RMIT UNIVERSITY

ISYS2099 - Database Applications

Database Project

Lecturer: Dr. Tri Dang

Group: 7

Doan Thien Di	s3926977
Tran Thuc Ai Quynh	s3872104
Do Quang Thang	s3891873

TABLE OF CONTENTS

I. Database Design

In this project we design a database that includes both relational and non-relational databases. In it, we used collections to manage product categories, and tables to manage all other entities. Below are images simulating our relational schema and entity relational diagram.

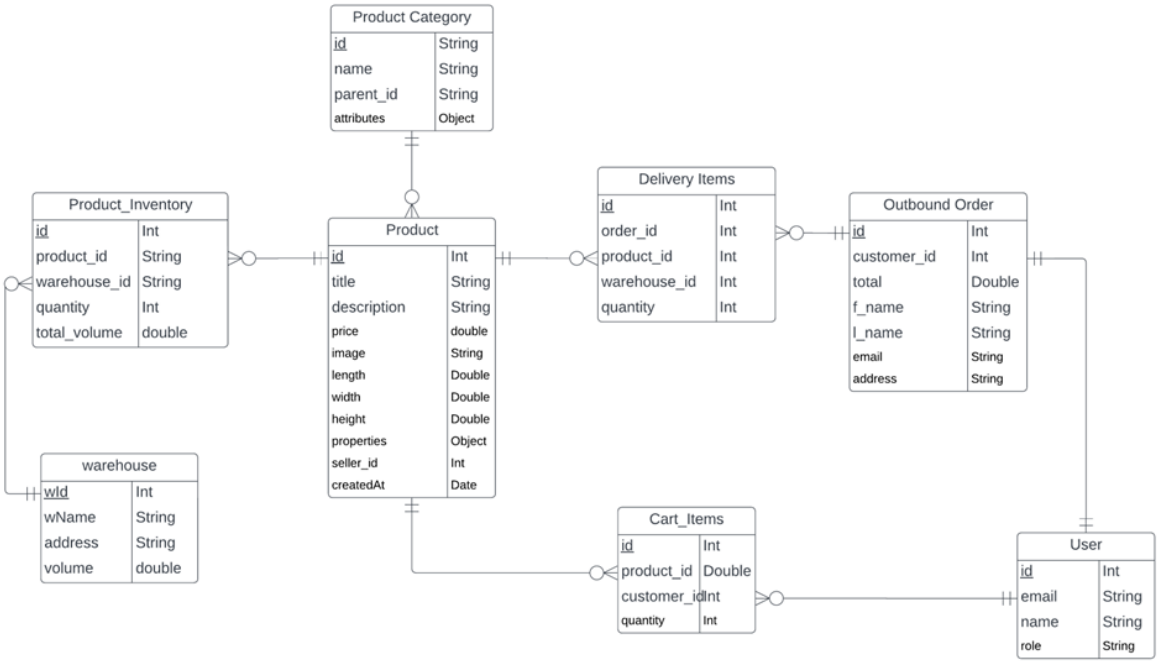

Figure 1. Relational Schema

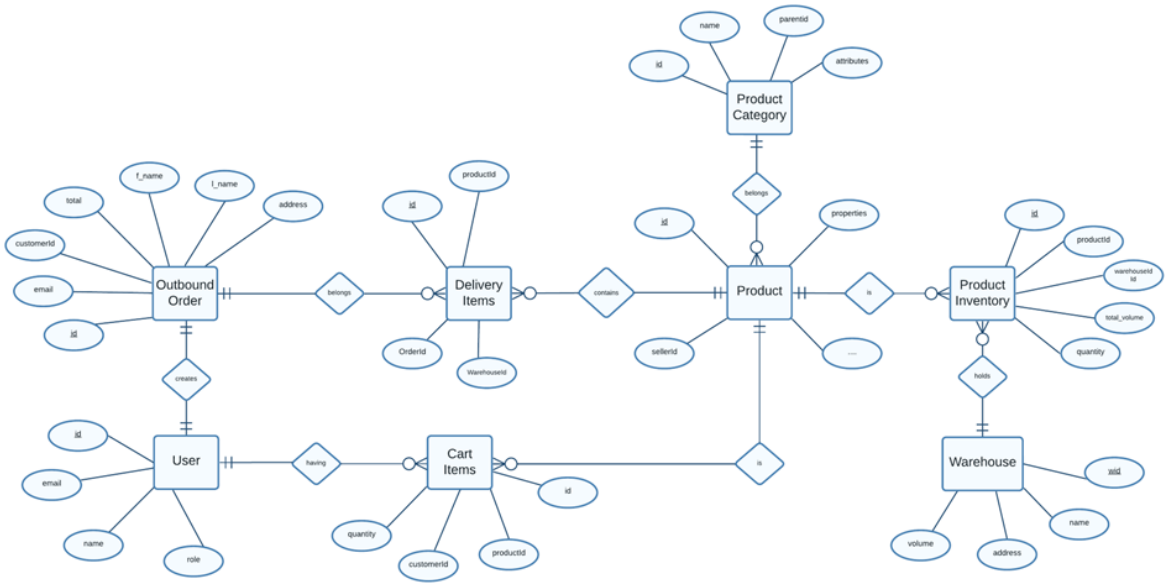

Figure 2. ERD Diagram

As you can see, the entire project includes a total of 7 tables and 1 collection, namely product category, product, warehouse, product inventory, cart items, outbound order, delivery items, and user. This design makes it easier for us to manage data and ensure data consistency. First let's take a look at the category collection model.

Figure 3. Categories collection model

This is the only collection in the project, the reason we use noSql for category is because this data type includes values that refer to other objects like parentCategory and dynamic data types that cannot be predefined like in relational databases. As you can see, the document parent has a data type of object and properties has a data type of an array of objects with unspecified values. Therefore, managing this data using a relational database is extremely difficult. The relationship between product table and category collection is one to many because each product belongs to only 1 category and each category can contain many different products. Other data such as products, warehouses, etc. are designed using a relational database to optimise performance and ensure transactions between users.

As a warehouse admin, in addition to category management, the admin also has the ability to manage information of the company's warehouses. We realised that the warehouse table and product table should have a many to many relationship, so we created a new table including 2 foreign keys, productId and warehouseId, to manage this data, called table product_inventory. Table product and warehouse will have a one to many relationship with the new table, in addition, this table also contains additional information about quantity of each product in a warehouse and total volume of each product. Admins can also move products back and forth between warehouses they manage. We have implemented a stored procedure for this task, called moveProduct. This Stored procedure also used transactions to ensure the total number of products in the warehouse before and after the move.

As a seller, users can manage their products and send their products to the appropriate warehouse. The stored procedure for this task is named inboundOrder. In particular, the system will arrange warehouses according to available space and insert products into each warehouse until there is no warehouse available or there is no more product to add.

Finally, the customer role allows users to order and place orders on the system. Tables involved in this process include product, cart_items, user, outbound_order, and delivery_items. In particular, for simplicity, we assume that each account can only create one order until the entire delivery process is finished, then that user can create a new order. According to that hypothesis, like product and warehouse, we realise that the relationship between product and customer(user) tables is many-to-many, so we create a new table to link these two tables, called table "cart_items". This table includes 2 foreign keys: productId and customerId. Users can interact with this table through actions such as adding products to the cart, or increasing or decreasing the number of products. After choosing the desired product, users can move to the checkout stage. We have created a procedure for this task, called the "checkout" procedure. In which, the system will check the valid quantity of each product in the cart, compare it with the quantity of each product in the warehouse, and remove the invalid product. This process involves transaction and concurrency techniques. After filling in the necessary information in the checkout form, the user can place the order. Specifically, the system will create an outbound order for that account, and the products in the cart will be inserted into the delivery items table along with the warehouseId of that product. Finally, when the user accepts or rejects an order, the system will activate a trigger named delivery. This trigger will deduct the product from the warehouse appropriately if the order is accepted, otherwise, return the product to the corresponding warehouse if the order is rejected. After that, all cart_items, outbound_order, delivery_items related to this user will be deleted from the system. The relationship between the user table and outboundOrder is one to one because each user can only create 1 order at the time. The relationship between the products table and Delivery_items is one to many, similar to the outbound_order table and the Delivery_items table.

II. Performance Analysis

- **Index:**

While working on the Customer role, there are several different filter functions that require a lot of search queries, which we then decided to use Index.

```
-- OPTIMIZATION FOR BROWSING
ALTER TABLE product
ADD INDEX idx_product_title (title);

ALTER TABLE product
ADD INDEX idx_product_description (description);

ALTER TABLE product
ADD INDEX idx_product_category (category);

ALTER TABLE product
ADD INDEX idx_product_price (price);

ALTER TABLE product
ADD INDEX idx_product_createdDate (createdAt);
```

Figure 4: Indexes of table Product

We create *index idx_product_title* and *idx_product_description* for search queries, and *idx_product_category* for browse by category queries. Indexes for price and createdDate were also made for sorting purposes in browsing through our products. By doing this it makes the filter work more efficiently for Customer uses.

- **Query optimization:**

There are a few points that we made in our project to help the query optimization.

1. Use necessary indexes: as stated above, 5 indexes were created to help our customers to use our website more conveniently. Besides, we also did not create any indexes that would not be used to reduce the potential wasted time.
2. Use the wildcards at the end of the phrases only: to retrieve data faster, it is recommended to use wildcards at the end of the phrases, which was what we did for our search function.
3. Avoid too many JOINs: as the server may be overloaded if we add too many tables together in a specific query, it is necessary to not use too many JOINs.

On the other hand, there are many limitations in our system of supporting the query performances. For instance:

1. No partitions for sorting functions
2. Use SELECT * instead of SELECT field names
3. Use SELECT WHERE instead of SELECT INNER JOIN

Acknowledging the shortcomings of our system, we learnt from our mistakes and will change if we have a chance to work on the project again

- **Concurrent access:**

For concurrent access, we use the *Serializable* for the isolation level. As the highest isolation level, serializability requires both read and write locks at the end of each transaction. By implementing this isolation level, each record is ensured to be consistent in different transactions.

III. Data Consistency

Data Consistency. In this section, you must describe/explain how you use triggers and transaction management features to ensure data consistency.

```
create procedure moveProduct(old_wid int, new_wid int, inventory int, product int, moveQuantity int)
begin
    declare product_volume double;
    declare exist_inventory int;
    declare message varchar(225);

    declare `_rollback` int default 0;
    declare CONTINUE HANDLER FOR SQLEXCEPTION set `_rollback` = 1;

    set session transaction isolation level serializable;
    start transaction;
```

Figure 5: moveProduct procedure

Transaction at isolation level serialisation458 is applied when moving products from one warehouse to another in order to maintain data consistency. When a specific product is moved, the transaction will automatically convert all plain SELECT statements to SELECT ... FOR SHARE, stopping other actions to modify the affected tables. After the product has been

successfully moved in, the transaction ends for other transactions to continue, the stock number will remain consistent.

```
create procedure checkout(customerId int)
begin
    declare item_count int default 0;
    declare itemId int;
    declare itemQuantity int;
    declare pid int;
    declare `_rollback` bool default 0;
    declare continue handler for sqlexception set `_rollback` = 1;

set session transaction isolation level serializable;
start transaction;
```

Figure 6: checkout procedure

Transaction is also used to ensure availability of products, as customers can only buy instock products. During the transaction, the inventory table will be locked to check for availability. Concurrency problems such as non-repeatable read, as a share lock will be imposed on the affected tables.

```
create procedure placeOrder(total double, customer_id int, f_name varchar(255), l_name varchar(255), email varchar(255), address varchar(255), delivery_s
begin
    declare item_count int default 0;
    declare orderId int;
    declare itemId int;
    declare itemQuantity int;
    declare pid int;
    declare maxWhItemId int;
    declare maxWhId int;
    declare maxWhItemQuantity int;

    declare remainItem int;
    declare `_rollback` bool default 0;
    declare continue handler for sqlexception set `_rollback` = 1;

set session transaction isolation level serializable;
start transaction;
```

Figure 7: placeOrder procedure

A transaction is applied in the process of placing order to ensure that the stock number is consistent. Transactions will prevent modification on the product_inventory table when an order is placed, preventing lost update problems when changing the inventory.

```
drop trigger if exists delivery;
delimiter $$
create trigger delivery after update on outbound_order for each row
begin

    if new.delivery_status = "accept" then
        call acceptDelivery(new.id);
    elseif new.delivery_status = "reject" then
        call rejectDelivery(new.id);
    end if;

    delete from cart_items where cart_items.customer_id = new.customer_id;
    delete from delivery_items where order_id = new.id;
end $$
```

Figure 8: deliver trigger

Triggers are used to automatically update the inventory. Whenever the delivery status of an order is changed, a corresponding procedure will be called to excuse the inventory update.

IV. Data Security

Data Security. In this section, you must describe/explain the usage of database permissions, SQL injection prevention, password hashing, or additional security mechanisms you have applied in this project.

- **Database permissions:**

After analysing the requirements of Lazada, we know that there are 3 most basic accounts needed in the system: Warehouse Administration for categories and warehouse management, Seller for products and inbound orders management, and Customer for their shopping activities and transactions.

1. Warehouse management:

With Warehouse Administration's (Admin) permissions: we have granted full access for these tables: warehouse, products_inventory, product, product_volume, available_space, cart_items, outbound_orderm and delivery_items. Moreover, Admin can use 4 procedures: moveProduct, deleteWarehouse, and createWarehouse.

```
-- USER MANAGEMENT
-- ADMIN
drop user if exists 'admin'@'localhost';
create user 'admin'@'localhost' identified by 'admin';

ALTER USER 'admin'@'localhost' IDENTIFIED WITH mysql_native_password BY 'admin';

drop role if exists admin_role;
create role admin_role;

grant select, insert, update, delete on lazada.warehouse to admin_role;
grant select, insert, update, delete on lazada.product_inventory to admin_role;
grant select, insert, update, delete on lazada.product to admin_role;
grant select, insert, update, delete on lazada.product_volume to admin_role;
grant select, insert, update, delete on lazada.available_space to admin_role;
grant select, insert, update, delete on lazada.cart_items to admin_role;
grant select, insert, update, delete on lazada.outbound_order to admin_role;
grant select, insert, update, delete on lazada.delivery_items to admin_role;
grant execute on procedure moveProduct to admin_role;
grant execute on procedure deleteWarehouse to admin_role;
grant execute on procedure createWarehouse to admin_role;
grant execute on procedure selectWarehouse to admin_role;

grant admin_role to 'admin'@'localhost';
set default role 'admin_role'@'%' to 'admin'@'localhost';
flush privileges;
```

Figure 9. Create and grant to admin account

2. Seller Management:

Our sellers will have full control over the *product* and *product_volume* table and one procedure of getSellerProduct where it fetches all products of a specific seller.

```
-- SELLER
drop user if exists 'seller'@'localhost';
create user 'seller'@'localhost' identified by 'seller';

ALTER USER 'seller'@'localhost' IDENTIFIED WITH mysql_native_password BY 'seller';

drop role if exists seller_role;
create role seller_role;

grant select, insert, update, delete on lazada.product to seller_role;
grant select, insert, update, delete on lazada.product_volume to seller_role;
grant execute on procedure getSellerProduct to seller_role;
-- grant execute on procedure inboundOrder to seller_role;
-- grant execute on procedure insert_inbound to seller_role;

grant seller_role to 'seller'@'localhost';
set default role 'seller_role'@'%' to 'seller'@'localhost';
flush privileges;
```

Figure 10. Create and grant to seller account

3. Customer Management:

Our customers will obviously have full access on their own cart where they can manage which items they want to proceed to check out. Besides, they will have the access to select items in the *product* table where it fetches products per filter and sort purposes.

*

```
-- CUSTOMER
drop user if exists 'customer'@'localhost';
create user 'customer'@'localhost' identified by 'customer';

ALTER USER 'customer'@'localhost' IDENTIFIED WITH mysql_native_password BY 'customer';

drop role if exists customer_role;
create role customer_role;

grant select on lazada.product to customer_role;
grant select, insert, update, delete on lazada.cart_items to customer_role;

grant customer_role to 'customer'@'localhost';
set default role 'customer_role'@'%' to 'customer'@'localhost';
flush privileges;
```

Figure 10. Create and grant to customer account

- **SQL injections:**
    1. Parameterized Query:

This method is to ensure that strings are escaped properly without getting an SQL injection attack. By doing this we do not input the raw data but the question mark "?" will sanitise the input.

```
//placed order
app.post("/placeOrder", (req, res) => {
  const q = "call placeOrder(?, ?, ?, ?, ?, ?, ?)";
  const values = [req.body.total, req.body.customer_id, req.body.f_name, req.body.l_name,
  connection.query(q, values, (err, data) => {
    if (err) return res.json(err);
    return res.json(data);
  });
});
```

Figure 11. Using question mark in a query

2. Input validation:

Whenever anyone tries to sign up, we assign a common format for the email address to validate and make sure it does not contain any code injections. Additionally, before signing in, we check the email one more time by validating the existence within the system. Again, we also use Parameterized Query for validation to prevent any SQL injection attempt.

```
var mailformat = /^([A-Za-z0-9_\-\.])+\@([A-Za-z0-9_\-\.])+\.([A-Za-z]{2,4})$/;
const getUserSql = "SELECT * FROM user WHERE email = ?";

connection.query(getUserSql, [req.body.email], (err, data) => {
  if (err) return res.json({ Error: "Failed to find email in system" });
  if (req.body.email.match(mailformat)) {
```

Figure 12. Email validation

- **Password hashing:**

For password management, we use a hashing algorithm called Bcrypt. Whenever users input a plain password, Bcrypt will convert it into a hash then store it in our database. However, the special feature of Bcrypt which enhanced its uniqueness is it used the concept of salt. By implementing this, it adds another level of security to our application where the password is appended to a random generated salt before fully hashed. This method helps the system to prevent the rainbow table attack as the attacker can guess the password but not the salt.

```
bcrypt.hash(req.body.password.toString(), salt, (err, hash) => {
  if (err) return res.json({ Error: "Error for hashing password!" });

  const values = [req.body.name, req.body.role, req.body.email, hash];

  connection.query(sql, [values], (error, result) => {
    if (error) return res.json({ Error: "Failed to insert data to server" });
    return res.json({ Status: "Success" });
  });
});
```

Figure 13. Password hashing using bcrypt

- **JSON Web Token:**

Since our system is set up for Single Sign On, it is common to use JSON Web Tokens for authentication purposes. When a user wants to log in, the system will send a HTTP request to the server, then it validates the credentials by the input validation and the hashed password comparison. Once it passes the authentication, a token is created, which is used for the user's session. Now at this stage, the token will be used for 2 jobs: the token is stored in the database related to that user, and a cookie response will be joined to the token with a session's expiration time.Then the cookie will be handled with every request or response made between the client and the server. On the client side, every time a request is made, the attached cookie from the server with the access token will check with the database associated with that user. Once it all checks out, the access for the user will be granted.

```
const token = jwt.sign({ id, name, role }, "jwt-secret-key", {
  expiresIn: "1d",
});

res.cookie("token", token);
```

Figure 14. Generating Json web token

V. Appendix:

- Github: https://github.com/doquangthang-zet/Simple_Lazada_FBL_model/
- Youtube: https://youtu.be/vnP9z6DBgCI
- Backup Drive link: https://drive.google.com/file/d/141RTbobM2yBUkVkZSfV2fq_mWDJ5QWPb/view?usp=sharing