DROP KEYSPACE vitb;

❖ Cassandra CRUD operations on table(also called as column family)

1

- ➢ CRUD stands for create, read, update, delete
- ➢ Create
  - You can create a table using the command CREATE TABLE. Given below is the syntax for creating a table.

    Syntax
    CREATE (TABLE | COLUMNFAMILY) <tablename>
    ('<column-definition>' , '<column-definition>')
    (WITH <option> AND <option>)
  - Example

```
CREATE TABLE emp(
  emp_id int PRIMARY KEY,
  emp_name text,
  emp_city text,
  emp_sal varint,
  emp_phone varint
  );
```

- ➢ Read
  - SELECT clause is used to read data from a table in Cassandra. Using this clause, you can read a whole table, a single column, or a particular cell. Given below is the syntax of SELECT clause.

    SELECT FROM <tablename>
  - Example

```
Select * from emp;
Select * from emp where emp_id=123;
Select emp_sal from emp where emp_city='Hyd';
```

- ➢ Update
  - UPDATE is the command used to update data in a table. The following keywords are used while updating data in a table −

    Where − This clause is used to select the row to be updated.
    Set − Set the value using this keyword.
    Must − Includes all the columns composing the primary key.

  - While updating rows, if a given row is unavailable, then UPDATE creates a fresh row. Given below is the syntax of UPDATE command −

    UPDATE <tablename>
    SET <column name> = <new value>

    <column name> = <value>....
    WHERE <condition>
  - Example
    - UPDATE emp SET emp_city='Delhi',emp_sal=50000 WHERE emp_id=2;
- ➢ Delete
  - You can delete data from a table using the command DELETE. Its syntax is as follows −

    DELETE FROM <identifier> WHERE <condition>;
  - Example
    - DELETE emp_sal FROM emp WHERE emp_id=3;

2 In Apache Cassandra, replication strategies determine how data is distributed across the nodes in a cluster and how data redundancy is maintained for fault tolerance and high availability. Cassandra provides several replication strategies to meet different use cases and requirements. Let's describe some of the commonly used replication strategies along with examples:

1. **SimpleStrategy**:

- SimpleStrategy is the default replication strategy in Cassandra. It distributes data across the cluster nodes in a ring topology, where each node acts as a replica for a certain range of data.

- With SimpleStrategy, all replicas for a given row are placed on nodes determined by the partitioner's token range calculation.

- This strategy is typically used for single data center deployments or development environments.

Example:

```yaml
CREATE KEYSPACE example_keyspace
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3};
```

In this example, a keyspace named "example_keyspace" is created with SimpleStrategy replication, specifying a replication factor of 3. This means that each piece of data will be replicated across 3 nodes in the cluster.

2. **NetworkTopologyStrategy**:

- NetworkTopologyStrategy allows users to define how replicas are distributed across multiple data centers and racks within a data center.

- It offers more control over replication placement by specifying the number of replicas in each data center and rack.

- NetworkTopologyStrategy is suitable for multi-data center deployments to ensure fault tolerance and data locality.

Example:

```yaml
CREATE KEYSPACE example_keyspace
WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1': 3, 'DC2': 2};
```

In this example, a keyspace named "example_keyspace" is created with NetworkTopologyStrategy replication. There are 3 replicas in data center 'DC1' and 2 replicas in data center 'DC2'. This configuration ensures fault tolerance and data locality across multiple data centers.

3. **LocalStrategy**:

- LocalStrategy is used when you want to restrict replication to a single data center but have multiple replicas within that data center.

- It is useful when you have a multi-node cluster within a single data center and want to ensure fault tolerance without replicating data across multiple data centers.

Example:

```yaml
CREATE KEYSPACE example_keyspace
WITH replication = {'class': 'LocalStrategy', 'replication_factor': 3};
```

In this example, a keyspace named "example_keyspace" is created with LocalStrategy replication, specifying a replication factor of 3. This means that each piece of data will be replicated across 3 nodes within the same data center.

These are some of the replication strategies available in Cassandra, each catering to different deployment scenarios and requirements for fault tolerance, data locality, and scalability. The choice of replication strategy depends on factors such as the number of data centers, fault tolerance requirements, and the desired level of consistency and performance.

3a

❖ **MongoDB datatypes**

- **String** − This is the most commonly used datatype to store the data. The string in MongoDB must be UTF-8 valid.

- **Integer** − This type is used to store a numerical value. Integer can be 32-bit or 64-bit, depending upon your server.
- **Boolean** − This type is used to store a boolean (true/ false) value.
- **Double** − This type is used to store floating point values.
- **Min/ Max keys** − This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** − This type is used to store arrays or lists or multiple values into one key.
- **Timestamp** − timestamp. This can be handy for recording when a document has been modified or added.
- **Object** − This datatype is used for embedded documents.
- **Null** − This type is used to store a Null value.
- **Symbol** − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** − This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating an object of Date and passing a day, month, or year into it.
- **Object ID** − This datatype is used to store the document's ID.
- **Binary data** − This datatype is used to store binary data.
- **Code** − This datatype is used to store JavaScript code in the document.
- **Regular expression** − This datatype is used to store regular expressions.

MongoDB is a NoSQL database that stores data in a flexible, JSON-like format called BSON (Binary JSON). BSON supports a variety of data types, allowing for rich and dynamic data modeling within documents. Some commonly used data types in MongoDB documents include:

1. **String**:
   - Represents a sequence of characters. Strings are commonly used for storing textual data.
   - Example:
   ```json
   {
     "name": "John Doe",
     "email": "john@example.com"
   }
   ```

2. **Integer**:
   - Represents a whole number without a fractional component.
   - Example:
   ```json
   {
     "age": 30,
     "quantity": 5
   }
   ```

3. **Double**:
   - Represents a floating-point number with a fractional component.
   - Example:
   ```json
   {
     "price": 10.99,
     "weight": 2.5
   }
   ```

```

4. **Boolean**:

- Represents a logical value of true or false.

- Example:

```json
{
  "is_active": true,
  "is_admin": false
}
```

5. **Date**:

- Represents a date and time value. Dates are stored as milliseconds since the Unix epoch (January 1, 1970).

- Example:

```json
{
  "created_at": ISODate("2024-03-21T08:00:00Z"),
  "updated_at": ISODate("2024-03-21T12:30:00Z")
}
```

6. **Array**:

- Represents a list of values. Arrays can contain elements of different data types, including other arrays or documents.

- Example:

```json
{
  "tags": ["mongodb", "database", "nosql"],
  "scores": [85, 90, 75, 88]
}
```
```

```

7. **Object (Embedded Document)**:

- Represents a nested document within a MongoDB document. Embedded documents allow for hierarchical data structures.

- Example:

```json
{
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "zipcode": "10001"
  }
}
```

8. **ObjectId**:

- Represents a unique identifier for documents in a collection. ObjectId values are generated automatically by MongoDB.

- Example:

```json
{
  "_id": ObjectId("62fea5c934f51c71f14f502d"),
  "name": "Alice"
}
```

These are some of the commonly used data types in MongoDB documents. MongoDB's flexible schema allows for easy integration of various data types within a document, making it suitable for a wide range of use cases and data models.
```

3b

Optimizing performance in MongoDB involves various strategies aimed at improving query execution, data retrieval, storage efficiency, and overall system throughput. Here are several strategies for optimizing performance in MongoDB:

1. **Indexing**:

- Proper indexing can significantly enhance query performance by allowing MongoDB to efficiently locate and retrieve documents based on indexed fields.

- Create indexes on fields frequently used in queries, especially those involved in equality matches, range queries, sorting, and data retrieval operations.

- Consider using compound indexes for queries that involve multiple fields.

- Regularly review and optimize indexes based on query patterns and workload changes.

2. **Schema Design**:

- Designing an appropriate schema tailored to the application's requirements can have a substantial impact on performance.

- Denormalize data when necessary to reduce the need for complex joins and improve query performance.

- Use embedded documents and arrays strategically to minimize the number of read operations required to retrieve related data.

- Balance between read and write optimizations based on the application's workload characteristics.

3. **Query Optimization**:

- Optimize queries by ensuring they leverage indexes effectively and minimize unnecessary data retrieval.

- Use projection to retrieve only the required fields rather than fetching entire documents.

- Utilize aggregation pipelines for complex data processing tasks instead of multiple individual queries.

- Employ query hints and query plan analysis tools to identify and resolve performance bottlenecks.

4. **Sharding**:

- Sharding distributes data across multiple shards (physical servers or replica sets) to horizontally scale MongoDB deployments.

- Choose appropriate sharding keys to evenly distribute data and queries across shards, preventing hotspots and uneven workload distribution.

- Monitor and rebalance shards periodically to maintain optimal performance and data distribution.

5. **Replication**:

- Replication improves fault tolerance, data availability, and read scalability by maintaining multiple copies (replicas) of data across different nodes.

- Configure replica sets with an appropriate number of nodes to ensure high availability and durability.

- Use read preferences to route read operations to specific replica set members based on latency, consistency requirements, and workload characteristics.

6. **Storage Optimization**:

- Optimize storage configuration, including disk layout, filesystem settings, and storage engine options, to maximize I/O performance and storage efficiency.

- Choose the appropriate storage engine (WiredTiger or mmapv1) based on workload requirements, data access patterns, and performance considerations.

- Monitor storage usage, fragmentation, and disk I/O metrics to identify and address storage-related performance issues proactively.

7. **Caching**:

- Utilize caching solutions like MongoDB's in-memory storage engine (MMAPv1) or external caching systems (e.g., Redis, Memcached) to cache frequently accessed data and reduce query latency.

- Implement application-level caching mechanisms to cache query results, computed data, or frequently accessed documents to minimize database load and improve response times.

8. **Monitoring and Performance Tuning**:

- Monitor MongoDB performance metrics, including query execution times, index usage, server resource utilization (CPU, memory, disk I/O), and operation throughput.

- Use MongoDB's built-in monitoring tools (e.g., mongostat, mongotop) or third-party monitoring solutions to identify performance bottlenecks and optimize resource allocation.

- Continuously tune and optimize MongoDB configuration parameters (e.g., cache size, connection pool settings, write concern) based on workload patterns, performance metrics, and hardware capabilities.

By implementing these strategies and continuously monitoring performance metrics, MongoDB deployments can achieve optimal performance, scalability, and reliability to meet the demands of modern applications.

4a

1. **Single Field Index**:
   - Indexes a single field of a document.
   - Helpful for fetching data in ascending or descending order.
   - Example: `` `db.students.createIndex({studentsId: 1})` ``
2. **Compound Index**:
   - Combines multiple fields for indexing.
   - Useful for queries involving multiple fields.
   - Example: `` `db.students.createIndex({studentAge: 1, studentName: 1})` ``
3. **Multikey Index**:
   - Indexes values stored in arrays.
   - Automatically created by MongoDB for fields containing array values.
   - Example: `` `db.students.createIndex({skillsets: 1})` ``
4. **Geospatial Indexes**:
   - Supports querying geospatial data for location-based operations.
   - Two types: 2d indexes and 2d sphere indexes.
   - Example: `` `db.industries.createIndex({location: "2dsphere"})` ``
5. **Text Index**:
   - Enables full-text search on string content.
   - Searches for string content in specified fields or collections.
   - Example: `` `db.accessories.createIndex({name: "text", description: "text"})` ``
6. **Hash Index**:
   - Maintains entries with hashes of indexed field values.
   - Useful for even distribution of data across a sharded cluster.
   - Example: `` `db.collection.createIndex({_id: "hashed"})` ``
7. **Wildcard Index**:
   - Indexes unknown or arbitrary fields in documents.
   - Supports queries for fields that are not explicitly indexed.
   - Example: `` `db.book.createIndex({"authorTags.$**": 1})` ``

Each index type has its own purpose and use cases. By strategically creating and utilizing indexes based on the application's query patterns and workload characteristics, developers can significantly enhance the performance and efficiency of MongoDB databases.

4b

To create a CSV file from a MongoDB collection, you can use various methods, including MongoDB Compass, MongoDB shell (mongo), or programming languages like Python. Here, I'll provide steps for using MongoDB Compass and Python:

Using MongoDB Compass:

1. **Connect to MongoDB**: Open MongoDB Compass and connect to your MongoDB deployment.

2. **Navigate to Collection**: Select the collection from which you want to export data.

3. **Export Data**: Click on the "Export Data" dropdown menu in the top-right corner of the collection view.

4. **Select Export Option**: Choose "Export Collection" to export the entire collection or "Export Pipeline" if you want to export data based on a query or aggregation pipeline.

5. **Choose File Format**: Select "CSV" as the export format.

6. **Specify Export Location**: Choose the location where you want to save the CSV file and click "Export".

7. **Configure Export Options**: MongoDB Compass allows you to configure additional export options such as field selection, data type conversion, and CSV options. Adjust these settings as needed and proceed with the export.

8. **Export**: Click "Export" to initiate the export process. MongoDB Compass will generate the CSV file containing the exported data from the MongoDB collection.

Using Python:

You can also use Python to export data from a MongoDB collection to a CSV file. Here's a basic example using the pymongo library:

```python
```

```
from pymongo import MongoClient
import csv

# Connect to MongoDB
client = MongoClient('mongodb://localhost:27017/')
db = client['your_database']
collection = db['your_collection']

# Fetch data from MongoDB collection
cursor = collection.find()

# Specify fields to include in CSV
fields = ['field1', 'field2', 'field3'] # Adjust field names as needed

# Write data to CSV file
with open('output.csv', 'w', newline='') as csvfile:
    writer = csv.DictWriter(csvfile, fieldnames=fields)
    writer.writeheader()
    for document in cursor:
        writer.writerow(document)
```

Replace `'mongodb://localhost:27017/'`, `'your_database'`, and `'your_collection'` with appropriate values for your MongoDB connection and collection. Adjust the `fields` list to include the fields you want to include in the CSV file.

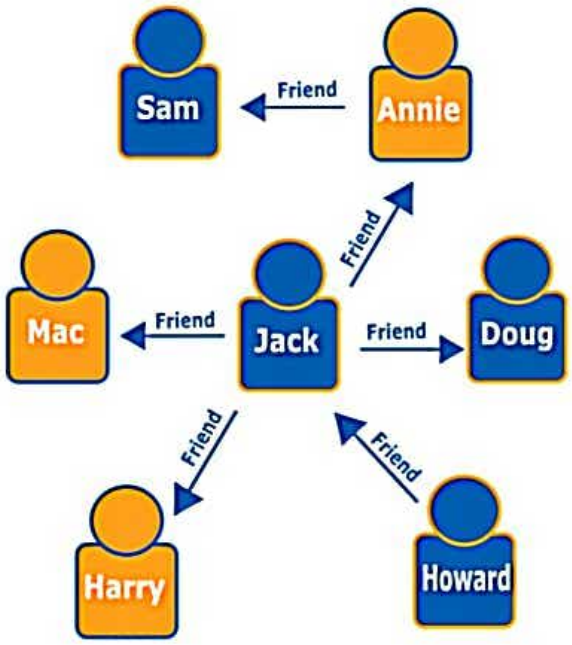
This Python script connects to MongoDB, fetches data from the specified collection, and writes it to a CSV file named `output.csv` using the `csv.DictWriter` class. Each document from the MongoDB collection is written as a row in the CSV file, with fields corresponding to the specified field names.

Choose the method that best fits your requirements and preferences to create a CSV file from a MongoDB collection.

5a

❖ What Is a Graph Database?

- ➢ Graph databases are purpose-built to store and navigate relationships. Relationships are first-class citizens in graph databases, and most of the value of graph databases is derived from these relationships. Graph databases use nodes to store data entities, and edges to store relationships between entities. An edge always has a start node, end node, type, and direction, and an edge can describe parent-child relationships, actions, ownership, and the like. There is no limit to the number and kind of relationships a node can have.
- ➢ A graph in a graph database can be traversed along specific edge types or across the entire graph. In graph databases, traversing the joins or relationships is very fast because the relationships between nodes are not calculated at query times but are persisted in the database. Graph databases have advantages for use cases such as social networking, recommendation engines, and fraud detection, when you need to create relationships between data and quickly query these relationships.
- ➢ The following graph shows an example of a social network graph. Given the people (nodes) and their relationships (edges), you can find out who the "friends of friends" of a particular person are—for example, the friends of Howard's friends.



Graph databases offer several advantages over other types of NoSQL databases, such as document-oriented or key-value stores:

1. **Rich Data Model**: Graph databases provide a rich data model that allows for the representation of complex relationships between data entities. This makes them well-suited for scenarios where relationships are as important as the data itself, such as social networks, recommendation systems, fraud detection, and network analysis.
2. **Native Graph Processing**: Graph databases are optimized for traversing and querying graph structures. They use graph-specific query languages (such as Cypher for Neo4j) that are tailored for graph traversal and pattern matching. This allows for efficient querying of highly interconnected data and complex graph algorithms.
3. **Schema Flexibility**: Graph databases typically have flexible schemas that can evolve over time without requiring predefined schema definitions. This allows for agile development and adaptation to changing business requirements without the need for extensive schema migrations.
4. **Query Performance**: Graph databases excel at querying relationships and patterns within the data. They can efficiently traverse large graphs to find connections and relationships between nodes, even in highly connected datasets. This makes them well-suited for use cases requiring complex queries and graph analytics.
5. **Graph Analytics**: Graph databases support advanced graph analytics and algorithms for tasks such as pathfinding, community detection, centrality analysis, and recommendation systems. These built-in graph processing capabilities enable powerful data insights and decision-making.

Data modeling in Neo4j revolves around creating a graph schema that defines the structure of the graph database, including nodes, relationships, and properties. The principles of data modeling in Neo4j are centered on representing real-world entities and their relationships as nodes and edges in a graph. Here's a discussion of the principles of data modeling in Neo4j and how graph schemas are designed:

5b

1. **Identify Entities and Relationships**:
   - Begin by identifying the entities (nodes) in your domain and the relationships (edges) between them. Entities can represent various real-world objects, such as people, places, events, products, or concepts, while relationships capture how these entities are connected or related.
2. **Define Node Labels**:
   - Node labels categorize nodes into different types based on their roles or characteristics in the domain. Labels provide semantic meaning to nodes and help organize the graph database. Each node can have one or more labels associated with it. For example, in a social network graph, nodes representing users might be labeled as "Person" or "User."
3. **Determine Relationship Types**:
   - Relationship types define the nature of connections between nodes. They represent the semantics of the relationships and convey meaningful information about how entities are related. Relationship types should be descriptive and intuitive, reflecting the real-world interactions between entities. For instance, in a social network graph, relationship types could include "FRIENDS_WITH," "FOLLOWS," or "LIKES."
4. **Model Properties**:
   - Properties provide additional attributes or metadata associated with nodes and relationships. They capture details about entities and relationships, such as names, dates, quantities, or any other relevant information. Properties can be indexed for efficient querying and retrieval of data. It's essential to determine which properties are crucial for describing nodes and relationships accurately.
5. **Design Graph Schema**:
   - Based on the identified entities, relationships, labels, and properties, design the graph schema using the Cypher query language. Define node labels, relationship types, and property keys to create the schema. The schema serves as a blueprint for structuring and organizing the graph database. It provides a clear understanding of the data model and guides data insertion, querying, and maintenance operations.

querying, and maintenance operations.

6. **Consider Graph Constraints**:
   - Neo4j allows you to define constraints to enforce data integrity and ensure consistency within the graph database. Constraints can include uniqueness constraints to enforce uniqueness of node or relationship properties, existence constraints to ensure certain properties exist, and relationship constraints to define rules for relationships between nodes.
7. **Iterate and Refine**:
   - Data modeling in Neo4j is an iterative process that involves continuous refinement based on evolving requirements and feedback. As you gain insights into the domain and use cases, refine the graph schema to accommodate new entities, relationships, or properties. Regularly review and optimize the schema to improve performance, scalability, and maintainability.

Overall, the principles of data modeling in Neo4j emphasize representing real-world entities and their relationships in a graph structure, designing intuitive node labels and relationship types, and defining properties to capture relevant information. A well-designed graph schema facilitates efficient querying, navigation, and analysis of data in Neo4j graph databases.

6a

❖ **CQL DataTypes**

➢ CQL stands for Cypher Query Language. It is a query language for Neo4j just like SQL is a query language for Oracle or MySQL.

➢ Neo4j CQL Features

- CQL is a query language for Neo4j Graph Database.
- Is a declarative pattern-matching language.
- The syntax of CQL is same like SQL syntax.
- Syntax of CQL is very simple and in human readable format.

➢ Similarity between Oracle SQL and Neo4j CQL

- Oracle and Neo4j CQL both has simple commands to do database operations.
- Both support clauses like WHERE, ORDER BY, etc., to simplify complex queries.
- Oracle and Neo4j CQL supports some Relationship Functions and functions such as String, Aggregation.

➢ The Neo4j CQL data types are similar to Java language data types. They are used to define properties of a node or a relationship.

➢ A list of Neo4j CQL data types:

CQL Data Type	Usage
Boolean	It is used to represent Boolean literals: True, False.
byte	It is used to represent 8-bit integers.
short	It is used to represent 16-bit integers.
int	It is used to represent 32-bit integers.
long	It is used to represent 64-bit integers.
float	Float is used to represent 32-bit floating-point numbers.
double	Double is used to represent 64-bit floating-point numbers.
char	Char is used to represent 16-bit characters.

String	String is used to represent strings.

❖ **Neo4j CQL Operators**

➢ Neo4j CQL Operators can be categorized in following types:

- Mathematical Operators: i.e. +, -, *, /, %, ^
- Comparison Operators: i.e. +, <>, <, >, <=, >=
- Boolean Operators: i.e. AND, OR, XOR, NOT
- String Operators: i.e. +
- List Operators: i.e. +, IN, [X], [X?..Y]
- Regular Expression: i.e. =-
- String matching: i.e. STARTS WITH, ENDS WITH, CONSTRAINTS

➢ Let's see the two most used Neo4j CQL Operators:

➢ Boolean Operators

Following is a list of Boolean operators which are used in Neo4j CQL WHERE clause to support multiple conditions:

Boolean Operators	Description
AND	It is a neo4j CQL keyword to support AND operation. It is like SQL AND operator.
OR	It is a Neo4j CQL keyword to support OR operation. It is like SQL AND operator.
NOT	It is a Neo4j CQL keyword to support NOT operation. It is like SQL AND operator.
XOR	It is a Neo4j CQL keyword to support XOR operation. It is like SQL AND operator.

Comparison Operators

➢ A list of Neo4j CQL Comparison Operators used with WHERE clause:

Boolean operators	Description
=	It is a Neo4j CQL "equal to" operator.
< >	It is a Neo4j CQL "not equal to" operator.
<	It is a Neo4j CQL "less than" operator.
>	It is a Neo4j CQL "greater than" operator.
<=	It is a Neo4j CQL "less than or equal to" operator.
> =	It is a Neo4j CQL"greater than or equal to" operator.

6b Graph databases offer a range of features that make them uniquely suited for modeling and querying highly connected data. Let's discuss these features with a suitable example:

Feature 1: Graph Data Model

- Graph databases use a graph data model consisting of nodes (vertices) and relationships (edges) to represent and store data.
- Example: Consider a social network where users are represented as nodes, and relationships between users such as "FRIENDS_WITH" or "FOLLOWS" are represented as edges. Each node represents a user profile, and edges capture relationships between users based on interactions.

Feature 2: Highly Connected Data

- Graph databases excel at managing highly connected data where relationships between entities are crucial.
- Example: In a recommendation system, nodes represent users and items (e.g., movies, books), and edges represent interactions such as "LIKES" or "PURCHASED". By analyzing the graph structure, the system can recommend items to users based on their preferences and connections with similar users.

Feature 3: Schema Flexibility

- Graph databases typically offer schema flexibility, allowing developers to evolve the data model without rigid schema constraints.
- Example: In a knowledge graph representing academic publications, nodes can represent authors, papers, conferences, and topics. New types of nodes and relationships can be added dynamically as new research fields emerge without needing to modify the existing schema.

Feature 4: Graph Query Language

- Graph databases provide a query language optimized for traversing and querying graph structures.
- Example: In Neo4j, Cypher is a graph query language that allows users to express graph patterns and perform operations such as finding paths, filtering nodes based on properties, and aggregating results. For instance, a query can find the shortest path between two users in a social network graph.

Feature 5: Native Graph Processing

- Graph databases are optimized for native graph processing, enabling efficient storage and retrieval of graph data.
- Example: In a logistics network, nodes represent locations (e.g., warehouses, distribution centers), and edges represent transportation routes. By storing the network as a graph, the system can quickly find optimal routes for delivering goods based on factors like distance, cost, and available capacity.

Feature 6: Real-time Query Performance

- Graph databases offer real-time query performance for graph-related operations, enabling interactive exploration and analysis of data.
- Example: In a fraud detection system, nodes represent customers and transactions, and edges represent financial interactions. By analyzing the transaction graph in real-time, the system can detect suspicious patterns such as money laundering or fraudulent activities.

Feature 7: Scalability and Performance

- Graph databases are designed to scale horizontally and vertically to handle large-scale graphs and growing data volumes.
- Example: In a recommendation engine for e-commerce, as the number of users and items grows, the graph database can scale by adding more nodes and edges while maintaining query performance for personalized recommendations.