Sub:	NoSQL database					Sub Code:	18CS823
Date:	17/6/2022	Duration:	90 mins	Max Marks:		Sem/Sec:	OBE

Scheme

1	Write equivalent MongoDB queries for given SQL queries. i) SELECT * FROM order **(2 mark)** ii) SELECT orderId, orderDate FROM order WHERE customerId = "883c2c5b4e5b" **(2 marks)** iiii) SELECT * FROM customerOrder, orderItem, product WHERE customerOrder.orderId = orderItem.customerOrderId AND orderItem.productId = product.productId AND product.name LIKE '%Refactoring%' **(2 mark)** iv) Write SQL query and MongoDB query to select orderId and orderDate for a customer with id as IN BL 12181**. (4 marks)**		
2	Explain features of document databases. features **-8 marks** document database schema**-2 marks**		
3	Write code snippets to find shortest path between two nodes in Neo4J. Also write and explain the pseudo code for Dijkstra's algorithm. Shortest path code snippet- **3 marks** Dijkstra's algorithm and explanation – **7 marks**		
4	Describe the procedure to add indexing for the nodes in the Neo4J database. Write the code snippet to traverse the nodes in Neo4J using breadth first search. procedure to add indexing for the nodes in the Neo4J database – **6 marks** traversing the nodes in Neo4J using breadth first search**- 4 marks**		
5	Explain schema changes and incremental migration in MongoDB Schema changes – **5 marks** Incremental migration – **5 marks**		
6	How to ensure consistency and availability in MongoDB? Consistency in MongoDB – **5 marks**		

Availability in MongoDB - **5 marks**

Solution

Q1.

Write equivalent MongoDB queries for given SQL queries.

i) SELECT * FROM order **(2 mark)**

db.order.find()

ii) SELECT orderId, orderDate FROM order WHERE customerId = "883c2c5b4e5b" **(2 marks)**

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%' **(2 mark)**

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**. (4 marks)**

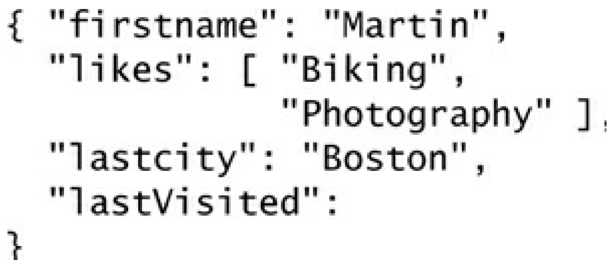
db.order.find({"customerId":"883c2c5b4e5b"})
SELECT orderId,orderDate FROM order WHERE customerId = "883c2c5b4e5b"

Q2

Explain features of document databases.
features -**8 marks**
document database schema-**2 marks**

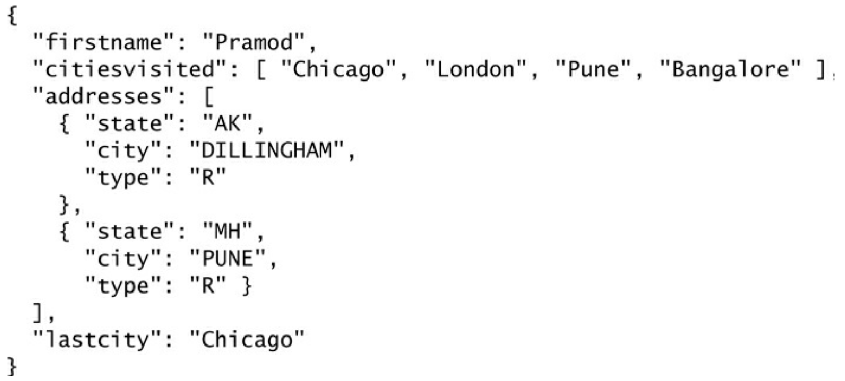
- Documents are the main concept in document databases. **The database stores and retrieves documents, which can be XML, JSON, BSON, and so on.**
- **These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values.**

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.



```
{ "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", "Bangalore" ],
  "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**
- **Embedding child documents as subobjects inside documents provides for easy access and better performance.**

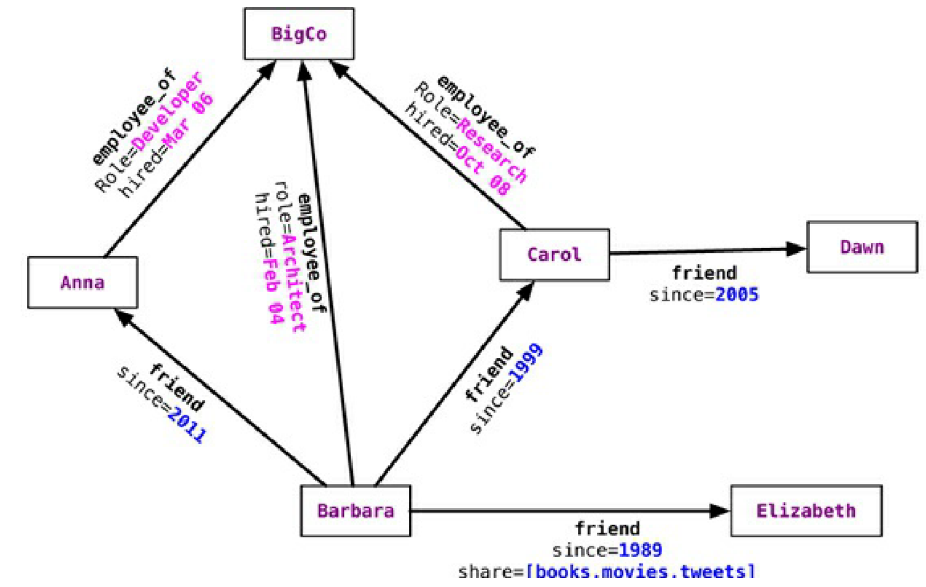
Eg. If you look at the documents, you will see that some of **the attributes are similar**, such as firstname or city. At the same time, there are attributes in the second document which **do not exist in the first document**, such as addresses, while likes is in the first document but not the second.

- **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 thewell-known and often reviled Lotus Notes [Notes Storage Facility] that uses document storage.

Q3. Write code snippets to find shortest path between two nodes in Neo4J. Also write and explain the pseudo code for Dijkstra's algorithm.

```
START beginingNode = (beginning node specification)
MATCH (relationship, pattern matches)
WHERE (filtering condition: on data in nodes and relationships)
RETURN (What to return: nodes, relationships, properties)
ORDER BY (properties to order by)
SKIP (nodes to skip from top)
LIMIT (limit results)
```

Q4 Describe the procedure to add indexing for the nodes in the Neo4J database. Write the code snippet to traverse the nodes in Neo4J using breadth first search.



● **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.**

Q5. Explain schema changes and incremental migration in MongoDB

- An RDBMS database has to be changed before the application is changed. This is what the schema free, or schema less, approach tries to avoid, aiming at flexibility of schema changes per entity.
- **Frequent changes to the schema are needed to react to frequent market changes and product innovations.**
- When developing with NoSQL databases, in some cases **the schema does not have to be thought about beforehand.**
- We still **have to design and think about other aspects**, such as the types of **relationships (with graph databases), or the names of the column families, rows, columns, order of columns (with column databases), or how are the keys assigned and what is the structure of the data inside the value object** (with key-value stores).
- Even if didn’t think about these up front, or if we want to change our decisions, it is easy to do so.
- The claim that NoSQL databases are entirely schemaless is misleading; while they store the data without regard to the schema the data adheres to, that schema has to be defined by the application, because the data stream has to be parsed by the application when reading the data from the database.
- Also, the application has to create the data that would be saved in the database. If the application cannot parse the data from the database, we have a schema mismatch even if, instead of the RDBMS database throwing a error, this error is now encountered by the application.
- Thus, even in schemaless databases, the schema of the data has to be taken into consideration when refactoring the application.

● Schema changes especially matter when there is a deployed application and existing production data.
● For the sake of simplicity, assume we are using a document data store like MongoDB [MongoDB] and we have the same data model as before: customer, order, and orderItems.

```
{"
_id": "4BD8AE97C47016442AF4A580",
"customerid": 99999,
"name": "Foo Sushi Inc",
"since": "12/12/2012",
"order": {
"orderid": "4821-UXWE-122012","orderdate": "12/12/2001",
"orderItems": [{"product": "Fortune Cookies",
"price": 19.99}]
}
}
```

Application code to write this document structure to MongoDB:

```
BasicDBObject orderItem = new BasicDBObject();
orderItem.put("product", productName);
orderItem.put("price", price);
orderItems.add(orderItem);
```

Code to read the document back from the database:

```
BasicDBObject item = (BasicDBObject) orderItem;
String productName = item.getString("product");
Double price = item.getDouble("price");
```

● Changing the objects to add preferredShippingType does not require any change in the database, as the database does not care that different documents do not follow the same schema.
● This allows for faster development and easy deployments.
● All that needs to be deployed is the application—no changes on the database side are needed. The code has to make sure that documents that do not havethe preferredShippingType attribute can still be parsed—and that’s all.

Let’s look at the schema change we made before: introducing discountedPrice and renaming price to fullPrice. To make this change, we rename the price attribute to fullPrice and add discountedPrice attribute. The changed document is

```
{"
_id": "5BD8AE97C47016442AF4A580",
"customerid": 66778,
"name": "India House",
"since": "12/12/2012",
"order": {
"orderid": "4821-UXWE-222012",
"orderdate": "12/12/2001",
"orderItems": [{"product": "Chair Covers",
"fullPrice": 29.99,
"discountedPrice":26.99}]
}
}
```
Once we deploy this change, new customers and their orders can be saved and read back without problems, but for existing orders the price of their product cannot be read, because now the code is looking for fullPrice but the document has only price.

12.3.1. Incremental Migration

- Schema mismatch trips many new converts to the NoSQL world.
- When schema is changed on the application, we have to make sure to convert all the existing data to the new schema (depending on data size, this might be an expensive operation).
- Another option would be to make sure that **data, before the schema changed, can still be parsed by the new code, and when it's saved, it is saved back in the new schema.** This technique, known as **incremental migration**, will migrate data over time;some data may never get migrated, because it was never accessed.

We are reading both price and fullPrice from the document:
```
BasicDBObject item = (BasicDBObject) orderItem;
String productName = item.getString("product");
Double fullPrice = item.getDouble("price");
if (fullPrice == null) {
fullPrice = item.getDouble("fullPrice");
}
Double discountedPrice = item.getDouble("discountedPrice");
```

When writing the document back, the old attribute price is not saved:

```
BasicDBObject orderItem = new BasicDBObject();
orderItem.put("product", productName);
orderItem.put("fullPrice", price);
orderItem.put("discountedPrice", discountedPrice);
orderItems.add(orderItem);
```

- **When using incremental migration, there could be many versions of the object on the application side that can translate the old schema to the new schema; while saving the object back, it is saved using the new object.**
- **This gradual migration of the data helps the application evolve faster**.
- **The incremental migration technique will complicate the object design**, especially as new changes are being introduced yet old changes are not being taken out.
- **This period between the change deployment and the last object in the database migrating to the new schema is known as the transition period (**Figure 12.6). Keep it as short as possible and focus it to the minimum possible scope—this will help you keep your objects clean.

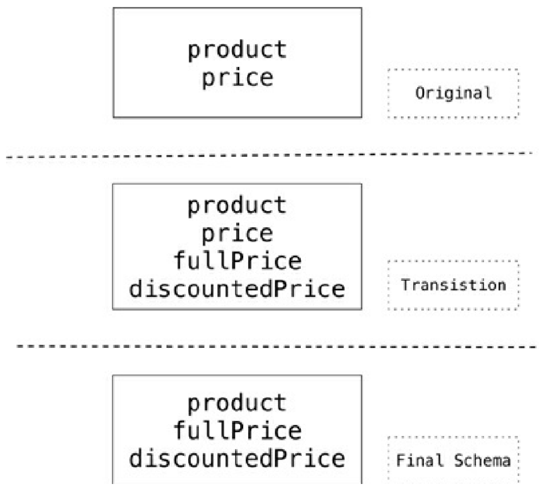


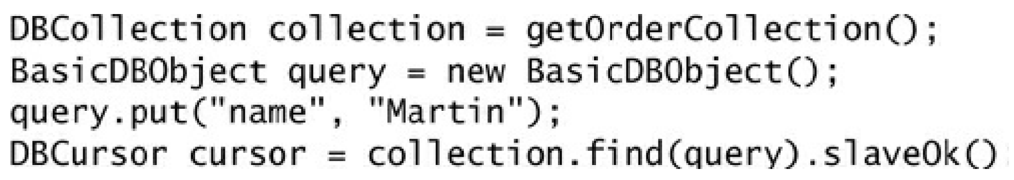
Figure 12.6. Transition period of schema changes

● **The incremental migration technique can also be implemented with a schema_version field on the data, used by the application to choose the correct code to parse the data into the objects.**
● When saving, the data is migrated to the latest version and the schema_version is updated to reflect that.
● **Having a proper translation layer between your domain and the database is important so that, as the schema changes, managing multiple versions of the schema is restricted to the translation layer and does not leak into the whole application**.
● Mobile apps create special requirements. Since we cannot enforce the latest upgrades of the application, the application should be able to handle almost all versions of the schema.

Q6. How to ensure consistency and availability in MongoDB?
● Consistency in MongoDB database is configured by using the **replica sets** and **choosing to wait forthe writes to be replicated to all the slaves or a given number of slaves.**
● **Every write can specify thenumber of servers the write has to be propagated to before it returns as successful.**
● A command like db.runCommand({ getlasterror : 1 , w : "majority" }) tells thedatabase 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 be**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 trade-off 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 of nodes in MongoDB:
● **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 number 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.

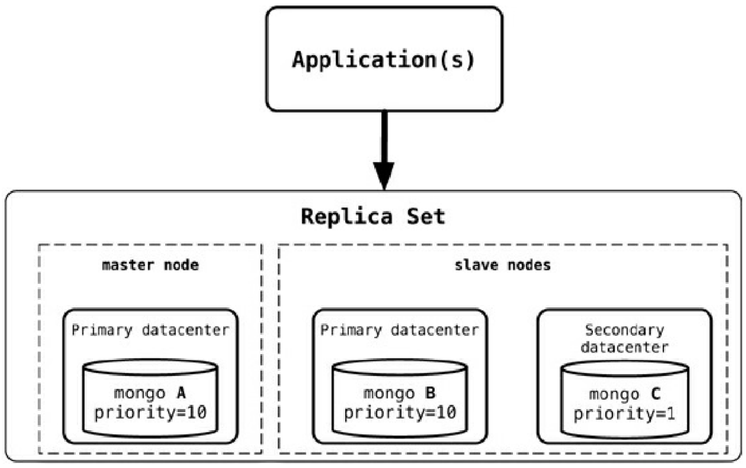


Figure 9.1. Replica set configuration with higher priority assigned to nodes in the same Datacentre

- 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**
- Using replica sets gives you the ability to have highly available document data store.
- **Replica sets are generally used for data redundancy, automated failover, read scaling, server maintenance without downtime, and disaster recovery.**