DATABASE MANAGEMENT SYSTEM

NOSQL AND NEW DATABASE TECHNOLOGIES (MONGODB)

Relational Database Management System (RDBMS)

Definition:

- Contains information about a particular enterprise.
- Consists of a **collection of interrelated data** (called a database).
- Includes a **set of programs** to access and manage the data.
- Provides an **environment** that is **both convenient and efficient** to use for data storage, retrieval, and manipulation.

Key Characteristics:

- **Data Storage**:
  Data is stored in **tables (relations)**, organized into **rows and columns**.
- **Relationships**:
  Relationships between data are **represented through tables** and keys (like **foreign keys**).
- **Data Definition Language (DDL)**:
  - Used to **define** database structure (e.g., CREATE, ALTER, DROP tables).
- **Data Manipulation Language (DML)**:
  - Used to **insert**, **update**, **delete**, and **query** data (e.g., INSERT, UPDATE, DELETE, SELECT).
- **Transactions and ACID Properties**:
  Transactions in RDBMS ensure:
  - **Atomicity** (all-or-nothing),
  - **Consistency** (database remains valid),
  - **Isolation** (transactions don't interfere),
  - **Durability** (changes are permanent after commit).
- **Abstraction from Physical Layer**:
  Users interact with **logical structures** (tables, columns) without worrying about **physical storage** details.

- **Database Applications:**
  - Banking: all transactions
  - Airlines: reservations, schedules
  - Universities: registration, grades
  - Sales: customers, products, purchases
  - Online retailers: order tracking, customized recommendations
  - Manufacturing: production, inventory, orders, supply chain
  - Human resources: employee records, salaries, tax deductions

ACID Properties of Transactions

A **transaction** is a **unit of program execution** that accesses and possibly updates various **data items**.
To **preserve the integrity** of the database, the system must ensure the **ACID** properties:

Property	Definition	Explanation
Atomicity	All operations of a transaction are completed successfully or none are.	No partial updates; ensures the transaction is "all-or-nothing."
Consistency	Transaction execution preserves the consistency of the database.	Database moves from one valid state to another valid state.
Isolation	Transactions are executed independently, without interference.	Intermediate states of a transaction are invisible to other transactions.
Durability	Once a transaction commits, changes are permanent even after a system failure.	Committed data is saved permanently in the database.

Structured Data (RDBMS) vs Unstructured Data:

Aspect	Structured Data (RDBMS)	Unstructured Data
Definition	Data is organized in a predefined manner. Data organized in tables (rows and columns).	Data without a predefined format or structure.
Storage	Stored in **relational databases** (e.g., MySQL, Oracle).	Stored in **NoSQL databases**, files, or cloud.
Schema	Fixed schema with predefined structure.	No fixed schema; flexible and dynamic.
Data Types	Numeric, text, date, boolean.	Text, multimedia (images, audio, video).
Ease of Access	Accessed using **SQL queries**.	Requires **advanced tools** for extraction.
Analysis Tools	SQL-based and BI tools.	Machine learning, big data tools (e.g., Hadoop).
Examples	University database, Customer data, transactions.	Social media posts, videos, emails, Facebook, Email body, twitter
Scalability	Vertical scaling (hardware upgrade).	Horizontal scaling (more servers).
Best Used For	Transactions, relational data (banking, inventory).	Media, IoT, irregular data (text, images).

SQL vs NoSQL

Aspect	SQL (Relational Databases)	NoSQL (Non-relational Databases)
Data Structure	Structured tables with rows and columns.	Document, key-value, graph, or wide-column stores.
Schema	Fixed schema (needs to be defined first).	Dynamic or flexible schema (can evolve over time).
Scalability	Vertical scaling (scale by upgrading hardware).	Horizontal scaling (scale by adding more servers).
ACID Compliance	Strong ACID compliance for transactions.	Some NoSQL databases relax ACID for better performance (eventual consistency).
Query Language	Uses Structured Query Language (SQL).	Uses various query languages depending on the database (e.g., MongoDB Query Language).
Examples	MySQL, PostgreSQL, Oracle DB, Microsoft SQL Server.	MongoDB, Cassandra, CouchDB, Redis, Neo4j.
Best Used For	Complex queries, multi-row transactions, financial systems.	Large volumes of unstructured data, real-time applications, scalable applications.

Motives Behind NoSQL

Motive	Explanation
Big Data	Ability to **collect, store, organize, analyze, and share** massive volumes of data. Traditional databases cannot keep up with the growth.
Scalability	Need for **horizontal scalability** (scale-out by adding servers) rather than just **vertical scalability** (scale-up with bigger servers).

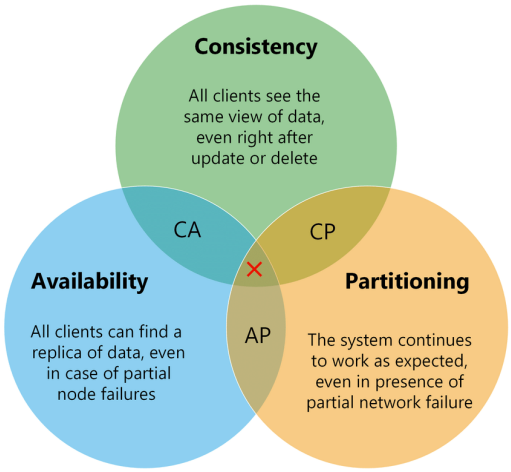
Motive	Explanation
Manageability	Easier management with **less downtime**, **fault tolerance**, and **shared-nothing architecture** using cheap, commodity hardware.
Data Format	Support for **flexible and evolving data models**, handling both structured and unstructured data easily.

CAP Theorem (Brewer's Theorem)

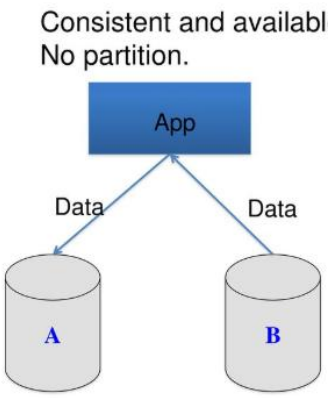
- Proposed by **Prof. Eric Brewer** (2000, University of Berkeley).

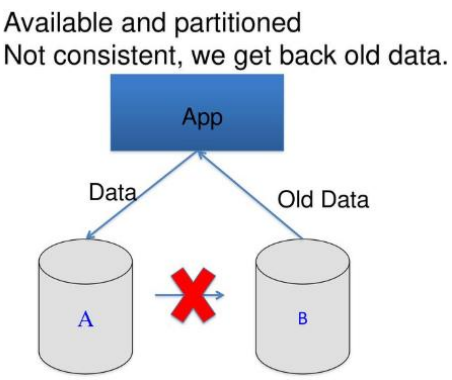
◆ In a **distributed data system**, you can achieve **only two out of three** at any given time:

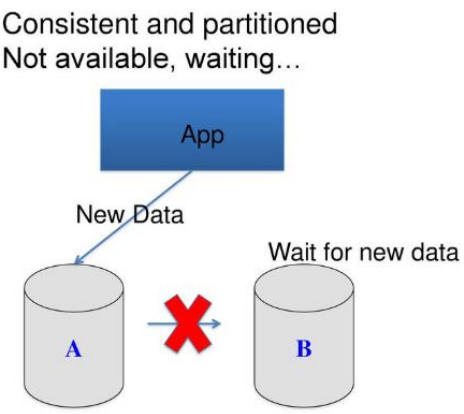
1. **Consistency (C)** – All clients see the same data, always.
2. **Availability (A)** – Every request gets a (non-error) response, even if it's not the latest.
3. **Partition Tolerance (P)** – System continues to operate despite network failures that split communication.



Link: [Brewer's CAP Theorem Paper]







BASE in Databases

BASE is an acronym used in **NoSQL databases** as a softer alternative to ACID properties. It stands for:

Letter	Meaning	Explanation
B	**Basically Available**	The system guarantees availability — responses will be given (even if some data is not the most recent).
A	**Soft state**	The system state may change over time, even without new input (due to eventual consistency).
SE	**Eventual consistency**	The system will **eventually** become consistent, but **not immediately** after a transaction.

ACID vs BASE

Aspect	ACID	BASE
Full form	Atomicity, Consistency, Isolation, Durability	Basically Available, Soft state, Eventual consistency
Focus	Data integrity and correctness	High availability and scalability
Consistency	Strict — database remains consistent after every operation	Relaxed — database may be temporarily inconsistent
System Type	Traditional relational databases (SQL)	NoSQL and distributed databases
Transaction Behavior	All operations succeed or none (atomic transactions)	Transactions may not be immediately consistent, but will eventually be
Failure Handling	Strong rollback and recovery mechanisms	Tolerates partial failures; system keeps working

Aspect	ACID	BASE
Availability	Sometimes sacrificed for consistency (CAP theorem)	Prioritizes availability over immediate consistency
Examples	MySQL, PostgreSQL, Oracle DB	Cassandra, DynamoDB, Couchbase, MongoDB (in some configurations)

NoSQL

NoSQL is a class of database management systems that **do not strictly follow** the relational database model (RDBMS) and **SQL** standards.

Originally meant "**Non-SQL**" but now stands for "**Not Only SQL**."

- **Origin and Popularity**:
    - Emerged due to the need for **web-scale** data management.
    - Companies like **Google**, **Facebook**, **Amazon**, **Twitter**, etc., developed systems for data **where SQL was not the best fit**.

Why NoSQL?

- **High Availability**: Systems must stay accessible even during failures.
- **Flexible Data Models**: Supports various structures (documents, graphs, key-value, wide-column stores).
- **Real Location Independence**: Data can be distributed globally.
- **Schema-Free**: No rigid schema; dynamic and evolving data structures.
- **Ease of Use**: Designed for developers' productivity and faster deployment.

NoSQL Distinguishing Characteristics

- **Handles Large Data Volumes**: Example — Google's "Big Data."
- **Scalable Replication and Distribution**:
    - Thousands of machines.

    - Geographically distributed databases.
- **Fast Query Response**:
    - Optimized for quick reads.
- **Mostly Reads, Few Updates**:
    - Heavy read loads.
    - Fewer updates compared to traditional systems.
- **Asynchronous Inserts and Updates**:
    - Writes are often processed asynchronously to boost performance.
- **Schema-less**:
    - No rigid tables; documents/records can have different structures.
- **BASE Model**:
    - ACID properties are **not strictly needed**.
    - Follows **BASE** (Basically Available, Soft state, Eventually consistent).

Advantages	Disadvantages
High Scalability: Easily scales horizontally across multiple servers.	**No Standardization**: Each NoSQL database has its own APIs, query languages, and techniques.
Distributed Computing: Supports distributed architectures natively.	**Limited Query Capabilities**: Complex queries like JOINs and aggregations are harder or less efficient.
Lower Cost: Open-source and runs on commodity hardware.	**Eventual Consistency**: Often sacrifices strong consistency for availability and partition tolerance.
Schema Flexibility: Supports dynamic, evolving data.	**Maturity Issues**: Newer databases might lack robust tooling and enterprise features.

Advantages

No Complex Relationships: Simpler data modeling without the need for complex JOINs.

Faster for Big Data and Real-Time Applications: Ideal for large-scale and real-time processing.

High Availability and Fault Tolerance: Minimizes downtime during failures.

Disadvantages

Less Support for ACID Transactions: Full ACID compliance is not guaranteed.

Data Duplication: Denormalization can lead to redundancy and maintenance challenges.

SQL vs NoSQL: Key Differences

1. **Scaling**
    - **SQL**: Traditionally, SQL databases scale **vertically** (scale-up), meaning you increase the capacity of a single machine by upgrading its hardware (more RAM, CPU). This approach limits scalability and becomes expensive as the application grows.
    - **NoSQL**: NoSQL databases are designed for **horizontal scaling** (scale-out). They distribute data across multiple servers or cloud instances, enabling better handling of large volumes of data and high traffic, making it easier to scale applications without the need for more powerful hardware.
2. **Modeling**
    - **SQL**: SQL databases follow a **relational model** where data is highly **normalized** into tables with predefined schemas. This requires a well-thought-out design before inserting data, which can lead to data consistency and integrity but can be rigid when adapting to changes.
    - **NoSQL**: NoSQL databases use a more flexible approach, allowing **schema-less** designs. This means data can be stored without a predefined structure, allowing for more agility and rapid development, especially when dealing with unstructured or semi-structured data.

NoSQL Database Types

1. Key-Value Store (e.g., Redis):

- **Structure**: Stores data as key-value pairs (hash table).
- **Use Cases**: Fast lookups, caching, session storage.
- **Example**: Redis, used for real-time data retrieval.

2. Column Store (e.g., Cassandra):

- **Structure**: Data is stored in columns, not rows, for optimized reads of specific data.
- **Use Cases**: Large-scale data aggregation, data warehousing, time-series data.
- **Example**: Cassandra, used for high-volume, distributed systems.

3. Document Store (e.g., MongoDB):

- **Structure**: Stores data in documents (JSON/BSON), no fixed schema.
- **Use Cases**: Flexible, semi-structured data (e.g., content management, user profiles).
- **Example**: MongoDB, great for applications with varying data formats.

4. Graph-Based Store (e.g., Neo4j):

- **Structure**: Stores data as nodes and edges, representing entities and relationships.
- **Use Cases**: Complex relationships, social networks, recommendation systems.
- **Example**: Neo4j, used for analyzing networked data.

Data Model	Performance	Scalability	Flexibility	Complexity
Key value	High	High	High	None

Column	High	High	Moderate	Low
Document	High	Variable	High	Low
Graph	variable	Variable	High	High

Document Stores:

- **Schema-Free**: No predefined schema; documents can have varying structures.
- **Data Format**: Usually use a **JSON-like** format (e.g., BSON in MongoDB).
- **Query Model**: Queries are typically written in **JavaScript** or a custom query language (specific to the database, like MongoDB's query syntax).
- **Aggregations**: Often handled using **Map/Reduce** operations to process large datasets.
- **Indexes**: Typically use **B-Trees** for indexing to ensure efficient searching and retrieval of documents.

Column Stores:

- Column stores can be classified into two types:

Feature	**Row-Oriented Column Store**	**Column-Oriented Column Store**
Data Storage	Stores data by rows, but columns are stored separately.	Data is stored by columns (column family).
Query Efficiency	More efficient for row-based operations, but can be slower for column-based queries.	Optimized for column-based queries, making it faster for read-heavy operations involving specific columns.

Feature	Row-Oriented Column Store	Column-Oriented Column Store
Use Case	Hybrid use cases needing both row-based and column-based access.	Best suited for analytical workloads, aggregation, and data warehousing.
Compression	Less efficient compression since rows have different types of data.	Better compression due to similar data types stored together in columns.
Example	Apache HBase.	Apache Cassandra, Google Bigtable.
Performance for Aggregations	Slower for large-scale aggregations.	Faster for aggregations and large-scale data operations.
Data Retrieval	Faster when accessing entire rows.	Faster when accessing a few specific columns in large datasets.

- **Row oriented**

Id	username	Email	Department
1	John	john@foo.com	Sales
2	Mary	mary@foo.com	Marketing
3	Yoda	yoda@foo.com	IT

- **Column oriented**

Id
1
2
3

Username
John
Mary
Yoda

email
john@foo.com
mary@foo.com
yoda@foo.com

Department
Sales
Marketing
IT

MongoDB

MongoDB is a **open source, document-oriented** NOSQL database that stores data in flexible, JSON-like documents. It is designed to handle large volumes of unstructured or semi-structured data with high performance and scalability. Unlike traditional relational databases, MongoDB does not use tables and rows but instead stores data in collections of documents.

1. **Data Types**:
    - **bool**: Boolean values (true/false).
    - **int**: Integer values.
    - **double**: Floating point numbers.
    - **string**: Textual data.
    - **object (BSON)**: Embedded documents.
    - **ObjectId**: Unique identifier (typically auto-generated).
    - **array**: Ordered list of values.
    - **null**: Represents null values.
    - **date**: Stores date and time.
2. **Database and Collections**:
    - **Automatic Creation**: MongoDB automatically creates databases and collections when data is inserted.
3. **Capped Collections**:
    - Fixed-size collections with a **FIFO (First-In, First-Out)** structure.
    - Ideal for high-speed, log-like data storage (e.g., logs or time-series data).
    - **No Indexes**: Data is added in a continuous stream, and old entries are automatically overwritten once the limit is reached.
4. **ObjectId**:
    - **Generated by client**: 12-byte identifier.
        - **4 bytes**: Timestamp (seconds since Unix epoch).

        - **3 bytes**: Machine identifier (unique to the server).
        - **2 bytes**: Process ID (PID).
        - **3 bytes**: Incremental counter.
    - The ObjectId ensures uniqueness across different machines.
5. **Referencing vs. Embedding**:
    - **Referencing**: Possible to reference documents in different collections, but embedding is more efficient for related data, reducing the need for joins.
    - **Embedding**: Storing related data within a document, making queries faster and more efficient.
6. **Replication**:
    - **Easy Setup**: MongoDB supports replication, where data can be replicated across multiple servers (called **Replica Sets**).
    - **Read from Slaves**: You can configure MongoDB to read from secondary replicas (slaves), distributing the read load and improving performance.

Key Features of MongoDB:

1. **High Performance**:
    - **Fast Data Persistence**: Efficient data storage and retrieval.
    - **Embedded Data Models**: Reduces need for joins and minimizes I/O.
    - **Indexes**: Improves query speed, including support for embedded documents and arrays.
2. **High Availability**:
    - **Replica Sets**: Ensures data availability with automatic failover and redundancy.
    - **Automatic Failover**: Quickly switches to a secondary server if the primary fails.
3. **Automatic Scaling**:

- **Horizontal Scalability**: Distributes data across multiple machines.
- **Sharding**: Automatically partitions data across clusters for load balancing.
- **Eventual Consistency**: Provides low-latency reads in high-throughput systems.

Binary JSON (BSON):

- **Data Model**: BSON is a binary format of JSON (JavaScript Object Notation), which is used by MongoDB to store data. It allows for easy mapping to modern **object-oriented languages** without needing a complex Object-Relational Mapping (ORM) layer.
- **Structure**: BSON stores data as key-value pairs, where each pair is a single entity, making it efficient and easy to traverse.
- **Features**:
  - **Lightweight**: The binary format is compact, requiring less storage space.
  - **Traversable**: Data is easy to navigate through.
  - **Efficient**: Optimized for fast storage and retrieval of data.

Document Database (MongoDB):

- **Document Structure**: In MongoDB, a record is stored as a **document**, which is a data structure made up of **field and value pairs**. These documents are similar to **JSON objects**, and the values of fields can include:
  - Other **documents** (nested objects),
  - **Arrays**,
  - **Arrays of documents**.
- **Example Document**:

```
{
 RollNo: 3201,
 name: "Rahul",
 Class: "TE Comp",
 Subject: ["DBMSA", "TOC"]
}
```

Advantages of Document Databases:

1. **Natural Mapping**: Documents map directly to native data types in many programming languages (e.g., objects in JavaScript).
2. **Reduced Joins**: **Embedded documents** and **arrays** reduce the need for expensive joins by storing related data together.
3. **Dynamic Schema**: Supports **fluent polymorphism** (flexibility in data structure), meaning documents can have different structures without altering the database schema.

Term Mapping Between RDBMS and MongoDB:

RDBMS	**MongoDB**
Table	Collection
Row	Document
Column	Field
Primary Key	_id (ObjectId)
Foreign Key	Embedded Document or Reference
Index	Index
SQL Query	MongoDB Query (BSON)
Schema	Dynamic Schema (No predefined schema)

Replication: Replica Sets vs. Master-Slave

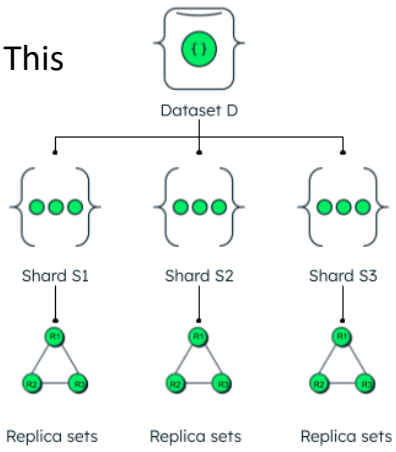
- **Replica Sets**: A **replica set** consists of multiple MongoDB servers that keep the same data, offering **high availability** and **data redundancy**. Only the **primary** server handles **writes**, while **secondaries** replicate data and can handle **reads**.
- **Master-Slave**: In **master-slave**, only the **master** accepts writes, and slaves replicate the data. Reads can happen from slaves, but writes are restricted to the master.

Why Use Replica Sets:

1. **Data Redundancy**: Multiple copies of data ensure fault tolerance.
2. **Automated Failover**: If the primary fails, a secondary becomes the new primary automatically.
3. **Read Scaling**: Offload reads to secondaries, improving performance.
4. **Maintenance**: Perform maintenance without downtime by reading from secondaries.
5. **Disaster Recovery**: **Delayed secondaries** allow recovery from recent data loss.

Sharding in MongoDB:

- **Sharding** is the process of dividing data across multiple machines in a **horizontal scaling** approach. This helps handle large datasets by distributing the load across several servers.
- **Shard Keys**: To distribute data, a **shard key** is chosen. This key determines how the data is partitioned. Example key patterns:
    - { state: 1 }
    - { name: 1 }



The shard key must have **high cardinality** (many unique values) to allow the data to be split into multiple **chunks** and distributed across different shards efficiently.

- **BSON Document**: Each **BSON document** (which may contain embedded data) resides on **one** and only **one** shard in the cluster, ensuring the data is partitioned correctly.

CRUD Operations in MongoDB:

CRUD stands for **Create, Read, Update, and Delete** — the four basic operations to interact with data.

1. Create (Insert Data):

- **Insert one document**:

db.people.insert(mydoc)

This adds one document (record) to the collection.

- **Batch Insert** (Insert multiple documents at once, size limit of 16 MB):

db.collection.insert([doc1, doc2, doc3])

2. Read (Retrieve Data):

- **Find documents** that match a query:

db.collection.find(<query>, <projection>)

  - **<query>**: Specifies the condition to match documents.
  - **<projection>**: Specifies which fields to return (optional).

- **Find one document**:

db.collection.findOne(<query>, <projection>)

This returns the first matching document.

3. Update (Modify Data):

- **Update a document** (by query):

db.collection.update(<query>, <update>, <options>)

- **<options>**: Can include upsert: true to insert a document if it doesn't exist.

4. Delete (Remove Data):

- **Remove specific document(s)**:

db.collection.remove(<query>, <justOne>)

- **<justOne>**: If true, only one matching document is removed.

- **Remove all documents** in a collection:

db.mydb.remove()

- **Remove a collection**:

db.people.drop()

This deletes the collection and all of its documents.

Document Creation and Insertion in MongoDB:

1. Create a Document Format:

```
{
  Teacher_id: "RSCOE1",
  Teacher_Name: "Ravi",
  Dept_Name: "IT",
  Sal: 30000,
  status: 'A'
}
```

2. Create a Collection (optional, MongoDB auto-creates if missing):

db.createCollection("Teacher")

3. Insert Documents into Collection:

db.Teacher.insert({ Teacher_id: "RSCOE1", Teacher_Name: "Ravi", Dept_Name: "IT", Sal: 30000, status: "A" })

db.Teacher.insert({ Teacher_id: "RSCOE2", Teacher_Name: "Ravi", Dept_Name: "IT", Sal: 20000, status: "A" })

db.Teacher.insert({ Teacher_id: "RSCOE3", Teacher_Name: "Akshay", Dept_Name: "Comp", Sal: 25000, status: "N" })

Retrieving data from Mongodb:

- **> db.Teacher.find()**

SQL VS MongoDB Commands

Operation	SQL Command	MongoDB Command
Create Database	CREATE DATABASE mydb;	use mydb (auto-creates)
Create Table/Collection	CREATE TABLE Teacher (...);	db.createCollection("Teacher")
Insert Data (Single)	INSERT INTO Teacher VALUES (...);	db.Teacher.insert({field1: value1, field2: value2})
Insert Data (Multiple)	INSERT INTO Teacher VALUES (...), (...);	db.Teacher.insertMany([{...}, {...}])
Select All Data	SELECT * FROM Teacher;	db.Teacher.find()
Select with Condition	SELECT * FROM Teacher WHERE Sal > 20000;	db.Teacher.find({Sal: {$gt: 20000}})
Select Specific Columns	SELECT name, Sal FROM Teacher;	db.Teacher.find({}, {name: 1, Sal: 1})
WHERE with AND Condition	SELECT * FROM Teacher WHERE Sal > 20000 AND Dept = 'IT';	db.Teacher.find({Sal: {$gt: 20000}, Dept: 'IT'})

Operation	SQL Command	MongoDB Command
WHERE with OR Condition	SELECT * FROM Teacher WHERE Sal > 20000 OR Dept = 'IT';	db.Teacher.find({$or: [{Sal: {$gt: 20000}}, {Dept: 'IT'}]})
Update Data	UPDATE Teacher SET Sal = 35000 WHERE Teacher_id='RSCOE1';	db.Teacher.update({Teacher_id: "RSCOE1"}, {$set: {Sal: 35000}})
Update Many Records	UPDATE Teacher SET Sal = 30000 WHERE Dept = 'IT';	db.Teacher.updateMany({Dept: "IT"}, {$set: {Sal: 30000}})
Delete Specific Data	DELETE FROM Teacher WHERE Teacher_id='RSCOE1';	db.Teacher.remove({Teacher_id: "RSCOE1"})
Delete All Records	DELETE FROM Teacher;	db.Teacher.remove({})
Drop Table/Collection	DROP TABLE Teacher;	db.Teacher.drop()
Count Rows/Documents	SELECT COUNT(*) FROM Teacher;	db.Teacher.countDocuments()
Sorting Data	SELECT * FROM Teacher ORDER BY Sal DESC;	db.Teacher.find().sort({Sal: -1})
Limit Results	SELECT * FROM Teacher LIMIT 5;	db.Teacher.find().limit(5)
GROUP BY Operation	SELECT Dept, COUNT(*) FROM Teacher GROUP BY Dept;	db.Teacher.aggregate([{$group: {_id: "$Dept", total: {$sum: 1}}}])
Aggregation with Conditions	SELECT Dept, AVG(Sal) FROM Teacher GROUP BY Dept HAVING AVG(Sal)>25000;	db.Teacher.aggregate([{$group: {_id: "$Dept", avgSal: {$avg: "$Sal"}}}, {$match: {avgSal: {$gt: 25000}}}])

Operation	SQL Command	MongoDB Command
Join Tables/Collections	SELECT * FROM Teacher INNER JOIN Dept ON Teacher.DeptID=Dept.ID;	MongoDB $lookup: db.Teacher.aggregate([{ $lookup: { from: "Dept", localField: "DeptID", foreignField: "ID", as: "Department_Info" } }])
Distinct Values	SELECT DISTINCT Dept FROM Teacher;	db.Teacher.distinct("Dept")

Aggregation:

- The aggregation framework lets you transform and combine documents in a collection.
- Basically, you build a pipeline that processes a stream of documents through several building blocks:
- filtering, projecting, grouping, sorting, limiting, and skipping.
- For example,
    - If you had a collection of magazine articles, you might want find out who your most prolific authors were. Assuming that each article is stored as a document in MongoDB, you could create a pipeline with several steps:

Aggregation Pipeline

- **Pipeline** = series of **stages**.
- Each stage transforms the documents and passes them to the next stage.
- Common stages:
    - $match – filters documents (like WHERE)
    - $group – groups documents by some field
    - $sort – sorts documents
    - $project – reshapes the documents (choose fields)

Example:

```
db.Teacher.aggregate([
  { $match: { Dept_Name: "IT" } },  // filter
  { $group: { _id: "$Dept_Name", avgSal: { $avg: "$Sal" } } },  // group and average
  { $sort: { avgSal: -1 } }  // sort descending
])
```

MongoDB Aggregation Commands:

Command	Meaning	Use
$project	Select or reshape fields.	Include, exclude, or create new fields.
$match	Filter documents.	Works like SQL WHERE.
$limit	Limit the number of documents.	Returns only the specified number of documents.
$skip	Skip a number of documents.	Used for pagination (e.g., skip first 10 docs).
$unwind	Deconstruct arrays.	Flattens array fields into multiple documents.
$group	Group documents.	Used for operations like SUM, AVG, COUNT, etc.
$sort	Sort documents.	Arrange documents ascending or descending.

Example:

Command	Example	Meaning
$project	{ $project: { name: 1, age: 1 } }	Show only name and age.
$match	{ $match: { status: "A" } }	Filter documents where status is "A".

Command	Example	Meaning
$limit	{ $limit: 5 }	Only first 5 documents.
$skip	{ $skip: 5 }	Skip first 5 documents.
$unwind	{ $unwind: "$hobbies" }	Break hobbies array into separate docs.
$group	{ $group: { _id: "$dept", total: { $sum: 1 } } }	Group by department, count entries.
$sort	{ $sort: { age: -1 } }	Sort by age in descending order.

What are Pipeline Expressions in MongoDB?

- **Pipeline expressions** are **operations** used **inside aggregation stages** (like $project, $group, $match, etc.) to **calculate**, **transform**, or **filter** data.
- They are **small operations** that **work on fields** and **produce computed values**.

Examples:

Expression	Meaning	Example
$sum	Adds values.	{ $sum: "$salary" }
$avg	Calculates average.	{ $avg: "$marks" }
$min	Finds minimum value.	{ $min: "$price" }
$max	Finds maximum value.	{ $max: "$price" }
$push	Push values into an array.	{ $push: "$name" }
$first	First document's value.	{ $first: "$date" }
$last	Last document's value.	{ $last: "$date" }
$addToSet	Unique array values.	{ $addToSet: "$tag" }

- For example, the following expression would sum the "salary" and "bonus" fields:

>db.employees.aggregate({"$project":{"total Pay":{ "$add" : ["$salary", "$bonus"] } } })

Date Expressions in MongoDB

- MongoDB provides **date expressions** to **extract parts of a date**.
- Used inside **aggregation pipelines** (like $project or $group).
- **Important:** You can only use them on fields of **date type**, not numbers.

Expression	Purpose
$year	Extracts the year from a date.
$month	Extracts the month (1–12).
$week	Returns the ISO week number.
$dayOfMonth	Returns the day (1–31).
$dayOfWeek	Returns the day of the week (1–7).
$dayOfYear	Returns the day number in the year (1–366).
$hour	Returns the hour (0–23).
$minute	Returns the minute (0–59).
$second	Returns the second (0–59).

Example:

```
db.orders.aggregate([
 {
   $project: {
    year: { $year: "$orderDate" },
    month: { $month: "$orderDate" },
```

```
    day: { $dayOfMonth: "$orderDate" }
   }
  }
])
```

String Expressions in MongoDB

- **String expressions** are used to **manipulate and transform text data** inside aggregation pipelines.
- These are used inside stages like $project, $group, etc.

Expression	Purpose
$concat	Joins two or more strings.
$substr	Extracts part of a string (substring). *(Deprecated in favor of $substrBytes and $substrCP)*
$toLower	Converts string to lowercase.
$toUpper	Converts string to uppercase.
$strLenBytes	Returns length of a string in bytes.
$strLenCP	Returns length of a string in Unicode code points.
$split	Splits a string into an array.
$trim	Removes whitespace or specific characters from start and end of a string.
$regexMatch	Checks if a string matches a regular expression.

Example:

```
db.users.aggregate([
```

```
  {
    $project: {
      fullName: { $concat: ["$firstName", " ", "$lastName"] },
      lowerCaseEmail: { $toLower: "$email" }
    }
  }
])
```

SORT:

```
> db.employees.aggregate(
... {
... "$project" : {
... "compensation" : {
... "$add" : ["$salary", "$bonus"]
... },
... "name" : 1
... }
... },
... {
... "$sort" : {"compensation" : -1, "name" : 1}
... })
```

- This example would sort employees by compensation, from highest to lowest, and then name from A-Z.
- Possible sorts are 1 (for ascending) and −1 (for descending).

MapReduce in MongoDB:

- MapReduce is a tool in MongoDB used to process and analyze large amounts of data in a **flexible** way, similar to how SQL uses **GROUP BY** to group and summarize data.
- **MapReduce** is a powerful tool for **aggregating data**.

- **How it works**:
    1. **Map**: First, the map function goes through the data and creates **key-value pairs**. Think of it as categorizing data into groups.
    2. **Reduce**: Next, the reduce function looks at these groups and does some work on them, like adding up numbers or combining values.

Steps in MapReduce:

1. **Map Function**:
    - Takes input data and **maps it** into key-value pairs.
    - Each document in the collection is processed and a key-value pair is emitted.
2. **Reduce Function**:
    - Takes the **key-value pairs** output from the map function.
    - Groups values by key and **aggregates** or processes them to produce the final result.
3. **Finalize Function (Optional)**:
    - A final step to process the output of the reduce function before returning it.

Example:

1. Insert Books into the books Collection:

First, you insert records for books:

```
book1 = { name : "Understanding JAVA", pages : 100 };
book2 = { name : "Understanding JSON", pages : 200 };
db.books.save(book1);
db.books.save(book2);

book = { name : "Understanding XML", pages : 300 };
db.books.save(book);

book = { name : "Understanding Web Services", pages : 400 };
db.books.save(book);

book = { name : "Understanding Axis2", pages : 150 };
db.books.save(book);
```

Now, your books collection contains 5 documents, each with a name and pages.

2. Map Function:

This function categorizes books into "Big Books" (books with 250 or more pages) and "Small Books" (books with fewer than 250 pages).

```
var map = function() {
  var category;
  if (this.pages >= 250) {
    category = 'Big Books';
  } else {
    category = "Small Books";
  }
  emit(category, { name: this.name });
};
```

- **emit(category, { name: this.name })**: This emits a category (Big Books or Small Books) with the name of the book. It creates a key-value pair.

---

3. Reduce Function:

This function counts the number of books in each category.

```
var reduce = function(key, values) {
  var sum = 0;
  values.forEach(function(doc) {
    sum += 1;
  });
  return { books: sum };
};
```

- The reduce function groups the books by their category (Big Books or Small Books) and counts how many books are in each category.

---

4. Running MapReduce:

You now run the **MapReduce** function, which will process the data and store the results in a new collection called book_results.

```
var count = db.books.mapReduce(map, reduce, { out: "book_results" });
```

This will:

- Apply the map function to all documents in books.
- Apply the reduce function to group and count the books by category.
- Store the results in the book_results collection.

5. Check the Output:

Finally, you check the output of the mapReduce operation:

```
db.book_results.find();
```

This returns the following result:

```
{ "_id" : "Big Books", "value" : { "books" : 2 } }
{ "_id" : "Small Books", "value" : { "books" : 3 } }
```

- **Big Books** category contains 2 books.
- **Small Books** category contains 3 books.

Summary:

- The **Map function** categorizes books based on their pages into "Big Books" and "Small Books".
- The **Reduce function** counts the number of books in each category.
- The result is saved into a new collection called book_results.