USN | | | | | | | | | | |

Internal Assessment Test 3 – December 2024

Sub:	NOSQL database					Sub Code:	21CS745	Branch:	CSE
Date:	/12/24	Duration:	90 mins	Max Marks:	50		Sem/Sec:7/ A/B/C		OBE

Answer any FIVE FULL Questions

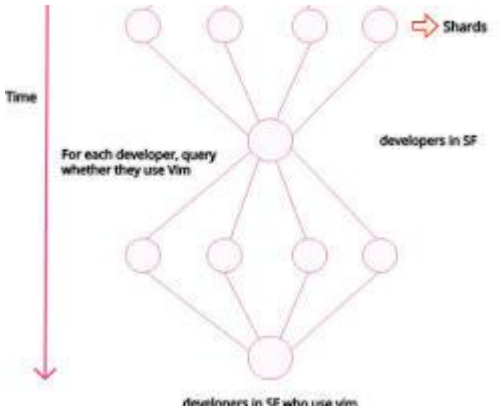
		MARKS	CO	RBT
1	Provide a clear explanation, accompanied by a well-structured diagram, illustrating three methods of scaling a graph database.	[10]	CO3	L2
2	Explain the process of adding indexes for nodes in a Neo4j database, and provide a detailed explanation of how to add incoming and outgoing relationships.	[10]	CO3	L2
3	Write Cypher queries for i) Consider Barbara is connected to Jill by two distinct paths; How to find all these paths and the distance between Barbara and Jill along those different paths? (4 marks) ii) Find all outgoing relationships with the type of FRIEND, and return the friends' names of "Ajay" for greater depth (3 marks) iii) Find relationships where a particular relationship property exists. Filter on the properties of relationships and query if a property exists or not.(3 marks)	[10]	CO3	L3
4	Clarify the principles of consistency and availability in MongoDB, with the help of clear diagrams. Consistency (5 marks) Availability (5 marks)	[10]	CO4	L2
5	Explain the concept of horizontal sharding in MongoDB, i) Covering the procedure for adding a new node to an existing replica set (4+1 diagram), ii) Detailing the setup where each shard is a replica set- 4+1 diagram	[10]	CO4	L3
6	Provide a brief overview of document databases and highlight their differences from SQL and key-value databases.	[10]	CO4	L2

CI **CCI** **HOD**

1. Graph databases, like Neo4j, are designed to handle complex relationships between data. As data grows, it's crucial to scale these databases efficiently. Here are three common methods:

1. Vertical Scaling (Sharding):

- **Concept: Partitioning the graph into smaller subgraphs based on node properties or relationships.**
- **Diagram:**

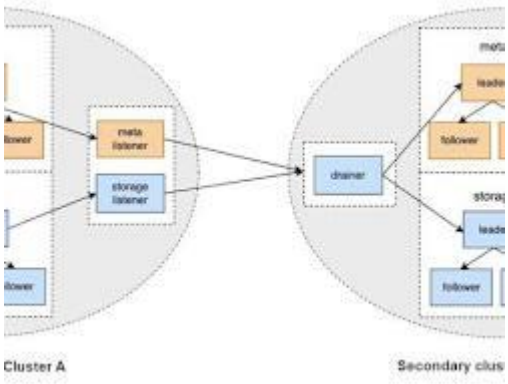


Graph Database Sharding Diagram

- **Implementation: Neo4j offers built-in sharding capabilities. You define rules to distribute nodes and relationships across different partitions.**

2. Horizontal Scaling (Replication):

- **Concept: Creating multiple copies of the entire graph database on different servers.**
- **Diagram:**

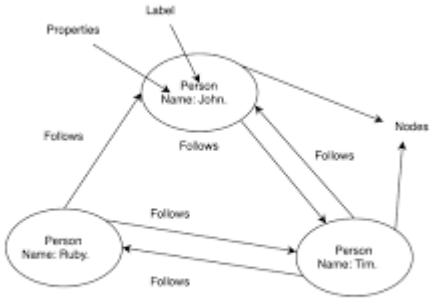


Graph Database Replication Diagram

- **Implementation: Neo4j supports replication through read replicas. This allows read operations to be distributed across multiple servers, improving performance.**

3. Hybrid Approach:

- **Concept: Combining sharding and replication for optimal scalability.**
- **Diagram:**



Graph Database Hybrid Scaling Diagram

- **Implementation: You can shard the graph and then replicate each shard for high availability and read scalability.**

2. Adding Indexes:

Create an Index: Use the `CREATE INDEX` statement, specifying the label and property to index.

Cypher

```
CREATE INDEX ON :Person(name)
```

1.

Create a Unique Constraint: Ensure uniqueness for a property using `CREATE CONSTRAINT`.

Cypher

```
CREATE CONSTRAINT ON (p:Person) ASSERT p.name IS UNIQUE
```

2.

Adding Relationships:

Create a Relationship: Use the `MATCH` clause to find nodes, then create a relationship between them using the `CREATE` clause.

Cypher

```
MATCH (person1:Person {name: 'Alice'}), (person2:Person {name: 'Bob'})

CREATE (person1)-[:KNOWS]->(person2)
```

1.

Add Properties to Relationships: Assign properties to the relationship during creation.

Cypher

```
MATCH (person1:Person {name: 'Alice'}), (person2:Person {name: 'Bob'})

CREATE (person1)-[:KNOWS {since: 2022}]->(person2)
```

2.

Add Incoming and Outgoing Relationships: Use the `<-[]-` syntax for incoming relationships and `-[]->` for outgoing relationships.

Cypher

```
MATCH (person:Person {name: 'Alice'})

RETURN (person)<-[:KNOWS]-() // Incoming relationships

RETURN (person)-[:KNOWS]->() // Outgoing relationships
```

3.

3. Cypher Queries

(i) Finding Paths and Distances:

Cypher

```
MATCH p=(b:Person {name: 'Barbara'})-[*]-(j:Person {name: 'Jill'})

RETURN p, length(p) AS distance
```

(ii) Finding Friends of "Ajay":

Cypher

```
MATCH (a:Person {name: 'Ajay'})-[:FRIEND]->(f)

RETURN f.name
```

(iii) Finding Relationships with a Property:

Cypher

```
MATCH ()-[r]-()

WHERE EXISTS(r.property_name)

RETURN r
```

4: Consistency and Availability in MongoDB

Consistency:

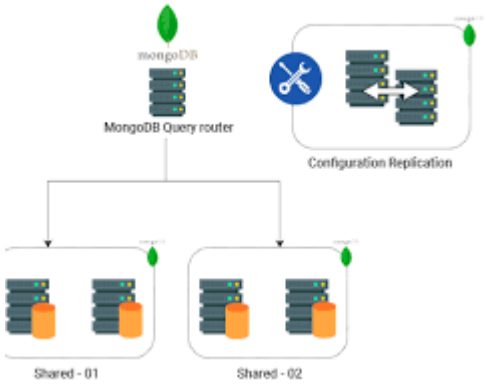
- **Concept: All nodes in a replica set have the same data.**
- **Diagram:**

MongoDB Consistency Diagram

- **Implementation: MongoDB uses the write concern option to enforce consistency.**

Availability:

- **Concept: The database is always accessible, even if some nodes fail.**
- **Diagram:**



MongoDB Availability Diagram

- **Implementation: MongoDB uses replica sets to achieve high availability.**

5: Horizontal Sharding in MongoDB

(i) Adding a New Node to an Existing Replica Set:

1. **Add the new node: Use the `rs.add()` command.**
2. **Initialize the new node: Run `rs.initiate()` on the primary node.**

(ii) Setup where each shard is a replica set:

1. **Create replica sets: Create multiple replica sets, each containing multiple nodes.**
2. **Configure sharding: Use the `sh.addShard()` command to add each replica set as a shard.**

6: Document Databases vs. SQL and Key-Value Databases

- **Document Databases: Store data in flexible, JSON-like documents.**
- **SQL Databases: Store data in tables with fixed schemas.**
- **Key-Value Databases: Store data as key-value pairs.**

Differences:

- **Schema: Document databases are schema-less or have flexible schemas, while SQL databases have rigid schemas.**
- **Data Model: Document databases are better suited for complex, hierarchical data, while key-value databases are good for simple data.**
- **Querying: Document databases support flexible queries based on document content, while key-value databases are limited to key-based lookups.**