**Database Design Assignment**

**Inventory Module**

**Assumptions**

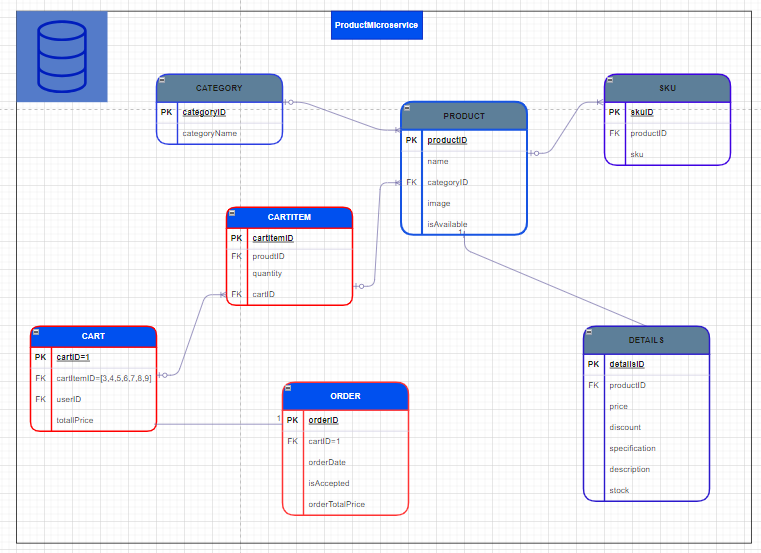
* There are complex filters on product search similar to amazon.
* Products have a fixed number of parameters to be stored.

Note: If there are no complex filters then NOSQL will be better here because of its scalability and Flexibility.

**Database choice**

* SQL (PostgreSQL/Cockroach DB)

**Entity Relationship diagram**



**Search**

* Defining proper constraints for fields.
* Write efficient SQL queries, avoid unnecessary joins, and leverage query hints and execution plans when needed.
* **We can create indexes for key fields like price, category, name etc. This will improve the search speed and since the number of write operations is limited, the write overhead because of indexing should not be an issue**.
* **Implement caching mechanisms (e.g., in-memory cache, Redis) to store frequently accessed data and reduce database load.**
* **We can also use Elastic search or ELK stack for better search performance and analytics.**

**Replication**

* **We should use Transactional Replication: Replicates individual transactions in real-time from the source to the target servers. It has real-time consistency, which is important for real-time stock management and price updating.**

**Normalization and Denormalization**

* Apply 3rd Normal Form (3NF) to eliminate redundant data and ensure data integrity.
* Consider denormalization for performance optimization in read-heavy scenarios, carefully evaluating trade-offs with data consistency.

**Indexing**

* **We can create indexes for key fields like price, category, name etc. This will improve the search speed and since the number of write operations are fairly limited, the write overhead because of indexing should not be an issue**.

**Scaling**

**\*Cockroach DB’s serverless services can handle Vertical and Horizontal scaling and Connection pooling automatically.**

**1. Vertical Scaling:**

Increase the resources of the existing PostgreSQL server (CPU, RAM, storage).

Pros:

* Simple to implement.
* Suitable for moderate increases in workload.

Cons:

* Limited scalability compared to horizontal scaling.
* Can become cost-prohibitive for significant resource upgrades.

**2. Horizontal Scaling (Sharding):**

Distribute data across multiple PostgreSQL servers (shards).

Pros:

* Enhanced scalability for large datasets and high transaction volumes.
* Improved read/write performance.

Cons:

* Complex to implement and manage.
* Sharding key selection requires careful consideration.

**3. Read Replicas:**

Create read-only replicas of the primary PostgreSQL server to offload read queries.

Pros:

* Improved read scalability.
* Allows for distributing read traffic across multiple servers.

Cons:

* Eventual consistency between replicas (Should not be an issue because of **Transactional Replication**).
* Write-intensive workloads still rely on the primary server (Should not be an issue because of **low write operations**).

**4. Connection Pooling:**

Implement a connection pooler (e.g., PgBouncer) to efficiently manage and reuse database connections.

Pros:

* Reduces the overhead from establishing new database connections.
* Helps handle a larger number of concurrent connections.

Cons:

* Requires careful configuration to avoid connection-related issues.

**Order/Cart Module**

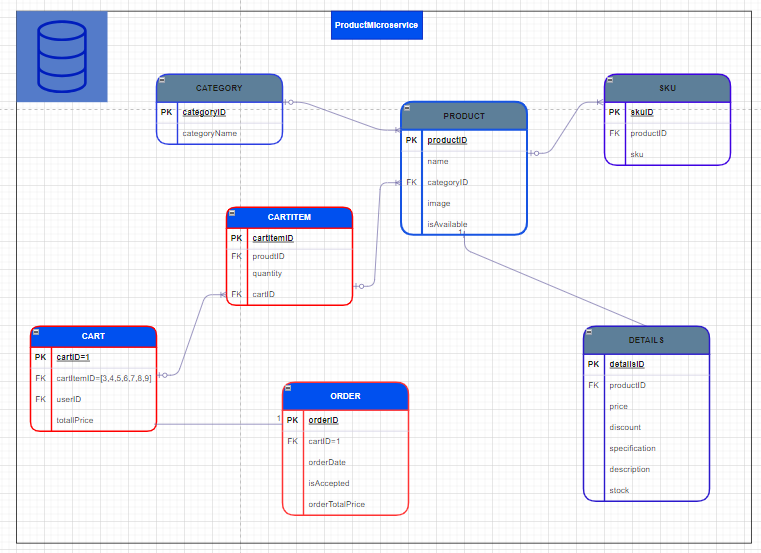
**Assumptions**

* There are complex filters on seller dashboard for products.
* Products have a fixed number of parameters to be stored.

**Database choice**

* SQL (Postgres/Cockroach DB)

**Entity Relationship diagram**



**Search (Same as Inventory Module)**

* Defining proper constraints for fields.
* Write efficient SQL queries, avoid unnecessary joins, and leverage query hints and execution plans when needed.
* **We can create indexes for key fields like price, category, name etc. This will improve the search speed and since the number of write operations are fairly limited, the write overhead because of indexing should not be an issue**.
* **Implement caching mechanisms (e.g., in-memory cache, Redis) to store frequently accessed data and reduce database load.**
* **We can also use Elastic search or ELK stack for better search performance and analytics.**

**Replication (Same as Inventory Module)**

* **We should use Transactional Replication: Replicates individual transactions in real-time from the source to the target servers. It has real-time consistency, which is important for real-time stock management and price updating.**

**Normalization and Denormalization (Same as Inventory Module)**

* Apply 3rd Normal Form (3NF) to eliminate redundant data and ensure data integrity.
* Consider denormalization for performance optimization in read-heavy scenarios, carefully evaluating trade-offs with data consistency.

**Indexing (Same as Inventory Module)**

* **We can create indexes for key fields like price, category, name etc. This will improve the search speed and since the number of write operations are fairly limited, the write overhead because of indexing should not be an issue**.

**Scaling (Same as Inventory Module)**

**\*Cockroach DB’s serverless services can handle Vertical and Horizontal scaling and Connection pooling automatically.**

**1. Vertical Scaling:**

Increase the resources of the existing PostgreSQL server (CPU, RAM, storage).

Pros:

* Simple to implement.
* Suitable for moderate increases in workload.

Cons:

* Limited scalability compared to horizontal scaling.
* Can become cost-prohibitive for significant resource upgrades.

**2. Horizontal Scaling (Sharding):**

Distribute data across multiple PostgreSQL servers (shards).

Pros:

* Enhanced scalability for large datasets and high transaction volumes.
* Improved read/write performance.

Cons:

* Complex to implement and manage.
* Sharding key selection requires careful consideration.

**3. Read Replicas:**

Create read-only replicas of the primary PostgreSQL server to offload read queries.

Pros:

* Improved read scalability.
* Allows for distributing read traffic across multiple servers.

Cons:

* Eventual consistency between replicas (Should not be an issue because of **Transactional Replication**).
* Write-intensive workloads still rely on the primary server (Should not be an issue because of **low write operations**).

**4. Connection Pooling:**

Implement a connection pooler (e.g., PgBouncer) to efficiently manage and reuse database connections.

Pros:

* Reduces the overhead from establishing new database connections.
* Helps handle a larger number of concurrent connections.

Cons:

* Requires careful configuration to avoid connection-related issues.

**Notification Module**

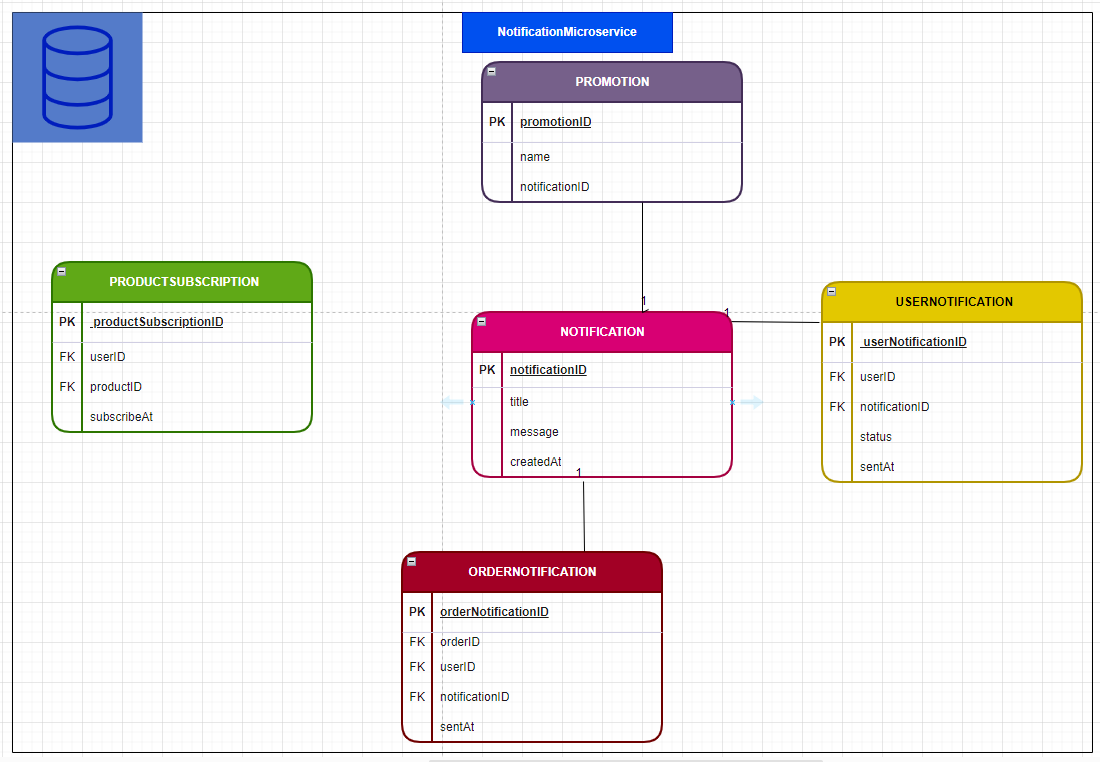
**Assumptions**

* We only need to hold notification information.
* A lot of notifications are being generated and do not have complex schema.

**Database choice**

* NOSQL (MongoDB/CASSANDRA)

**Entity Relationship diagram**



**Search**

* No special consideration needed

**Replication**

* **We should use Merge Replication: Allows updates to occur independently on both the source and target servers. Changes are then merged at scheduled intervals. Suitable for scenarios where updates can occur in multiple locations, and these changes need to be merged into a single dataset.**
* **MongoDB Realm Sync allows you to synchronize data between MongoDB Realm and mobile devices, providing offline capabilities. While it's not traditional merge replication, it provides similar functionality for mobile applications.**

**Normalization and Denormalization**

* No special consideration needed

**Indexing**

* **We can create indexes for key fields like region, user id, subscription id etc.**

**Scaling**

**1. Sharding**

Distribute data across multiple servers or shards (very easy in mongo db.).

Pros:

* Horizontal scalability for large datasets.
* Improved read and write performance.

Cons:

* Sharding key selection requires careful consideration.

**2. Replication**

Create replica sets for high availability and fault tolerance (very easy in mongo db.).

Pros:

* Data redundancy and automatic failover.
* Improved read scalability using read replicas.

Cons:

* Eventual consistency between replicas.
* Write-intensive workloads still rely on the primary replica.

**3. Read Replicas**

Create read-only replicas to distribute read queries.

Pros:

* Improved read scalability.
* Allows for distributing read traffic across multiple servers.

Cons:

* Eventual consistency between replicas.
* Write-intensive workloads still rely on the primary replica.

**4. Capped Collections**

Use capped collections for fixed-size collections.

Pros:

* Ideal for scenarios where old data can be replaced by new data.
* Simplifies collection management.

**5. TTL Indexes**

Use Time-To-Live (TTL) indexes to automatically remove documents after a specified time.

Pros:

* Automatic data expiration.
* Useful for managing time-based data.

Cons:

* Limited use cases (which is fine in this case).

**6. Database Sharding:**

Implement sharding at the database level.

Pros:

* Improved horizontal scalability for multiple collections.
* Simplified management compared to collection-level sharding.

Cons:

* Increased complexity in setup.

**Authentication & Authorization**

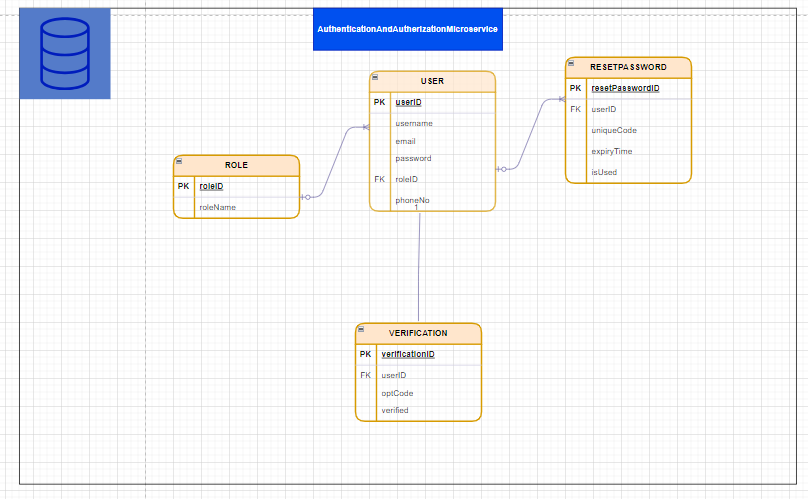
**Assumptions**

* We need to hold user information and preferences (as well as his settings).
* There are no complex queries in this module.

**Database choice**

* NOSQL (MongoDB/CASSANDRA)

**Entity Relationship diagram**



**Search**

* Creating an index for email and phone no. should be sufficient.

**Replication**

* **We should use Transactional Replication: Replicates individual transactions in real-time from the source to the target servers. It has real-time consistency, which is important for real-time password change and other critical user features.**
* **Change Streams in Replication of MongoDB: MongoDB Change Streams allows you to listen for changes that occur in a specific collection. You can use this feature to capture changes and propagate them to other MongoDB instances.**

**Normalization and Denormalization**

* No special consideration needed

**Indexing**

* **We can create indexes for key fields like region, user id, subscription id etc.**

**Scaling**

**1. Sharding**

Distribute data across multiple servers or shards (very easy in mongo db.).

Pros:

* Horizontal scalability for large datasets.
* Improved read and write performance.

Cons:

* Sharding key selection requires careful consideration.

**2. Replication**

Create replica sets for high availability and fault tolerance (very easy in mongo db.).

Pros:

* Data redundancy and automatic failover.
* Improved read scalability using read replicas.

Cons:

* Eventual consistency between replicas.
* Write-intensive workloads still rely on the primary replica.

**3. Read Replicas**

Create read-only replicas to distribute read queries.

Pros:

* Improved read scalability.
* Allows for distributing read traffic across multiple servers.

Cons:

* Eventual consistency between replicas.
* Write-intensive workloads still rely on the primary replica.

**Note: We can use Redis for Caching in all the modules**