UNIT - 5

Crud operations in nosql dbs

CRUD operations, which stand for Create, Read, Update, and Delete, are fundamental data management operations that can be performed in NoSQL databases. While the specific syntax and methods for CRUD operations vary depending on the type of NoSQL database (document-based, key-value, wide-column, or graph), the underlying principles remain the same. Below, I'll provide a general overview of how CRUD operations work in NoSQL databases:

1. Create (C):

- **Create (or Insert) operation** involves adding new data to the database. In NoSQL databases, this typically means creating a new record, document, or data item.

- **In NoSQL databases:**

  - In a document-based database (e.g., MongoDB), you would insert a new document into a collection.

  - In a key-value database (e.g., Redis), you would set a new key-value pair.

  - In a wide-column database (e.g., Cassandra), you would insert a new row or record into a column family.

  - In a graph database (e.g., Neo4j), you would create nodes and relationships.

2. Read (R):

- **Read operation** involves retrieving data from the database based on a specific key, identifier, or query.

- **In NoSQL databases:**

  - In a document-based database, you would query for documents using conditions.

  - In a key-value database, you would retrieve the value associated with a specific key.

  - In a wide-column database, you would read data from specific rows or column families.

  - In a graph database, you would traverse the graph to find nodes and relationships.

3. Update (U):

- **Update operation** involves modifying existing data in the database. This can include changing field values, adding new data, or removing data.

- **In NoSQL databases:**
  - In a document-based database, you would update document fields.
  - In a key-value database, you would set new values for a specific key.
  - In a wide-column database, you would update values in a row or column family.
  - In a graph database, you would modify properties of nodes or relationships.

4. Delete (D):

- **Delete operation** involves removing data from the database.

- **In NoSQL databases:**
  - In a document-based database, you would delete documents that meet certain criteria.
  - In a key-value database, you would remove a specific key-value pair.
  - In a wide-column database, you would delete rows or data associated with a specific column family.
  - In a graph database, you would delete nodes and relationships.

It's important to note that the exact syntax and methods for CRUD operations can vary between different NoSQL databases and even different database management systems within the same category. Therefore, it's essential to refer to the documentation and API specific to the NoSQL database you are working with.

CRUD operations are fundamental for interacting with NoSQL databases and are used in a wide range of applications, from web development and mobile apps to data analytics and IoT systems.

Transaction control language (commit, rollback, savepoint) with commands

Transaction Control Language (TCL) is a set of SQL commands that are used to manage transactions in a relational database management system (RDBMS). These commands allow you to control the start, end, and outcome of a transaction. The three primary TCL commands are `COMMIT`, `ROLLBACK`, and `SAVEPOINT`. Here's how each of them works:

1. **COMMIT:**

- `COMMIT` is used to permanently save the changes made during a transaction. Once a `COMMIT` command is executed, all changes (insertions, updates, deletions) made during the transaction are finalized and become permanent in the database. After a `COMMIT`, other database sessions can see the changes.

Syntax:

```sql
COMMIT;
```

2. **ROLLBACK:**

- `ROLLBACK` is used to undo the changes made during a transaction and return the database to the state it was in before the transaction started. This command is typically used when an error occurs during a transaction, and you want to discard any changes made so far.

Syntax:

```sql
ROLLBACK;
```

3. **SAVEPOINT:**

- `SAVEPOINT` is used to set a savepoint within a transaction. A savepoint is a point in the transaction to which you can later roll back. It allows you to partially undo the changes made since the savepoint while leaving the rest of the transaction intact.

Syntax (Setting a Savepoint):

```sql
SAVEPOINT savepoint_name;
```

Syntax (Rolling Back to a Savepoint):

```sql
ROLLBACK TO savepoint_name;
```

Example:

```sql
SAVEPOINT my_savepoint;
-- Make some changes
-- If something goes wrong, you can roll back to the savepoint
ROLLBACK TO my_savepoint;
```

TCL commands are essential for maintaining data consistency and integrity in a database. They ensure that changes made during a transaction are correctly and safely managed. When a `COMMIT` is issued, all changes are made permanent; when a `ROLLBACK` is issued, all changes are discarded, and when a `SAVEPOINT` is set, you can roll back to that point if needed, allowing for more fine-grained control over transaction management.

Data consistency with nosql dbs

Data consistency in NoSQL databases is a crucial aspect of ensuring the accuracy and reliability of your data. However, achieving consistency in NoSQL databases can be more complex than in traditional SQL databases due to their distributed and often schema-flexible nature. Here's an overview of data consistency in NoSQL databases:

1. **CAP Theorem:**

The CAP theorem, proposed by computer scientist Eric Brewer, is often used to frame discussions about data consistency in distributed systems, including NoSQL databases. It states that in a distributed system, you can achieve at most two out of the following three properties:

- **Consistency (C):** All nodes in the system see the same data at the same time.

- **Availability (A):** Every request to the system receives a response (not necessarily with the most recent data).

- **Partition Tolerance (P):** The system continues to function even in the presence of network partitions or message loss.

NoSQL databases often prioritize either consistency and partition tolerance (CA) or availability and partition tolerance (AP) based on the specific use case. However, you may have to make trade-offs between these properties.

2. **Eventual Consistency:**

Many NoSQL databases, particularly those designed for high availability and partition tolerance, provide eventual consistency. In an eventually consistent system, all nodes will eventually converge to a consistent state, but there may be temporary inconsistencies immediately after data changes. This approach balances availability and partition tolerance.

3. **Strong Consistency:**

Some NoSQL databases, like Google Cloud Spanner, offer strong consistency, ensuring that all nodes see the same data at the same time. However, achieving strong consistency may come at the cost of reduced availability or increased latency.

4. **Read and Write Concerns:**

NoSQL databases typically allow you to specify read and write concerns for individual operations. For example, you can choose to perform a "strong" read or a "causal" read, depending on your consistency requirements.

5. **Conflict Resolution:**

NoSQL databases provide mechanisms for resolving conflicts when multiple nodes update the same data concurrently. Conflict resolution strategies vary by database type and configuration.

6. **Quorums and Consensus Algorithms:**

Some NoSQL databases use quorum-based approaches and consensus algorithms like Raft or Paxos to ensure consistency. These mechanisms allow for distributed agreement on data changes.

7. **Replication and Data Distribution:**

Replication and data distribution strategies play a crucial role in data consistency. NoSQL databases may replicate data across multiple nodes or data centers to improve availability and redundancy.

8. **Transactions and Isolation Levels:**

NoSQL databases may support transactions and isolation levels to ensure data consistency for specific operations. These features vary between database systems.

9. **Schema Design:**

Well-designed schemas can help ensure data consistency in NoSQL databases. Consider your data access patterns and use schema design best practices to avoid potential data inconsistencies.

It's important to choose a NoSQL database and configure it according to your application's specific needs and requirements for consistency, availability, and partition tolerance. Understanding the CAP theorem and the trade-offs involved is essential for making informed decisions regarding data consistency in a NoSQL database.

Difference between nosql and traditional dbs with respect to crud operations

The key differences between NoSQL (Not Only SQL) databases and traditional relational databases (SQL databases) with respect to CRUD (Create, Read, Update, Delete) operations lie in the data models, schema flexibility, and the handling of these operations. Here's a comparison of how CRUD operations differ between the two types of databases:

1. Data Model:

- **SQL Databases:**

  - SQL databases follow a tabular, structured data model where data is organized into tables with fixed schemas.

  - Data must adhere to a predefined schema with well-defined relationships between tables.

- **NoSQL Databases:**

  - NoSQL databases employ various data models, including document-based, key-value, wide-column, and graph models.

- Data in NoSQL databases is often schema-flexible, allowing for different data structures and relationships within the same database.

2. Create (C):

- **SQL Databases:**

- In SQL databases, data must conform to the schema, meaning you need to define the structure of the data before inserting it.

- You perform inserts using SQL statements like `INSERT INTO`.

- **NoSQL Databases:**

- In NoSQL databases, creating data is generally more flexible, as you can insert data without needing to adhere to a predefined schema.

- How data is created depends on the specific NoSQL data model (e.g., `INSERT` in document-based databases, `SET` in key-value stores).

3. Read (R):

- **SQL Databases:**

- SQL databases use SQL SELECT statements for data retrieval, often involving complex joins and queries.

- Data must conform to the schema to be retrievable.

- **NoSQL Databases:**

- NoSQL databases offer different ways to retrieve data, including simple key-based lookups, query languages specific to the data model, and graph traversal for graph databases.

- Data retrieval can be faster and more straightforward, especially for use cases where schema flexibility is advantageous.

4. Update (U):

- **SQL Databases:**

- Updates in SQL databases require adherence to the schema, with changes made using SQL UPDATE statements.

- Transactions are often used to ensure data consistency during updates.

- **NoSQL Databases:**

- Updates in NoSQL databases can be more flexible, allowing for dynamic changes to data without modifying the entire schema.

- Some NoSQL databases support atomic, single-field updates, while others may require a replacement of the entire document or data structure.

5. Delete (D):

- **SQL Databases:**

- In SQL databases, data deletion is performed using SQL DELETE statements. Referential integrity and foreign key constraints are often enforced.

- CASCADE and SET NULL options may be used for cascading deletions.

- **NoSQL Databases:**

- Deletions in NoSQL databases can vary by data model. Key-value stores typically support simple key-based deletion, while other models may involve more complex deletion operations.

- NoSQL databases may not enforce foreign key constraints, requiring you to manage data integrity in the application layer.

The primary differences between NoSQL and traditional SQL databases in terms of CRUD operations are flexibility and performance. NoSQL databases are often preferred when flexibility in data modeling and faster read and write operations are needed, especially in scenarios where data structures are evolving or when dealing with large volumes of unstructured or semi-structured data. In contrast, traditional SQL databases excel in maintaining data integrity and enforcing strict schemas, which is beneficial for applications with complex relationships and well-defined structures. The choice between the two types of databases depends on the specific needs of your application.

DDL commands with Mongodb

MongoDB is a NoSQL database, and it uses a flexible schema-less data model. As a result, MongoDB does not have traditional Data Definition Language (DDL) commands like those found in relational databases. Instead, MongoDB uses collections to store data, and the structure of the documents within these collections can evolve over time. However, there are some operations that can be considered equivalent to DDL actions in the context of MongoDB:

1. **Creating a Collection:**

- MongoDB does not have a specific DDL command to create collections because collections are created automatically when you insert data into them. When you insert the first document into a

collection, MongoDB creates the collection. For example, if you want to create a collection called "users," you can simply start inserting documents into the "users" collection.

```javascript
db.users.insert({ name: "John", age: 30 });
```

2. **Dropping a Collection:**

- You can drop (delete) a collection in MongoDB using the `drop` method. This is similar to the SQL DDL command `DROP TABLE`.

```javascript
db.users.drop();
```

3. **Renaming a Collection (Starting from MongoDB 4.2):**

- You can rename a collection using the `renameCollection` command. This is similar to the SQL DDL command `ALTER TABLE`.

```javascript
db.adminCommand({ renameCollection: "mydb.myoldcollection", to: "mydb.mynewcollection" });
```

4. **Creating Indexes:**

- While not directly related to DDL, you can create indexes in MongoDB to improve query performance. Indexes are created using the `createIndex` method, which is similar in concept to the SQL DDL command `CREATE INDEX`.

```javascript
db.users.createIndex({ email: 1 });
```

5. **Data Validation Rules (Starting from MongoDB 3.2):**

- MongoDB introduced data validation rules that allow you to define a schema-like structure for your collections. While not equivalent to traditional SQL DDL, this feature provides some data validation capabilities.

```javascript
db.createCollection("contacts", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "email"],
      properties: {
        name: {
          bsonType: "string",
          description: "Must be a string and is required",
        },
        email: {
          bsonType: "string",
          pattern: "@mongodb\\.com$",
          description: "Must be a string and match the regular expression pattern",
        },
      },
    },
  },
});
```

It's important to note that MongoDB's schema-less design allows for greater flexibility in data modeling. While you can define data validation rules and create indexes, there is no strict schema enforcement, making it easier to adapt to changing data requirements. This flexibility is one of the key distinctions between MongoDB and traditional relational databases.

DML commands with Cassandra

Cassandra is a NoSQL database known for its distributed and scalable architecture. It uses the CQL (Cassandra Query Language) for data manipulation, and it provides a set of Data Manipulation Language (DML) commands similar to those used in traditional relational databases. Here are some of the common DML operations in Cassandra using CQL:

1. Insert (Create):

- The `INSERT` statement is used to add new rows (records) to a table. In Cassandra, each `INSERT` statement is typically associated with a particular partition key, which determines how the data is distributed across the cluster.

```sql
INSERT INTO tablename (column1, column2, ...) VALUES (value1, value2, ...);
```

2. Update (Update):

- The `UPDATE` statement is used to modify existing rows in a table. You can specify one or more columns to update and use conditions to target specific rows.

```sql
UPDATE tablename
SET column1 = new_value1, column2 = new_value2
WHERE condition;
```

3. Delete (Delete):

- The `DELETE` statement is used to remove rows from a table based on specific conditions.

```sql
DELETE FROM tablename WHERE condition;
```

4. Batch (Insert/Update/Delete in Bulk):

- Cassandra supports batch statements that allow you to group multiple `INSERT`, `UPDATE`, and `DELETE` operations into a single batch for execution.

```sql
BEGIN BATCH
  INSERT INTO tablename (column1, column2, ...) VALUES (value1, value2, ...);
  UPDATE tablename SET column1 = new_value1 WHERE condition;
  DELETE FROM tablename WHERE condition;
APPLY BATCH;
```

5. Select (Read):

- The `SELECT` statement retrieves data from one or more columns of a table based on specified conditions. You can perform various types of queries in Cassandra, including simple SELECTs, filtering, and aggregation.

```sql
SELECT column1, column2, ... FROM tablename WHERE condition;
```

6. TTL (Time To Live):

- Cassandra allows you to set a time-to-live (TTL) value for each insert or update operation. This value determines how long data remains in the database before it is automatically deleted.

```sql
INSERT INTO tablename (column1, column2) VALUES (value1, value2) USING TTL 3600;
```

7. Counter (Increment/Decrement):

- Cassandra supports counter data types that allow you to increment or decrement values in a column.

```sql
UPDATE tablename SET counter_column = counter_column + 1 WHERE condition;
```

It's important to note that Cassandra's data model and query language differ from traditional SQL databases in several ways. Cassandra is designed for high write throughput and scalability, making it well-suited for distributed, large-scale applications. Its unique partitioning, clustering, and replication strategies require careful schema design and consideration of access patterns.

Additionally, Cassandra's consistency level and eventual consistency model can be configured for each query, providing flexibility and control over data consistency. The choice of consistency level depends on the specific needs of your application.

Comparison of Mongodb and Cassandra

MongoDB and Apache Cassandra are both popular NoSQL databases, but they have different designs and use cases. Here's a comparison of MongoDB and Cassandra across various dimensions:

1. Data Model:

- **MongoDB:** MongoDB uses a document-based data model. Data is stored in flexible, JSON-like BSON documents, which can have varying structures within the same collection.

- **Cassandra:** Cassandra employs a wide-column data model. Data is organized into tables with predefined column families, but each row can have different columns, providing schema flexibility within the same column family.

2. Query Language:

- **MongoDB:** MongoDB uses a query language for document-based databases, and you can perform complex queries using a rich set of operators.

- **Cassandra:** Cassandra uses CQL (Cassandra Query Language), which is similar to SQL but tailored to its wide-column data model. It supports a subset of SQL features.

3. Consistency Model:

- **MongoDB:** MongoDB offers tunable consistency. You can configure read and write concerns to balance consistency and availability based on your application requirements.

- **Cassandra:** Cassandra provides tunable consistency through its quorum-based architecture. You can choose consistency levels for each read and write operation.

4. Scalability:

- **MongoDB:** MongoDB can scale horizontally by adding more servers to the cluster. It supports automatic sharding to distribute data across nodes.

- **Cassandra:** Cassandra is designed for high write and read throughput and provides linear scalability. It can handle large, distributed deployments.

5. Data Distribution and Replication:

- **MongoDB:** MongoDB provides replica sets for data replication, ensuring high availability and fault tolerance. Data is distributed based on replica set configuration.

- **Cassandra:** Cassandra uses peer-to-peer architecture for data distribution. Data is automatically partitioned and distributed across nodes using a consistent hashing algorithm.

6. Complex Queries:

- **MongoDB:** MongoDB is well-suited for a wide range of queries and supports aggregation and indexing for complex queries.

- **Cassandra:** Cassandra is optimized for simple read and write operations and may not be as suitable for complex analytical queries.

7. Use Cases:

- **MongoDB:** MongoDB is suitable for various use cases, including content management systems, catalogs, user profiles, and applications requiring flexible, semi-structured data.

- **Cassandra:** Cassandra is ideal for time-series data, event logging, sensor data, real-time analytics, and applications requiring high write and read scalability.

8. Schema Evolution:

- **MongoDB:** MongoDB is flexible with schema changes, allowing for dynamic schema evolution and addition of new fields.

- **Cassandra:** Cassandra is more rigid in terms of schema changes, and schema alterations often require planning and migration.

9. Ecosystem:

- **MongoDB:** MongoDB has a mature ecosystem with a wide range of drivers, tools, and cloud-based services, making it developer-friendly.

- **Cassandra:** Cassandra has a robust ecosystem, particularly for large-scale, distributed, and real-time applications.

10. ACID Transactions:

- **MongoDB:** MongoDB introduced multi-document transactions in version 4.0, providing ACID compliance for multiple operations within a single document.

- **Cassandra:** Cassandra focuses on high availability and partition tolerance, and it does not provide ACID transactions in the traditional sense.

In summary, MongoDB and Cassandra have different strengths and weaknesses. MongoDB is a good fit for applications requiring flexible data models and complex queries, while Cassandra excels in write-intensive, highly scalable, and distributed scenarios. The choice between the two depends on your specific use case and requirements for scalability, data modeling, and data consistency.