**Question 1- Explain about searching performance. How will you handle replication in SQL for searching & Reporting?**

**Answer –** To increase searching performance we can apply Elasticsearch in our database design .

Step 1- First define the mapping is like a schema for that index. The mapping defines the data types and configurations for each field in the index. We can create mapping for **Seller , Customer , Order** etc .

Like for Seller we can define mapping for Seller index –

sellers\_mapping = {

"mappings": {

"properties": {

"seller\_id": {"type": "integer"},

"name": {"type": "text"},

"email": {"type": "keyword"},

"password": {"type": "keyword"},

"phone\_number": {"type": "long"},

"otp": {"type": "integer"},

"reset\_password\_token": {"type": "text"},

"reset\_password\_token\_expiration": {"type": "text"}

}

}

}

Step 2 – Then we have to do indexing in our data this involves converting our into json data format then apply indexing to json data in the corresponding Elasticsearch index

sellers\_data = [

{"seller\_id": 1, "name": "Seller A", "email": "sellerA@example.com", ...},

{"seller\_id": 2, "name": "Seller B", "email": "sellerB@example.com", ...},

# Add more seller records

]

Step 3 Then we can search queries by elastic search client library .

Example –

query = {

"query": {

"match": {

"name": "Seller A"

}

}

}

Like this we can perform this for other Entities of our database .

To handle replication in SQL for searching & reporting we can implement Transactional Replication method for our database design.

Step – **First identify the master and slave node –** Choose one database as a master and slave where all write operation occur . And choose one slave node for replicating data from master to slave for read operations .

Step 2 – Enable the logging on master so that this log will record all updates of master node .

Step 3 – Slave node must subscribe the master node for replication process .

Step 4 - Take a snapshot of the master database to create a full copy of the data. Then copy the snapshot to the slave node and store it to create a replica of master node .

Step 5 – Connect the master and slave node and start the replication process .

Step 6 - **Load Balancing** :Implement load balancing for read queries in our application to distribute read queries across the available slave server.

**Question -2 Explain what major factors are taken into consideration for performance.**

**Answer -** 1- Indexing - Elastic search strategy is applied for our database design to enhance its searching performance.

2- Database storage - Estimate the expected data volume based on the number of active users (100k) and the product inventory.

Ensure sufficient storage capacity and choose appropriate hardware to handle the data load.

3- Normalization and Denormalization: I have apply Normalization to our database design to eliminate redundancy and maintain data integrity.

4- Hardware and Server Configuration: Optimize the database server's memory, CPU, and disk settings based on our database workload .

5- Load Balancing and Replication: Implement load balancing to distribute read queries across multiple database servers for better performance.

Set up transactional replication to replicate data from the master to slave nodes to ensure high availability and load distribution in our database design .

6- Database Sharding -We can do database sharding if the data grows significantly beyond the capacity of a single server.

7- Security Considerations: Implement robust security measures to prevent unauthorized access and protect sensitive data. For our database we have created Authorization entity in our database design for system security .

**Question 3 - Mention about Indexing, Normalization and Denormalization.**

**Answer –**

Indexing: Indexing the columns that are frequently used in search, filtering, and joining operations. Potential candidates for indexing include:

name column in Sellers, Product, and Customer tables for search queries based on names.

email column in Sellers, Customer, and Notification tables for efficient email-based searches.

product\_id column in Inventory, Skus, OrderDetails, and Notification tables for fast retrieval of product-related information.

customer\_id column in Order, Notification, and AuthenticationTokens tables for efficient retrieval of customer-related data.

Consider the appropriate type of indexes for each column, such as B-tree indexes for equality searches and range queries.

Normalization: Normalize our database design to avoid data redundancy and maintain data integrity.

For example, consider normalizing the Address table by creating a separate table for cities (City) and using the city\_id as a foreign key reference.

In the Order table, order\_description could be moved to a separate table if it contains repeated text to reduce data duplication.

Denormalization: Denormalization may be considered in certain scenarios to optimize read performance and avoid complex joins. However, it should be approached with caution to maintain data consistency.

For example, denormalization could be used to store some pre-aggregated data in the Order table for frequently generated reports.

Considerations:

Use normalization for most of our tables to avoid data anomalies and ensure data integrity.

Utilize indexing to speed up data retrieval for frequently accessed columns.

Evaluate the need for denormalization on a case-by-case basis and only apply it when justified by specific performance requirements.

Regularly monitor and optimize the database performance by analysing query execution plans, index usage, and database statistics.

It's important to note that the decision to implement indexing, normalization, or denormalization should be based on a thorough understanding of our application's specific use cases, query patterns, and performance requirements. Additionally, regular maintenance and performance tuning are essential to ensure the database operates efficiently as data volume and user load increase.

**Question 4 - How will you handle scaling, if required at any point of time.**

Answer- I will handle scaling by Horizontal Scaling with Sharding.

**Horizontal Scaling with Sharding**: Sharding involves distributing data across multiple database servers or nodes, where each server handles a specific subset of data. This approach allows our database to handle increasing data volumes and user loads by distributing the workload across multiple servers, which can result in improved read and write performance.

Steps to implement horizontal sharding in our database design –

Step 1 - Determine the sharding key, which is a column or combination of columns that will be used to distribute data across shards. In our database design, potential sharding keys would be customer\_id, city\_id, or product\_id.

Step 2 - Partition Data into Shards: Based on the selected sharding key, partition our data into multiple shards. Each shard will hold a specific range of data based on the sharding key's value. For example, one shard could store data for customers with customer\_id between 1 and 1000, another shard for customers with customer\_id between 1001 and 2000, and so on.

Step 3 - Load Balancing: Set up a load balancer to distribute incoming database requests across the different database shards. The load balancer ensures that each shard receives an equal share of the workload.

Step 4 - Replication within Shards: To ensure data availability and fault tolerance, we can set up replication within each shard. Create read replicas for each shard to handle read-intensive queries.

Step 5 - Routing Layer: Implement a routing layer in our application that determines which shard to route a specific database request based on the sharding key. This ensures that queries are directed to the correct shard.

**Question 5 - Mention all the assumptions you are taking for solutions.**

**Answer -**  1- **E-commerce Application:** The database design is assumed to be for an e-commerce application that involves sellers, customers, products, orders, inventory, notifications, and promotional activities.

2- **Data Integrity**: The database design is assumed to ensure data integrity through primary keys and foreign keys, maintaining relationships between entities.

3- **Single Database Management System**: The design code is written using DBML, but the assumption is that the entire database will be managed within a single database management system

4- The database design is assumed to handle data for an e-commerce application with 100k active users

**Summary of my database design are the following –**

Table Name: Sellers

Primary Key: seller\_id

Fields: name, email, password, phone\_number, otp, reset\_password\_token, reset\_password\_token\_expiration

Inventory Table:

Table Name: Inventory

Primary Key: inventory\_id

Foreign Key: seller\_id (References Sellers.seller\_id)

Fields: product\_id, price, discount, specification

Product Table:

Table Name: Product

Primary Key: product\_id

Fields: name, category, description

Customer Table:

Table Name: Customer

Primary Key: customer\_id

Fields: name, email, password, otp, reset\_password\_token, reset\_password\_token\_expiration

Address Table:

Table Name: Address

Primary Key: address\_id

Foreign Key: customer\_id (References Customer.customer\_id)

Fields: customer\_id, address, city\_id

City Table:

Table Name: City

Primary Key: city\_id

Fields: city\_name, state

Skus Table:

Table Name: Skus

Primary Key: sku\_id

Foreign Key: product\_id (References Product.product\_id)

Fields: product\_id, sku\_code, quantity

Order Table:

Table Name: Order

Primary Key: order\_id

Foreign Key: customer\_id (References Customer.customer\_id)

Fields: customer\_id, order\_date, order\_description, order\_amount

OrderDetails Table:

Table Name: OrderDetails

Primary Key: order\_details\_id

Foreign Keys: order\_id (References Order.order\_id), product\_id (References Product.product\_id)

Fields: order\_id, product\_id, quantity

Notification Table:

Table Name: Notification

Primary Key: notification\_id

Foreign Key: customer\_id (References Customer.customer\_id), promotional\_id (References Promotional.promotional\_id)

Fields: customer\_id, promotional\_id, message, date\_send, status, notification\_type, related\_entity\_type, related\_entity\_id

PushNotification Table:

Table Name: PushNotification

Fields: message, date\_sent, notification\_type, related\_entity\_type, related\_entity\_id

Promotional Table:

Table Name: Promotional

Primary Key: promotional\_id

Fields: title, description, start\_date, end\_date

**In summary, the cardinality relationships between the entities are as follows:**

One Seller can have Many Inventories and Many Authentication Tokens.

One Inventory belongs to One Seller and One Product.

One Product can have Many Inventories, Many Skus, and Many OrderDetails.

One Customer can have Many Addresses, Many Orders, Many Authentication Tokens, and Many Notifications.

One Address belongs to One Customer and One City.

One City can have Many Addresses.

One Sku belongs to One Product.

One Order belongs to One Customer and can have Many OrderDetails.

One OrderDetail belongs to One Order and One Product.

One Notification belongs to One Customer and One Promotional.

One Promotional can have Many Notifications.