Module-3

Introduction to Mongo DB

MongoDB

MongoDB is a **document-oriented NoSQL database** designed for storing, querying, and managing large volumes of unstructured or semi-structured data.
Instead of storing data in rows and columns like traditional relational databases (RDBMS), MongoDB stores data in **JSON-like documents** called **BSON** (Binary JSON).

Key Features:

- Schema-less (fields can vary from document to document).
- Horizontal scaling through **sharding**.
- Rich query language for filtering, aggregation, and indexing.
- High availability through **replication**.

6.1 WHAT IS MONGODB?

MongoDB is:

- Cross-platform.
- Open source.
- Non-relational.
- Distributed.
- NoSQL.
- Document-oriented data store.

6.2 WHY MONGODB?

Traditional RDBMS face challenges with:

- Handling large volumes of data.
- Managing a rich variety of data, especially unstructured data.
- Scaling to meet enterprise needs.
- MongoDB addresses these by:
- Scaling out or horizontally.
- Offering schema flexibility.
- Being fault-tolerant.

- Maintaining consistency and partition tolerance.
- Allowing easy distribution over many nodes in a cluster.

6.2.1 Using JavaScript Object Notation (JSON)

- JSON is highly expressive for data representation.
- MongoDB doesn't directly use JSON; it uses **BSON** (Binary JSON).
- BSON is an open standard designed to store complex data structures.

Advantages of MongoDB (Figure 6.1)

1. **Auto Sharding** – Distributes data across multiple servers automatically for scalability.
2. **Document Oriented** – Stores data in flexible, JSON-like documents.
3. **High Performance** – Optimized for read and write operations.
4. **Fast In-place Updates** – Allows efficient updates without rewriting entire documents.
5. **Replication** – Ensures data redundancy and high availability.
6. **High Availability** – Continuous data access even during failures.
7. **Easy Scalability** – Simple to expand storage and processing capacity.
8. **Rich Query Language** – Supports expressive queries and aggregations.
9. **Full Index Support** – Indexing capabilities for faster data retrieval.

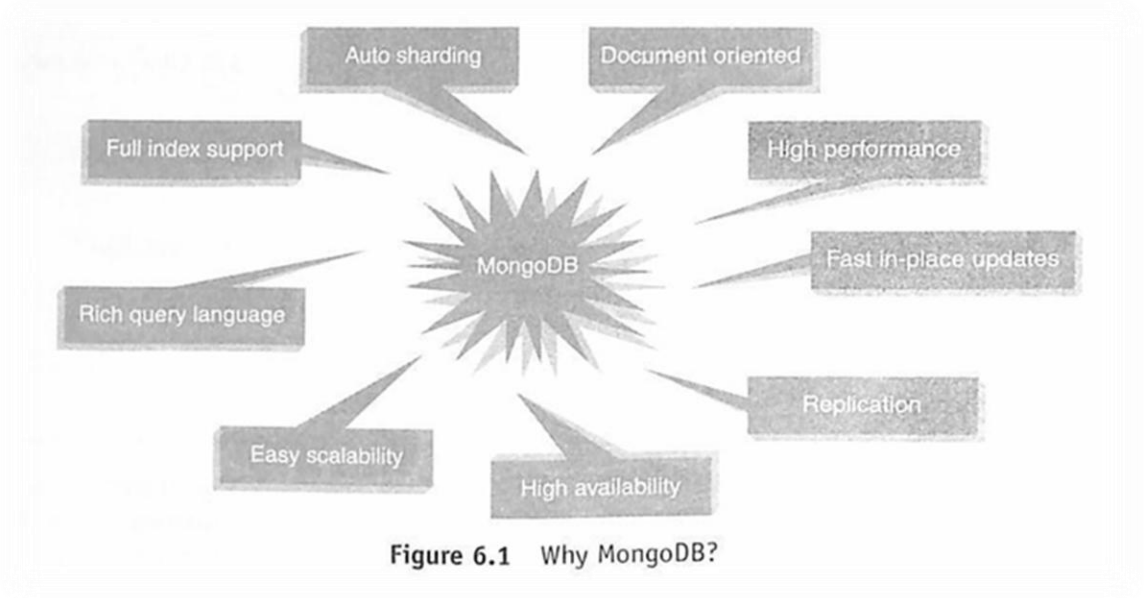


Figure 6.1 Why MongoDB?

Example: How employee contact data can be represented in different formats — CSV, XML, and JSON — and why JSON (and BSON) can be advantageous.

1. CSV Format

- **Structure:** Simple, flat table where each column is separated by commas, and each row is separated by a carriage return.
- **Problem:**
    - Can't easily store multiple values for a single field (e.g., multiple contact numbers or email addresses per employee).
    - Different departments might store data differently, making merging files messy.
    - Limited in handling complex, hierarchical, or repeating data.

2. XML Format

- **Advantage:** Highly extensible and good at representing structured and hierarchical data.
- **Drawback:** More complex to set up and work with, especially for simple data exchange.
- **Use Case:** Best for complex and highly structured datasets, but overkill for simple ones.

3. JSON Format

- **Advantage:**
    - Readable and easy to store multiple values in arrays (e.g., multiple contact numbers).
    - Flexible structure without the rigidity of CSV and without XML's verbosity.
    - No confusion in representing multiple fields of the same type.
- **Example:**

```
{
 "FirstName": "John",
 "LastName": "Mathews",
 "ContactNo": ["+123 4567 8900", "+123 4444 5555"]
}
```

4. BSON

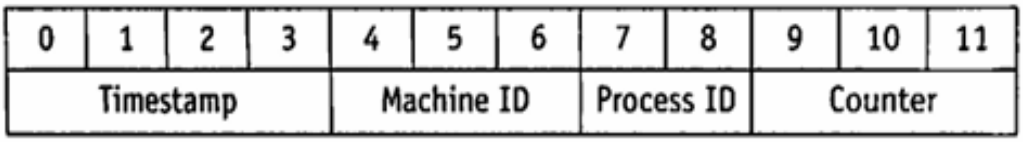
- **Definition:** Binary form of JSON, used in systems like MongoDB.
- **Benefits:**
  - Consumes less space than text-based JSON.
  - Faster to parse into native data structures.
  - Supported by many programming languages through MongoDB drivers.
  - Allows quick and efficient storage/retrieval of documents in a database.

Summary:

- **CSV** → Simple but flat, no repeating fields.
- **XML** → Powerful and extensible but verbose and complex.
- **JSON** → Flexible, readable, good for most use cases.
- **BSON** → Optimized binary JSON, great for database performance.

6.2.2 Creating or Generating a Unique Key

- Every JSON document should have a **unique identifier**: the _id key.
- _id is similar to a **primary key** in relational databases.
- Benefits:
  - Facilitates quick document searches.
  - Automatically indexed by MongoDB.
- You can either:
  - Provide the _id value yourself.
  - Let MongoDB auto-generate it.



- **Auto-generated _id** is a **12-byte value** composed of:

1. **Timestamp** (first 4 bytes)
2. **Machine ID** (next 3 bytes)
3. **Process ID** (next 2 bytes)
4. **Counter** (last 3 bytes)

6.2.2.1 Database

- A **collection of collections** (like a container for them).
- Created when a collection is first referenced or on demand.
- Each database has its **own files** on the file system.
- One MongoDB server can store **multiple databases**.

6.2.2.2 Collection

- Analogy: **Table** in RDBMS.
- Created when a document is saved to it.
- Belongs to a single database.
- Holds multiple documents.
- **No schema enforcement**:
    - Documents in the same collection can have **different fields**.
    - Even with same fields, the **order of fields can vary**.

6.2.2.3 Document

- Analogy: **Row/Record/Tuple** in RDBMS.
- Has a **dynamic schema**:
    - Fields can differ from document to document within a collection.
    - Key-value pairs don't need to be uniform across the collection.

6.2.3 Support for Dynamic Queries

- MongoDB supports **dynamic queries**, similar to traditional RDBMS where both **static** and **dynamic** queries are possible.

- **Dynamic queries**: The ability to query data using conditions and filters at runtime without predefined structures.
- **CouchDB**:
  - Another document-oriented, schema-less NoSQL database.
  - Takes the opposite approach: supports **dynamic data** but **static queries**.

Figure 6.2 Example

- Shows a collection named **students** containing 3 different documents.

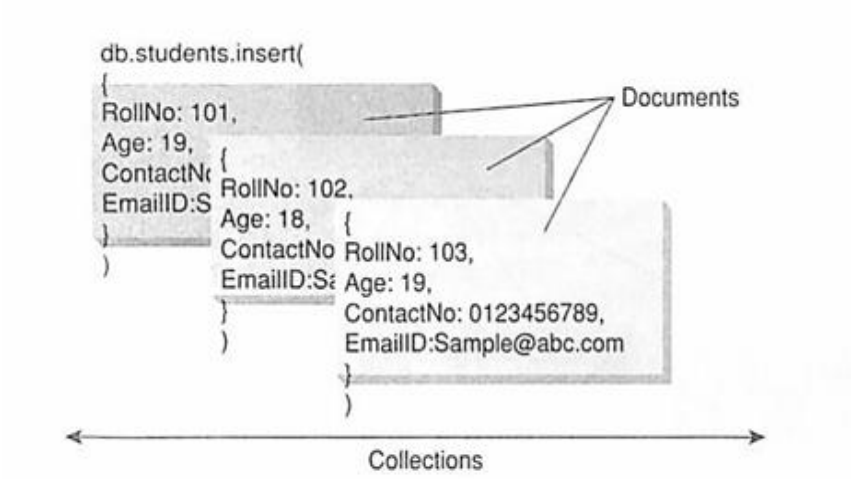


Figure 6.2 A collection "students" containing 3 documents.

Key Points from the Example

- Documents in the same collection **don't need identical fields** or values.
- Some documents may omit certain fields entirely.
- Field ordering may differ.
- The flexibility of MongoDB's schema-less design supports a variety of document structures while still allowing dynamic, powerful queries.

6.2.4 Storing Binary Data

- MongoDB supports binary data storage through **GridFS**.
- **Default limit**: Up to **4 MB** for a single piece of binary data.
  - Suitable for small media files, e.g.:
    - Profile pictures

    - Small audio clips
- For larger files (e.g., movie clips), MongoDB uses a **chunk-based storage method**.

How GridFS Works

1. **Metadata storage**:
    - Metadata (data about data) is stored in a collection named **file**.
    - Metadata contains:
        - File name
        - File size
        - Upload date
        - Content type (e.g., image/jpeg, audio/mpeg)
2. **Chunking process**:
    - File is split into small pieces called **chunks**.
    - Chunks are stored in a separate **chunks** collection.
3. **Scalability**:
    - Splitting into chunks makes it easier to store and retrieve large files efficiently.
    - Supports streaming of files without loading the entire file into memory.

6.2.5 Replication

Why replication?

- Protects against:
    - **Hardware failures**
    - **Service interruptions**
- Ensures **data redundancy** and **high availability**.

How it works in MongoDB

- Uses a **replica set**:
    - **One primary** node.
    - **Multiple secondary** nodes.
- **Write requests**:
    - Always sent to the **primary**.
    - Primary logs write operations into the **Oplog** (operations log).
- **Read requests**:
    - By default, directed to the **primary**.
    - Can be configured (read preference) to read from secondaries.

Oplog Role

- Secondary nodes **synchronize** their data with the primary by replicating entries from the Oplog.
- This ensures **consistency** across the replica set.

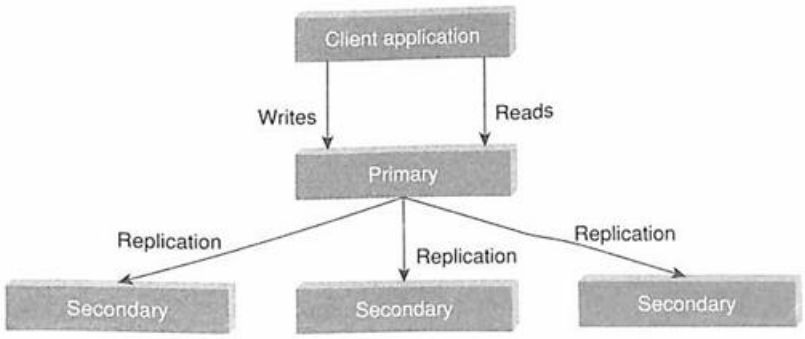


Figure 6.3 The process of **REPLICATION** in MongoDB.

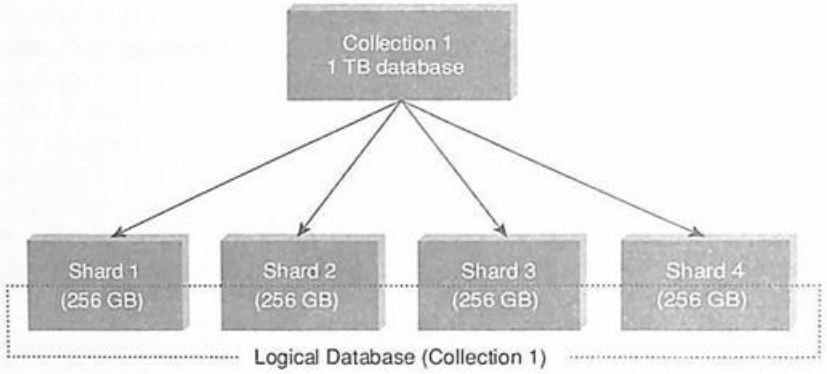


Figure 6.4 The process of **SHARDING** in MongoDB.

Figure 6.3 – Replication

- **Goal:** Data redundancy and high availability.
- **How it works:**
  1. The **primary** node handles all writes.
  2. **Secondary** nodes replicate data from the primary using the **Oplog**.
  3. Reads normally go to the primary but can be configured to read from secondaries.
- **Benefit:** Prevents data loss in case of node failure.

Figure 6.4 – Sharding

- **Goal:** Horizontal scaling of large datasets.
- **How it works:**
    1. A large collection (e.g., **1 TB database**) is split into smaller, manageable pieces called **shards**.
    2. Example: 1 TB is divided into **4 shards of 256 GB each**.
    3. Together, these shards represent the **logical database**.
- **Benefit:** Distributes data across multiple servers, improving performance and storage capacity.

Key Difference

- **Replication:** Copies the *same data* to multiple servers for **redundancy**.
- **Sharding:** Splits *different portions* of data across servers for **scalability**.

6.2.7 Updating Information In-Place

 Meaning: MongoDB updates the data directly where it is stored — no new space is allocated, and indexes remain unchanged.

 Lazy writes:

- MongoDB writes to disk about once per second (not immediately).
- This improves performance by reducing the number of disk operations.

 Trade-off:

- Faster performance but **no guarantee** that the most recent data is safely stored if a crash occurs before the next write to disk.

 Performance benefit:

- Fewer disk reads/writes → faster operations (since reading/writing to memory is much faster than disk I/O).

6.3 Terms Used in RDBMS and MongoDB

RDBMS Term	MongoDB Term	Notes
Database	Database	Same term in both
Table	Collection	A collection stores multiple documents
Record	Document	Each document is like a row in a table
Columns	Fields / Key-Value pairs	Fields hold the actual data inside documents
Index	Index	Works similarly in both
Joins	Embedded documents	MongoDB stores related data inside the same document
Primary Key	Primary key (_id is identifier)	Every document has a unique _id field

Database Server & Client Examples

Feature	MySQL	Oracle	MongoDB
Database Server	mysqld	oracle	mongod
Database Client	mysql	SQL Plus	mongo

6.3.1 Create Database

Syntax:

use DATABASE_Name

Example:

use myDB

Output:

switched to db myDB

- To check the current database:

db

Output:

myDB

- To list all databases:

show dbs

Example output:

admin (empty)

local 0.078GB

test 0.078GB

Note:

- A newly created database won't appear in show dbs until it has at least **one document**.
- Default database in MongoDB is **test**.
-

6.3.2 Drop Database

Syntax:

db.dropDatabase();

Steps to drop myDB:

use myDB

db.dropDatabase();

Example output:

{ "dropped" : "myDB", "ok" : 1 }

Note:

- If no database is selected, dropping will remove the default test database.

6.4 Data Types in MongoDB

Data Type	Description
String	Must be UTF-8 valid. Most commonly used type.
Integer	32-bit or 64-bit depending on the server.
Boolean	True/false values.
Double	Floating-point (real) values.
Min/Max keys	Compare a value against lowest/highest BSON elements.
Arrays	Store arrays or multiple values in one key.
Timestamp	Record when a document was modified or added.
Null	NULL value (missing/unknown).
Date	Current date/time in Unix format. Can store date objects.
Object ID	Document's unique ID.
Binary data	Binary data like images, binaries, etc.
Code	Store JavaScript code in the document.
Regular expression	Store regular expressions.

Useful MongoDB Commands

1. **Report current database:**

db

Example output:

test

2. **Display list of databases:**

show dbs

Example output:

admin (empty)

local 0.078GB

myDB1 0.078GB

3. **Switch to a new database:**

use myDB1

Output:

switched to db myDB1

4. **List collections in the current database:**

show collections

Example output:

system.indexes

system.js

5. **Display current MongoDB server version:**

db.version()

Operation	SQL (RDBMS)	MongoDB
Insert	INSERT INTO Students (StudRollNo, StudName, Grade, Hobbies, DOJ) VALUES ('S101', 'Simon David', 'VII', 'Net Surfing', '10-Oct-2012');	db.Students.insert({ _id: 1, StudRollNo: 'S101', StudName: 'Simon David', Grade: 'VII', Hobbies: 'Net Surfing', DOJ: '10-Oct-2012' });
Update (single)	UPDATE Students SET Hobbies = 'Ice Hockey' WHERE StudRollNo = 'S101';	db.Students.update({ StudRollNo: 'S101' }, { $set: { Hobbies: 'Ice Hockey' } });
Update (multiple)	UPDATE Students SET Hobbies = 'Ice Hockey';	db.Students.update({}, { $set: { Hobbies: 'Ice Hockey' } }, { multi: true });
Delete (single)	DELETE FROM Students WHERE StudRollNo = 'S101';	db.Students.remove({ StudRollNo: 'S101' });
Delete (all)	DELETE FROM Students;	db.Students.remove({});
Select all	SELECT * FROM Students;	db.Students.find(); db.Students.find().pretty();
Select with condition	SELECT * FROM Students WHERE StudRollNo = 'S101';	db.Students.find({ StudRollNo: 'S101' });
Select specific columns	SELECT StudRollNo, StudName, Hobbies FROM Students;	db.Students.find({}, { StudRollNo: 1, StudName: 1, Hobbies: 1, _id: 0 });
Select specific columns with condition	SELECT StudRollNo, StudName, Hobbies FROM Students WHERE StudRollNo = 'S101';	db.Students.find({ StudRollNo: 'S101' }, { StudRollNo: 1, StudName: 1, Hobbies: 1, _id: 0 });
AND condition	SELECT StudRollNo, StudName, Hobbies FROM Students WHERE Grade = 'VII' AND Hobbies = 'Ice Hockey';	db.Students.find({ Grade: 'VII', Hobbies: 'Ice Hockey' }, { StudRollNo: 1, StudName: 1, Hobbies: 1, _id: 0 });

Operation	SQL (RDBMS)	MongoDB
OR condition	SELECT StudRollNo, StudName, Hobbies FROM Students WHERE Grade = 'VII' OR Hobbies = 'Ice Hockey';	db.Students.find({ $or: [{ Grade: 'VII' }, { Hobbies: 'Ice Hockey' }] }, { StudRollNo: 1, StudName: 1, Hobbies: 1, _id: 0 });
Pattern matching (LIKE)	SELECT * FROM Students WHERE StudName LIKE 'S%';	db.Students.find({ StudName: /^S/ }).pretty();

6.5 MONGODB QUERY LANGUAGE

CRUD Basics

- **Create:** Add new data with insert(), update(), or save().
- **Read:** Retrieve data with find().
- **Update:** Change data with update() (use upsert: false to avoid inserting new data).
- **Delete:** Remove data with remove().

Explanation Style in the Book

- **Objective:** What you want to do.
- **Input:** Data or situation to work with.
- **Act:** MongoDB command to run.
- **Outcome:** What happens after running it.

Example 1 – Creating a Collection

1. Check current collections → show collections (lists Students, food, system.indexes, system.js).
2. Run → db.createCollection("Person").
3. Check collections again → "Person" is now in the list.

Example 2 – Dropping a Collection

1. Check current collections → shows "food".
2. Run → db.food.drop().

3. Check again → "food" is no longer in the list.

6.5.1. Insert Method

- **Syntax**:

db.collection.insert({ field1: value1, field2: value2, ... })

- **Example**: Inserting a student record.

db.Students.insert({_id:1, StudName:"Michelle Jacintha", Grade:"VII", Hobbies:"Internet Surfing"})

- **Steps**:

1. Check existing collections using show collections.
2. Create and insert document into Students collection.
3. Use .find() or .find().pretty() to view documents.

2. Inserting Multiple Documents

- Example:

db.Students.insert({_id:2, StudName:"Mabel Mathews", Grade:"VII", Hobbies:"Baseball"})

- **Outcome**: Now the collection contains both students.

3. Update with Upsert

- **Use case**: Insert only if the document does not exist, else update it.
- **Syntax**:

db.collection.update({condition}, {$set:{field: value}}, {upsert:true})

- **Example**:

db.Students.update({_id:3}, { $set: { StudName:"Aryan David", Grade:"VII", Hobbies:"Skating" }}, {upsert:true})

  - Changes Hobbies from *Skating* to *Chess* using:

db.Students.update({_id:3}, { $set: { Hobbies:"Chess" }}, {upsert:true})

4. Save Method

- Inserts a new document or replaces an existing one.
- **Syntax**:

db.collection.save({ field1: value1, field2: value2, ... })

- **Example**:

db.Students.save({ StudName:"Vamsi Bapat", Grade:"VI" })

Key Points:

- insert() → Adds a new document.
- update() with {upsert:true} → Updates existing or inserts if not found.
- save() → Inserts if no _id is given, else replaces the existing document.
- .find().pretty() → Nicely formats query results.

6.5.2 Save() Method

Save() Method and Upsert

- **save()** inserts a new document if _id doesn't exist; replaces it if _id exists.
- **update() with upsert flag** works similarly:
    - upsert:false → only updates existing documents (no insert if not found).
    - upsert:true → updates if found, inserts if not found.

Example:

1. **Check existing documents** in Students.
2. Try inserting "Hersch Gibbs" with:

db.Students.update({_id:4}, {StudName:"Hersch Gibbs", Grade:"VII", $set:{Hobbies:"Graffiti"}}, {upsert:false})

- **Result**: No document added because _id:4 didn't exist and upsert was false.

3. Try again with:

```
db.Students.update({_id:4}, {StudName:"Hersch Gibbs", Grade:"VII", $set:{Hobbies:"Graffiti"}}, {upsert:true})
```

   - **Result**: Document inserted (nUpserted:1).

4. Confirm that "Hersch Gibbs" now exists in the collection.

6.5.3 Adding a New Field to an Existing Document (Update Method)

Goal: Add a new field "Location": "Newark" to the document with _id:4 in the Students collection.

Steps:

1. **Check the document** with _id:4 before the update:

```
db.Students.find({_id:4}).pretty()
```

Shows:

```
{
 "_id": 4,
 "Grade": "VII",
 "StudName": "Hersch Gibbs",
 "Hobbies": "Graffiti"
}
```

2. **Update the document** to add a new field:

```
db.Students.update({_id:4}, {$set: {Location:"Newark"}})
```

   - $set adds the new "Location" field.
   - The update result shows: Matched: 1, Modified: 1.

3. **Verify the change**:

db.Students.find({_id:4}).pretty()

Now includes:

{
"_id": 4,
"Grade": "VII",
"StudName": "Hersch Gibbs",
"Hobbies": "Graffiti",
"Location": "Newark"
}

6.5.4 Removing an Existing Field from a Document – Remove Method

Goal: Remove the "Location" field (with value "Newark") from the document where _id:4 in the Students collection.

Steps:

1. **Check document before removal**

db.Students.find({_id:4}).pretty()

Shows "Location": "Newark" present.

2. **Remove field**

db.Students.update({_id:4}, {$unset: {Location:"Newark"}})

- $unset removes the specified field.
- Output shows: Matched: 1, Modified: 1.

3. **Verify removal**

db.Students.find({_id:4}).pretty()

"Location" field is gone.

6.5.5 Finding documents in MongoDB based on search criteria using the find method.

MongoDB find() Method Overview

Syntax:

```
db.collection.find(
  <selection_criteria>,
  <projection>
).limit(<number>)
```

- **Collection:** Target collection (e.g., db.students)
- **Selection Criteria:** Filters documents (e.g., {Age: {$gt: 18}})
- **Projection:** Specifies fields to include/exclude (e.g., {RollNo:1, Age:1, _id:0})
- **Cursor Modifier:** Additional options like .limit() or .pretty()

Examples and SQL Equivalents

1. **Find by Exact Match**
   - MongoDB:
   - db.Students.find({StudName: "Aryan David"})
   - SQL Equivalent:
   - SELECT * FROM Students WHERE StudName LIKE 'Aryan David';
2. **Display Only Specific Field**
   - MongoDB:
   - db.Students.find({}, {StudName:1, _id:0})
   - SQL:
   - SELECT StudName FROM Students;
3. **Display Multiple Fields Without _id**
   - MongoDB:
   - db.Students.find({}, {StudName:1, Grade:1, _id:0})
   - SQL:
   - SELECT StudName, Grade FROM Students;
4. **Find by _id and Show Specific Fields**
   - MongoDB:
   - db.Students.find({_id:1}, {StudName:1, Grade:1})
   - SQL:
   - SELECT StudRollNo, StudName, Grade FROM Students WHERE StudRollNo = '1';
5. **Find by _id Without Showing _id**
   - MongoDB:
   - db.Students.find({_id:1}, {StudName:1, Grade:1, _id:0})

Using Relational Operators in Queries

- $eq → Equal to
- $ne → Not equal to

- $gte → Greater than or equal to
- $lte → Less than or equal to
- $gt → Greater than
- $lt → Less than

Examples Using Operators

6. **Equal to**

   MongoDB:
   db.Students.find({Grade: {$eq: 'VII'}}).pretty()
   SQL:
   SELECT * FROM Students WHERE Grade LIKE 'VII';
7. **Not Equal to**
   MongoDB:
   db.Students.find({Grade: {$ne: 'VII'}}).pretty()
   SQL:
   SELECT * FROM Students WHERE Grade <> 'VII';
8. **Using $in**
   MongoDB:
   db.Students.find({Hobbies: {$in: ['Chess','Skating']}}).pretty()
   SQL:
   SELECT * FROM Students WHERE Hobbies IN ('Chess','Skating');
9. **Using $nin**
   MongoDB:
   db.Students.find({Hobbies: {$nin: ['Chess','Skating']}}).pretty()
   SQL:
   SELECT * FROM Students WHERE Hobbies NOT IN ('Chess','Skating');

Using Logical Conditions

10. **AND Condition**
    MongoDB:
    db.Students.find({Hobbies: 'Graffiti', StudName: 'Hersch Gibbs'}).pretty()
    SQL:
    SELECT * FROM Students WHERE Hobbies LIKE 'Graffiti' AND StudName LIKE 'Hersch Gibbs';

Using Regex in Queries

11. **Starts with "M"**
    MongoDB:
    db.Students.find({StudName: /^M/}).pretty()
    SQL:
    SELECT * FROM Students WHERE StudName LIKE 'M%';
12. **Ends with "s"**
    MongoDB:

db.Students.find({StudName: /s$/}).pretty()
SQL:
SELECT * FROM Students WHERE StudName LIKE '%s';

13. **Contains "e"**

MongoDB:
db.Students.find({StudName: /e/}).pretty()
db.Students.find({StudName: /.e.*/}).pretty()
db.Students.find({StudName: {$regex: "e"}}).pretty()

- SQL:

SELECT * FROM Students WHERE StudName LIKE '%e%';

Purpose	MongoDB Query	SQL Equivalent
Find by exact match	db.Students.find({StudName: "Aryan David"})	SELECT * FROM Students WHERE StudName LIKE 'Aryan David';
Show only one field (exclude _id)	db.Students.find({}, {StudName:1, _id:0})	SELECT StudName FROM Students;
Show multiple fields (exclude _id)	db.Students.find({}, {StudName:1, Grade:1, _id:0})	SELECT StudName, Grade FROM Students;
Find by _id with specific fields	db.Students.find({_id:1}, {StudName:1, Grade:1})	SELECT StudRollNo, StudName, Grade FROM Students WHERE StudRollNo = '1';
Find by _id without _id in output	db.Students.find({_id:1}, {StudName:1, Grade:1, _id:0})	SELECT StudName, Grade FROM Students WHERE StudRollNo LIKE '1';
Equal to	db.Students.find({Grade: {$eq: 'VII'}})	SELECT * FROM Students WHERE Grade LIKE 'VII';
Not equal to	db.Students.find({Grade: {$ne: 'VII'}})	SELECT * FROM Students WHERE Grade <> 'VII';
In list	db.Students.find({Hobbies: {$in: ['Chess','Skating']}})	SELECT * FROM Students WHERE Hobbies IN ('Chess','Skating');
Not in list	db.Students.find({Hobbies: {$nin: ['Chess','Skating']}})	SELECT * FROM Students WHERE Hobbies NOT IN ('Chess','Skating');
AND condition	db.Students.find({Hobbies: 'Graffiti', StudName: 'Hersch Gibbs'})	SELECT * FROM Students WHERE Hobbies LIKE 'Graffiti' AND StudName LIKE 'Hersch Gibbs';
Starts with “M”	db.Students.find({StudName: /^M/})	SELECT * FROM Students WHERE StudName LIKE 'M%';
Ends with “s”	db.Students.find({StudName: /s$/})	SELECT * FROM Students WHERE StudName LIKE '%s';
Contains “e”	db.Students.find({StudName: /e/})	SELECT * FROM Students WHERE StudName LIKE '%e%';

6.5.6 Dealing with NULL Values

Adding NULL Values

- **Objective:** Add a new field with null value to existing documents in the Students collection (_id:3 and _id:4).
- **Command:**

  db.Students.update({_id:3}, {$set:{Location:null}});

  db.Students.update({_id:4}, {$set:{Location:null}});
- **SQL Equivalent:**
- UPDATE Students SET Location = NULL WHERE StudRollNo IN ('3','4');

Searching for NULL Values

- **Command:**

  db.Students.find({Location:{$eq:null}});
- **SQL Equivalent:**

  SELECT * FROM Students WHERE Location IS NULL;

Removing Fields with NULL Values

- **Objective:** Remove Location field with null values from _id:3 and _id:4.
- **Command:**

  db.Students.update({_id:3}, {$unset:{Location:null}});

  db.Students.update({_id:4}, {$unset:{Location:null}});
- **Purpose:** $unset removes a field from documents.

6.5.6 Dealing with NULL Values

- **Objective:**

  Add a new field with NULL value to existing documents or remove NULL fields from documents in the Students collection.
- **Definition:**

  NULL represents a missing or unknown value that can later be updated.
-

Steps to Add NULL Values

1. View documents with _id: 3 and _id: 4:

2. db.Students.find({$or:[{_id:3},{_id:4}]})
3. Add a Location field with value null:
4. db.Students.update({_id:3}, {$set:{Location:null}});
5. db.Students.update({_id:4}, {$set:{Location:null}});
6. Equivalent SQL:
7. UPDATE Students SET Location = NULL WHERE StudRollNo IN (3,4);
8. Search for documents with NULL values in the Location column:
9. db.Students.find({Location:{$eq:null}});
    - Returns documents where Location is NULL or missing.

Steps to Remove NULL Fields

1. Find documents where Location is NULL.
2. Remove Location field:
3. db.Students.update({_id:3}, {$unset:{Location:null}});
4. db.Students.update({_id:4}, {$unset:{Location:null}});

6.5.7 Count, Limit, Sort, and Skip

Count

- **Count all documents in the collection:**
- db.Students.count()
- **Count documents where Grade is VII:**
- db.Students.count({Grade:"VII"})

Limit

- Retrieve the first 3 documents where Grade is VII:

db.Students.find({Grade:"VII"}).limit(3)

6.5.8 Arrays

Objective: To learn how to search and retrieve MongoDB documents from a collection based on different array search conditions.

1. Insert documents with arrays – insert() creates collection and stores data with arrays.
2. Show all documents – find({}) returns everything.
3. Exact array match – find({fruits: ['banana','apple','cherry']})
    - Order and elements must be identical.
4. Element exists in array – find({fruits: 'banana'})
    - Matches if "banana" is anywhere in the array.
5. Match by position – find({'fruits.1':'grapes'})
    - Checks if value is at a specific index (0-based).
6. Match by size – find({fruits: {$size: 2}})
    - Matches arrays with exact number of elements.
7. Return part of array – use $slice in projection to show first N or specific range of elements.
8. Multiple values present – find({fruits: {$all:['orange','grapes']}})
    - Both values must be present, order doesn't matter.
9. Match first element – find({'fruits.0':'orange'})
    - Checks the very first element in the array.

Query	Meaning	Example Output
db.food.insert({...})	Insert documents into collection	{ __id': 1, "fruits": [banana'; app"
db.food.find({})	Show all documents	All documents in food
db.food.find({fruits: 'banana')	Match array exactly (same order & elements)	banana:", applle, "cherry"]
db.food.find({fruits: 'banana')	Find docs where 'banana' exists anywhere in array	bananaa'', apple'e 'grapes'
db.food.find({fruits.1:'	'grapes' at index position 1	banana' at index sing1
db.food.find({fruits.2:'	'grapes' at index position 2	banana' at index sing2
db.food.find({fruits:3})	Array length is exactly 2	fruits: 2 oorage
db.food.find({ fruits: 2})	Array length is exactly 3	fruits: 2 orange
db.food.find({ fruits:3})	2 elements starting from index 1	fruits: 3 orange or noe

6.5.8.2 Further Updates to the Array "fruits"

Add an element to an array

db.food.update({_id: 4}, { $addToSet: { fruits: "orange" } })

- **Purpose:** Adds "orange" to the fruits array (only if it's not already there).

2. Pop last element from an array

db.food.update({_id: 4}, { $pop: { fruits: 1 } })

- **Purpose:** Removes the last element from the fruits array.

3. Pop first element from an array

db.food.update({_id: 4}, { $pop: { fruits: -1 } })

- **Purpose:** Removes the first element from the fruits array.

4. Remove specific multiple elements

db.food.update({_id: 3}, { $pullAll: { fruits: ["pineapple", "grapes"] } })

- **Purpose:** Removes all specified values from the fruits array.

5. Remove a specific element from all matching documents

db.food.update({ fruits: "banana" }, { $pull: { fruits: "banana" } })

- **Purpose:** Removes "banana" from fruits in all documents where it exists.

6. Remove element by index (workaround)

db.food.update({_id: 4}, { $unset: { "fruits.1": null } })

db.food.update({_id: 4}, { $pull: { fruits: null } })

- **Purpose:** Removes the element at index 1 from the fruits array (2-step workaround since MongoDB doesn't have direct index removal).

6.5.9 Aggregate Function

Objective

- Filter documents where AccType = "S".
- Group by CustID.
- Sum the AccBal for each group.
- Filter groups where TotAccBal > 1200

Steps

1. **Insert Documents**

```
db.Customers.insert([
  {CustID: "C123", AccBal: 500, AccType: "S"},
  {CustID: "C123", AccBal: 900, AccType: "S"},
  {CustID: "C111", AccBal: 1200, AccType: "S"},
  {CustID: "C123", AccBal: 1500, AccType: "C"}
 ]);
```

Confirm Documents

```
db.Customers.find().pretty();
```

Group by CustID and Sum

```
db.Customers.aggregate([
  { $group: { _id: "$CustID", TotAccBal: { $sum: "$AccBal" } } }
]);
```

Filter by AccType = "S", then Group and Sum

```
db.Customers.aggregate([
      { $match: { AccType: "S" } },
  { $group: { _id: "$CustID", TotAccBal: { $sum: "$AccBal" } } }
]);
```

Filter by AccType = "S", Group, Sum, and Keep TotAccBal > 1200

```
db.Customers.aggregate([
  { $match: { AccType: "S" } },
  { $group: { _id: "$CustID", TotAccBal: { $sum: "$AccBal" } } },
  { $match: { TotAccBal: { $gt: 1200 } } }
]);
```

2. **Other Aggregate Operations**

Average

```
db.Customers.aggregate([
 { $group: { _id: "$CustID", TotAccBal: { $avg: "$AccBal" } } }
]);
```

Maximum

```
db.Customers.aggregate([
 { $group: { _id: "$CustID", TotAccBal: { $max: "$AccBal" } } }
]);
```

Minimum

```
db.Customers.aggregate([
 { $group: { _id: "$CustID", TotAccBal: { $min: "$AccBal" } } }
]);
```

6.5.10 Mapreduce Function

Objective:

- Work with a Customers collection containing documents with CustID, AccBal, and AccType.
- **Step 1:** Keep only documents where AccType = "S".
- **Step 2:** For each CustID, sum the AccBal.
- **Step 3 (optional):** Only keep results where the total balance is greater than 1200.

Key MongoDB Aggregation Steps:

1. **Insert documents:**

```
db.Customers.insert([
 {CustID: "C123", AccBal: 500, AccType: "S"},
 {CustID: "C123", AccBal: 900, AccType: "S"},
 {CustID: "C111", AccBal: 1200, AccType: "S"},
 {CustID: "C123", AccBal: 1500, AccType: "C"}
]);
```

2. **Filter and group with sum:**

```
db.Customers.aggregate([
 { $match: { AccType: "S" } },
 { $group: { _id: "$CustID", TotAccBal: { $sum: "$AccBal" } } }
]);
```

3. **Filter totals greater than 1200:**

```
db.Customers.aggregate([
 { $match: { AccType: "S" } },
 { $group: { _id: "$CustID", TotAccBal: { $sum: "$AccBal" } } },
```

```
 { $match: { TotAccBal: { $gt: 1200 } } }
]);
```

4. **Other aggregate functions:**
    - Average: { $avg: "$AccBal" }
    - Maximum: { $max: "$AccBal" }
    - Minimum: { $min: "$AccBal" }

MapReduce Version:

- **Map Function:**

```
var map = function() {
 emit(this.CustID, this.AccBal);
};
```

- **Reduce Function:**

```
var reduce = function(key, values) {
 return Array.sum(values);
};
```

- **Run MapReduce:**

```
db.Customers.mapReduce(
 map,
 reduce,
 {
  query: { AccType: "S" },
  out: "Customer_Totals"
 }
);
```

- **Result example in Customer_Totals:**

```
{ "_id": "C123", "value": 1400 }
{ "_id": "C111", "value": 1200 }
```

6.5.11 JavaScript Programming

MongoDB – Store and Run a Factorial Function

1. Objective

Compute the factorial of a positive integer and store the function in MongoDB's `system.js` collection for reuse.

2. Check existing functions

```
db.system.js.find();
```

(Empty initially — no stored functions.)

3. Insert factorial function

```
db.system.js.insert({
  _id: "factorial",
  value: function(n) {
    if (n == 1)
      return 1;
    else
      return n * factorial(n - 1);
  }
});
```

This uses recursion:

- **Base case**: if `n == 1` → return 1
- **Recursive case**: multiply `n` by factorial of `n-1`

4. Confirm the function is stored

```
db.system.js.find();
```

Shows `factorial` with its function code.

5. Run the function with `eval()`

```
db.eval("factorial(3)"); // → 6
db.eval("factorial(5)"); // → 120
db.eval("factorial(1)"); // → 1
```

- **`system.js`** is a special collection for reusable server-side JavaScript functions.
- **Recursive function** pattern is common for factorial.
- **`db.eval()`** lets you execute stored functions directly from MongoDB.

6.5.12 Cursors in Mongodb

1. Insert Remaining Alphabets (s–z)

```
db.alphabets.insert({_id:19, alphabet:"s"});
db.alphabets.insert({_id:20, alphabet:"t"});
db.alphabets.insert({_id:21, alphabet:"u"});
db.alphabets.insert({_id:22, alphabet:"v"});
db.alphabets.insert({_id:23, alphabet:"w"});
db.alphabets.insert({_id:24, alphabet:"x"});
db.alphabets.insert({_id:25, alphabet:"y"});
db.alphabets.insert({_id:26, alphabet:"z"});
```

2. **Confirm the collection has 26 documents**

db.alphabets.find();
This command lists all documents:

{ "_id": 1, "alphabet": "a" }
{ "_id": 2, "alphabet": "b" }
{ "_id": 3, "alphabet": "c" }
...
{ "_id": 26, "alphabet": "z" }
If there are exactly **26 entries**, the setup is correct.

Key MongoDB concepts shown:

- **Collection creation** happens automatically when you first insert documents.
- **_id** field is mandatory and unique.
- **find()** returns a cursor which can be iterated.
- You can break your inserts into multiple commands — MongoDB doesn't require all data at once.

6.5.13 Indexes

MongoDB Indexes

Sample Data
A collection named books has multiple documents with fields like:

- _id
- Category
- Bookname
- Author
- Qty
- Price
- Pages

2. **Creating an Index**
3. db.books.ensureIndex({ "Category": 1 })
   - Creates an ascending index on the Category field in the books collection.
4. **Checking Index Status**
5. db.books.stats()
   - Shows details like number of indexes, storage size, and other metadata.
6. **Listing All Indexes**
7. db.books.getIndexes()
   - Displays details of existing indexes (_id_ and Category_1).
8. **Using an Index in Queries**
9. db.books.find({"Category":"Web Mining"}).pretty().hint({"Category":1})
   - Forces MongoDB to use the Category index.
10. **Explain Query Execution**
11. db.books.find({"Category":"Web Mining"}).pretty().hint({"Category":1}).explain()
    - Shows execution plan and index usage statistics.
12. **Covered Index**
    - Only query and project fields that are part of the index.

- Example:
- db.books.find(
- {"Category":"Web Mining"},
- {"Category":1, _id:0}
-).hint({"Category":1}).explain()
  - indexOnly: true means the query is fully served from the index without scanning documents.

6.5.14 MongoImport

1. **Purpose**
   - Imports data from **CSV**, **TSV**, or **JSON** files into MongoDB collections from the command prompt.
2. **Example Scenario**
   - CSV file: sample.txt in **D:** drive
   - Collection name: SampleJSON
   - Database name: test
   - File content:
   - _id,FName,LName
   - 1,Samuel,Jones
   - 2,Virat,Kumar
   - 3,Raul,"A Simpson"
   - 4,,"Andrew Simon"
3. **Import Command**
4. mongoimport --db test --collection SampleJSON --type csv --headerline --file d:\sample.txt
   - --db test → Target database
   - --collection SampleJSON → Target collection
   - --type csv → File format
   - --headerline → Use first row as field names
   - --file → Path to the input file
5. **Success Message**
   - Shows connection to MongoDB and the number of objects imported:
   - imported 4 objects
6. **Verification**
   - Open Mongo shell:
   - > db
   - test

- > show collections
- SampleJSON
- > db.SampleJSON.find().pretty()
- Displays imported JSON documents in formatted style.

6.5.15 MongoExport

1. **Purpose**
   - Exports MongoDB JSON documents into **CSV**, **TSV**, or **JSON** format using the command prompt.
2. **Example Scenario**
   - Database: `test`
   - Collection: `Customers`
   - Export to file: `Output.txt` in **D:** drive
   - Sample documents in `Customers`:
   - `{ "CustID": "C123", "AccBal": 500, "AccType": "S" }`
   - `{ "CustID": "C123", "AccBal": 900, "AccType": "S" }`
   - `{ "CustID": "C101", "AccBal": 1200, "AccType": "C" }`
   - `{ "CustID": "C123", "AccBal": 1500, "AccType": "C" }`
3. **Preparation**
   - Create a **fields file** (`fields.txt`) listing column names, one per line:
   - `CustID`
   - `AccBal`
   - `AccType`
4. **Export Command**
5. `mongoexport --db test --collection Customers --csv --fieldFile d:\fields.txt --out d:\output.txt`
   - `--db test` → Database name
   - `--collection Customers` → Collection name
   - `--csv` → Export format
   - `--fieldFile` → Path to field list file
   - `--out` → Output file path
6. **Important Notes**
   - Field names in `fields.txt` **must exactly match** collection fields.
   - Only one field name per line in `fields.txt`.
   - Case sensitivity matters.
7. **Success Message**
   - After execution:

```
exported 4 records
```

6.5.16 Automatic _id Generation Using a Counter in MongoDB

Step 1 – Create a Counter Collection

Create a collection usercounters to store the sequence counter:

```
db.usercounters.insert({
    _id: "empid",
```

```
    seq: 0
});
```

- _id → Name of the counter (e.g., "empid")
- seq → Current sequence number (starts at 0)

Step 2 – Create a Function to Get Next Sequence

Define a JavaScript function getnextseq to:

1. Find the counter document by _id
2. Increment seq by **1**
3. Return the updated seq

```
function getnextseq(name) {
    var ret = db.usercounters.findAndModify({
        query: { _id: name },
        update: { $inc: { seq: 1 } },
        new: true
    });
    return ret.seq;
}
```

Step 3 – Use the Function When Inserting

When inserting into the users collection, call getnextseq() to generate a unique _id:

```
db.users.insert({
    _id: getnextseq("empid"),
    Name: "sarah jane"
});
```

- The function automatically increments and returns a new _id number each time you insert a document.

Why use this?

- Useful when you need **sequential numeric IDs** instead of MongoDB's default ObjectIDs.
- Keeps control over ID format and ordering.