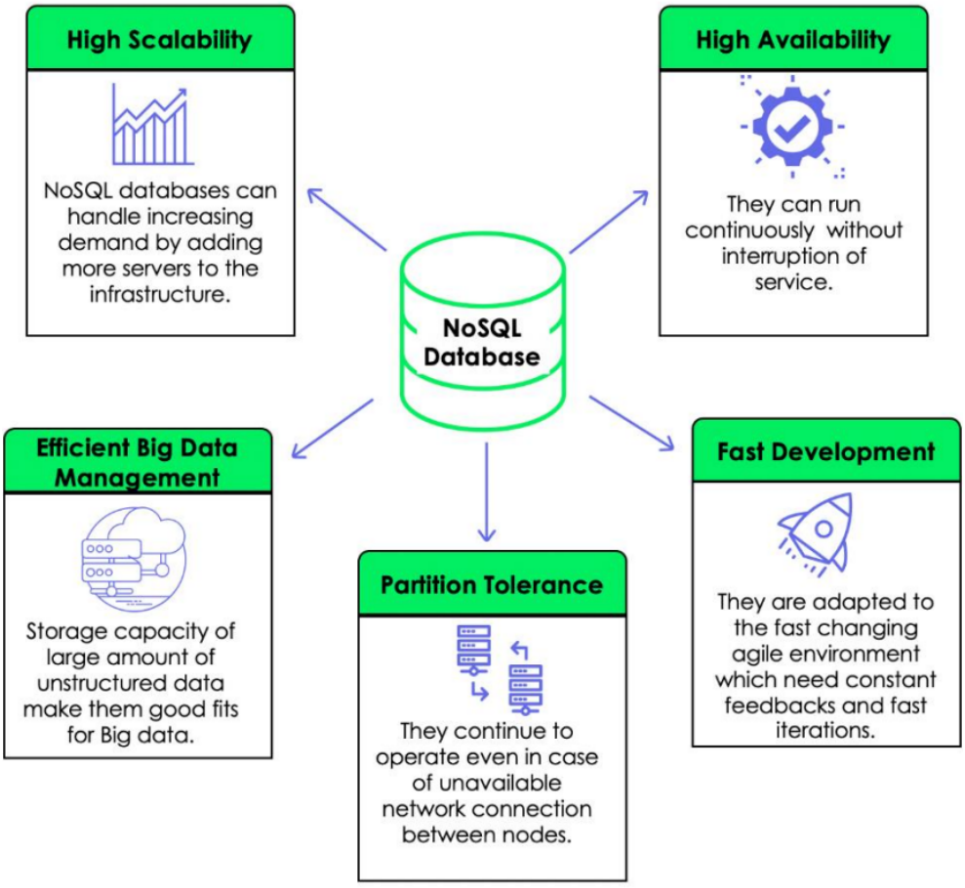
Second Unit

NOSQL Storage Architecture



NoSQL (Not Only SQL) is a type of database that stores and retrieves data using **non-relational mechanisms.** NoSQL databases are often used for **large-scale data** storage and **high-velocity data processing**.

Common characteristics of NoSQL storage architecture:

1. **Schema-less data model**
2. **Distributed and scalable**
3. **Different data models**: NoSQL databases use a variety of data models, such as document, key-value, graph, and column-family models.
4. **High performance**
5. **Eventual consistency**

6. **No support for joins**: NoSQL databases do not support joins, which are used in relational databases to combine data from multiple tables.

NOSQL Basics with MongoDB

1. **Document-Oriented**: MongoDB is a document-oriented database, which means that data is stored as JSON-style documents. Each document can have its own fields, including nested sub-documents.
   MongoDB is a document-oriented database, which means that it stores data as documents in a collection instead of using traditional relational tables and rows. A document is a data structure that contains key-value pairs, where the keys represent the field names and the values represent the field values. Documents can have different structures, as long as they belong to the same collection.
   a. **Flexible Data Modeling**
   b. **Scalability**
   c. **Performance**
   d. **Easy to use**
   e. **Rich Data Types**: MongoDB supports rich data types such as dates, arrays, and nested documents, allowing you to store and manipulate complex data structures.
2. **Dynamic Schema**: MongoDB uses a dynamic schema, which means that you can add or remove fields to a document without having to change the schema of the entire collection. This makes it easy to adapt to changes in the data structure over time.
   a. **Flexibility**
   b. **Simplicity**
   c. **Improved performance**
   d. **Supports Polymorphic Data**: MongoDB's dynamic schema allows you to store documents with different structures in the same collection. This is particularly useful when working with data that is inherently polymorphic, such as user profiles or products, where each document may have a different set of fields.
   e. **Reduced Overhead**: With dynamic schema, you don't have to spend time defining and maintaining a schema, freeing you up to focus on other

important tasks.

3. **Collections:** In MongoDB, collections are equivalent to tables in a traditional relational database. Each collection can have many documents, and each document can have many fields.
   Collections are a way to organize and store data. A collection is a grouping of related documents, which can have different structures, but must belong to the same database.
   a. **Dynamic schema**
   b. **Auto-sharding**: Collections in MongoDB can be automatically sharded, meaning they can be divided and distributed across multiple servers in a cluster, improving performance and scalability.
   c. **Indexing**: MongoDB supports creating indexes on collections to improve query performance. You can create indexes on specific fields in documents, allowing for faster querying of the data.
   d. **Scalability**
   e. **Easy to use**
4. **Indexing:** MongoDB supports indexing on any field in a document, which makes it fast to search and retrieve data. You can create compound indexes to improve the performance of complex queries.
   Indexing is a way to optimize the performance of queries in MongoDB. An index is a data structure that provides a mapping between the values in a collection and their location on disk, allowing for faster querying of the data.
   a. **Improved query performance**
   b. **Supports multiple types of queries**: MongoDB supports multiple types of indexes, including single field indexes, compound indexes, text indexes, geospatial indexes, and more. This allows you to create indexes that are optimized for different types of queries, improving performance and scalability.
   c. **Flexibility**
   d. **Saves disk space**
   e. **Supports scalability**
5. **Replication**: MongoDB supports replication, which allows you to have multiple copies of the same data on different servers for increased reliability and

availability.
Replication in MongoDB is a way to ensure the high availability and data durability of your applications. Replication involves creating multiple copies of the data in a MongoDB deployment and distributing them across multiple servers.

a. **High availability**
b. **Data durability**
c. **Load balancing**: Replication can also be used for load balancing, by distributing the read operations across multiple replicas. This can help to improve the performance and scalability of your applications.
d. **Flexible topologies**: MongoDB supports multiple replication topologies, including replica sets and sharded clusters, allowing you to choose the configuration that best fits your needs.
e. **Easy to manage**

6. **Sharding**: MongoDB supports sharding, which allows you to horizontally partition data across multiple servers to improve performance and scale horizontally.
Sharding in MongoDB refers to the process of distributing data across multiple servers, to improve the scalability and performance of a MongoDB deployment.

a. **Horizontal scalability**
b. **Improved performance**
c. **Easy management**
d. **Supports multiple topologies**
e. **Supports different sharding strategies**: MongoDB supports different sharding strategies, including range-based sharding and hash-based sharding, allowing you to choose the strategy that best fits your data and query patterns.

CRUD

```
import pymongo

# Connect to the MongoDB server
```

```
client = pymongo.MongoClient("mongodb://localhost:27017/")

# Get the database and collection
db = client["testdb"]
collection = db["testcollection"]

# Create a document to insert
my_document ={"name":"Sachin Tendulkar", "age": 35, "city":  "Mumbai"}

# Insert the document into the collection
collection.insert_one(my_document)

# Insert many documents into the collection
collection.insert_many([
    {"name": "Sachin Tendulkar", "age": 35, "city": "Mumbai"},
    {"name": "Virat Kohli", "age": 32, "city": "Delhi"},
    {"name": "MS Dhoni", "age": 39, "city": "Ranchi"}
])

# Verify that the documents have been inserted
results = collection.find()

# Alternative find usage
results = collection.find({"age": {"$gte": 35}})

# Find documents that match a complex condition
results = collection.find({
    "$and": [
        {"age": {"$gte": 35}},
        {"$or": [
            {"city": "Mumbai"},
            {"city": "Delhi"}
        ]}
    ]
})

# Display found results
for result in results:
    print(result)
```

Create

To insert data into a collection in MongoDB, you can use the **insertOne()** or **insertMany()** method. The insertOne() method inserts a single document into the collection, while the insertMany() method inserts an array of documents.

insertOne()

`db.collection.insertOne({data})`

```
db.employees.insertOne({
 firstName: "John",
 lastName: "King",
```

```
 email: "john.king@abc.com"
})
```

insertMany()

`db.collection.insertMany([{document1}, {document2}, …, {documentn}])`

```
db.employees.insertMany([
  {
   firstName: "John",
   lastName: "King",
   email: "john.king@abc.com"
  },
  {
   firstName: "Sachin",
   lastName: "T",
   email: "sachin.t@abc.com"
  },
  {
   firstName: "James",
   lastName: "Bond",
   email: "jamesb@abc.com"
  }
])
```

Read

To retrieve data from a collection in MongoDB, you can use the **find()** method. The find() method returns a cursor to the documents in the collection that match the specified query criteria. You can also use the **findOne()** method which returns the first document that is matched with the specified criteria.

findOne()

`db.collection.findOne(query, projection)`

Parameters:

query: Optional. Specifies the criteria using query operators.

projection: Optional. Specifies the fields to include in a resulting document.

```
db.employees.findOne({salary: {$gt: 8000}})
```

In the above example, the query operator criteria for the salary field is written inside another document as {field: {operator: value}}. The {salary: {$gt: 8000}} criteria returns the first document where salary is greater than 8000.

find()

```
db.employees.find({salary: {$gt: 7000, $lt: 8000}})
```

Projection

Use the projection parameter to specify **the fields to be included in the result.** The projection parameter format is {<field>: <1 or true>, <field>: <1 or true>...} where 1 or true **includes** the field, and o or false **excludes** the field in the result.

```
db.employees.find({salary: 7000}, {firstName:1, lastName:1})
```

Operations

Operation	Syntax	Example
Equality	{<key>:{$eg:<value>}}	db.mycol.find({”by”:”my_point”}).pretty()
Less than	{<key>:{$lt:<value>}}	db.mycol.find({”likes”:{$lt:50}}).pretty()
Less than equals	{<key>:{$lte:<value>}}	db.mycol.find({”likes”:{$lte:50}}).pretty()
Greater than	{<key>:{$gt:<value>}}	db.mycol.find({”likes”:{$gt:50}}).pretty()
Greater than equals	{<key>:{$gte:<value>}}	db.mycol.find({”likes”:{$gte:50}}).pretty()
Not equals	{<key>:{$ne:<value>}}	db.mycol.find({”likes”:{$ne:50}}).pretty()
Values in an array	{<key>:{$in:[<value1>, …, <valuen>]}}	db.mycol.find({”name”:{$in:[”Raj”, ”Ram”, ”Raghu”]}})
Values not in array	{<key>:{$nin:[<value1>, …, <valuen>]}}	db.mycol.find({”name”:{$nin:[”Raj”, ”Ram”, ”Raghu”]}})

AND, OR, NOR, NOT

>db.mycol.find({"likes": {$gt:10}, $or: [{"by": " RDBMS Book’ "}, {"title": "MongoDB Overview"}]})

>**db.users.find({$and: [{age: {$gte: 25}}, {status: "active"}]})**

>db.empDetails.find({ $nor:[{"First_Name": "Radhika"},{"Last_Name": "Christopher"}]})

>db.empDetails.find({"Age": { $not: {$gt: "25"}}})

Update

To update documents in a collection in MongoDB, you can use the **updateOne()** or **updateMany()** method. The updateOne() method updates a single document that matches the specified query criteria, while the updateMany() method updates all documents that match the specified query criteria.

updateMany()

`db.COLLECTION_NAME.update(<filter>, <update>)`

```
db.empDetails.updateMany({Age:{ $gt: "25" }}, { $set: { Age: '00'}})
```

updateOne()

`db.COLLECTION_NAME.update(<filter>, <update>)`

```
db.collection.update_one({"name": "John Doe"}, {"$set": {"age": 36}})
```

save()

The save() method replaces the existing document with the new document passed in the save() method.

`db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})`

```
db.mycol.save({"_id" : ObjectId("507f191e810c19729de860ea"), "title": "NoSQL Introduct
ion", "by": "MongoDB Introduction "})
```

findOneAndUpdate()

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

```
db.empDetails.findOneAndUpdate(
{First_Name: 'Radhika'}, {$set: { Age: '30',e_mail: 'radhika_newemail@gmail.com'}})
```

Delete

To delete documents from a collection in MongoDB, you can use the **deleteOne()** or **deleteMany()** method. The deleteOne() method deletes a single document that matches the specified query criteria, while the deleteMany() method deletes all documents that match the specified query criteria.

deleteOne()

```
collection.delete_one({"name": "John Doe"})
```

deleteMany()

```
collection.delete_many({"age": {"$lt": 36}})
```

Limit()

`db.COLLECTION_NAME.find().limit(NUMBER)`

```
db.mycol.find({},{"title":1,_id:0}).limit(2)
```

Skip()

`db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)`

```
db.mycol.find({},{"title":1,_id:0}).limit(1).skip(1)
```

Above example will display only the second document.

Aggregation

Aggregation in MongoDB refers to the **process of processing data records and returning computed results.** It is used to perform operations on multiple documents and return the processed data in a single result set. The aggregation framework in MongoDB provides several operations that can be used to **process and manipulate data, including filtering, grouping, transforming, and reducing data.**

$match

$group

$project

$sort

$limit

Index

An index in MongoDB is **a data structure that provides a fast and efficient** way to query and retrieve data from a collection. It works by organizing the data in a specific order, so that MongoDB can locate the required data quickly and efficiently. Indexes in MongoDB can be created on any field or combination of fields in a collection, and they can be used to support various types of queries, such as equality queries, range queries, and text searches.

dropIndex()

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

getIndexes()

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