouchDB, Cloudant

Importance of NoSQL Databases

1. **Scalability:** NoSQL databases are highly scalable, supporting horizontal scaling (by adding more servers) rather than vertical scaling (increasing power to an existing server). This makes them ideal for applications with rapid growth in data volume and user load.
2. **Flexible Schema:** Unlike relational databases, NoSQL databases don't require a fixed schema. This flexibility allows for dynamic modification of the data structure without the need for migration or downtime, which is advantageous in agile development and handling varied data sources.
3. **Performance for Big Data:** NoSQL databases are designed for handling large volumes of data, offering faster data retrieval and storage capabilities, especially for unstructured and semi-structured data types, which are common in big data applications.
4. **Handling Unstructured Data:** Many modern applications generate unstructured or semi-structured data (e.g., JSON, XML, multimedia files). NoSQL databases like document stores are optimized for storing and querying these types of data, making them a better fit than RDBMS for web and mobile applications.
5. **High Availability:** With distributed architectures and replication capabilities, NoSQL databases provide high availability and reliability. Data replication across multiple nodes ensures system uptime and fault tolerance, which is critical for mission-critical applications.

6. **Speed and Flexibility for Real-Time Applications:** NoSQL databases, particularly in-memory databases like Redis, offer extremely low latency for real-time applications like social media feeds, online gaming, and live analytics.

MongoDB Introduction:

MongoDB stands out as a leading **NoSQL** database, offering an open-source, document-oriented approach that diverges from traditional relational databases.

Unlike **SQL** databases, MongoDB stores data in BSON format, akin to JSON, allowing for more flexible data storage and retrieval. In this article, We will get a in **detailed** knowledge about **MongoDB**.

What is MongoDB?

- [MongoDB](#) the most popular [NoSQL](#) database, is an **open-source document-oriented** database. The term 'NoSQL' means '**non-relational**'.
- It means that MongoDB isn't based on the **table-like relational database** structure but provides an altogether different mechanism for the **storage** and **retrieval** of data. This format of storage is called [BSON](#) (similar to **JSON** format).

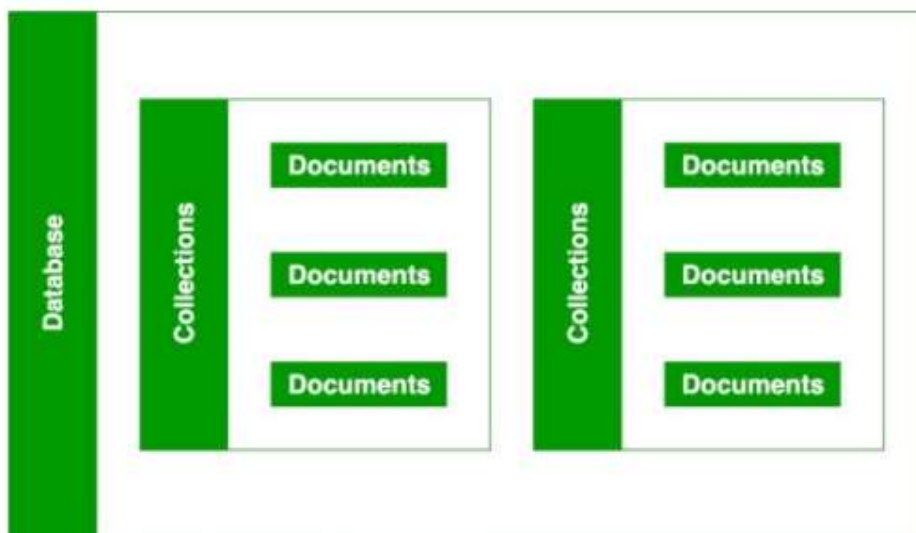
A simple MongoDB document Structure:

```
{
  title: 'Cbit',
  by: 'Harshit Gupta',
  url: 'https://cbit.ac.in',
  type: 'NoSQL'
}
```

- [SQL](#) databases store data in tabular format. This data is stored in a predefined data model which is not very much flexible for today's real-world highly growing applications.
- **Modern applications are more networked, social and interactive than ever.** Applications are storing more and more data and are accessing it at higher rates.
- [Relational Database Management System\(RDBMS\)](#) is not the correct choice when it comes to handling big data by the virtue of their design since they are not **horizontally scalable**. If the database runs on a single server, then it will reach a scaling limit.
- NoSQL databases are more scalable and provide superior performance. MongoDB is such a NoSQL database that scales by adding more and more servers and increases productivity with its flexible document model.

RDBMS	MongoDB
It is a relational database .	It is a non-relational and document-oriented database.
Not suitable for hierarchical data storage.	Suitable for hierarchical data storage .
It is vertically scalable i.e increasing RAM.	It is horizontally scalable i.e we can add more servers.
It has a predefined schema.	It has a dynamic schema.
It is quite vulnerable to SQL injection.	It is not affected by SQL injection .
It centers around ACID properties (Atomicity, Consistency, Isolation, and Durability).	It centers around the CAP theorem (Consistency, Availability, and Partition tolerance).
It is row-based.	It is document-based.
It is slower in comparison with MongoDB.	It is almost 100 times faster than RDBMS.
Supports complex joins.	No support for complex joins.
It is column-based.	It is field-based.
It does not provide JavaScript client for querying.	It provides a JavaScript client for querying.
It supports SQL query language only.	It supports JSON query language along with SQL .

Databases, collections, documents are important parts of MongoDB without them you are not able to store data on the MongoDB server. A Database contains a collection, and a collection contains documents and the documents contain data, they are related to each other.



Database

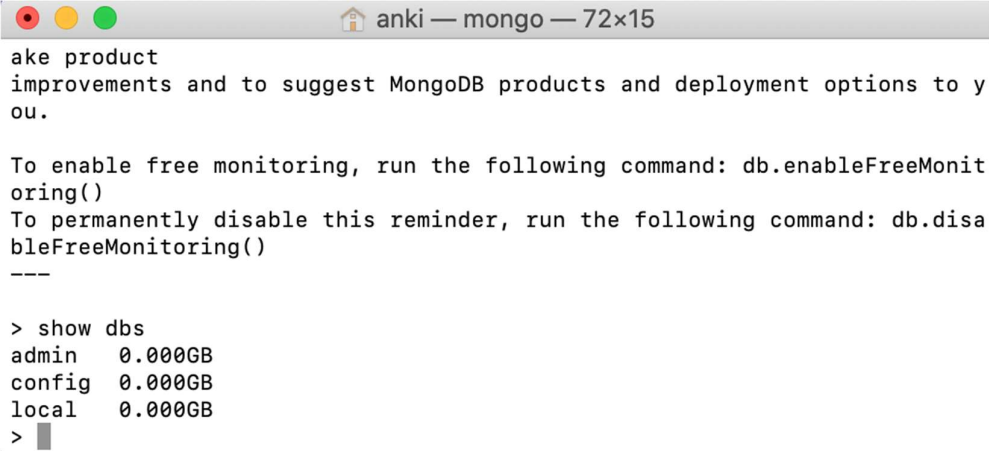
In MongoDB, a database contains the collections of documents. One can create multiple databases on the MongoDB server.

View Database:

To see how many databases are present in your MongoDB server, write the following statement in the mongo shell:

```
show dbs
```

For Example:



```
ake product
improvements and to suggest MongoDB products and deployment options to y
ou.

To enable free monitoring, run the following command: db.enableFreeMonit
oring()
To permanently disable this reminder, run the following command: db.disa
bleFreeMonitoring()
---
```

```
> show dbs
admin    0.000GB
config   0.000GB
local    0.000GB
>
```

Here, we freshly started MongoDB so we do not have a database except these three default databases, i.e, admin, config, and local.

Naming Restriction for Database:

Before creating a database you should first learn about the naming restrictions for databases:

- In MongoDB, the names of the database are case insensitive, but you must always remember that the database names cannot differ only by the case of the characters.
- For windows user, MongoDB database names cannot contain any of these following characters:

```
/\ . "$* : | ?
```

- For Unix and Linux users, MongoDB database names cannot contain any of these following characters:

```
/\ . "$
```

- MongoDB database names cannot contain null characters(in windows, Unix, and Linux systems).
- MongoDB database names cannot be empty and must contain less than 64 characters.

Creating Database:

In the mongo shell, you can create a database with the help of the following command:

```
use database_name
```

This command actually switches you to the new database if the given name does not exist and if the given name exists, then it will switch you to the existing database. Now at this stage, if you use the show command to see the database list where you will find that your new database is not present in that database list because, in MongoDB, the database is actually created when you start entering data in that database.

For Example:

```
anki — mongo — 72x20
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---

> show dbs
admin    0.000GB
config  0.000GB
local    0.000GB
[> use GeeksforGeeks
switched to db GeeksforGeeks
[> show dbs
admin    0.000GB
config  0.000GB
local    0.000GB
> █
```

Collection

Collections are just like tables in relational databases, they also store data, but in the form of documents. A single database is allowed to store multiple collections.

Schemaless:

As we know that MongoDB databases are schemaless. So, it is not necessary in a collection that the schema of one document is similar to another document. Or in other words, a single collection contains different types of documents like as shown in the below example where mystudentData collection contain two different types of documents:

```
anki — mongo — 89x17
[> db.mystudentData.find().pretty()]
{
  "_id" : ObjectId("5e37b67303ab1253cde7afe6"),
  "name" : "Sumit",
  "branch" : "CSE",
  "course" : "DSA",
  "amount" : 4999,
  "paid" : "Yes"
}
{
  "_id" : "geeks_for_geeks_201",
  "name" : "Rohit",
  "branch" : "ECE",
  "course" : "Sudo Gate",
  "year" : 2020
}
> █
```

Naming Restrictions for Collection:

Before creating a collection you should first learn about the naming restrictions for collections:

- Collection name must start with an underscore (`_`) or a letter (a-z or A-Z)
- Collection name should not start with a number, and does not contain \$, empty string, null character and does not begin with prefix `system.` as this is reserved for MongoDB system collections.
- The maximum length of the collection name is 120 bytes(including the database name, dot separator, and the collection name).


Creating collection:

After creating database now we create a collection to store documents. The collection is created using the following syntax:

```
db.collection_name.insertOne({..})
```

Here, insertOne() function is used to store single data in the specified collection. And in the curly braces {} we store our data or in other words, it is a document.

For Example:



```
[> use GeeksforGeeks
switched to db GeeksforGeeks
[> db.Author.insertOne({name: "Ankita"})
{
  "acknowledged" : true,
  "insertedId" : ObjectId("5e3799ff1993ad62dcde4f0d")
}
>
```

In this example, we create a collection named as the Author and we insert data in it with the help of insertOne() function. Or in other words, {name: "Ankita"} is a document in the Author collection, and in this document, the name is the key or field and "Ankita" is the value of this key or field. After pressing enter we got a message(as shown in the above image) and this message tells us that the data enters successfully (i.e., "acknowledge": true) and also assigns us an automatically created id. It is the special feature provided by MongoDB that every document provided a unique id and generally, this id is created automatically, but you are allowed to create your own id (must be unique).

Document

In MongoDB, the data records are stored as BSON documents. Here, BSON stands for binary representation of JSON documents, although BSON contains more data types as compared to JSON. The document is created using field-value pairs or key-value pairs and the value of the field can be of any BSON type.

Syntax:

```
{
field1: value1
field2: value2
...
fieldN: valueN
}
```

Naming restriction of fields:

Before moving further first you should learn about the naming restrictions for fields:

- The field names are of strings.
- The `_id` field name is reserved to use as a primary key. And the value of this field must be unique, immutable, and can be of any type other than an array.

- The field name cannot contain null characters.
- The top-level field names should not start with a dollar sign (\$).

JSON	BSON
JSON is javascript object notation.	BSON is Binary Javascript Object notation.
It is a standard file format type.	It is a binary file format type.
JSON contains some basic data types like string, numbers, Boolean, null.	BSON contains some additional data types like date, timestamp, etc.
Databases like AnyDB, redis, etc. stores the data into the JSON format.	The data in MongoDB is stored in a BSON format.
JSON requires less space as compared to BSON.	BSON requires more space as compared to JSON.
It is comparatively less faster than BSON.	It is faster as compared to BSON.
It is used for the transmission of data.	It is used for the storage of data.
It does not have encoding and decoding technique.	It has faster encoding and decoding technique.
If we want to read any particular information from the JSON file then it needs to go through whole content.	In BSON, indexes concept is used that skips all the content which are not in use.
JSON format does not require to be parsed as it is already human readable.	It requires to be parsed as it can be easily parsed by the machines.
JSON is a combination of objects and arrays where an object is a collection of key-value pairs while array is an ordered list of elements.	The binary encoding technique provides some additional information such as length of the string and object subtypes. BinData and Date are the additional data types supported by BSON over the JSON.

MongoDB Documents: Fields & Data Types

The **document**, which is comprised of field/value pairs, is at the heart of the MongoDB data structure. Most interactions with MongoDB occur at the document level.

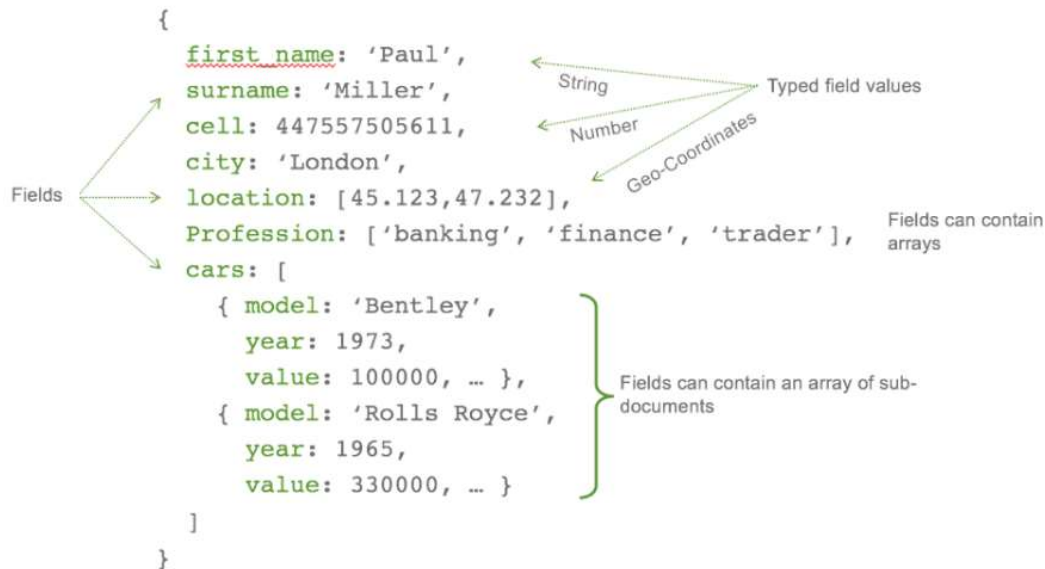
A **field** can contain a single value, multiple fields, or multiple elements.

```

db.users.insert (  ← collection
{
  name: "sue",      ← field: value
  age: 26,          ← field: value
  status: "A"       ← field: value
}
)                  } document

```


A value made up of multiple fields is referred to as an embedded document and is assigned the Object data type (see field cars in the screenshot). When rendered in the JSON format, an embedded document is enclosed in curly braces and adheres to the same structure as the outer (main) document.



A value made up of multiple elements is referred to as an **array** and is assigned the Array [data type](#) (see field Profession in the screenshot). When rendered in the JSON format, an array is enclosed in square brackets, with each element separated by a comma. An element can be a [scalar value](#) (like in the field Profession) or an [embedded document](#) (like in the field cars).

MongoDB Datatypes:

Remote procedure calls in MongoDB can be made by using the BSON format. MongoDB has a unique way of representing data types in which each data type is associated with an alias as well as a number that is usually used to search or find any specific record within a MongoDB database. MongoDB allows its users to implement different variations of data types:

Integer

Integer is a data type that is used for storing a numerical value, i.e., integers as you can save in other programming languages. 32 bit or 64-bit integers are supported, which depends on the server.

Example:

```
Copy Codedb.TestCollection.insert({"Integer example": 62})
```

Output:


```
> use my_project_db
switched to db my_project_db
> db.TestCollection.insert({"Integer example": 62})
WriteResult({ "nInserted" : 1 })
> db.TestCollection.find()
{ "_id" : ObjectId("5d08724bce0d9292a5121a78"), "Integer example" : 62 }
>
```

Boolean

Boolean is implemented for storing a Boolean (i.e., true or false) values.

Example:

```
db.TestCollection.insert({"Nationality Indian": true})
```

Output:

```
> db.TestCollection.insert({"Nationality Indian": true})
WriteResult({ "nInserted" : 1 })
> db.TestCollection.find()
{ "_id" : ObjectId("5d08724bce0d9292a5121a78"), "Integer example" : 62 }
{ "_id" : ObjectId("5d087412ce0d9292a5121a79"), "Nationality Indian" : true }
>
```

Double:

Double is implemented for storing floating-point data in MongoDB.

```
db.TestCollection.insert({"double data type": 3.1415})
```

Output:

```
> db.TestCollection.insert({"double data type": 3.1415})
WriteResult({ "nInserted" : 1 })
> db.TestCollection.find()
{ "_id" : ObjectId("5d08724bce0d9292a5121a78"), "Integer example" : 62 }
{ "_id" : ObjectId("5d087412ce0d9292a5121a79"), "Nationality Indian" : true }
{ "_id" : ObjectId("5d0874dcce0d9292a5121a7a"), "double data type" : 3.1415 }
```

Min/Max Keys:

Min / Max keys are implemented for comparing a value adjacent to the lowest as well as highest BSON elements.

String

String is one of the most frequently implemented data type for storing the data.

Example:

```
Copy Codedb.TestCollection.insert({"string data type" : "This is a sample message."})
```

Output:

```
> db.TestCollection.insert({"string data type" : "This is a sample message."})
WriteResult({"nInserted" : 1 })
> db.TestCollection.find()
{ "_id" : ObjectId("5d08724bce0d9292a5121a78"), "Integer example" : 62 }
{ "_id" : ObjectId("5d087412ce0d9292a5121a79"), "Nationality Indian" : true }
{ "_id" : ObjectId("5d0874dcce0d9292a5121a7a"), "double data type" : 3.1415 }
{ "_id" : ObjectId("5d087547ce0d9292a5121a7b"), "string data type" : "This is a sample message." }
```

Arrays:

Arrays are implemented for storing arrays or list type or several values under a single key.

```
var degrees = ["BCA", "BS", "MCA"]
```

```
db.TestCollection.insert({" Array Example" : " Here is an example of array",
" Qualification" : degrees})
```

Output:

```
> var degrees = ["BCA", "BS", "MCA"]
> db.TestCollection.insert({" Array Example" : " Here is an example of array",
" db.TestCollection.insert({" Array Example" : " Here is an example of array",
" Qualification" : degrees})
WriteResult({"nInserted" : 1 })
> db.TestCollection.find()
{ "_id" : ObjectId("5d0a6bc0940707e25658e62c"), "Integer example" : 62 }
{ "_id" : ObjectId("5d0a6c2e28323ade3e6287f9"), "Nationality Indian" : true }
{ "_id" : ObjectId("5d0a6c3528323ade3e6287fa"), "double data type" : 3.1415 }
{ "_id" : ObjectId("5d0a6c3d28323ade3e6287fb"), "string data type" : "This is a sample message." }
{ "_id" : ObjectId("5d0a6ee828323ade3e628800"), " Array Example" : " Here is an example of array", " Qualification" : [ "BCA", "BS", "MCA" ] }
```

Object:

Object is implemented for embedded documents.

```
var embeddedObject={"English" : 94, "ComputerSc." : 96, "Maths" : 80,
    "GeneralSc." : 85}

db.TestCollection.insert({"Object data type" : "This is Object",
    "Marks" : embeddedObject})
```

Output:

```
> var embeddedObject = {"English" : 94, "ComputerSc." : 96, "Maths" : 80, "GeneralSc." : 85}
> db.TestCollection.insert({"Object data type" : "This is Object", "Marks" : embeddedObject})
WriteResult({ "nInserted" : 1 })
> db.TestCollection.find()
{ "_id" : ObjectId("5d0a6bc0940707e25658e62c"), "Integer example" : 62 }
{ "_id" : ObjectId("5d0a6c2e28323ade3e6287f9"), "Nationality Indian" : true }
{ "_id" : ObjectId("5d0a6c3528323ade3e6287fa"), "double data type" : 3.1415 }
{ "_id" : ObjectId("5d0a6c3d28323ade3e6287fb"), "string data type" : "This is a sample message." }
{ "_id" : ObjectId("5d0a6ee828323ade3e628800"), "Array Example" : " Here is an example of array", "Qualification" : [ "BCA", "BS", "MCA" ] }
{ "_id" : ObjectId("5d0a715828323ade3e628801"), "Object data type" : "This is Object", "Marks" : { "English" : 94, "ComputerSc." : 96, "Maths" : 80, "GeneralSc." : 85 } }
```

Symbol:

Symbol is implemented to a string and is usually kept reticent for languages having specific symbol type.

Null:

Null is implemented for storing a Null value.

```
db.TestCollection.insert({"EmailID" : null})
```

Output:

```
> db.TestCollection.insert({"EmailID" : null})
WriteResult({ "nInserted" : 1 })
> db.TestCollection.find()
{ "_id" : ObjectId("5d0a731828323ade3e628803"), "EmailID" : null }
>
```

Date:

Date is implemented for storing the current date and time as UNIX-time format.

```
var date=new Date()

    var date2=ISODate()

    var month=date2.getMonth()

db.TestCollection.insert({"Date":date, "Date2":date2, "Month":month})
```

Output:

```
> var date=new Date()
> var date2=ISODate()
> var month=date2.getMonth()
> db.TestCollection.insert({"Date":date, "Date2":date2, "Month":month})
WriteResult({ "nInserted" : 1 })
> db.TestCollection.find()
{ "_id" : ObjectId("5d0a73fa28323ade3e628804"), "Date" : ISODate("2019-06-19T17:41:34.598Z"), "Date2" : ISODate("2019-06-19T17:41:45.071Z"), "Month" : 5 }
>
```

Timestamp:

Timestamp stores 64-bit value, in which the first 32 bits are time_t value (seconds epoch) and the other 32 bits are ordinal to operate within a given second.

Binary data:

Binary data is implemented for storing binary data.

Object ID:

Object ID is implemented for storing the ID of the document.

Regular expression:

Regular expression is implemented for storing regular expression.

Code:

Code is implemented for storing JavaScript code for your MongoDB document.

Data Modelling & Schema Design

Data modeling in MongoDB is the process of designing and creating the structure of collections and documents that will be stored in the database.

Maintaining data in an organized manner is very important for database efficiency. It also ensures data security, data accuracy, and better functioning. To maintain an organized database, it is important to learn data modeling.

Data modeling in MongoDB is the process of designing and creating the structure of collections and documents that will be stored in the database.

Maintaining data in an organized manner is very important for database efficiency. It also ensures data security, data accuracy, and better functioning. To maintain an organized database, it is important to learn data modeling.

What is Data Modeling in MongoDB?

MongoDB Data modeling is the process of **arranging unstructured data** from a real-world event into a logical data model in a database.

There is no need to build a schema before adding data to [MongoDB database](#) because MongoDB is flexible. This means that MongoDB supports a **dynamic database schema**.

To create a perfect Data model in MongoDB, always balance **application needs**, **database engine performance features**, and **data retrieval patterns**.

When creating data models, always account for both the application uses of the data, such as queries, updates, and data processing, as well as the fundamental design of the data itself.

MongoDB Data Model Designs

For modeling data in MongoDB, two strategies are available. These strategies are different and it is recommended to analyze scenario for a better flow.

The two methods for data model design in MongoDB are:

1. Embedded Data Model
2. Normalized Data Model

1. Embedded Data Model (Denormalization)

This method, also known as the **de-normalized** data model, allows you embed all of the **related documents in a single document**.

These nested documents are also called **sub-documents**.

Embedded Data Model example

If we obtain student information in three different documents, Personal_details, Contact, and Address, we can embed all three in a single one, as shown below.

```
Personal_details {  
  _id: ,  
  Std_ID: "987STD001"  
  Personal_details:{  
    First_Name: "Rashmika",  
    Last_Name: "Sharma",  
    Date_Of_Birth: "1999-08-26"  
  },  
  Contact: {  
    e-mail: "rashmika_sharma.123@gmail.com",  
    phone: "9987645673"  
  },  
  Address: {  
    city: "Karnataka",  
    Area: "BTM2ndStage",  
    State: "Bengaluru"  
  }  
}
```

```
}  
}
```

2. Normalized Data Model (normalization)

In a normalized data model, object references are used to express the **relationships between documents and data objects**. Because this approach **reduces data duplication**, it is relatively simple to document **many-to-many relationships** without having to repeat content.

Normalized data models are the most effective technique to model large **hierarchical data** with cross-collection relationships.

Normalized Data Model Example

Here we have created multiple collections for storing students data which are linked with `_id`.

Student:

```
{  
  _id: <StudentId101>,  
  Std_ID: "10025AE336"  
}
```

Personal_Details:

```
{  
  _id: <StudentId102>,  
  stdDocID: " StudentId101",  
  First_Name: "Rashmika",  
  Last_Name: "Sharma",  
  Date_Of_Birth: "1999-08-26"  
}
```

Contact:

```
{  
  _id: <StudentId103>,  
  stdDocID: " StudentId101",  
  e-mail: "rashmika_sharma.123@gmail.com",  
  phone: "9987645673"  
}
```

Address:

```
{  
  _id: <StudentId104>,  
  stdDocID: " StudentId101",  
  city: "Karnataka",  
  Area: "BTM2ndStage",  
  State: "Bengaluru"  
}
```

Advantages Of Data Modeling in MongoDB

Data modeling in MongoDB is essential for a successful application, even though at first it might just seem like one more step. In addition to increasing **overall efficiency** and **improving development cycles**, data modeling helps you better understand the data at hand and identify future business requirements, which can save time and money.

In particular, applying suitable data models:

- Improves application performance through better database strategy, design, and implementation.
- Allows faster application development by making object mapping easier.
- Helps with better data learning, standards, and validation.
- Allows organizations to assess long-term solutions and model data while solving not just current projects but also future application requirements, including maintenance.

For example:

let us take an example of a client who needs a database design for his website. His website has the following requirements:

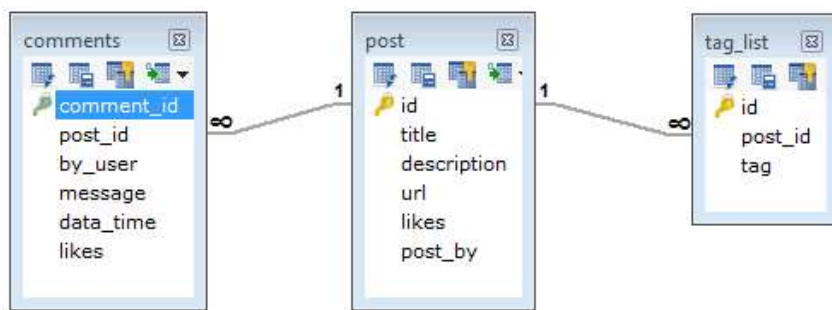
Every post is distinct (contains unique title, description and url).

Every post can have one or more tags.

Every post has the name of its publisher and total number of likes.

Each post can have zero or more comments and the comments must contain user name, message, data-time and likes.

For the above requirement, a minimum of three tables are required in RDBMS.



But in MongoDB, schema design will have one collection post and has the following structure

```
{
  _id: POST_ID
  title: TITLE_OF_POST,
  description: POST DESCRIPTION,
  by: POST_BY,
  url: URL_OF_POST,
  tags: [TAG1, TAG2, TAG3],
  likes: TOTAL LIKES,
  comments: [
    {
      user: 'COMMENT BY',
      message: TEXT,
      datecreated: DATE TIME,
      like: LIKES
    },
    {
      user: 'COMMENT BY',
      message: TEST,
      dateCreated: DATE TIME,
      like: LIKES
    }
  ]
}
```


Indexing:

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require MongoDB to process a large volume of data.

Indexes are special data structures, that store a small portion of the 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 as specified in the index.

The createIndex() Method

To create an index, you need to use createIndex() method of MongoDB.

Syntax

The basic syntax of **createIndex()** method is as follows().

```
>db.COLLECTION_NAME.createIndex({KEY:1})
```

Here key is the name of the field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

Example

```
>db.mycol.createIndex({"title":1})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
>
```

In **createIndex()** method you can pass multiple fields, to create index on multiple fields.

```
>db.mycol.createIndex({"title":1,"description":-1})
>
```

This method also accepts list of options (which are optional). Following is the list –

Parameter	Type	Description
background	Boolean	Builds the index in the background so that building an index does not block other database activities. Specify true to build in the background. The default value is false .
unique	Boolean	Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index. Specify true to create a unique index. The default value is false .
name	string	The name of the index. If unspecified, MongoDB generates an index name by concatenating the names of the indexed fields and the sort order.
sparse	Boolean	If true, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). The default value is false .
expireAfterSeconds	integer	Specifies a value, in seconds, as a TTL to control how long MongoDB retains documents in this collection.
weights	document	The weight is a number ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score.
default_language	string	For a text index, the language that determines the list of stop words and the rules for the stemmer and tokenizer. The default value is English .
language_override	string	For a text index, specify the name of the field in the document that contains, the language to override the default language. The default value is language.

The dropIndex() method

You can drop a particular index using the dropIndex() method of MongoDB.

Syntax

The basic syntax of DropIndex() method is as follows().

```
>db.COLLECTION_NAME.dropIndex({KEY:1})
```

Here, "key" is the name of the file on which you want to remove an existing index. Instead of the index specification document (above syntax), you can also specify the name of the index directly as:

```
dropIndex("name_of_the_index")
```

Example

```
> db.mycol.dropIndex({"title":1})
{
  "ok" : 0,
  "errmsg" : "can't find index with key: { title: 1.0 }",
  "code" : 27,
  "codeName" : "IndexNotFound"
}
```

The dropIndexes() method

This method deletes multiple (specified) indexes on a collection.

Syntax

The basic syntax of DropIndexes() method is as follows() –

```
>db.COLLECTION_NAME.dropIndexes()
```

Example

Assume we have created 2 indexes in the named mycol collection as shown below –

```
> db.mycol.createIndex({"title":1,"description":-1})
```

Following example removes the above created indexes of mycol –

```
>db.mycol.dropIndexes({"title":1,"description":-1})
{ "nIndexesWas" : 2, "ok" : 1 }
>
```

The getIndexes() method

This method returns the description of all the indexes int the collection.

Syntax

Following is the basic syntax od the getIndexes() method –

```
db.COLLECTION_NAME.getIndexes()
```

Example

Assume we have created 2 indexes in the named mycol collection as shown below –

```
> db.mycol.createIndex({"title":1,"description":-1})
```

Following example retrieves all the indexes in the collection mycol –

```

> db.mycol.getIndexes()
[
  {
    "v" : 2,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "test.mycol"
  },
  {
    "v" : 2,
    "key" : {
      "title" : 1,
      "description" : -1
    },
    "name" : "title_1_description_-1",
    "ns" : "test.mycol"
  }
]
>

```

Aggregation

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In SQL `count(*)` and `with group by` is an equivalent of MongoDB aggregation.

The aggregate() Method

For the aggregation in MongoDB, you should use **aggregate()** method.

Syntax

Basic syntax of **aggregate()** method is as follows –

```
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```

Example

In the collection you have the following data –

```

{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by_user: 'tutorials',
  url: 'http://www.tutorials.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100
},
{
  _id: ObjectId(7df78ad8902d)
  title: 'NoSQL Overview',
  description: 'No sql database is very fast',

```

```

    by_user: 'tutorials',
    url: 'http://www.tutorial.com',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 10
  },
  {
    _id: ObjectId(7df78ad8902e)
    title: 'Neo4j Overview',
    description: 'Neo4j is no sql database',
    by_user: 'Neo4j',
    url: 'http://www.neo4j.com',
    tags: ['neo4j', 'database', 'NoSQL'],
    likes: 750
  },

```

Now from the above collection, if you want to display a list stating how many tutorials are written by each user, then you will use the following **aggregate()** method –

```

> db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum :
1}}}}]
{ "_id" : "tutorials", "num_tutorial" : 2 }
{ "_id" : "Neo4j", "num_tutorial" : 1 }
>

```

Expression	Description	Example
\$sum	Sums up the defined value from all documents in the collection.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$sum : "\$likes"}}}}])
\$avg	Calculates the average of all given values from all documents in the collection.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$avg : "\$likes"}}}}])
\$min	Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$min : "\$likes"}}}}])
\$max	Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$max : "\$likes"}}}}])
\$push	Inserts the value to an array in the resulting document.	db.mycol.aggregate([{\$group : {_id : "\$by_user", url : {\$push: "\$url"}}}}])
\$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([{\$group : {_id : "\$by_user", url : {\$addToSet : "\$url"}}}}])
\$first	Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "\$sort"-stage.	db.mycol.aggregate([{\$group : {_id : "\$by_user", first_url : {\$first : "\$url"}}}}])
\$last	Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "\$sort"-stage.	db.mycol.aggregate([{\$group : {_id : "\$by_user", last_url : {\$last : "\$url"}}}}])

Pipeline Concept

In UNIX command, shell pipeline means the possibility to execute an operation on some input and use the output as the input for the next command and so on. MongoDB also supports same concept in aggregation framework. There is a set of possible stages and each of those is taken as a set of documents as an input and produces a resulting set of documents (or the final resulting JSON document at the end of the pipeline). This can then in turn be used for the next stage and so on.

- **\$project** – Used to select some specific fields from a collection.
- **\$match** – This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.
- **\$group** – This does the actual aggregation as discussed above.
- **\$sort** – Sorts the documents.
- **\$skip** – With this, it is possible to skip forward in the list of documents for a given amount of documents.
- **\$limit** – This limits the amount of documents to look at, by the given number starting from the current positions.
- **\$unwind** – This is used to unwind document that are using arrays. When using an array, the data is kind of pre-joined and this operation will be undone with this to have individual documents again. Thus with this stage we will increase the amount of documents for the next stage.

OR

1. Aggregation Pipeline

The aggregation pipeline consists of stages that apply sequential operations on a collection. Each stage takes documents as input, processes them, and passes the output to the next stage. Some of the common stages are:

- **\$match**: Filters documents to include only those that meet specific criteria, similar to the `find()` method.
- **\$group**: Groups documents by a specified field and performs aggregate functions like `$sum`, `$avg`, `$max`, `$min`, etc., on grouped documents.
- **\$project**: Reshapes documents, including or excluding fields, creating new fields, or reformatting fields.
- **\$sort**: Sorts documents by specified field(s).
- **\$limit**: Limits the number of documents to pass along the pipeline.
- **\$skip**: Skips a specified number of documents in the pipeline.
- **\$unwind**: Deconstructs an array field from documents into individual documents for each element of the array.
- **\$lookup**: Performs a left outer join to another collection to bring in additional data.
- **\$facet**: Performs multiple independent pipelines within a single stage and outputs multiple results.

2. Example Aggregation Queries

Basic Grouping with \$group

Here's a simple example that groups documents by a field and counts them:

```
db.myCollection.aggregate([
  { $group: { _id: "$fieldToGroupBy", count: { $sum: 1 } } }
])
```

Filtering and Grouping

Suppose you want to filter by a specific date range and then group by a field:

```
db.myCollection.aggregate([
  { $match: { dateField: { $gte: ISODate("2023-01-01"), $lte:
ISODate("2023-12-31") } } },
  { $group: { _id: "$category", totalSales: { $sum: "$sales" } } }
])
```

Projecting Specific Fields with \$project

To only show selected fields or create a new calculated field:

```
db.myCollection.aggregate([
  { $project: { item: 1, totalCost: { $multiply: ["$price", "$quantity"] } } }
])
```

Sorting Results with \$sort

To sort the output by a field (e.g., descending order of count):

```
db.myCollection.aggregate([
  { $group: { _id: "$by_user", num_tutorial: { $sum: 1 } } },
  { $sort: { num_tutorial: -1 } }
])
```

3. Example: Using Multiple Stages

A more complex example combining multiple stages:

```
db.orders.aggregate([
  { $match: { status: "completed" } }, // Filter completed orders
  { $group: { _id: "$customerId", totalSpent: { $sum: "$amount" } } }, //
Group by customer
  { $sort: { totalSpent: -1 } }, // Sort by total spent, descending
  { $limit: 5 } // Show top 5 customers
])
```

4. Aggregation Operators

Each stage supports various operators. Some of the most common ones are:

- **Arithmetic Operators:** \$add, \$subtract, \$multiply, \$divide
- **Array Operators:** \$push, \$addToSet, \$size, \$arrayElemAt

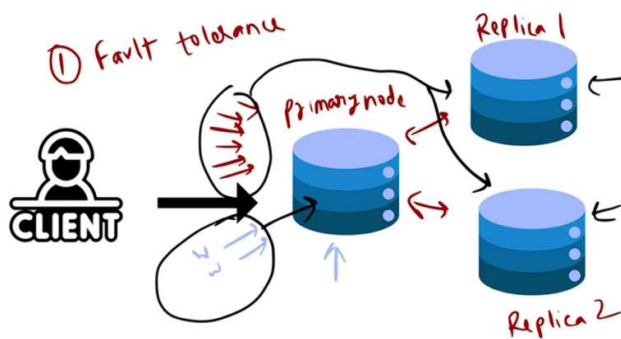
- **String Operators:** \$concat, \$substr, \$toUpper, \$toLower
- **Date Operators:** \$year, \$month, \$dayOfMonth, \$hour, \$minute, \$second

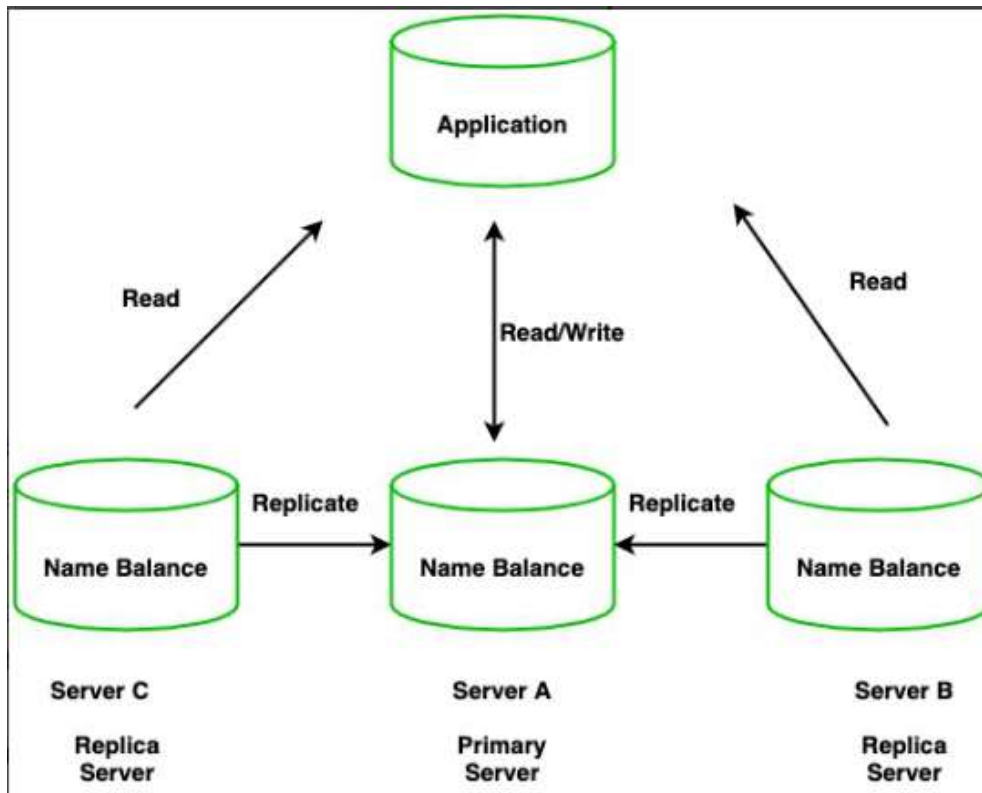
Replication:

Replication and **sharding** are two key features of MongoDB that enhance **data availability, redundancy**, and performance. Replication involves **duplicating data** across multiple servers by ensuring high availability and fault tolerance. On the other hand, **sharding** distributes **large datasets across several servers to manage large volumes of data** and handle high throughput operations.

What is Replication in MongoDB?

- **Replication** is the method of duplication of data across multiple servers in [MongoDB](#).
- **For example**, we have an application that reads and writes data to a database and says **server A** has a name and balance which will be **copied/replicated** to two other servers in two different locations.





Key Features of Replication

- Replica sets are the clusters of N different nodes that maintain the same copy of the data set.
- The primary server receives all write operations and record all the changes to the data i.e, oplog.
- The secondary members then copy and apply these changes in an asynchronous process.
- All the secondary nodes are connected with the primary nodes. there is one heartbeat signal from the primary nodes. If the primary server goes down an eligible secondary will hold the new primary.

Advantages of Replication

- High Availability of data disasters recovery
- No downtime for maintenance (like backups index rebuilds and compaction)
- Read Scaling (Extra copies to read from)

Configuring a Replica Set

To set up replication in MongoDB, you need to configure a replica set:

1. Initiate a Replica Set:

- Start each MongoDB instance with the `--replSet` option and a name for the replica set.

```
bash
mongod --replSet "rs0" --port 27017 --dbpath /data/db1
```

2. Connect to MongoDB and Initialize the Replica Set:

- Connect to the primary MongoDB instance using the `mongo` shell.

```
rs.initiate()
```

3. Add Members to the Replica Set:

- Add secondary nodes to the replica set.

```
rs.add("hostname2:27018") // Add a secondary  
rs.addArb("hostname3:27019") // Add an arbiter
```

5. Example: Checking Replica Set Status

To view the status of your replica set, use the `rs.status()` command:

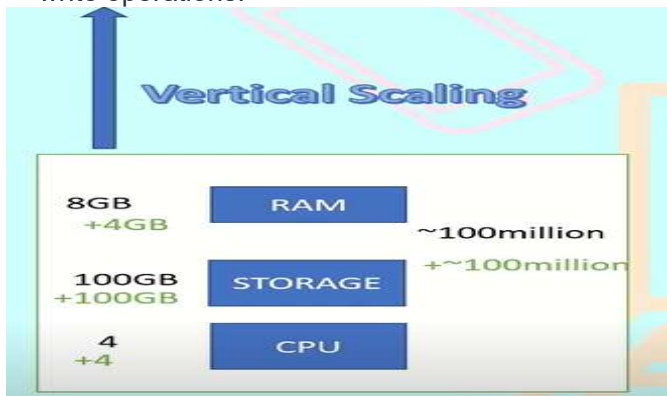
```
rs.status()
```

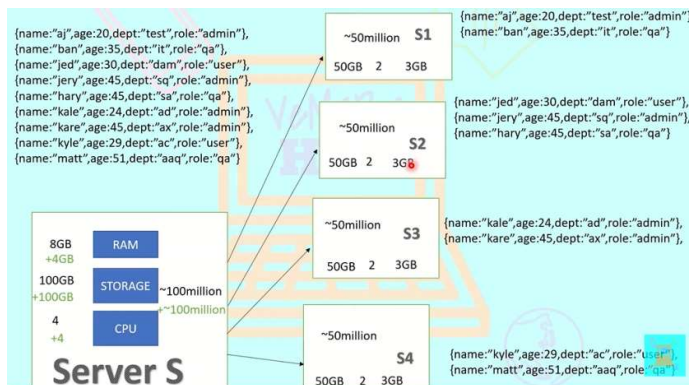
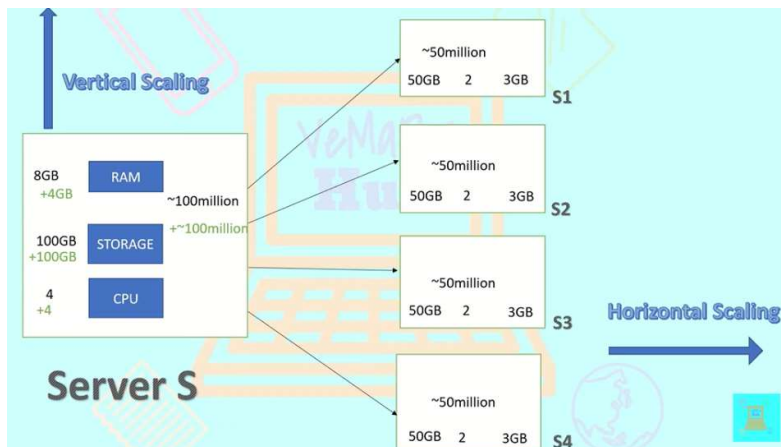
This command displays information about each node in the replica set, including their roles, statuses, and replication lag.

Sharding:

[Sharding](#) is a method for distributing large [collection](#) (dataset) and allocating it across multiple servers. MongoDB uses sharding to help deployment with very big data sets and high volume operations.

- Sharding combines more devices to carry data extension and the needs of read and write operations.





Chunks and Operations

1. Split – split key range into two key ranges when size of chunk is too big.
2. Migrate – To maintain balance migrate chunk from one shard to another.

Chunks and Operations

1. Split – split key range into two key ranges when size of chunk is too big.

Shard S1 Data:

- {name:"jed",age:30,dept:"dam",role:"user"},
- {name:"jery",age:45,dept:"sq",role:"admin"},
- {name:"john",age:45,dept:"sq",role:"admin"},
- {name:"hary",age:25,dept:"sa",role:"qa"},
- {name:"heff",age:34,dept:"sa",role:"qa"}

Shard S2 Data:

- {name:"jed",age:30,dept:"dam",role:"user"},
- {name:"jery",age:45,dept:"sq",role:"admin"},
- {name:"john",age:45,dept:"sq",role:"admin"},
- {name:"hary",age:25,dept:"sa",role:"qa"},
- {name:"heff",age:34,dept:"sa",role:"qa"}

2. Migrate – To maintain balance migrate chunk from one shard to another.

Shard S1 Data:

- {name:"jed",age:30,dept:"dam",role:"user"},
- {name:"jery",age:45,dept:"sq",role:"admin"},
- {name:"john",age:45,dept:"sq",role:"admin"},

Shard S2 Data:

- {name:"hary",age:25,dept:"sa",role:"qa"},
- {name:"heff",age:34,dept:"sa",role:"qa"}

Selection of Shard Key

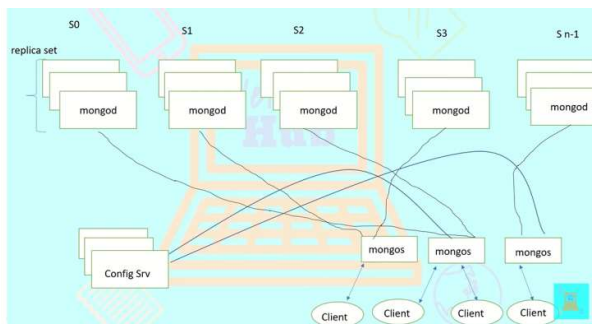
- The shard key is common in queries for the collection
- Good 'cardinality'/granularity
- Consider compound shardKeys
- Is the key monotonically increasing? – eg timestamp – BSON object ID do this.

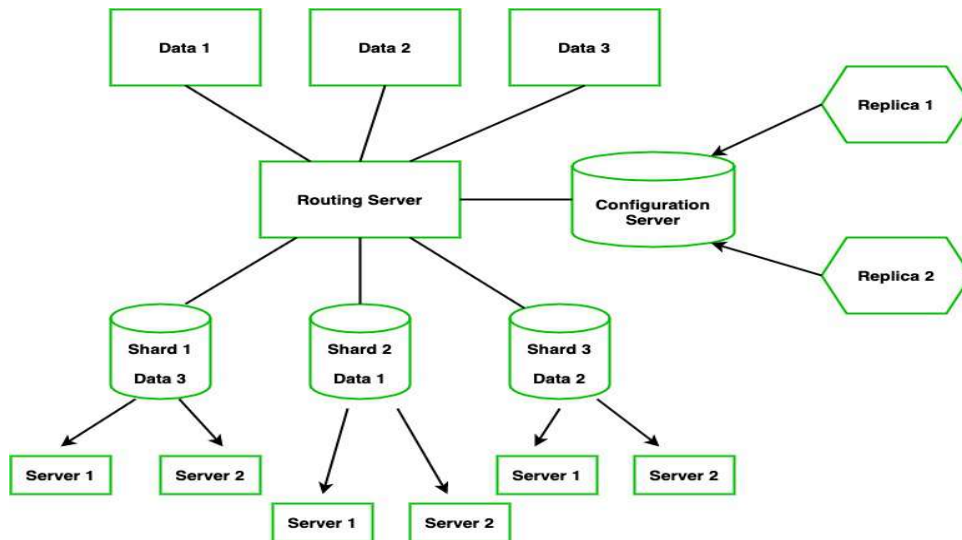
Shard Key Selection Example

- Either `_id`
- Or compound `{company:1,date:1}`

Sharding Process

- **config servers** – Small mongods where metadata of clusters are stored.
- **mongos** – client connects to view whole cluster as single entity.
- **mongod** – Process for data store .





Now, instead of one signal as primary, we have multiple servers called **Shard**. We have different routing servers that will route data to the shard servers.

For example: Let say we have **Data 1, Data 2, and Data 3** this will be going to the routing server which will route the data (i.e, Different Data will go to a particular Shard). Each Shard holds some pieces of data.

Here the **configuration server** will hold the metadata and it will configure the routing server to integrate the particular data to a shard however configure server is the MongoDB instance if it goes down then the entire server will go down, So it again has **Replica Configure database**.

Sharding in MongoDB is a method of horizontally scaling the database by distributing data across multiple servers, or "shards." This enables MongoDB to handle large datasets and high-throughput applications efficiently, ensuring high availability, load balancing, and performance even as the data grows.

Here's an overview of MongoDB sharding, its components, and best practices:

1. Why Use Sharding?

Sharding addresses the limitations of a single server by distributing data across multiple machines, which helps in:

- **Handling Large Datasets:** Sharding allows databases to scale out as data grows.
- **Improving Write and Read Scalability:** By spreading data across multiple shards, MongoDB can handle more read and write operations simultaneously.
- **Maintaining High Availability:** Sharded clusters allow for failover and redundancy, so the system can remain operational even if some nodes are unavailable.

2. Sharding Architecture Components

A sharded MongoDB cluster consists of several components:

- **Shards:** Each shard holds a subset of the data and can be a single MongoDB instance or a replica set (for data redundancy). Shards are responsible for storing data.
- **Config Servers:** These servers store the metadata and configuration information for the cluster, including details on how data is distributed across shards. A typical setup includes three config servers to ensure redundancy.
- **Mongos Routers:** These act as query routers, directing client requests to the appropriate shards based on data location and shard key. Applications connect to the mongos routers, which then interact with the shards.

3. Choosing a Shard Key

The shard key is a critical part of the sharding setup. It determines how MongoDB will distribute data across the shards. A shard key can be a single field or a combination of fields, and it should be carefully chosen based on data access patterns.

Characteristics of a Good Shard Key:

- **High Cardinality:** A shard key with many unique values helps distribute data evenly.
- **Even Distribution:** Avoid "hot spots" by ensuring that data can be evenly spread across shards.¹¹
- **Query Isolation:** Choose a shard key that supports common query patterns so that queries target only specific shards rather than querying all shards.

Common shard key types include:

- **Ranged Sharding:** Distributes documents based on a range of shard key values. It's suitable when data access patterns align with ranges, but it can lead to uneven data distribution if data inserts or updates are concentrated in certain ranges.
- **Hashed Sharding:** Uses a hashed value of the shard key, which helps distribute data more evenly but may impact range-based queries.

4. Setting Up a Sharded Cluster

Here's a high-level overview of setting up sharding in MongoDB:

1. Start Config Servers:

- Launch config servers for metadata storage. In a production setup, use three config servers for redundancy.

```
bash
mongod --configsvr --replSet configReplSet --port 27019 --dbpath
/data/config1
```

2. Initiate Config Server Replica Set:

- Connect to one of the config servers and initiate a replica set.

```
javascript
rs.initiate({
  _id: "configReplSet",
  configsvr: true,
```



```

members: [
  { _id: 0, host: "hostname1:27019" },
  { _id: 1, host: "hostname2:27019" },
  { _id: 2, host: "hostname3:27019" }
]
}))

```

3. Start Shards:

- Start each shard as either a standalone server or a replica set.

bash

```

mongod --shardsvr --replSet shardReplSet --port 27018 --dbpath
/data/shard1

```

4. Initiate Shard Replica Sets (if using replica sets for shards):

javascript

```

rs.initiate({
  _id: "shardReplSet",
  members: [
    { _id: 0, host: "hostname1:27018" },
    { _id: 1, host: "hostname2:27018" }
  ]
})

```

5. Start Mongos Instances:

- Start one or more mongos instances to act as query routers.

bash

```

mongos --configdb
configReplSet/hostname1:27019,hostname2:27019,hostname3:27019 --port
27017

```

6. Add Shards to the Cluster:

- Connect to a mongos instance and add each shard to the cluster.

```

sh.addShard("shardReplSet/hostname1:27018,hostname2:27018")

```

7. Enable Sharding on a Database and Collection:

- Enable sharding for the target database and specify the shard key for collections.

```

sh.enableSharding("myDatabase")
sh.shardCollection("myDatabase.myCollection", { shardKey: 1 })

```

5. Monitoring and Managing Sharded Clusters

- **Sharding Status:** Use the `sh.status()` command to check the status of the sharded cluster, including shard distribution.
- **Balancing:** MongoDB includes a balancer process that redistributes data across shards as data volume changes to maintain an even distribution. You can enable or disable the balancer as needed.

- **Chunk Splitting:** MongoDB automatically splits large data ranges, called "chunks," to keep them manageable. This prevents any single chunk from becoming too large.

Link: <https://www.mongodb.com/try/download/shell>

1) Type **mongosh** for getting current version and connecting to mongodb server.

2) For displaying existing databases

```
test> show dbs
DemoDb    160.00 KiB
admin     40.00 KiB
config    108.00 KiB
local     72.00 KiB
student   72.00 KiB
```

3) For Creating new Database

```
test> use aids
switched to db aids
```

4) For Creating Collection

```
aids> db.createCollection("fifthsem")
```

5) For displaying number of collection

```
aids> show collections
fifthsem
```

6) **Insert Document**

In MongoDB, the **db.collection.insert()** method is used to add or insert new documents into a collection in your database.

Syntax

```
>db.COLLECTION_NAME.insert(document)
```

```
aids> db.fifthsem.insert(
...   {
...     course: "java",
...     details: {
...       duration: "6 months",
...       Trainer: "Sonoo jaiswal"
...     },
...     Batch: [ { size: "Small", qty: 15 }, { size: "Medium", qty: 25 } ],
...     category: "Programming language"
...   }
... )
```

7) **Check the inserted documents**

```

aids> db.fifthsem.find()
[
  {
    _id: ObjectId('66c77d038f80a83e292710bc'),
    course: 'java',
    details: { duration: '6 months', Trainer: 'Sonoo jaiswal' },
    Batch: [ { size: 'Small', qty: 15 }, { size: 'Medium', qty: 25 } ],
    category: 'Programming language'
  }
]

```

8) MongoDB insert multiple documents

If you want to insert multiple documents in a collection, you have to pass an array of documents to the `db.collection.insert()` method.

Create an array of documents for inserting multiple elements

```

aids> var Allcourses =
...  [
...    {
...      Course: "SE",
...      details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
...      Batch: [ { size: "Medium", qty: 25 } ],
...      category: "Programming Language"
...    },
...    {
...      Course: "EAD",
...      details: { Duration: "6 months", Trainer: "Prashant Verma" },
...      Batch: [ { size: "Small", qty: 5 }, { size: "Medium", qty: 10 } ],
...      category: "Programming Language"
...    },
...    {
...      Course: "CN",
...      details: { Duration: "3 months", Trainer: "Rashmi Desai" },
...      Batch: [ { size: "Small", qty: 5 }, { size: "Large", qty: 10 } ],
...      category: "Programming Language"
...    }
...  ]

```

```
...   ];
```

Inserts the documents

Pass this Allcourses array to the db.collection.insert() method to perform a bulk insert.

```
aids> db.fifthsem.insert( Allcourses );
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('66c77f698f80a83e292710bd'),
    '1': ObjectId('66c77f698f80a83e292710be'),
    '2': ObjectId('66c77f698f80a83e292710bf')
  }
}
```

8) for checking inserted data

```
aids> db.fifthsem.find()
```

```
aids> db.fifthsem.find()
[
  {
    _id: ObjectId('66c77d038f80a83e292710bc'),
    course: 'java',
    details: { duration: '6 months', Trainer: 'Sonoo jaiswal' },
    Batch: [ { size: 'Small', qty: 15 }, { size: 'Medium', qty: 25 } ],
    category: 'Programming Language'
  },
  {
    _id: ObjectId('66c77f698f80a83e292710bd'),
    Course: 'SE',
    details: { Duration: '6 months', Trainer: 'Sonoo Jaiswal' },
    Batch: [ { size: 'Medium', qty: 25 } ],
    category: 'Programming Language'
  },
  {
    _id: ObjectId('66c77f698f80a83e292710be'),
    Course: 'EAD',
    details: { Duration: '6 months', Trainer: 'Prashant Verma' },
    Batch: [ { size: 'Small', qty: 5 }, { size: 'Medium', qty: 10 } ],
    category: 'Programming Language'
  },
  {
    _id: ObjectId('66c77f698f80a83e292710bf'),
    Course: 'CN',
    details: { Duration: '3 months', Trainer: 'Rashmi Desai' },
    Batch: [ { size: 'Small', qty: 5 }, { size: 'Large', qty: 10 } ],
    category: 'Programming Language'
  }
]
```

MongoDB update documents

In MongoDB, update() method is used to update or modify the existing documents of a collection.

Syntax:

```
db.COLLECTION_NAME.update(SELECTIOIN_CRITERIA, UPDATED_DATA)
```

Example

Consider an example which has a collection name fifthsem. Insert the following documents in collection:

```
db.fifthsem.insert(
{
  course: "java",
  details: {
    duration: "6 months",
    Trainer: "Sonoo jaiswal"
  },
  Batch: [ { size: "Small", qty: 15 }, { size: "Medium", qty: 25 } ],
  category: "Programming language"
}
)
```

Update the existing course "java" into "android":

```
>db.fifthsem.update({'course':'java'},{$set: {'course':'android'}})
```

```
>db.fifthsem.find()
```

Delete documents

In MongoDB, the db.collection.remove() method is used to delete documents from a collection. The remove() method works on two parameters.

1. Deletion criteria: With the use of its syntax you can remove the documents from the collection.
2. JustOne: It removes only one document when set to true or 1.

Syntax: db.collection_name.remove (DELETION_CRITERIA)

Remove all documents

Syntax:

```
db.fifthsem.remove({})
```

Remove all documents that match a condition

If you want to remove a document that match a specific condition, call the `remove()` method with the `<query>` parameter.

The following example will remove all documents from the `fifthsem` collection where the `type` field is equal to `programming language`.

```
db.fifthsem.remove( { type : "programming language" } )
```

Remove a single document that match a condition

If you want to remove a single document that match a specific condition, call the `remove()` method with `justOne` parameter set to `true` or `1`.

The following example will remove a single document from the `fifthsem` collection where the `type` field is equal to `programming language`.

```
db.fifthsem.remove( { type : "programming language" } )
```

Remove a single document that match a condition

If you want to remove a single document that match a specific condition, call the `remove()` method with `justOne` parameter set to `true` or `1`.

The following example will remove a single document from the `fifthsem` collection where the `type` field is equal to `programming language`.

```
db.fifthsem.remove( { type : "programming language" }, 1 )
```

Drop collection

The `db.collection.drop()` method does not take any argument and produce an error when it is called with an argument. This method removes all the indexes associated with the dropped collection.

Syntax: `db.collection.drop()`

Now check the collections in the database:

```
>show collections
```