9/8/2023

Database Application Group Project Report

Instructor: Tri Dang Tran

Group 14

Team Members:

Dinh Pham: S3878568

Loc Phan: S3938497

Kien Chau: S3790421

Contents

I. Database Design

1. ERD Diagrams

The entity relational diagram visually describes the database conceptual and logical design for the project's specific requirements. Our ERD was drawn using draw.io and exported as a PNG file. Since the files drawing was quite large, therefore, zoom-in with a higher rate would be recommended. Each entity is marked with square brackets, attributes are marked with pill-shaped, and non-relational documents are marked using classes. Attributes with red color are primary keys and blue colors are foreign keys.

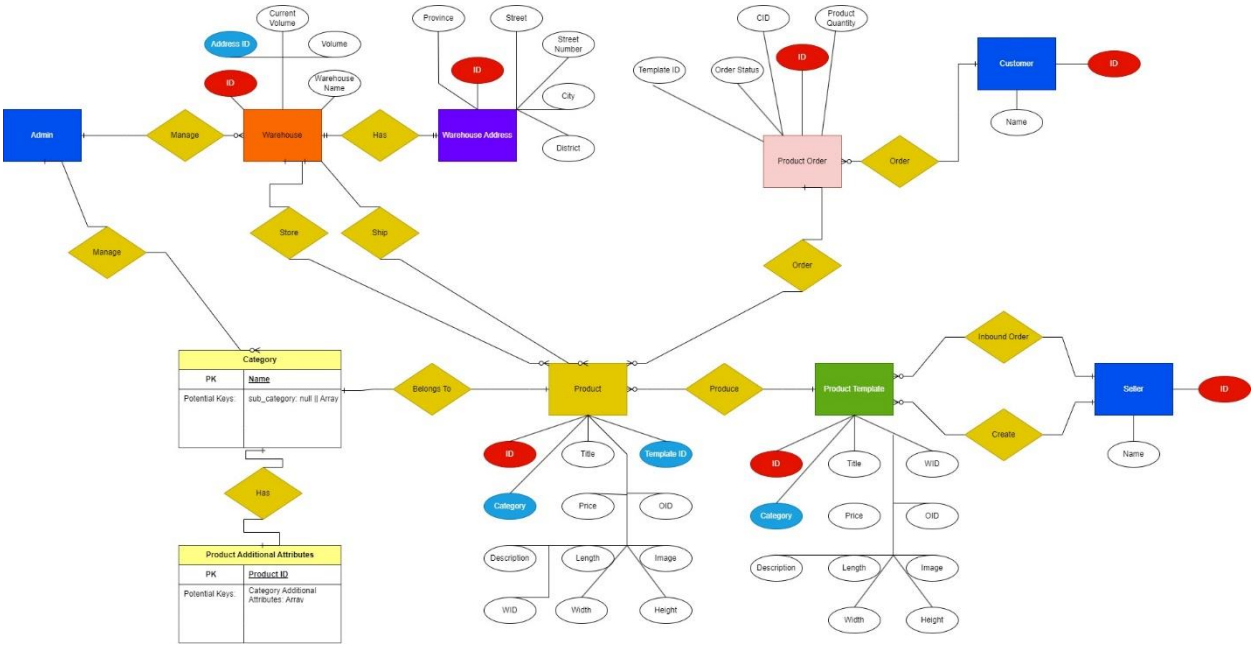


Figure 1. ERD Diagram

Explanation will be provided to elaborate the relationships between entities provided in the diagram above as follow from left to right and top to bottom:

- Each customer can make one or many orders
- Each product order can consume/order one or many products
- Each product belongs to one single category
- Each product can have additional attributes of the category it belongs to
- A warehouse can store one or many products, as long as its volume is sufficient
- A warehouse can ship one or many products
- A product template can produce one or many products based on the inbound order quantity of seller
- A seller can create one or many product templates
- A seller can make one or many inbound orders using created product template(s)

- An admin can manage warehouses with full CRUD support
- An admin can manage categories with full CRUD support

The above ERD can be translated into the following Relational Schema to better show the table structures:

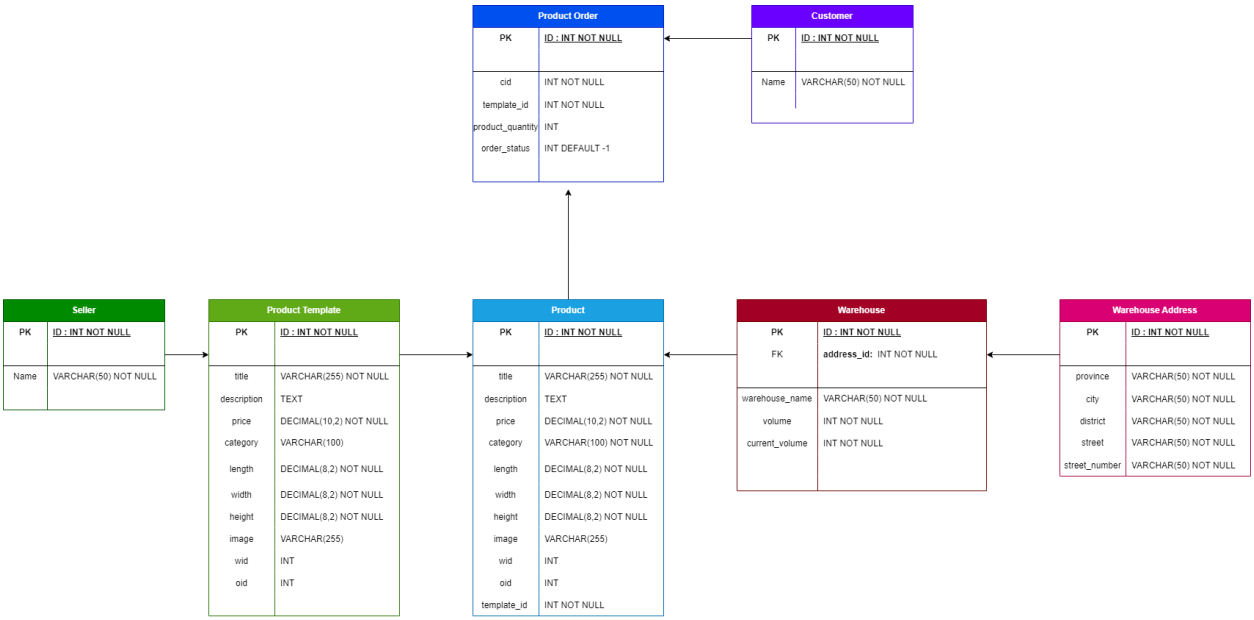


Figure 2. Relational Schema

This section will further explain and clarify the data analysis and database design that our team has implemented in this project.

Since there is a special step when a seller can make an inbound order to order many products, we think it is much more appropriate to first let the user create their product template using the **product_template** table, so these templates can be supported with read, update, and delete operations. A template can be updated & deleted if and only if this template currently has no products left to keep product consistency in the database. This table will act as a prototype to create products. Created products will have a one-to-one relationship with the **product_template**, like one factory's supply chain can create many products, before the supply chain's prototype is changed.

When the seller wants to order products, they will specify the quantity, and the server will create the requested products. Each product will get similar information that can't be updated by the seller later because in the real world, created products can't change their physical structures. However, since they are real instances, they will have a distinctive **id** in the **product** table. Therefore, these products can be read and deleted, but not updated. We believe this would help product ordering and product management much easier.

Each created product will get stored to a warehouse that has the most available volume. When all warehouses are full, the extra product will have the warehouse ID set to null, and these products will be sent to admin, so admin can choose to move them to the warehouse when

volumes are available later, or discard them (sending them back to the seller). We discussed discarding excess products in the first place by terminating them when volume exceeded, however, this approach would add an interesting feature to our application.

Each warehouse's address will be vertically partitioned into another table **warehouse_address** (MySQL doesn't support vertical partition) so the database is much more organized.

When a customer orders products, our team will use the **template_id** attribute of the **product** table to let the user order many items at the same time. For each template in the cart, there will be a new order added into the product table. For instance, in the cart, the customer may have 3 different templates, then there will be 3 orders made to fulfill the customer's cart. Essentially, when these products are added to a customer cart, these products won't be available to other customers. In the future development, these orders can have a set amount of time before expiration and returned to their original warehouses rather than waiting for the customer to accept/reject the whole cart order. One of the examples is **duong sat Viet Nam** (https://dsvn.vn) where they allow customers to reserve the train ticket for 500 seconds before the ticket is released and be available to other customers.

By having the attribute **oid** in the table product not constrained and nullable, we can simply make a many to one relationship between products and an order by setting **oid** of each product to the current order. When the customer accepts the order, then all these products with current order id will be removed from the warehouse. Otherwise, we just need to set those **oid** columns back to null.

2. Constraint Relationships

In this project, our team wanted to have flexibility on **product** table's order tracking and warehouse tracking. Specifically, sellers can make as many inbound orders as they want, and the extra products that have not been stored in any warehouse due to insufficient storage will be sitting on a wait-list. This wait-list can be viewed by the admin and the admin can decide to either move these waiting products to any warehouse that has volume availability after, or decline the waiting product.

Therefore, there is one foreign key constraint only in the **warehouse** table where a warehouse must have an address id that references an id in the **warehouse_address** table.

Last but not least, all tables have a primary key constraint using the **id** column.

3. Non-relational documents

In this project, MongoDB was used as a non-relational database to store a **category** table/collection. Additionally, it also stores a **product_template** collection that purposefully stores created products with their additional attributes inherited from the **category** the product belongs to. Because the categories can have dynamic attributes and structures, therefore, a non-relational database is needed.

As described in the ERD above, each category will mainly have a **name** as primary key, with potential **sub_category** attribute if there are sub categories. Any additional attributes will have a key that represents their name, and the value can either be text or number, required or not required. The documentation follows the **BSON/JSON** format. An example for this would be:

```
{
    “phone” : {
        type: “text/number”,
        require: true/false
    }
}
```

Combining with the name primary key column, we can have a category structure as follow:

```
{
    “name: “electronic”,
    “phone” : {
        type: “text/number”,
        require: true/false
    },
    …,
    sub_category: [
        {
            name: “phone”,
            …
        },

    ]
}
```

As illustrated, each category can be a top category, and each has a sub-category. We designed the format to only contain two levels of parent and child category because for CREATE, READ, and DELETE operations are very easy to handle. However, for UPDATE operation, it’s relatively very hard to handle for our team when the category loops go deeper into 3+ levels. Therefore, to support CRUD, our team would like to keep the data in a consistent state with only two levels.

II. Performance Analysis

1. Indexes

In this section, our team would like to present the indexes we have created or not created for each table and elaborate why the indexes were created or not created on that particular table using the table below:

Table 1. Indexes

Table Name	Possible index fields	Indexes	Rational
warehouse	**volume**	**none**	The id field is the most accessed one to identify the warehouse. However, it's already the primary key. The **address_id** key is also accessed a lot. However, in real practice, a warehouse should only contain one address. Therefore, there is no usage making an index on this field. The volume is an integer and can be sorted. However, in a warehouse, users are more interested and accessing the available volume field more than the maximum volume. Therefore, creating an index wouldn't boost up query processes that much. Since the available volume will be frequently updated, it is no use to create an index there as well.
warehouse_address	**N/A**	**None**	The id field is the most accessed one to identify the warehouse address. However, it's already the primary key.
product_template	**N/A**	**None**	The id field is the most accessed one to identify the template. However, it's already the primary key.

			All the other columns are just placeholders to create products. Therefore, there are no usage creating indexes on them
product	**wid, oid, category**	**warehouse_index(wid) order_index(oid) category_index(category)**	All these columns are integers, frequently accessed to retrieve rows, and can be shared by many products
product_order	**cid, template_id**	**customer_product_order_index(cid), product_order_template_index(template_id)**	These two fields are frequently visited to identify products within an order, as well as the customer. They can also be used for data analysis on products & customer behaviors. Therefore, it's very useful to create indexes on these two columns.
customer	**N/A**	**None**	The id field is the most accessed one to identify the customer. However, it's already the primary key.
seller	**N/A**	**None**	The id field is the most accessed one to identify the seller. However, it's already the primary key.

2. Partitions

In this section, our team would like to present the partitions we have created or not created for each table and elaborate why the partitions were created or not created on that particular table using the table below:

Table 2. Partitions

Table Name	Possible partition fields	Partition Types	Partition Column	Rational
warehouse	**Volume**	**N/A**	**N/A**	There are constraints in the table, as well

				as the warehouse table isn't expected to have lots of data.
warehouse_address	**province**	**N/A**	**N/A**	The warehouse table isn't expected to have lots of data, therefore warehouse_address as well. Unless the project can scale as large as The Gioi Di Dong or Dien May Xanh where there are branches across 68 provinces in Vietnam, we can make a partition using LIST. However, since the input for warehouse addresses is very dynamic right now, we can't define the exact number of partitions we want to make for this table.
product_template	**price, length, width, height**	**N/A**	**N/A**	We allow users to input decimal numbers such as 12.04, 14.08 and so on to calculate taxes in the future, therefore Range partition is not possible. We can partition by hash on the price, however, it's very unpredictable on the partitions and the appropriate partition's number is still unknown for us.
product	**oid, wid, price, length, width, height, template_id**	**N/A**	**N/A**	Template_id, oid, and wid all have their own indexes already, using partitions won't be a wise choice. These values can be a lot and they can also be null as well, therefore, indexes work better on these fields. We allow users to input decimal numbers such as 12.04, 14.08 and so on to calculate taxes in the future, therefore Range partition is not possible. We can partition by hash on the price, however, it's very unpredictable on the partitions and the appropriate partition's number is still unknown for us.

product_order	**order_status**	**LIST**	**order_status**	Since status has three forms: ● -1: Waiting customer ● 0: Order Rejected ● 1: Order Accepted Therefore, we can partition this table into 3 using LIST partition.
customer & seller	**None**	**RANGE**	**None**	These two tables only have id, and a name. ID is already a primary key.
		LIST		
		HASH		
		KEY		

3. Query Optimization

In this project, aside from using partitions and indexes to optimize queries, all data is called and returned on O(1) complexity using stored procedures. Any loop mechanism is limited to O(n) with possible early termination.

Most of the looping is done by the frontend/**NodeJS** client code to make sure the server's responsibility is returning appropriate and single data on each call. Matching search strings, sorting ascending, sorting descending, etc. are mostly done on the front-end side because sorting and finding rows using matching patterns with wildcards are very expensive in database management. Therefore, we make API calls to the server to get the whole data in one call, and process them on the front-end.

Moreover, all stored procedures only select certain variables/fields to return data rather than all columns (SELECT x, y, x rather than SELECT *).

By avoiding many to many relationships between tables, we can avoid the loops of joining table to perform complex queries. On table joins, we can also use INNER JOIN of primary keys and foreign keys rather than using WHERE condition.

4. Concurrent Access

Most of the transactions in this database uses **SERIALIZABLE** isolation level offered by MySQL to ensure data consistency, avoiding possible dirty reads and writes, non-repeatable reads, and phantom read phenomena.

Our team is aware that by applying this isolation level, the transactions' concurrency will be very minimal compared to **REPEATABLE-READS** isolation level. Nonetheless, when a customer places the orders in his/her cart, many products in the cart need to be updated, as well as the warehouse's volume and the product order. Therefore, we need to sacrifice performance to lock related rows for updating and keeping data consistency.

III. Data Consistency

1. Triggers

In this project, we have built exactly one trigger in the whole application to satisfy this requirement: **"Use triggers (combined with stored procedures) to automate the inventory update once an order status is updated to Accept or Reject."**.

The trigger is built to check whether there is an update made in the table **product_order.** Because an order is expected to be inserted once, updated once, and potentially be deleted after implementing business logic with stored procedures and transactions. On each update, the order is expected to be updating only the **order_status** field.

This trigger is called after the update on table **product_order** to ensure all orders are in their stable state. This trigger will then execute either **accept_order** or **reject_order** stored procedures based on the condition of the **order_status** on each row. If the status is 0 then **reject_order** is called and **accept_order** is called when status is 1, else nothing happens.

2. Transaction Management

In this project, we have built two transaction management which are **move_product()** and **order_product()**.

move_product() is used to lock a specific warehouse and the product rows that need to be updated. During this transaction, the server is updating the two warehouse's available volumes, as well as the product that's being moved, therefore, these rows are locked with exquisite locks using SERIALIZABLE isolation level. During this time, we don't want any product's inbound orders or product orders to happen. Any attempt to make dirty reads, writes, non-repeatable reads and phantom reads will be blocked to ensure the volume is consistent.

Similarly, **order_product()** is used to lock a specific product order and the product rows that need to be updated. During this transaction, the server doesn't know which warehouse all products belong to since they are assigned randomly, as well as the server is potentially updating many warehouse available volumes at once if the customer accepts the order (when products are shipped, they return more space to warehouse's available volume), therefore, these rows are locked with exquisite locks using SERIALIZABLE isolation level. Additionally, this also makes sure that stocks should be updated consistently as well if the product orders are rejected by the customer and the products are returned to their original warehouses. Any

attempt to make dirty reads, writes, non-repeatable reads and phantom reads will be blocked to ensure the volume is consistent. This also makes sure any customer making the order first will be served first.

IV. Data Security

1. Database permissions

In our database system, we have created several database accounts aside from root accounts to provide appropriate accessibility to the database system. The account along with their roles, permissions, and business usage will be discussed in the table below:

Table 3. Database Permissions

User	Roles	Permissions	Business Logic
root	root user	- All privileges in the database management system	- Create by default
lazada_admin	admin	- Read/write the whole lazada database - Grant other users with temporary privileges	- Since the root user can navigate through all databases, we think having an additional admin will provide another security layer to the database system when this account doesn't modify anything on other databases, but it still has enough flexibility to perform everything in the lazada database.
lazada_seller	seller/tester	- SELECT table product_order - UPDATE, DELETE rows in table product and product_template to make inbound orders - Additional privileges granted by lazada_admin on contract -	- This account is used to stimulate seller actions for testing purpose, as well as it can be provided to potential clients for evaluating the database business logic

lazada_customer	customer/tester	- View table product - UPDATE rows in table product to order products - Additional privileges granted by lazada_admin on contract	- This account is used to stimulate customer actions for testing purpose

2. SQL Injection Prevention

Since our group was using NodeJS MySQL client to drive the queries from frontend to the database, the client offers one of the SQL Injection Prevention methods on its own. The following image describes how SQL Injection Prevention is done using NodeJS:

```
const client = require('../database/mongoDB.js');
const db = require('../database/mySQL.js');

async function countProducts(req, res) {
    console.log("Called");
    var categoryName = req.params.categoryName;
    var query = `
        SELECT COUNT(product.id) as count
        FROM product
        WHERE product.category = ?;
    `;
    db.query(query, [categoryName], async (err, response) => {
        if (err) {
            console.log(err);
            res.send({status : 400});
        }
        else {
            res.send(response[0]);
            return;
        }
    });
}
```

Figure 3. Code Snippet

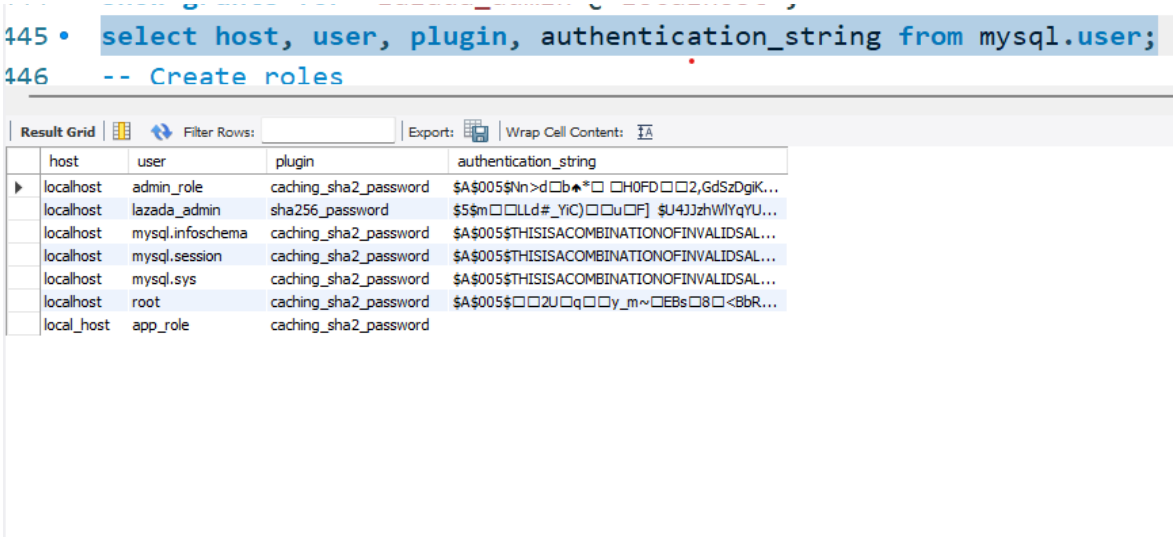
In this image, the query's statement is prepared and computed first, then the dependencies are injected later using the appropriate passed in array values.

We have also implemented another SQL Injection Prevention method, which is using stored procedures. Each stored procedure only accepts certain types of IN parameters with predefined variable types such as INT.

3. Password hashing

By default, on each user creation, MySQL automatically hashes the user password using either MD5 or SHA-1/SHA-2 authentication plugins by default **[1]**. These two hashing functions are

relatively close to each other. As the picture below describes, when running the query on default, MySQL will automatically convert any password string in “IDENTIFIED BY” using declared hashing functions. We used SHA-256 in this case for consistency and to illustrate we have used a password hashing method in this project.



```
145 • select host, user, plugin, authentication_string from mysql.user;
146   -- Create roles
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

host	user	plugin	authentication_string
localhost	admin_role	caching_sha2_password	A005$Nn>d□b♠*□ □H0FD□□2,GdSzDgiK...
localhost	lazada_admin	sha256_password	5m□□LLd#_YiC)□□u□F] $U4JJzhWlYqYU...
localhost	mysql.infoschema	caching_sha2_password	A005$THISISACOMBINATIONOFINVALIDSAL...
localhost	mysql.session	caching_sha2_password	A005$THISISACOMBINATIONOFINVALIDSAL...
localhost	mysql.sys	caching_sha2_password	A005$THISISACOMBINATIONOFINVALIDSAL...
localhost	root	caching_sha2_password	A005$□□2U□q□□y_m~□EBs□8□<BbR...
local_host	app_role	caching_sha2_password	

Figure 4. Password Hashing

As illustrated, other accounts are created using the default plugin of MySQL, which is caching SHA2. However, our accounts are created using SHA256 hash functions to diversify the security layer. For further protection, a salt value can be used to create the hash password. However, we think this is a small application and it would be fine using this SHA256 hash function. It requires high computational power to brute force and to break into any system, the weakest link is the people who hold the high-privileged accounts that have access to the database.

4. Additional security mechanisms

a. MySQL connections

Connections to the database are made via MySQL connection pool using NodeJS dependencies. On each query to the database, the data is fetched using API and connection is released immediately and returned to the pool after receiving the response from the server. Short connections will ensure there won’t be potential breaches during the established section between frontend and backend.

b. Other potential approaches

These **application-layer** security approaches are potential practices that can be used for future projects to further enhance the application security. Even though we have not implemented these practices due to time constraint and human resources, it’s very worth to mention these for future students who can use these for their projects **[3]:**

Table 4. Potential Security Approaches

Method	Rationale
Data Encryption	- This is an extra step that uses encryption to transform plain-text data into more secured data using asymmetric encryption such as RSA.
Real-time database monitoring	Databases should be scanned regularly for breaching attempts to come up with intermediate actions.
Regular Database Back-ups	Databases should be backed up regularly to protect and retrieve data when the server breaks down.
Multi-factor authentication	Add more security mechanisms to verify user access.

References

[1] MySQL, "Chapter 13 Creating User Accounts," N/A. [Online]. Available: https://dev.mysql.com/doc/mysql-secure-deployment-guide/5.7/en/secure-deployment-user-accounts.html. [Accessed 6 September 2023].

[2] Azure, "What is database security?," [Online]. Available: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-database-security/#what-is-database-security. [Accessed 7 September 2023].

[3] K. Thompson, "10 Database Security Best Practices You Should Know," 2 March 2023. [Online]. Available: https://www.tripwire.com/state-of-security/database-security-best-practices-you-should-know. [Accessed 7 September 2023].