USN										

Internal Assessment Test 3 – May 2023 Scheme and solution

Sub:	NOSQL database				Sub Code:	18CS823	Branch:	CSE
Date:	13/05/23	Duration:	90 mins	Max Marks: 50			Sem/Sec:8 A/B/C	90 mins

scheme

		MARKS	CO	RBT
1	Explain Graph database with neat diagram. Identify places where graph database is applicable.	[10]	CO2	L2
2	Describe the procedure to add indexing for the nodes in the NEo4J database. Elaborate on addition of incoming and outgoing links.	[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 "AAAAAA" for greater depth (3 marks) iv) 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	Explain scaling in Document database and Graph database.	[10]	CO2	L2
5	Write equivalent MongoDB and SQL queries for i) Find all orders of the customers with order ID and date on which the customer placed the order.(3 marks) ii) Selecting the orders for a single customer Id.(3 marks) iii) All the orders where one of the items ordered has a name like Vanilla(4 marks).	[10]	CO2	L3
6	Briefly describe the relation in Graph databases with neat diagram.	[10]	CO2	L2

Solution:

Q1. Explain Graph database with neat diagram. Identify places where graph database is applicable.

- Graph databases allow you **to store entities and relationships between these entities. Entities are also known as nodes, which have properties**. Think of a **node as an instance of an object in the application.**
- **Relations are known as edges that can have properties. Edges have directional significance;** nodes are organized by relationships which allow you to find interesting patterns between the nodes.
- **The organization of the graph lets the data to be stored once and then interpreted in different ways based on relationships.**
- In the example graph in Figure 11.1, we see a bunch of nodes related to each other. Nodes are entities that have properties, such as name. The node of Martin is actually a node that has property of name set to Martin.

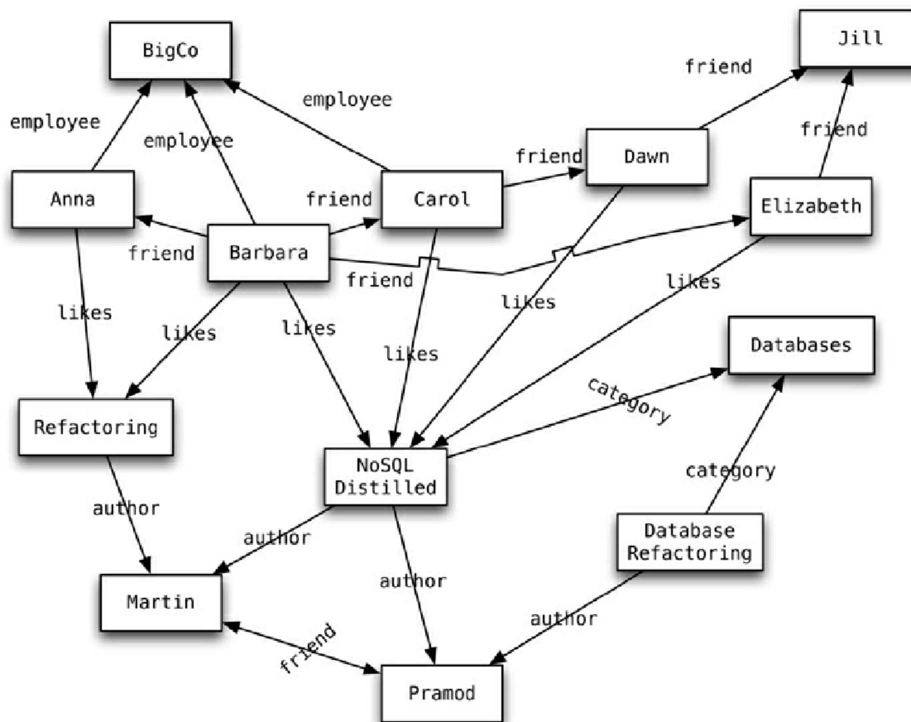


Figure 11.1. An example graph structure

We also see that edges have types, such as likes, author, and so on. These properties let us organize the nodes; for example, the nodes Martin and Pramod have an edge connecting them with a relationship type of friend.

- **Edges can have multiple properties**. We can assign a property of sinceon the friend relationship type between Martin and Pramod.
- **Relationship types have directionalsignificance; the friend relationship type is bidirectional but 'likes' is not**. When Dawn likesNoSQL Distilled, it does not automatically mean NoSQL Distilled likes Dawn.
- Once we have a graph of these nodes and edges created, we can query the graph in many ways, such as "get all nodes employed by Big Co that like NoSQL Distilled."
- **A query on the graph isalso known as traversing the graph. An advantage of the graph databases is that we can change the traversing requirements without having to change the nodes or edges**.

- If we want to “get all nodes thatlike NoSQL Distilled,” we can do so without having to change the existing data or the model ofthe database, because we can traverse the graph any way we like.
- **Usually, when we store a graph-like structure in RDBMS, it’s for a single type of relationship**(“who is my manager” is a common example).
- **Adding another relationship to the mix usually means a lot of schema changes and data movement, which is not the case when we are using graph databases.**
- Similarly, in relational databases we model the graph beforehand based on the Traversal we want; if the Traversal changes, the data will have to change.
- In graph databases**, traversing the joins or relationships is very fast.** The relationship between nodes is not calculated at query time but is actually persisted as a relationship.
- **Traversing persisted relationships is faster than calculating them for every query.**
- Nodes can have different types of relationships between them, allowing you to both represent **relationships between the domain entities and to have secondary relationships for things like category, path, time-trees, quad-trees for spatial indexing, or linked lists for sorted access.**

  Since there is no limit to the number and kind of relationships a node can have, all they can be represented in the same graph database

some suitable use cases for graph databases.

11.3.1. Connected Data

- Social networks are where graph databases can be deployed and used very effectively.
- These socialgraphs don’t have to be only of the friend kind; for example, they can represent employees, their knowledge, and where they worked with other employees on different projects.
- Any link-rich domain is well suited for graph databases.
- If you have relationships between domain entities from different domains (such as social, spatial,commerce) in a single database, you can make these relationships more valuable by providing theability to traverse across domains.

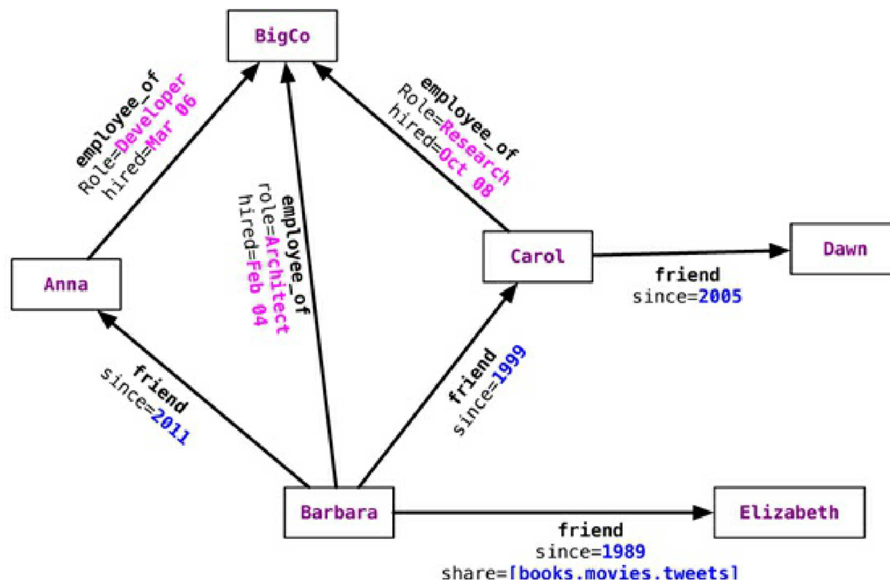
11.3.2. Routing, Dispatch, and Location-Based Services

- Every location or address that has a delivery is a node, and all the nodes where the delivery has to bemade by the delivery person can be modelled as a graph of nodes.
- Relationships between nodes can have the property of distance, thus allowing you to deliver the goods in an efficient manner.
- Distance and location properties can also be used in graphs of places of interest, so that your application can provide recommendations of good restaurants or entertainment options nearby.
- You can also create nodes for your points of sales, such as bookstores or restaurants, and notify the users when they are close to any of the nodes to provide location-based services.

11.3.3. Recommendation Engines

- As nodes and relationships are created in the system, they can be used to make recommendations like “your friends also bought this product” or “when invoicing this item, these other items are usually invoiced.” Or, it can be used to make recommendations to travellers mentioning that when other visitors come to Barcelona they usually visit Antonio Gaudi’s creations.
- An interesting side effect of using the graph databases for recommendations is that as the data size grows, the number of nodes and relationships available to make the recommendations quickly increases.
- The same data can also be used to mine information—for example, which products are always bought together, or which items are always invoiced together; alerts can be raised when these conditions are not met.
- Like other recommendation engines, graph databases can be used to search for patterns in relationships to detect fraud in transactions.

Q2. Describe the procedure to add indexing for the nodes in the NEo4J database. Elaborate on addition of incoming and outgoing links.



● **Neo4J allows you to query the graph for properties of the nodes, traverse the graph, or navigate the nodes relationships using language bindings.**

● **Properties of a node can be indexed using the indexing service.** Similarly, **properties of relationships or edges can be indexed, so a node or edge can be found by the value**.

● Indexes should be queried to find the starting node to begin a traversal. Let's look at searching for the node using node indexing.

● If we have the graph shown in Figure 11.1, we can index the nodes as they are added to the database, or we can index all the nodes later by iterating over them. **We first need to create an index for the nodes using the IndexManager**.

```
Index<Node>nodeIndex = graphDb.index().forNodes("nodes");
```

● We are indexing the nodes for the name property. Neo4J uses **Lucene** [Lucene] as its indexing service.

● **When new nodes are created, they can be added to the index.**

```
Transaction transaction = graphDb.beginTx();
try {
    Index<Node> nodeIndex = graphDb.index().forNodes("nodes")
    nodeIndex.add(martin, "name", martin.getProperty("name"))
    nodeIndex.add(pramod, "name", pramod.getProperty("name"))
    transaction.success();
} finally {
    transaction.finish();
}
```

● **Adding nodes to the index is done inside the context of a transaction.**

● **Once the nodes are indexed, we can search them using the indexed property. If we search for the node with the name of Barbara, we would query the index for the property of name to have a value of Barbara**.

```
Node node = nodeIndex.get("name", "Barbara").getSingle();
```

● **We get the node whose name is Martin; given the node, we can get all its relationships.**

```
Node martin = nodeIndex.get("name", "Martin").getSingle();
allRelationships = martin.getRelationships();
```

- **We can get both INCOMING or OUTGOING relationships.**

```
incomingRelations = martin.getRelationships(Direction.INCOMING);
```

- **Graph databases are really powerful when you want to traverse the graphs at any depth and specify a starting node for the traversal**. This is especially useful when you are trying to find nodes that are related to the starting node at more than one level down. **As the depth of the graph increases, it makes more sense to traverse the relationships by using a Traverser where you can specify that you are looking for INCOMING, OUTGOING, or BOTH types of relationships**. You can also make the **traverser go top-down or sideways on the graph by using Order values of BREADTH_FIRST or DEPTH_FIRST**. The traversal has to start at some node—in this example, **we try to find all the nodes at any depth that are related as a FRIEND with Barbara:**

```
Node barbara = nodeIndex.get("name", "Barbara").getSingle();
Traverser friendsTraverser = barbara.traverse(Order.BREADTH_FIRST,
StopEvaluator.END_OF_GRAPH,
ReturnableEvaluator.ALL_BUT_START_NODE,
EdgeType.FRIEND,
Direction.OUTGOING);
```

- **The friendsTraverser provides us a way to find all the nodes that are related to Barbara wherethe relationship type is FRIEND. The nodes can be at any depth—friend of a friend at any level—allowing you to explore tree structures.**

Q4. Explain scaling in Document database and Graph database.

Graph db

In NoSQL databases, one of the commonly used scaling techniques is sharding, where data is split and distributed across different servers.

- **With graph databases, sharding is difficult, as graphdatabases are not aggregate-oriented but relationship-oriented.**
- **Since any given node can be related to any other node, storing related nodes on the same server is better for graph traversal.**
- **Traversing a graph when the nodes are on different machines is not good for performance.**

Knowing this limitation of the graph databases, **we can still scale them using some common techniques** described by Jim Webber [Webber Neo4J Scaling].

Generally speaking, **there are three ways to scale graph databases.**

- We can **add enough RAM to the server so that the working set of nodes andrelationships is held entirely in memory.** This technique is only helpful if the dataset that we areworking with will fit in a realistic amount of RAM.
- We can **improve the read scaling of the database by adding more slaves with read-only access tothe data, with all the writes going to the master**. This pattern of writing once and reading from manyservers is a proven technique in MySQL clusters and is really useful when the dataset is large enoughto not fit in a single machine's RAM, but small enough to be replicated across multiple machines.
- **Slaves can also contribute to availability and read-scaling, as they can be configured to never become a master, remaining always read-only.**
- When the dataset size makes replication impractical, **we can shard the data from the application side using domain-specific knowledge**. For example, nodes thatrelate to the North America can be created on one server while the nodes that relate to Asia on

another. This application-level sharding needs to understand that nodes are stored on physically different databases (Figure 11.3).

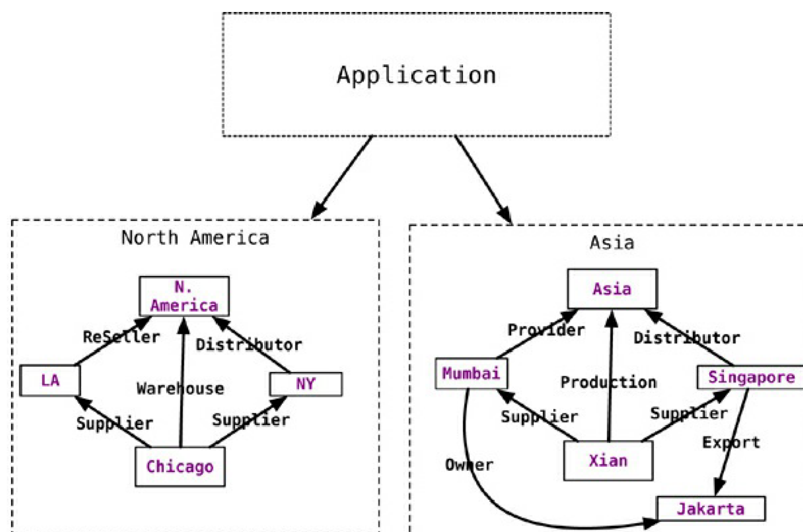


Figure 11.3. Application-level sharding of nodes

Document database:

- The idea of scaling is **to add nodes or change data storage without simply migrating the database to a bigger box**.
- We are **not talking about making application changes to handle more load; instead, we are interested in what features are in the database so that it can handle more load**.
  Scaling for reads (explain horizontal scaling for reads.)
- **Scaling for heavy-read loads can be achieved by adding more read slaves, so that all the reads can be directed to the slaves**. Given a heavy-read application, with our 3-node replica-set cluster, we can add more read capacity to the cluster as the read load increases just by adding more slave nodes to the replica set to execute reads with the *slaveOk* flag (Figure 9.2). This is horizontal scaling for reads.

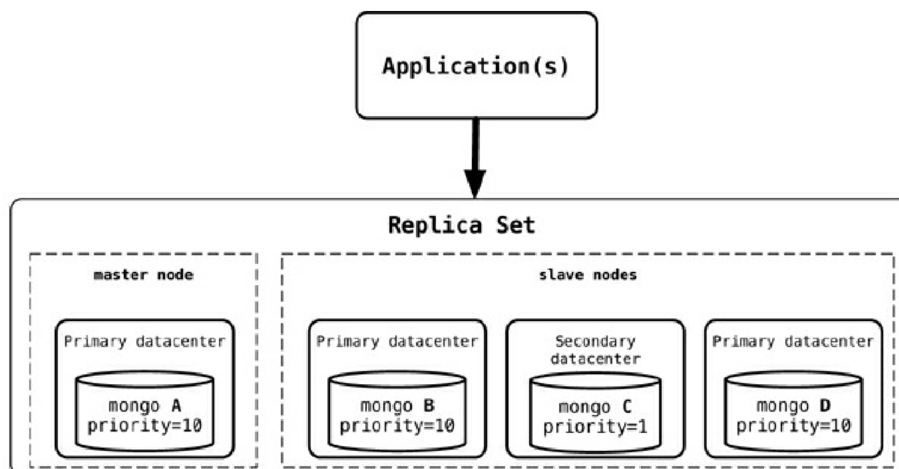


Figure 9.2. Adding a new node, mongo D, to an existing replica-set cluster

- Once the new node, mongo D, is started, it needs to be added to the replica set.

  rs.add("mongod:27017");

  When a new node is added, it will sync up with the existing nodes, join the replica set as secondary node, and start serving read requests.

- An advantage of this setup is that **we do not have to restart any other nodes, and there is no downtime** for the application either.

Scaling for writes

- When **we want to scale for write**, we can start sharding the data.
- Sharding is similar to partitions in RDBMS where we split data by value in a certain column, such as state or year. With RDBMS, partitions are usually on the same node, so the client application does not have to query a specific partition but can keep querying the base table; the RDBMS takes care of finding the right partition for the query and returns the data.
- **In sharding, the data is also split by certain field, but then moved to different Mongo nodes.**
- **The data is dynamically moved between nodes to ensure that shards are always balanced.**
- We can add more nodes to the cluster and increase the number of writable nodes, enabling horizontal scaling for writes.

```
db.runCommand( { shardcollection : "ecommerce.customer", key : {firstname :
1} } )
```

- **Splitting the data** on the first name of the customer **ensures that the data is balanced across the shards for optimal write performance**; furthermore**, each shard can be a replica set ensuring better read performance within the shard** (Figure 9.3).
- When we add a new shard to this existing sharded cluster, the data will now be balanced across four shards instead of three.
- **As all this data movement and infrastructure refactoring is happening, the application will not experience any downtime,** although the cluster may not **perform optimally** when large amounts of data are being moved to rebalance the shards.

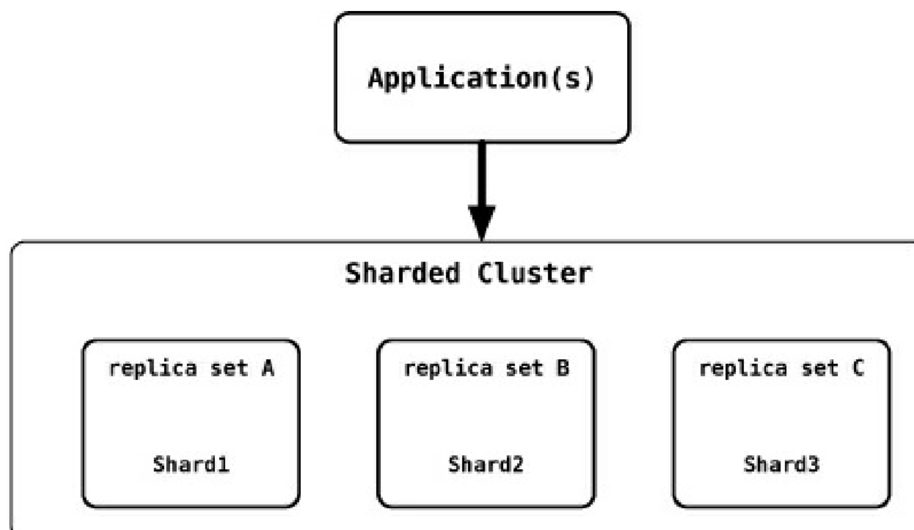


Figure 9.3. MongoDB sharded setup where each shard is a replica set

- **The shard key plays an important role**. You may want to place your MongoDB database shards closer to their users, so sharding based on user location may be a good idea.
- When sharding by customer location, all user data for the East Coast of the USA is in the shards that are served from the East Coast, and all user data for the West Coast is in the shards that are on the West Coast.

Q5. Write equivalent MongoDB queries for given SQL queries.

i) SELECT * FROM order

db.order.find()

ii) SELECT orderId, orderDate FROM order WHERE customerId = "883c2c5b4e5b"

db.order.find({"customerId":"883c2c5b4e5b"})

iiii) SELECT * FROM customerOrder, orderItem, product
WHERE
customerOrder.orderId = orderItem.customerOrderId
AND orderItem.productId = product.productId
AND product.name LIKE '%Refactoring%'

The equivalent Mongo query would be:
db.orders.find({"items.product.name":/Refactoring/})

iv) Write SQL query and MongoDB query to select orderId and orderDate for a customer with id as IN_BL_12181**.**

db.order.find({"customerId":"883c2c5b4e5b"})
SELECT orderId,orderDate FROM order WHERE customerId = "883c2c5b4e5b"

Q6 Briefly describe the relation in Graph databases with neat diagram.

There are many graph databases available, such as Neo4J [Neo4J], Infinite Graph [Infinite Graph], OrientDB [OrientDB], or FlockDB [FlockDB] (which is a special case: a graph database that only supports single-depth relationships or adjacency lists, where you cannot traverse more than one level deep for relationships).

Neo4J features:

- In Neo4J, creating a graph is as simple as creating two nodes and then creating a relationship. Let'screate two nodes, Martin and Pramod:

```
Node martin = graphDb.createNode();
martin.setProperty("name", "Martin");

Node pramod = graphDb.createNode();
pramod.setProperty("name", "Pramod");
```

We have assigned the name property of the two nodes the values of Martin and Pramod. Once we have more than one node, we can create a relationship:

```
martin.createRelationshipTo(pramod, FRIEND);

pramod.createRelationshipTo(martin, FRIEND);
```

- **We have to create relationship between the nodes in both directions, for the direction of the relationship matters:**

  For example, a product node can be liked by user but the product cannot like the user. This directionality helps in designing a rich domain model (Figure 11.2). Nodes know about INCOMING and OUTGOING relationships that are traversable both ways.

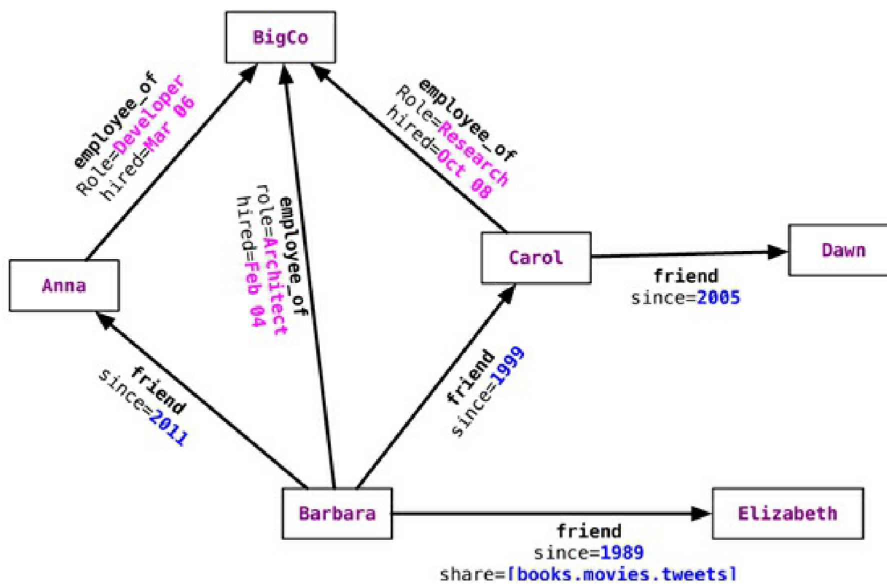


Figure 11.2. Relationships with properties

- **Relationships are first-class citizens in graph databases; most of the value of graph databases is derived from the relationships.**
- **Relationships don't only have a type, a start node, and an end node, but can have properties of their own**. Using these properties on the relationships, we can **add intelligence to the relationship—for example, since when did they become friends, what is the distance between the nodes, or what aspects are shared between the nodes. These properties on the relationships can be used to query the graph.**
- **Since most of the power from the graph databases comes from the relationships and their properties, a lot of thought and design work is needed to model the relationships** in the domain that we are trying to work with.
- **Adding new relationship types is easy; changing existing nodes and their relationships is similar to data migration, because these changes will have to be done on each node and each relationship in the existing data.**