**NoSQL - 08.01.2025**

**RDMS:**

1. **Referential Integrity**
2. **ACID**
   * **Atomicity**
   * **Consistency**
   * **Isolation**
   * **Durability**

**Disadvantages of RDMS:**

* **Cannot store Unstructured Data. To rectify that, we use NoSQL.**

**NoSQL:**

1. **Used to handle both Unstructured and Structured Data**
2. **Non-relational SQL**

**5 Main Variants of NoSQL:**

1. **Key-Value Pair**
   * **Dynamo**
   * **Redis (In-memory key-value store)**
   * **Amazon SimpleDB**
2. **Column (Aggregation Queries)**
   * **Cassandra**
   * **HBase**
   * **Hypertable**
3. **Document**
   * **MongoDB**
   * **CouchDB**
4. **Graph**
   * **Neo4J**
   * **InfiniteGraph**
5. **Multi-Model**
   * **OrientDB**
   * **ArangoDB**

**Key Features of NoSQL Database:**

1. **Multiple Data Model Compatibility**
2. **Enhanced Scalability and Availability**
3. **Global Data Distribution**

**Sharding:**

* **Splitting many datasets into many partitions is known as Sharding in NoSQL.**

**Vertical Scaling:**

* **Helps to scale up the processor and RAM.**

**Horizontal Scaling:**

* **Helps to scale up the servers.**
* **Does not support NoSQL.**

**CAP Theorem (Brewer’s Theorem):**

1. **C - Consistency**
   * **Every read operation will return the most recent write (or an error) for a given piece of data, ensuring all nodes have the same data at any point in time.**
2. **A - Availability**
   * **Every request (read or write) will receive a response, even if some of the nodes are unavailable. The system remains operational but may not guarantee consistency.**
3. **P - Partition Tolerance**
   * **The system will continue to function even if network partitions occur, meaning the system can tolerate the failure of network communication between nodes.**

**Only support CP in NoSQL.**

**What is MongoDB?**

**MongoDB** is a popular, open-source **NoSQL** database that uses a **document-oriented** data model. It is designed to store and manage **unstructured** or **semi-structured data** efficiently, which is different from traditional **relational databases** that use tables and rows to store data.

**Why we use MongoDB?**

1. **MongoDB** for a variety of reasons, especially when there is a need for **flexibility**, **scalability**, and the ability to handle complex data.
2. **Document Oriented Storage** – Data is stored in the form of JSON Style documents
3. **Fast in-place updates** (cannot see old values when you update)

**Collection** is a group of MongoDB Documents.it is the equivalent of table in RDBMS

**Documents** is a set of Key-value store Documents have dynamic schema Documents within the same collection may have different schema

**MongoDB \_id Field Structure**

The \_id field in MongoDB is a 12-byte **hexadecimal** number, which ensures the uniqueness of every document. The structure of this 12-byte value is as follows:

1. **First 4 bytes**: Represent the **current timestamp** (in seconds).
2. **Next 3 bytes**: Represent the **machine ID** (unique to the machine running MongoDB).
3. **Next 2 bytes**: Represent the **process ID** of the MongoDB server.
4. **Remaining 3 bytes**: Represent a simple **incremental value** (auto-incremented counter to ensure uniqueness within the same second).

You can also provide the \_id value while inserting a document manually if needed.

1. **show dbs**: Lists all databases in the MongoDB instance.
2. **use test**: Switches to the test database or creates it if it does not exist.
3. **db.authors.insertOne({name:"abc"})**: Inserts a document with the name abc into the authors collection in the test database.
4. The **db.createCollection('Authors')** command is used to **explicitly create a new collection** named **Authors** in the current database
5. **show collections**: Lists all collections in the current database.
6. **db.authors.drop()**: Drops (removes) the authors collection from the current database.
7. The **db.dropDatabase()** command is used to **delete the User defined database**. It removes the **database** along with all its **collections** and **documents**. In the case of **db.dropDatabase('test'),** it will drop the database named **test**.
8. **db.authors.find()** command is used to **query** and **retrieve** documents from the **authors collection** in the current MongoDB database.
9. **db.Stats.deleteMany({})** This command will **remove all documents** in the Stats collection, as the empty object {} means no filtering criteria.
10. **db.Stats.replaceOne({\_id:ObjectId('677e43ac037760e70f7fd6b9')},{country:"Nigeria",continent:"Africa",population:13.463})**
11. **db.Stats.find({name:'Delhi'})** This is a query that retrieves documents from the Stats collection where the name field is equal to 'Delhi'.
12. **The db.Stats.updateOne({ name: 'Delhi' }, { $set: { population: 28.000 } })** command updates the first document in the Stats collection where name is 'Delhi', setting its population field to 28.000. If the population field does not exist, it will be added.
13. **db.Stats.deleteOne({ name: 'Tokyo' })** deletes the **first document** in the Stats collection where the name field is 'Tokyo'.
14. The **command db.Stats.insertOne({\_id:1, Name:'Sethu'})** inserts a **single document** with \_id: 1 and Name: 'Sethu' into the Stats collection.
15. The command **db.Stats.find({\_id:1},{\_id:0,Name:1})** retrieves the **document** with \_id: 1 from the Stats collection, **excluding the \_id field** and **including only the Name field**.

**Operators**

**Comparison:**

* 1. **$eq: Matches values that are equal to the specified value.**
  2. **$ne: Matches values that are not equal to the specified value.**
  3. **$gt: Matches values that are greater than the specified value.**
  4. **$gte: Matches values that are greater than or equal to the specified value.**
  5. **$lt: Matches values that are less than the specified value.**
  6. **$lte: Matches values that are less than or equal to the specified value.**
  7. **$in: Matches values that exist in the specified array.**
  8. **$nin : Matches values that do not exist in the specified array.**
  9. **$exists: Matches documents that have the specified field (can be true or false).**
  10. **$type: Matches values of the specified BSON data type**
* The command **db.Stats.find({population: {$gt: 25}})** retrieves all documents from the Stats collection where the population field is greater than 25.
* The command **db.Stats.find({name:/^S|^M/})** retrieves all documents from the Stats collection where the name field starts with either "S" or "M".
* The command **db.Stats.find({name:/s$/})** retrieves all documents from the Stats collection where the name field ends with the letter "s".

**Logical Operator:**

* The command **db.Stats.find({$and:[{continent:"Asia"},{population: {$gt: 20}}]})** retrieves all documents from the Stats collection where the continent is "Asia" and the population is greater than 20.

**09.01.2025**

**Aggregation Stages:**

**$match**: Filters documents based on specified conditions to pass only those that meet the criteria.

**$group**: Groups documents by specified fields and performs aggregation operations on them.

**$sort**: Sorts the documents based on specified fields in either ascending or descending order.

**Example:**

Pipeline is an array

Pipeline=[{$match:{..}}, {$group:{..}}, {$sort:{..}}]

**Example Code to Pratice:**

db.bookOrders.insertMany ( [

{ \_id: 0, first\_name: "Fyodor", last\_name: "Dostoyevsky", book\_title: 'Demons', genre: 'Fiction', quantity: 10, date: ISODate( "2022-10-21T11:19:30Z" ) },

{ \_id: 1, first\_name: "Fyodor", last\_name: "Dostoyevsky", book\_title: 'Brothers Karamosov', genre: 'Fiction', quantity: 25, date: ISODate( "2022-10-21T11:19:30Z" ) },

{ \_id: 2, first\_name: "Jacques", last\_name: "Derrida", book\_title: 'The Politics of Friendship', genre: 'Fiction', quantity: 5, date: ISODate( "2022-10-21T11:19:30Z" ) },

{ \_id: 3, first\_name: "Charles", last\_name: "Dickens", book\_title: 'Tale of Two Cities', genre: 'Fiction', quantity: 6, date: ISODate( "2022-10-21T11:19:30Z" ) },

{ \_id: 4, first\_name: "James", last\_name: "Joyce", book\_title: 'Ulysses', genre: 'Fiction', quantity: 30, date: ISODate( "2021-03-13T11:19:30Z" ) },

{ \_id: 5, first\_name: "Henry David", last\_name: "Thoreau", book\_title: 'Walden', genre: 'Nonfiction', quantity: 15, date: ISODate( "2021-03-13T11:19:30Z" ) },

{ \_id: 6, first\_name: "Virginia", last\_name: "Woolf", book\_title: "A Room of One's Own", genre: 'Nonfiction',

quantity: 18, date: ISODate( "2022-10-21T11:19:30Z" ) },

{ \_id: 7, first\_name: "Virginia", last\_name: "Woolf", book\_title: "Mr's Dalloway", genre: 'Fiction', quantity: 14, date: ISODate( "2022-10-21T11:19:30Z" ) },

{ \_id: 8, first\_name: "Zadie", last\_name: "Smith", book\_title: 'White Teeth', genre: 'Fiction', quantity: 8, date: ISODate( "2022-10-21T11:19:30Z" ) },

{ \_id: 9, first\_name: "Charles", last\_name: "Dickens", book\_title: 'The Old Curiousity Shop', genre: 'Fiction', quantity: 6, date: ISODate( "2022-10-21T11:19:30Z" ) }

] )

**Commands:**

* **db.bookOrder.aggregate([ { $match: { genre: "Fiction" } } ])**

The command to filter documents with the genre "Fiction" in the bookOrder collection using MongoDB's aggregation framework

* **db.bookOrders.aggregate([{$group: {\_id: "$last\_name",totalQuantity: {$sum: "$quantity"}} }])**

The command groups the documents in the `bookOrders` collection by `last\_name` and calculates the total sum of the `quantity` for each `last\_name`.

* **db.bookOrders.aggregate([{$match:{genre:"Fiction"}},{$count:"first\_name"}])**

The command filters documents in the bookOrders collection where the genre is "Fiction" and then counts the number of such documents, returning the result with the field first\_name representing the count.

* **db.bookOrders.aggregate([{$group: {\_id: "$genre", totalQuantity: { $sum: "$quantity"},averageQuantity: {$avg: "$quantity" },count: { $sum: 1 }}}])**

**Project operator**

* **db.bookOrders.find({},{first\_name:1,last\_name:1}).limit(2)**
* **db.bookOrders.aggregate([{$match: {genre: "Fiction"}},{$project: {last\_name: 1, quantity: 1}}])**

**Aggregate using OR Gate and match**

* **db.bookOrders.aggregate([ { $match: { $or: [ { genre: "Fiction" }, { genre:"Nonfiction" } ] } }])**

The **$lookup** operator is used to perform **left outer joins** between two collections. This allows you to combine documents from one collection with documents from another collection based on a related field, similar to how SQL joins work.

Example Code for Lookup:

orders[ {

"\_id": ObjectId("60f9d7ac345b7c9df348a86e"),

"order\_number": "ORD001",

"customer\_id": ObjectId("60f9d7ac345b7c9df348a86d"),

},

{

"\_id": ObjectId("60f9d7ac345b7c9df348a86f"),

"order\_number": "ORD002",

"customer\_id": ObjectId("60f9d7ac345b7c9df348a86d"),

},

{

"\_id": ObjectId("60f9d7ac345b7c9df348a870"),

"order\_number": "ORD003",

"customer\_id": ObjectId("60f9d7ac345b7c9df348a86e"),

},

{

"\_id": ObjectId("60f9d7ac345b7c9df348a871"),

"order\_number": "ORD004",

"customer\_id": ObjectId("60f9d7ac345b7c9df348a86f"),

},

{

"\_id": ObjectId("60f9d7ac345b7c9df348a872"),

"order\_number": "ORD005",

"customer\_id": ObjectId("60f9d7ac345b7c9df348a86f"),

}

]

**cutomers**

[

{

"\_id": ObjectId("60f9d7ac345b7c9df348a86d"),

"name": "John Doe",

"email": "john@example.com"

},

{

"\_id": ObjectId("60f9d7ac345b7c9df348a86e"),

"name": "Alice Smith",

"email": "alice@example.com"

},

{

"\_id": ObjectId("60f9d7ac345b7c9df348a86f"),

"name": "Bob Johnson",

"email": "bob@example.com"

}

]

**db.orders.aggregate([{**

**$lookup: {from: "customers",**

**localField: "customer\_id",**

**foreignField: "\_id",**

**as: "customer\_details” } }])**

* $lookup joins orders with customers based on customer\_id and \_id.
* The result adds customer\_details to each order document, containing matched customer data.