**1. SQL Databases**

SQL databases are **relational databases** that use a structured format to organize data. SQL databases follow a structured schema with tables, rows, and columns, making them ideal for applications that require **complex queries and transactions**.

**Key Characteristics of SQL Databases**

* **Structure**: Data is stored in tables with a fixed schema. Each table has rows (records) and columns (fields).
* **Schema**: A predefined schema or structure must be set before data entry. Any changes to the schema can require significant work.
* **ACID Compliance**: SQL databases are **ACID (Atomicity, Consistency, Isolation, Durability)** compliant, ensuring data reliability and integrity, especially for complex transactions.
* **SQL Language**: Data is accessed and managed using SQL, a powerful language that allows complex queries, joins, filtering, grouping, and aggregation.
* **Examples**: MySQL, PostgreSQL, SQLite, Oracle, and Microsoft SQL Server.

**Pros of SQL Databases**

* **Data Integrity**: ACID compliance ensures data consistency and reliability, making SQL databases a strong choice for financial applications and e-commerce platforms where transactional accuracy is crucial.
* **Complex Queries**: SQL offers sophisticated querying abilities with JOINs, aggregations, and filtering, allowing developers to retrieve specific datasets.
* **Standardization**: SQL is widely supported and standardized, meaning knowledge of SQL applies to many databases.

**Cons of SQL Databases**

* **Scalability**: SQL databases are vertically scalable, meaning scaling typically involves upgrading hardware rather than simply adding more servers. This can be limiting for applications requiring horizontal scalability.
* **Rigidity**: The need for a predefined schema makes SQL databases less flexible, requiring schema updates when data structure changes are needed.

**Best Use Cases for SQL Databases**

* **Banking Systems**: Where data accuracy and integrity are paramount.
* **E-commerce Platforms**: Complex transactions and data relationships require robust querying capabilities.
* **Customer Relationship Management (CRM)**: Structured data and relationships make SQL databases ideal for CRM systems.

**2. NoSQL Databases**

NoSQL databases are **non-relational databases** that offer a more flexible approach to data storage and retrieval. They are designed to handle large volumes of data, which may not fit neatly into tables, and allow for more diverse data models (document, key-value, column-family, and graph).

**Key Characteristics of NoSQL Databases**

* **Flexible Schema**: NoSQL databases allow a schema-less structure, making it easy to store unstructured or semi-structured data. This is useful for applications with evolving data models.
* **Scalability**: NoSQL databases are generally horizontally scalable, meaning they can scale across multiple servers, which is beneficial for high-volume applications.
* **Types of NoSQL Databases**:
  + **Document** (e.g., MongoDB): Stores data in document formats like JSON, ideal for handling nested or hierarchical data.
  + **Key-Value** (e.g., Redis): Uses a simple key-value pair, best for caching and real-time applications.
  + **Column-Family** (e.g., Cassandra): Data is stored in columns rather than rows, optimized for read and write speeds.
  + **Graph** (e.g., Neo4j): Manages relationships between data, ideal for social networks and recommendation engines.
* **Eventual Consistency**: Unlike SQL databases, which enforce immediate consistency, many NoSQL databases are designed for eventual consistency, prioritizing availability and partition tolerance over strict consistency.

**Pros of NoSQL Databases**

* **Flexibility**: No schema requirements make NoSQL databases adaptable to changing data models.
* **Scalability**: Horizontal scalability makes NoSQL ideal for handling large-scale, high-traffic applications by adding servers as needed.
* **Performance**: Some NoSQL databases are optimized for specific types of operations, providing faster access in certain scenarios (e.g., key-value lookups).

**Cons of NoSQL Databases**

* **Data Consistency**: Many NoSQL databases prioritize scalability and performance over strict consistency, which can be a disadvantage in applications requiring high data accuracy.
* **Complex Querying**: NoSQL databases typically lack the sophisticated query capabilities found in SQL, requiring developers to create more custom logic.

**SQL (Structured Query Language)** is a programming language specifically designed for managing and manipulating relational databases. It enables users to create, read, update, and delete (CRUD) data stored in structured tables. SQL databases are essential in applications where data integrity, consistency, and complex queries are prioritized.

**1. Relational Database Structure**

SQL databases, also known as **relational databases**, store data in structured tables with rows and columns. Each table represents an entity (e.g., “Customers”), where:

* **Rows** represent individual records (e.g., a specific customer).
* **Columns** represent fields of data within that entity (e.g., customer ID, name, and email).

Tables can be related to each other using **keys**:

* **Primary Key**: A unique identifier for each record within a table.
* **Foreign Key**: A reference to the primary key in another table, establishing relationships between tables.

**2. SQL Language Components**

SQL provides various commands, grouped primarily into four main categories:

* **Data Definition Language (DDL)**: Manages the structure of the database.
  + CREATE: Creates a new database or table.
  + ALTER: Modifies an existing table structure.
  + DROP: Deletes tables or databases.
* **Data Manipulation Language (DML)**: Manages data within tables.
  + INSERT: Adds new records.
  + UPDATE: Modifies existing records.
  + DELETE: Removes records from tables.
  + SELECT: Queries and retrieves specific data.
* **Data Control Language (DCL)**: Manages permissions for users.
  + GRANT: Provides access rights to users.
  + REVOKE: Removes access rights.
* **Transaction Control Language (TCL)**: Manages database transactions to ensure data integrity.
  + COMMIT: Saves a transaction permanently.
  + ROLLBACK: Reverts changes made by a transaction.
  + SAVEPOINT: Sets a point in a transaction to which one can roll back.

**3. ACID Compliance**

SQL databases follow **ACID (Atomicity, Consistency, Isolation, Durability)** properties to ensure data reliability:

* **Atomicity**: Ensures all parts of a transaction are completed; if one part fails, the entire transaction fails.
* **Consistency**: Guarantees that data is valid according to all defined rules.
* **Isolation**: Ensures transactions do not interfere with each other, even when occurring simultaneously.
* **Durability**: Once a transaction is committed, it remains saved even if the system crashes.

**4. Complex Querying Capabilities**

SQL is well-known for its robust querying capabilities, supporting operations like:

* **Joins**: Combine data from multiple tables (e.g., INNER JOIN, LEFT JOIN) based on relationships.
* **Aggregations**: Compute summaries of data with functions like COUNT, SUM, AVG, and MAX.
* **Filtering**: Use conditions (WHERE) to retrieve specific subsets of data.
* **Grouping**: Group records (GROUP BY) for analysis.
* **Ordering**: Sort results (ORDER BY) based on specific columns.

**5. SQL Database Management Systems (RDBMS)**

There are various SQL database management systems (RDBMS), each with unique features but commonly sharing SQL syntax and functionality:

* **MySQL**: Known for its speed and reliability, commonly used for web applications.
* **PostgreSQL**: Highly extensible and supports advanced data types, including JSON.
* **Oracle Database**: Known for enterprise-level performance, scalability, and security features.
* **Microsoft SQL Server**: Popular in enterprise environments, integrates well with other Microsoft products.

**6. Strengths and Limitations of SQL**

* **Strengths**:
  + **Data Integrity**: ACID compliance ensures reliability in transactional applications.
  + **Complex Querying**: SQL supports detailed data analysis and reporting.
  + **Standardized**: SQL is widely supported across different RDBMS platforms, enabling broad compatibility.
* **Limitations**:
  + **Scalability**: SQL databases are typically vertically scalable, meaning they are limited by the resources of a single server.
  + **Schema Rigidity**: Schema changes can be complex, making SQL less flexible for applications that frequently evolve.

**7. Common Use Cases for SQL Databases**

SQL databases are widely used in applications requiring structured data and transactional integrity:

* **Financial Systems**: Where consistent, reliable transactions are crucial.
* **E-commerce Applications**: To manage products, customers, and order transactions.
* **Customer Relationship Management (CRM)**: For managing customer records and interactions.
* **Healthcare Systems**: To handle structured patient records securely.

In summary, SQL databases are a reliable choice for applications that need strict consistency,

* **Less Standardization**: NoSQL databases vary widely in structure and functionality, meaning skills learned with one NoSQL database might not transfer easily to another.

**Best Use Cases for NoSQL Databases**

* **Real-Time Data Processing**: Applications that need fast read/write access to large data sets, like gaming leaderboards and social media feeds.
* **Big Data Applications**: NoSQL databases handle large amounts of unstructured data, making them ideal for big data analytics and IoT.
* **Content Management Systems (CMS)**: Flexible schema allows NoSQL databases to easily handle a variety of content types and structures.

**Entity Types and Key Attributes**

**1. Entity Type**

An Entity Type groups entities with the same basic attributes. Each entity type is a collection of entities that share common properties, which are represented as attributes. For example:

* EMPLOYEE Entity Type: Represents all employees with common attributes like EmployeeID, Name, Department, and Salary.
* PROJECT Entity Type: Represents projects with attributes like ProjectID, ProjectName, and Budget.

Each entity within an entity type has its own set of values for these attributes, such as a specific employee's name or a specific project's budget.

**2. Key Attribute (Primary Key)**

A Key Attribute is an attribute whose values are unique for each individual entity in an entity set, allowing it to uniquely identify each entity. This is also known as the Primary Key.

Primary Key Characteristics:

* Uniquely identifies each record.
* Ensures that each entity can be differentiated from every other entity in the set.

Example: For the EMPLOYEE entity type, SSN (Social Security Number) might be the primary key, as each employee’s SSN is unique.

**3. Candidate Key**

A Candidate Key is any attribute or combination of attributes that can uniquely identify a row in a table. There can be multiple candidate keys in a table, and any one of them can be chosen as the primary key.

* A candidate key is also known as a Minimal Super Key, meaning it uniquely identifies records without including any unnecessary attributes.

Example: In a Car table with attributes such as RegNo (Registration Number), ChassisNo, Color, Make, and Year:

* Both RegNo and ChassisNo could be candidate keys, as each one can uniquely identify a car in the table. Only one of these would be chosen as the primary key, while the other can be an alternate key.

**4. Alternate Key**

An Alternate Key is any candidate key that is not selected as the primary key. When there are multiple candidate keys in a table, one is chosen as the primary key, while the others are classified as alternate keys.

Example: In a Student table with StudentID, SSN, and Email as attributes, both StudentID and SSN could serve as candidate keys. If StudentID is chosen as the primary key, then SSN would be an alternate key.

**5. Foreign Key**

A Foreign Key is an attribute in one table that links to the primary key of another table. It is used to establish a relationship between two tables, ensuring referential integrity by linking records in related tables.

Example: In a database with an Orders table and a Customers table:

* The Customers table has CustomerID as its primary key.
* The Orders table includes a CustomerID column, which serves as a foreign key that references the CustomerID in the Customers table. This relationship ensures that every order is associated with a valid customer.

**6. Super Key**

A Super Key is any set of one or more attributes that, when combined, uniquely identify a record in a table. A super key can include additional, non-unique attributes beyond those necessary for unique identification.

Example: In an EMPLOYEE table with SSN, Fname (First Name), Lname (Last Name), DOB (Date of Birth), and Addr (Address):

* The attribute SSN alone is sufficient to uniquely identify each employee, making it a candidate key and a super key.
* The combination of SSN and Fname is also a super key, as it still uniquely identifies each record in the table. However, it’s not minimal, as SSN alone is enough to achieve uniqueness.

**Relationship Types, Relationship Sets, Roles**

**1. Relationship Types**

A **Relationship Type** describes a logical connection between two or more **Entity Types** in a database. Each relationship type defines how entities are associated with each other, establishing meaningful interactions that reflect real-world relationships.

**Characteristics of Relationship Types**

* **Defines Connection**: Specifies how two or more entity types relate to each other (e.g., an employee works on a project).
* **Describes Multiplicity**: Determines the number of entities that can participate on each side of the relationship, such as one-to-one, one-to-many, or many-to-many.
* **Establishes Constraints**: Imposes rules for participation, ensuring the integrity of relationships (e.g., each employee can work on multiple projects).

**Examples of Relationship Types**

* **Works\_On**: Represents the relationship between the **EMPLOYEE** and **PROJECT** entity types, where an employee works on one or more projects.
* **Manages**: Represents a relationship between **DEPARTMENT** and **EMPLOYEE** entity types, where a department has a manager.

**Types of Relationships by Cardinality**

1. **One-to-One (1:1)**: Each entity in one entity type is associated with only one entity in another entity type, and vice versa.
   * *Example*: Each **Employee** has one **Workstation**, and each **Workstation** is assigned to only one **Employee**.
2. **One-to-Many (1**

**)**: An entity in one entity type can be associated with multiple entities in another entity type, but the reverse is not true.

* + *Example*: One **Manager** oversees many **Employees**, but each **Employee** has only one **Manager**.

1. **Many-to-Many (M**

**)**: Entities in one entity type can be associated with multiple entities in another entity type, and vice versa.

* + *Example*: Many **Students** can enroll in many **Courses**, and each **Course** can have many **Students**.

**2. Relationship Sets**

A **Relationship Set** is the actual collection of relationships that occur between specific entities in the database. Each relationship in a relationship set connects individual instances (tuples) of entity types according to the defined **Relationship Type**.

**Characteristics of Relationship Sets**

* **Instance-Level Representation**: Relationship sets contain specific instances of the relationship type, rather than just describing the general relationship.
* **Analogous to Tables**: In a relational database, a relationship set is often represented as a separate table, especially in many-to-many relationships, where it stores foreign keys referencing the related entities.

**Example of a Relationship Set**

For a **Works\_On** relationship type between **EMPLOYEE** and **PROJECT**:

* **Relationship Type**: An employee can work on multiple projects, and each project can have multiple employees.
* **Relationship Set**: Contains specific employee-project associations, such as “Employee 101 works on Project X,” “Employee 102 works on Project Y,” etc.

In a database, this might be represented as a table:

| **EmployeeID** | **ProjectID** |
| --- | --- |
| 101 | X |
| 101 | Y |
| 102 | X |

Each row in this table is an instance of the **Works\_On** relationship.

**3. Roles**

**Roles** describe the function or part that an entity type plays within a relationship. They help clarify how entities are related, especially when an entity participates in a relationship multiple times or when the relationship needs additional context.

**Characteristics of Roles**

* **Defines Function**: Roles explain how an entity is involved in the relationship.
* **Clarifies Ambiguity**: When the same entity type participates multiple times in a relationship, roles are essential to avoid confusion.
* **Named Context**: Roles are typically given descriptive names that specify the nature of the relationship for each participating entity.

**Examples of Roles**

1. **Supervisor and Subordinate** in a **Supervises** relationship:
   * In a **Supervises** relationship within an **EMPLOYEE** entity type, each employee could be both a supervisor and a subordinate. Defining these roles clarifies which employee is managing (supervisor) and which is being managed (subordinate).
2. **Supplier and Receiver** in a **Supplies** relationship:
   * In a **Supplies** relationship between a **SUPPLIER** and **DEPARTMENT** entity, **SUPPLIER** plays the role of the supplier, while **DEPARTMENT** plays the role of the receiver.
3. **Borrower and Lender** in a **Loans** relationship:
   * In a **Loans** relationship involving a **CUSTOMER** and **BANK** entity, the **CUSTOMER** role is the borrower, and the **BANK** role is the lender.

**Putting It All Together: Example**

For a **Manages** relationship type between **EMPLOYEE** and **DEPARTMENT**:

* **Relationship Type**: **Manages** defines that each **DEPARTMENT** has one **EMPLOYEE** as a manager.
* **Relationship Set**: Specific instances, such as "Employee 101 manages Department A" and "Employee 102 manages Department B," form the relationship set.
* **Roles**: The **EMPLOYEE** plays the role of "manager," and the **DEPARTMENT** plays the role of "managed department."

**Relational Model and Relational Algebra**

**Relational Model**

The relational model is a way of organizing data into tables (or relations), which are collections of rows (or tuples). Each table is defined by a schema that specifies the attributes (columns) and their data types.

Key concepts in the **Relational Model**:

1. **Relation (Table)**: A table with rows and columns. Each row represents a record, and each column represents a field or attribute.
   * Example: A table Student might have columns like Student\_ID, Name, Age, etc.
2. **Tuple (Row)**: A single record in a table.
   * Example: A row in the Student table could be (12345, 'Alice', 22).
3. **Attribute (Column)**: A field in a table that contains a specific type of data.
   * Example: Name is an attribute in the Student table.
4. **Domain**: The set of all possible values for an attribute.
   * Example: The domain of the Age attribute might be positive integers.
5. **Primary Key**: A unique identifier for each tuple in a table. No two rows can have the same primary key value.
   * Example: Student\_ID could be the primary key in the Student table.
6. **Foreign Key**: An attribute that refers to the primary key of another table, establishing a relationship between the two tables.
   * Example: Course\_ID in a Enrollment table might reference Course\_ID in a Course table.

**Relational Algebra**

Relational Algebra is a procedural query language used to manipulate and retrieve data from relational databases. It consists of a set of operations that take one or more relations as input and produce a new relation as output.

Some basic operations in **Relational Algebra** include:

1. **Select (σ)**: Filters rows based on a given condition.
   * Syntax: σ(condition)(Relation)
   * Example: σ(Age > 21)(Student) selects all students whose age is greater than 21.
2. **Project (π)**: Selects specific columns from a relation.
   * Syntax: π(columns)(Relation)
   * Example: π(Name, Age)(Student) retrieves only the Name and Age columns of the Student table.
3. **Union (∪)**: Combines the results of two relations, returning all tuples that are in either of the relations.
   * Syntax: Relation1 ∪ Relation2
   * Example: Student ∪ Employee returns all unique tuples from both the Student and Employee tables.
4. **Set Difference (−)**: Returns tuples that are in the first relation but not in the second.
   * Syntax: Relation1 − Relation2
   * Example: Student − Employee returns all students who are not employees.
5. **Cartesian Product (×)**: Combines every tuple of the first relation with every tuple of the second relation.
   * Syntax: Relation1 × Relation2
   * Example: Student × Course produces a set of all possible combinations of Student and Course tuples.
6. **Rename (ρ)**: Renames a relation or its attributes.
   * Syntax: ρ(new\_relation\_name)(Relation)
   * Example: ρ(Student\_Info)(Student) renames the Student table to Student\_Info.
7. **Join (⨝)**: Combines related tuples from two relations based on a common attribute.
   * Syntax: Relation1 ⨝ Relation2
   * Example: Student ⨝ Enrollment combines the Student and Enrollment tables based on a common key, such as Student\_ID.
8. **Intersection (∩)**: Returns tuples that are present in both relations.
   * Syntax: Relation1 ∩ Relation2
   * Example: Student ∩ Employee gives students who are also employees.

**MongoDB**

**1. Introduction to MongoDB**

* **MongoDB** is a **NoSQL document database** that stores data in a flexible, JSON-like format. Instead of using tables and rows (as in relational databases), MongoDB uses collections and documents.
* **NoSQL Databases**: Unlike SQL databases, NoSQL databases are designed to handle a wide variety of data formats and scale easily. They are well-suited for large-scale, distributed data environments.
* **Document-Oriented**: MongoDB is based on a document model, where each document represents a record and is a JSON object with a flexible schema.

**2. MongoDB Structure and Data Model**

**Documents**

* In MongoDB, data is stored as **documents**. Each document is a JSON-like object (called BSON – Binary JSON) with key-value pairs, where:
  + **Keys** are field names (like columns in SQL).
  + **Values** are data values (which can be of various types like strings, numbers, arrays, objects, etc.).
* Example document in MongoDB:
* Each document can have its own unique structure, allowing for **schema flexibility**.

|  |
| --- |
| {  "\_id": ObjectId("507f1f77bcf86cd799439011"),  "name": "John Doe",  "age": 29,  "email": "johndoe@example.com",  "skills": ["JavaScript", "MongoDB", "Node.js"]  } |

**Collections**

* **Collections** in MongoDB are analogous to tables in relational databases. A collection is a group of documents, and documents within a collection can have different structures.
* Collections do not enforce a rigid schema, meaning that the documents within a collection can have varying fields and data types.

**BSON (Binary JSON)**

* MongoDB uses **BSON** (Binary JSON) format to store data, which is more efficient and supports additional data types (such as Date, Binary, and ObjectId) that aren’t available in regular JSON.

**3. Key Features of MongoDB**

**Schema Flexibility**

* MongoDB is **schema-less**, meaning each document in a collection can have different fields or data types, allowing for easy adjustments to the structure without complex migrations.

**Rich Query Language**

* MongoDB provides a powerful and flexible query language for reading and manipulating data. This includes:
  + **CRUD Operations**: Create, Read, Update, Delete.
  + **Aggregation Framework**: Allows for complex data processing and transformation using operators like $match, $group, $sort, etc.
  + **Indexes**: MongoDB supports indexes to speed up queries.

**Indexing**

* MongoDB uses indexes to improve the performance of searches within collections. Types of indexes include:
  + **Single Field Index**: Created on a single field, similar to traditional SQL indexes.
  + **Compound Index**: Created on multiple fields.
  + **Multikey Index**: Supports indexing on array fields.
  + **Geospatial Index**: Supports queries on geographical data (e.g., finding locations near a point).

**Replication and Sharding**

* **Replication**: MongoDB supports **replica sets** to provide data redundancy and high availability. A replica set consists of multiple copies of the same data (primary and secondary nodes).
* **Sharding**: MongoDB supports **horizontal scaling** by sharding. It divides large datasets across multiple servers or shards, enabling distributed storage and scaling.

**Aggregation Framework**

* The aggregation framework is used for complex data processing. It allows chaining of operations like filtering, grouping, sorting, and projecting data.
* **Aggregation Pipeline**: A series of stages where each stage processes and transforms the data. Key stages include:
  + **$match**: Filters documents based on a condition.
  + **$group**: Groups documents and performs aggregate calculations.
  + **$sort**: Orders documents.
  + **$project**: Selects specific fields.

**4. CRUD Operations in MongoDB**

**Create (Insert)**

* **insertOne**: Inserts a single document into a collection.
* **insertMany**: Inserts multiple documents at once.

|  |
| --- |
| db.users.insertOne({ "name": "Alice", "age": 25, "city": "New York" }); |

**Read (Find)**

* **find**: Retrieves documents from a collection based on a query. MongoDB provides a range of query operators ($gt, $lt, $in, $regex, etc.) for filtering.

|  |
| --- |
| db.users.find({ "age": { "$gt": 20 } }); |

**Update**

* **updateOne**: Updates a single document based on a query.
* **updateMany**: Updates multiple documents based on a query.
* MongoDB provides operators like $set, $unset, $inc, etc., for updating fields.

|  |
| --- |
| db.users.updateOne({ "name": "Alice" }, { "$set": { "city": "Los Angeles" } }); |

**Delete**

* **deleteOne**: Deletes a single document matching the query.
* **deleteMany**: Deletes all documents matching the query.

Example:

|  |
| --- |
| db.users.deleteOne({ "name": "Alice" }); |

**5. MongoDB Indexing and Performance**

* **Creating Indexes**: Indexes are created using the createIndex method.

db.users.createIndex({ "name": 1 }); // Creates an ascending index on the "name" field

* **Compound Indexes**: Created on multiple fields.

db.users.createIndex({ "name": 1, "age": -1 });

* **Analyzing Performance**: MongoDB provides tools like explain() to analyze the performance of queries and see if indexes are being utilized.

**6. Replication and High Availability**

* **Replica Sets**: MongoDB’s mechanism for replication involves replica sets with multiple nodes:
  + **Primary Node**: Accepts write operations and replicates them to secondary nodes.
  + **Secondary Nodes**: Copies of the primary data; can serve read operations if configured.
  + **Automatic Failover**: If the primary node fails, an election is held among the secondary nodes to determine a new primary.
* **High Availability**: Replica sets provide high availability, as even if one node fails, other nodes can continue serving data.

**7. Sharding and Scalability**

* **Sharding** is MongoDB’s solution for handling large-scale data across multiple servers:
  + **Shard Key**: A field or a combination of fields that distributes data across shards.
  + **Config Servers**: Manage metadata and the distribution of data.
  + **Routing**: Client applications interact with a **mongos** router, which determines the appropriate shard for each query.
* **Horizontal Scalability**: By adding more shards, MongoDB can handle growing data and workload volumes.

**8. Aggregation Framework in MongoDB**

* **Aggregation Pipelines**: MongoDB’s aggregation framework allows processing data in stages, where each stage applies transformations.
* **Common Aggregation Operators**:
  + **$sum**: Calculates the sum of values in a group.
  + **$avg**: Calculates the average value.
  + **$count**: Counts the number of documents.
  + **$push**: Adds values to an array within documents.

Example Aggregation Pipeline:

javascript

Copy code

db.orders.aggregate([

{ "$match": { "status": "completed" } },

{ "$group": { "\_id": "$customer\_id", "totalSpent": { "$sum": "$amount" } } },

{ "$sort": { "totalSpent": -1 } }

]);

**9. Transactions in MongoDB**

* **Multi-Document ACID Transactions**: MongoDB 4.0 introduced support for multi-document transactions to ensure **ACID compliance** (Atomicity, Consistency, Isolation, Durability).
* **Transactions in Replica Sets and Sharded Clusters**: MongoDB allows transactions across multiple documents within a replica set and, in MongoDB 4.2 and later, across sharded clusters.

Example Transaction:

javascript

Copy code

const session = client.startSession();

session.startTransaction();

try {

db.inventory.updateOne({ "item": "A" }, { "$inc": { "qty": -10 } }, { "session": session });

db.orders.insertOne({ "item": "A", "qty": 10 }, { "session": session });

session.commitTransaction();

} catch (error) {

session.abortTransaction();

} finally {

session.endSession();

}

**10. MongoDB Use Cases and Applications**

* **Real-Time Analytics**: MongoDB is well-suited for real-time data processing due to its fast read and write capabilities.
* **Content Management**: It’s widely used in CMS (Content Management Systems) because of its flexibility and schema-less structure.
* **Internet of Things (IoT)**: MongoDB is used in IoT for managing high volumes of unstructured data from various devices.
* **Cataloging and Inventory Systems**: Its document model is ideal for applications like product catalogs where each item has unique attributes.