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

Internal Assessment Test 1 – October 2024

Sub:	NoSQL Database				Sub Code:	21CS745	Branch:	CSE	
Date:	18.10.2024	90 mins	Max Marks:	50	Sem/Sec:	VII/ A, B, C		OBE	

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1	a) What do you understand by NoSQL?? b) Explain about aggregate data models with a neat diagram. Considering an example of relational data models.	(5+5)	CO1	L2
2	a) Define materialized view. How are they different from views. b) Assume you're a data engineer for a financial analytics company that needs to optimize its data processing pipeline for generating daily reports on stock market trends. Describe the concept of materialized views within the context of this scenario and elucidate two approaches to implementing them, providing relevant examples for each approach.	(3+7)	CO1	L2,L3
3	a) Compare and contrast the key-value and document data models in NoSQL databases. b) Discuss scenarios where each data model is most effective, and provide an example for both the key-value and document models.	(5+5)	CO1	L2

CI CCI HoD

4	As the leader of the backend development team for a high-traffic e-commerce website, you need to ensure data availability and scalability during peak hours.Consider the above scenario and Explain how master-slave replication can address these challenges, detailing its implementation, benefits, and a concrete example of its application in your e-commerce platform.	10	CO2	L3
5	a) Highlight the significance of consistency in database systems. b) enumerate the various types of consistency that are commonly recognized within this context.	(5+5)	CO2	L2
6	a) What is the CAP theorem? b) What are three properties which cannot be simultaneously guaranteed? How it is applicable to NoSql Systems.	10	CO2	L2

CI CCI HoD

1-a) **NoSQL** stands for "Not Only SQL" and refers to a broad category of database systems designed to handle unstructured, semi-structured, or structured data in ways that traditional relational databases (RDBMS) cannot. NoSQL databases are more flexible, scalable, and designed to handle massive amounts of data across distributed architectures.

- **Types of NoSQL databases**:
  - **Document-based** (e.g., MongoDB, CouchDB)
  - **Key-value stores** (e.g., Redis, DynamoDB)
  - **Column-family stores** (e.g., Cassandra, HBase)
  - **Graph databases** (e.g., Neo4j, ArangoDB)
- **Advantages**:
  - Scalability (horizontal scaling is easier)
  - Schema flexibility (can store semi-structured and unstructured data)
  - High performance for certain use cases like big data and real-time analytics
- **Disadvantages**:
  - Lack of standardization
  - Complex query capabilities compared to SQL-based systems

- Potential consistency issues (CAP theorem)

b) An **aggregate data model** in NoSQL is a model where data is stored and retrieved as collections of related items grouped together, rather than using relational tables with joins. This is commonly used in document-based databases like MongoDB, where a single document can store an entire set of related data.

Relational Data Model Example:

- In a **Relational Database (RDBMS)**, we might have tables like `Orders`, `Customers`, and `Products`:
  - `Customers`: CustomerID, Name, Email
  - `Orders`: OrderID, CustomerID, ProductID, Quantity
  - `Products`: ProductID, ProductName, Price
- These tables would use **foreign keys** to establish relationships between them, and you'd use `JOIN` operations to retrieve all relevant data for a customer's order.

In an **Aggregate Data Model** (e.g., in MongoDB), the same data might be represented in a single "Order" document that contains embedded documents for `Customer` and `Products`:
json
Copy code

```
{
  "_id": 123,
  "customer": {
    "customer_id": 1,
    "name": "John Doe",
    "email": "john.doe@example.com"
  },
  "items": [
    {
      "product_id": 101,
      "product_name": "Laptop",
      "quantity": 2,
      "price": 500
    },
    {
      "product_id": 102,
      "product_name": "Mouse",
      "quantity": 1,
      "price": 25
    }
  ],
  "order_date": "2024-10-12"
}
```

- In this model, the order document aggregates both customer and product details in a single document, which makes it easier to retrieve all data about an order in a single query. **No JOIN operations** are necessary in aggregate data models.

2-a) Materialized View: A **materialized view** is a precomputed view that stores the results of a query physically on disk. This view is refreshed periodically or manually, depending on the system, which means it can be faster to query than a standard view since it doesn't require recomputing the result each time it's accessed.

- **Difference from Views**:
  - **Views**: A view is a virtual table that is defined by a query. The data in a view is not stored physically, and every time you query a view, the underlying query is executed.
  - **Materialized Views**: Data is stored and updated periodically. Querying a materialized view does not require re-executing the query; instead, the stored data is retrieved.

b) Two approaches to implement materialized views:

1. **Pre-aggregation Approach**:
   - You can define a materialized view to store daily aggregates of stock prices, volumes, and trends. This would involve aggregating data for each stock per day (e.g., average price, total volume, etc.) into a materialized view.

Example:
sql
Copy code

```
CREATE MATERIALIZED VIEW daily_stock_trends AS

SELECT stock_id,

       DATE(transaction_date) AS date,

       AVG(price) AS average_price,

       SUM(volume) AS total_volume

FROM stock_transactions

GROUP BY stock_id, DATE(transaction_date);
```

2.
   - This view would store the results of the aggregation, and it would be updated daily. The advantage is that querying the materialized view for stock trends would be much faster than recalculating these aggregates every time.
3. **Incremental Update Approach**:
   - Instead of recalculating the entire dataset each time, the materialized view could be incrementally updated with new data. For example, each time new stock transactions are added, only the relevant portions of the materialized view are updated.

Example:
sql
Copy code

```
CREATE MATERIALIZED VIEW daily_stock_trends

REFRESH FAST ON COMMIT

AS

SELECT stock_id,
```

```
        DATE(transaction_date) AS date,

        AVG(price) AS average_price,

        SUM(volume) AS total_volume

FROM stock_transactions

GROUP BY stock_id, DATE(transaction_date);
```

4.
    - In this case, every time a new transaction is committed, the view would update incrementally, which could be more efficient than recalculating the entire dataset.

3.

1. **Key-Value Model**:
    - **Structure**: Simple structure where each entry consists of a key and a value. The value could be anything, from a string, a number, or a more complex data structure like JSON or binary data.
    - **Examples**: Redis, DynamoDB
    - **Use Cases**: Best for storing data where fast retrieval by a unique key is required. Examples include caching, session storage, and user preferences.
2. **Example**:
    - Key: `user123`
    - Value: `{ "name": "John Doe", "email": "john.doe@example.com", "age": 30 }`
3. **Document Model**:
    - **Structure**: Stores data as **documents**, typically in JSON or BSON format. Each document is a set of key-value pairs, but unlike key-value models, documents can be complex and nested.
    - **Examples**: MongoDB, CouchDB
    - **Use Cases**: Suitable for more complex or semi-structured data where relationships between data entities are embedded in documents, and schema flexibility is needed.
4. **Example**:
    - Document:

json
Copy code

```
{

  "_id": 123,

  "customer": { "name": "John Doe", "email": "john.doe@example.com" },

  "order": [

    { "product": "Laptop", "price": 500, "quantity": 2 },

    { "product": "Mouse", "price": 25, "quantity": 1 }

  ]

}
```

5.

b)

1. **Key-Value Model**:
    - **Effective Scenario**: When the application needs to store and retrieve data very quickly by a unique key. Key-value stores are ideal for use cases that require high-speed lookups and do not require complex querying.
    - **Example**: **Caching** user session data for a website. A session can be identified by a unique session ID, and its data (such as user preferences or authentication tokens) can be stored as a value in a key-value store.
2. **Document Model**:
    - **Effective Scenario**: When the data is semi-structured or nested and the relationships between entities can be represented within a single document. Document stores are useful when each "record" can vary in structure or when dealing with complex objects.
    - **Example**: **E-commerce platform** where each product has various attributes (price, description, reviews, etc.) that might change from one product to another. Instead of using a rigid schema, a document store allows storing different attributes for different products flexibly.

4. **Master-Slave Replication:**

In a **master-slave replication** setup:

- The **master** node is the authoritative source of data, where all write operations (INSERT, UPDATE, DELETE) occur.
- The **slave** nodes replicate the data from the master and can be used to handle **read operations** (SELECT queries).

This setup can effectively address the challenges of data availability and scalability, particularly for high-traffic systems like an e-commerce website.

Implementation:

1. **Replication Setup**:
    - You configure one database server as the **master** and several database servers as **slaves**.
    - Each time a change is made to the master database, those changes are propagated to the slave databases. This ensures that the data on the slaves is always up-to-date with the master.
2. **Read and Write Distribution**:
    - **Writes**: All data write operations (e.g., placing an order, updating product inventory) are handled by the master node.
    - **Reads**: For queries like viewing product details, listing products, user profiles, and browsing categories, the read traffic is distributed across multiple slave nodes, thus offloading the master server and improving the overall system’s scalability.
3. **Replication Lag**:
    - Depending on the replication method (synchronous or asynchronous), there may be a slight delay in propagating changes from the master to the slaves. However, this is typically manageable if read operations can tolerate eventual consistency.

Benefits of Master-Slave Replication in E-Commerce:

1. **Scalability**:
    - **Horizontal Scaling**: By adding more slave nodes, you can handle more read traffic, which is especially important during peak hours (e.g., holiday sales, flash sales, or promotions).

    - **Distributed Load**: This balances the database load, ensuring that the master database is not overwhelmed with read requests.
2. **Improved Availability**:
    - **Fault Tolerance**: If a slave node goes down, the system can continue to serve read requests from the remaining slave nodes. Additionally, if the master fails, a slave can be promoted to master, reducing downtime.
    - **High Availability**: Users can still browse the website, view product details, and place orders (write operations) as long as the master is operational.
3. **Reduced Latency**:
    - **Local Reads**: By placing slave nodes in geographically distributed regions (such as multiple data centers), you can reduce read latency for users across different regions, improving response time for browsing product catalogs, user profiles, etc.

Concrete Example in E-Commerce Platform:

Imagine an e-commerce website with a **catalog of products** and a **shopping cart**. During a **flash sale**, hundreds of thousands of users are accessing the site simultaneously.

- **Master Node**: Handles all **write operations** such as updating the inventory when an order is placed or when stock levels are updated.
- **Slave Nodes**: Handle **read operations** like fetching product details, displaying product images, and listing products for browsing. This reduces the load on the master node.

During the sale:

- As users browse products and add items to their carts, the **slave nodes** handle the majority of the requests, ensuring the website remains responsive despite the high traffic.
- **Write-heavy operations** such as updating the inventory after an order is placed are handled by the **master node**, but since write operations are less frequent than read operations, this ensures that the system remains efficient.

5-a) Significance of Consistency in Database Systems

Consistency in database systems ensures that a database remains in a valid state after every transaction. The principle guarantees that any transaction will bring the database from one valid state to another, and that no partial or inconsistent data will be visible to users.

In distributed databases, **consistency** is crucial because it ensures that all nodes in the system have the same data after a transaction, preventing issues like reading outdated or partial data. Without consistency, users could end up seeing stale or incorrect data, which is especially problematic for transactional systems like e-commerce, where order details, inventory counts, and user balances must always be accurate.

For example, in an **e-commerce platform**, if **consistency** is not maintained, a user could place an order for an item that has already been sold out, leading to customer dissatisfaction and operational issues.

b) Types of Consistency

1. **Strong Consistency**:
    - Every read will return the most recent write. This is the highest level of consistency.
    - Example: If a product's stock level is updated, any subsequent read will immediately reflect the new stock level.
    - Drawback: Can cause delays, as the system needs to ensure that all nodes are synchronized before a read operation.
2. **Eventual Consistency**:
    - The system will eventually reach consistency, but in the interim, some reads might return outdated data.

    - Example: In an e-commerce site, if inventory data is replicated across multiple servers, a customer might see an out-of-stock item as available until the replication process updates all the servers.
    - This is typically used in NoSQL systems where performance and availability are prioritized over immediate consistency.
3. **Causal Consistency**:
    - Ensures that operations that are causally related are seen by all nodes in the same order. Non-causally related operations may be seen in different orders on different nodes.
    - Example: If a customer places an order, then adds an item to their cart, the second action should always be seen after the first one. However, other unrelated actions can be seen in different orders.
4. **Read-after-Write Consistency**:
    - Guarantees that a read operation will always return the result of the most recent write for a given item.
    - Example: If a product's price is updated, any read operation for that product will immediately return the updated price.

6-a) The **CAP theorem**, proposed by Eric Brewer, states that a distributed database system can provide at most two of the following three guarantees:

1. **Consistency**: Every read operation will return the most recent write (or an error).
2. **Availability**: Every request (read or write) will receive a response (either success or failure).
3. **Partition Tolerance**: The system can continue to operate even if network partitions or failures occur, i.e., parts of the system are unable to communicate with each other.

b) Three Properties That Cannot Be Simultaneously Guaranteed

According to the CAP theorem, it's impossible for a distributed system to guarantee all three properties (Consistency, Availability, and Partition Tolerance) simultaneously. You must prioritize two properties at the expense of the third:

1. **Consistency and Availability** (CA):
    - The system guarantees both consistency and availability, but **partition tolerance** is not provided. In the case of a network partition, the system might stop serving requests.
    - Example: Traditional relational databases (RDBMS) like MySQL often prioritize consistency and availability, but they may struggle in partitioned environments.
2. **Consistency and Partition Tolerance** (CP):
    - The system guarantees consistency even in the case of network partitioning, but it might sacrifice availability. In the case of a partition, the system might not respond to requests at all.
    - Example: Some systems like HBase prioritize consistency and partition tolerance, but during a partition, they may block read/write operations.
3. **Availability and Partition Tolerance** (AP):
    - The system guarantees availability and partition tolerance, but it may not guarantee consistency. The system can respond to requests even if the data is not consistent across all nodes.
    - Example: Many NoSQL systems (like Cassandra or MongoDB) prioritize availability and partition tolerance but might allow data inconsistencies during network partitions.

Application to NoSQL Systems:

NoSQL databases often operate under the assumption that **partition tolerance** is critical due to their distributed nature, especially in cloud environments where network partitions can occur frequently. Hence, many NoSQL systems favor **availability** and **partition tolerance** (AP), but may relax **consistency** under certain conditions. For example:

- **Cassandra**: Prioritizes availability and partition tolerance, allowing eventual consistency.
- **MongoDB**: Offers configurable consistency levels but also prioritizes availability and partition tolerance.