Internal Assessment Test 3 – MAY 2024

Solution

Sub:	NOSQL database					Sub Code:	18CS823	Branch:	CSE
Date:	11/05/24	Duration:	90 mins	Max Marks:	50	Date:	11/05/24		

Question number	Question and solution
1	**Explain with neat diagram three ways with which graph database is scaled.** 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 and relationships is held entirely in memory.** This technique is only helpful if the dataset that we are working 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 to the data, with all the writes going to the master**. This pattern of writing once and reading from many servers is a proven technique in MySQL clusters and is really useful when the dataset is large enough to 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 that relate 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).

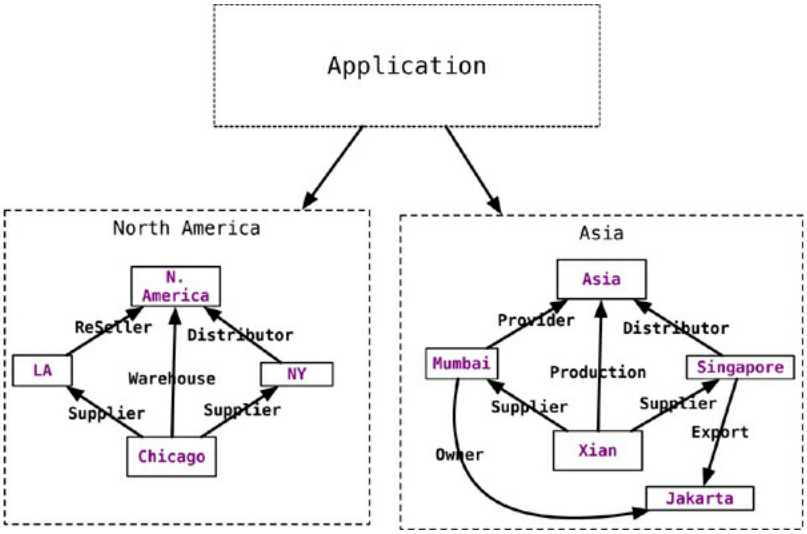


Figure 11.3. Application-level sharding of nodes

2

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.
  More about Lucene indexing : https://youtu.be/vLEvmZ5eEz0
- **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);
	
	• We can also apply directional filters on the queries when querying for a relationship. With the graph in Figure 11.1, **if we want to find all people who like NoSQL Distilled, we can find the NoSQL Distilled node and then get its relationships with Direction.INCOMING**. At this point we can also add the type of relationship to the query filter, since **we are looking only for nodes that LIKE NoSQL Distilled**.
	
	Node nosqlDistilled = nodeIndex.get("name",
	"NoSQL Distilled").getSingle();
	relationships = nosqlDistilled.getRelationships(INCOMING, LIKES);
	for (Relationship relationship : relationships) {
	likesNoSQLDistilled.add(relationship.getStartNode());
	}
	
	Finding nodes and their immediate relations is easy, but this can also be achieved in RDBMS databases.

3 Question number	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) Node barbara = nodeIndex.get("name", "Barbara").getSingle(); Node jill = nodeIndex.get("name", "Jill").getSingle(); PathFinder<Path> finder = **GraphAlgoFactory.allPaths(** Traversal.expanderForTypes(FRIEND,Direction.OUTGOING) ,MAX_DEPTH); Iterable<Path> paths = finder.findAllPaths(barbara, jill); ii) Find all outgoing relationships with the type of FRIEND, and return the

	friends' names of "Ajay" for greater depth (3 marks) START barbara=node:nodeIndex(name = "Ajay") MATCH path = barbara-[:FRIEND*1..3]->end_node RETURN barbara.name,end_node.name, length(path) 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) START barbara = node:nodeIndex(name = "Barbara") MATCH (barbara)-[relation]->(related_node) WHERE type(relation) = 'FRIEND' AND relation.share RETURN related_node.name, relation.since

4	Explain consistency and availability in MongoDB with neat diagrams.
	Consistency -5 marks
	Availability-5 marks
	• Consistency in MongoDB database is configured by using the **replica sets** and **choosing to wait for the writes to be replicated to all the slaves or a given number of slaves.**
	• **Every write can specify the number of servers the write has to be propagated to before it returns as successful.**
	• A command like db.runCommand({ getlasterror : 1 , w : "majority" }) tells the database how strong is the consistency you want.
	• For example, *if you have one server and specify the w as majority, the write will return immediately since there is only one node. If you have three nodes in the replica set and specify w as majority, the write will have to complete at a minimum of two nodes before it is reported as a success. You can increase the w value for stronger consistency but you will suffer on write performance, since now the writes have to complete at more nodes.*
	• **Replica sets also allow you to increase the read performance** by allowing reading from slaves by **setting slaveOk**; this parameter can be set on the connection, or database, or collection, or individually for each operation.
	
	`Mongo mongo = new Mongo("localhost:27017");`
	`mongo.slaveOk();`
	
	Here we are setting slaveOk per operation, so that we can decide which operations can work with data from the slave node.
	
	`DBCollection collection = getOrderCollection();`
	`BasicDBObject query = new BasicDBObject();`
	`query.put("name", "Martin");`
	`DBCursor cursor = collection.find(query).slaveOk();`
	
	• Similar to various options available for read, **you can change the settings to achieve strong write consistency, if desired.**
	• *By default,* ***a write is reported successful once the database receives it; you can change this so as to wait for the writes to be synced to disk or to propagate to two or more slaves.***

- **This is known as WriteConcern:** ***You make sure that certain writes are written to the master and some slaves by setting WriteConcern to REPLICAS_SAFE***. Shown below is code where we are setting the WriteConcern for all writes to a collection:

  ```
  DBCollection shopping = database.getCollection("shopping");
  shopping.setWriteConcern(REPLICAS_SAFE);
  ```

- **WriteConcern can also be set per operation by specifying it on the save command:**

  ```
  WriteResult result = shopping.insert(order, REPLICAS_SAFE);
  ```

- There is a tradeoff that you need to carefully think about, based on your application needs and business requirements, to decide what settings make sense for slaveOk during read or what safety level you desire during write with WriteConcern.

Availability

- The CAP theorem dictates that we can have only two of Consistency, Availability, and Partition Tolerance.
- **Document databases try to improve on availability by replicating data using the master-slave setup.**
- The same data is available on multiple nodes and the clients can get to the data even when the primary node is down.
- Usually, the application code does not have to determine **if the primary node is available or not.**
- MongoDB implements replication, providing high availability using **replica sets**.
- **In a replica set, there are two or more nodes participating in an asynchronous master-slave replication. The replica-set nodes elect the master, or primary, among themselves.**
- Assuming all the nodes have equal voting rights, some nodes can be favored for being closer to the other servers, for having more RAM, and so on; users can affect this by assigning a priority—a number between 0 and 1000—to a node.
- All requests go to the master node, and the data is replicated to the slave nodes. If the master node goes down, the remaining nodes in the replica set vote among themselves to elect a new master; all future requests are routed to the new master, and the slave nodes start getting data from the new master. When the node that failed comes back online, it joins in as a slave and catches up with the rest of the nodes by pulling all the data it needs to get current.

Figure 9.1 is an example configuration of replica sets. We have two nodes, **mongo A** and **mongo B**, running the MongoDB database in the primary data-center, and **mongo C** in the secondary datacenter. If we want nodes in the primary datacenter to be elected as primary nodes, we can assign them a higher priority than the other nodes. More nodes can be added to the replica sets without having to take them offline.

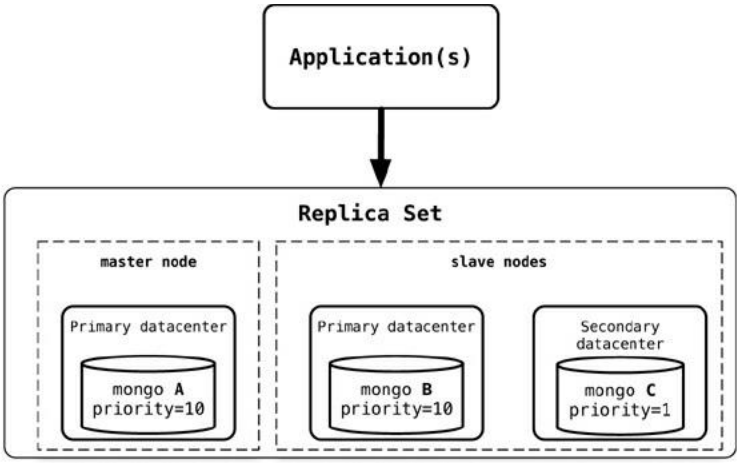


Figure 9.1. Replica set configuration with higher priority assigned to nodes in the same Datacentre

- The application writes or reads from the primary (master) node. When connection is established**, the application only needs to connect to one node (primary or not, does not matter) in the replica set,** and the rest of the nodes are discovered automatically.
- **When the primary node goes down, the driver talks to the new primary elected by the replica set**. The application does not have to manage any of the communication failures or node selection criteria.
- Using replica sets gives you the ability to have a highly available document data store.
- **Replica sets are generally used for data redundancy, automated failover, read scaling, server maintenance without downtime, and disaster recovery.**
- Similar availability setups can be achieved with CouchDB, RavenDB, Terrastore, and other products.

5r

Explain horizontal sharding in MongoDB,

i) For adding a new node to an existing replica-set - 4+1 diagram

ii) Where each shard is a replica set. - 4+1 diagram

- **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.

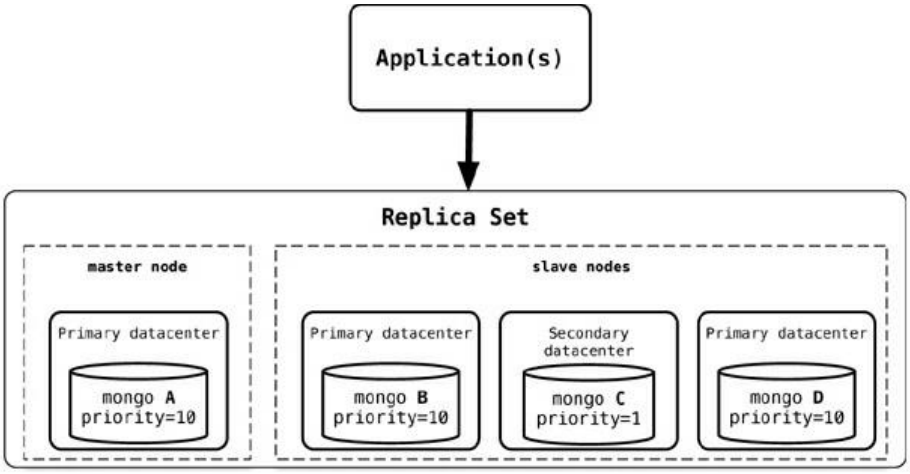


Figure 9.2. Adding a new node, mongo D, to an existing replica-set cluster

- 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.

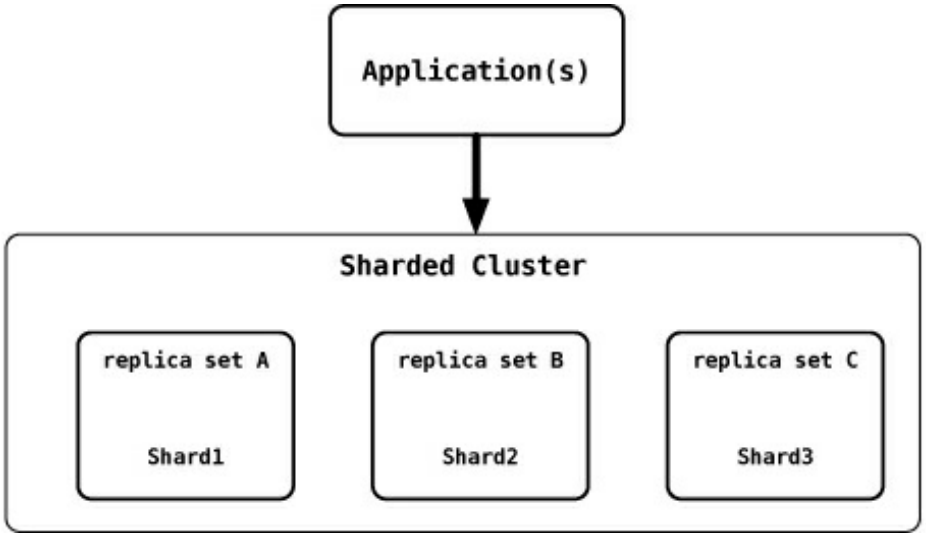
- An advantage of this setup is that **we do not have to restart any other nodes, and there is no downtime** for the application either.

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**
6	Briefly explain the document databases. Compare document databases with SQL and Key-value databases. • Documents are the main concept in document databases. **The database stores and retrieves documents, which can be XML, JSON, BSON, and so on.** • **These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values.** • **The documents stored are similar to each other but do not have to be exactly the same**. Document databases store documents in the value part of the key-value store; think about document databases as key-value stores where the value is examinable. • Let's look at how terminology compares in Oracle and MongoDB.

Oracle	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id
join	DBRef

- The _id is a special field that is found on all documents in Mongo, just like ROWID in Oracle. In MongoDB, _id can be assigned by the user, as long as it is unique.

9.1. What Is a Document Database?

```
{ "firstname": "Martin",
  "likes": [ "Biking",
             "Photography" ],
  "lastcity": "Boston",
  "lastVisited":
}
```

The above document can be considered a row in a traditional RDBMS. Let's look at another document:

```
{
  "firstname": "Pramod",
  "citiesvisited": [ "Chicago", "London", "Pune", "Bangal
  "addresses": [
    { "state": "AK",
      "city": "DILLINGHAM",
      "type": "R"
    },
    { "state": "MH",
      "city": "PUNE",
      "type": "R" }
  ],
  "lastcity": "Chicago"
}
```

- Looking at the documents, we can see that they are similar, but **have differences in attribute names. This is allowed in document databases**
- **The schema of the data can differ across documents, but these documents can still belong to the same collection**—unlike an RDBMS where every row in a table has to follow the same schema.

  Eg. We represent a list of citiesvisited as an array, or a list of addresses as list of documents embedded inside the main document.
- **Embedding child documents as subobjects inside documents provides for easy access and better performance.**

  Eg. If you look at the documents, you will see that some of **the attributes are similar**, such as firstname or city. At the same time, there are attributes in the second document which **do not exist in the first document**, such as addresses, while likes is in the first document but not the second.
- **This kind of different representation of data is not the same as in RDBMS where every column has to be defined, and if it does not have data it is marked as empty or set to null**.
- In documents, **there are no empty attributes**; if a given attribute is **not found, assume that it was not set or not relevant to the document**.
- Documents allow for new **attributes to be created without the need to define them or to change the existing documents.**
- Some of the popular document databases we have seen are MongoDB [MongoDB], CouchDB [CouchDB], Terrastore [Terrastore], OrientDB [OrientDB], RavenDB

	[RavenDB], and of course the well-known and often reviled Lotus Notes [Notes Storage Facility] that uses document storage.