

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION COMMUNICATION TECHNOLOGY

Project Report

Shopi – E-commerce Platform

Group members:

Nguyen Hoang Trung Kien - 20235960

Ngo Duc Huy - 20235946

Ha Hieu Minh - 20215223

Pham Thu Quynh - 202417189

Supervised by:

Professor Nguyen Thi Oanh

Capstone Project for IT3290E

Hanoi University of Science and Technology

January 1, 2025

1. Business Process Description

1.1 Business Context

Shopi functions within the e-commerce industry, providing a digital environment that connects individuals and organizations engaged in exchange of goods. The platform's purpose is to support online commercial activities, creating opportunities for trades and interactions between market participants.

1.2 Users

The Shopi platform is built on a three-tiered user structure. Each role is equipped with specialized tools to ensure the e-commerce operation proceeds optimally:

Customers

- Manage their account and personal information
- Browse products and place orders
- Leave reviews on product
- Leave complaints

Vendors

- Manage store information and profile
- Oversee product listing and inventory
- Process and track customer orders
- Handle refund and order related complaints

Admin

- Manage all user accounts (Customers and Vendors)
- Monitor and manage products and orders
- Review and resolve customer reports

1.3. Business Processes

As a **customer**, to access the shopping feature, they must first register or sign in to their account by providing their personal information such as full name, email address, password, phone number, and shipping address.

Once authenticated, customers can browse the product catalog or use the search function (which supports filters by category and price range) to find desired items. When choosing their desired items, user will be able to see its characteristic and most importantly its variant, such as size or color; the customer can then select the variant they want to purchase. Selected products will then be added to the shopping cart for later checkout.

When ready to purchase, customers review their cart, confirm their shipping address and select a preferred payment method to place the order. After an order is submitted, it is automatically forwarded to the corresponding vendor, who is responsible for processing the shipment and payment. Vendors are also required to manually update the shipping status in the system to notify the customer of the current stage of delivery.

Customers can track their order status through the platform. Upon receiving their order, the customer should confirm on the system that they received the product, and they may leave a product review, including a rating (1–5 stars) and an optional comment visible to other users.

If dissatisfied with the service provided by a vendor, customers may submit a complaint via the system. Small order related complaints like refund or returns, are issued through the view order history and are notified to the vendors then are handled manually by them. However, if customers feel a need to report more serious issues, they can through the report features. Reports must include a category, a comment, and the vendor that the user is reporting on; these reports are then forwarded to the admin which will be handling them.

Vendors can create a store through the platform's store creation feature, which is accessible to all users. During this process, they are required to provide store-related information and valid identification documents for verification.

Once the store is established, vendors can add and update product listings by providing details like Product Name, Category, Price, Stock, and Description. Vendors are responsible for keeping product stock and pricing up-to-date. New products must go through the approval process of an admin.

Vendors can view and manage customer orders; from this they can approve or reject incoming orders, view the customer information. When approving an order, vendors must contact the customer for payment. After that, vendors must prepare the product for shipment and update the shipping status manually at each stage of the fulfillment process, which include: "Not shipped", "Shipped", and "Received". If a complaint is filed on an order, its status will be updated to "Complaint filed" and will come with a comment from the customer whom order belongs to, after resolving the complaint, vendors can set the order status to "Complaint resolved". All actions that update the order status will be notified to the customer.

Vendors may also create promotional campaigns by defining the promotion type, the products involved, and the active period to boost sales and visibility.

Administrators are responsible for managing the overall platform, including updating the status of users, stores, and products. One of their main jobs is to regulate new products listed on the platform, new products need to be approved by an admin to be visible to everyone; once a product status is updated, it will be sent as a notification that may include a comment (or rejection reason) from the admin. Any banned or suspended entities will become inaccessible or hidden from public view.

Administrators also have full access to the notification system, enabling them to send messages directly to specific users.

In addition, they handle all customer reports submitted through the report system. Each report can be reviewed and replied to by an administrator; the response is then sent as a notification to the customer, and the report's status is updated accordingly.

1.4. Features

Based on the platform's defined business processes, these are the core database entities and their associated attributes:

Users:

- Username
- Password
- Name
- Phone
- Email
- Shipping address
- Privilege (user, admin)
- Status (active, suspended, banned)
- Registration Date

Stores:

- Store Name
- Owner
- Description
- Logo
- Phone
- Email
- Pickup Address
- Tax code
- Status (active, suspended, banned)
- Total sales
- Total Rating
- Create date

Products:

- Product Name
- Category
- Variant Types (eg. color, size)

- Price
- Stock
- Description
- Product Images
- Status (active, pending approval, removed)
- Total Rating
- Review

Promotion:

- Discount type (fixed price or percent discount)
- Discount value (depends on the type)
- Active period
- Applicable Products

Review:

- Rating (1 to 5 stars)
- Comment
- Replies (Include a comment and replier)
- Reviewer (User who wrote the review)

Order:

- Order Items (Include Product, Quantity, Price per item)
- Shipping address
- Payment method (cash or online transaction)
- Status (Rejected by vendor , Approved by vendor, Unresolved, Complaint filed, Complaint resolved)
- Order Date
- Shipping statuses
- Subtotal

Complaints:

- Related order
- Comment

Shipping status:

- Status (Not shipped, shipped, recieved)
- Update timestamp

Report:

- Category
- Vendor reported on

- Date
- Status (resolved, unresolved)
- Comment
- Sender
- Admin who resolved the report
- Admin note

Notification:

- Sender
- Receiver
- Timestamp
- Type
- Content
- Read status

1.5. Business Rules

User related:

- All users must register with a valid email and phone number and password before accessing the platform.
- Each account type (Customer, Admin) has different privileges and access scopes.
- Suspended or banned users cannot log in or perform any actions until their status is reinstated.

Vendor related:

- Only verified vendors (with approved business license or identification) may create a store.
- All store must have a valid phone, email
- Suspended stores automatically hide all listed products from the public catalog.

Product related:

- Product name, price and stock must be valid and are required for all products
- Vendors are responsible for keeping product stock and pricing accurate.
- Every product must belong to at least one category from the list of valid categories.
- New products require admin approval before becoming visible to customers.
- Product ratings are calculated as the average of all customer reviews.

Product category related:

- A category cannot be deleted if there are existing active products currently associated with it.
- Category names must be unique within the platform.

Promotion related:

- Discount campaigns must include a valid start and end date, discount value and list of applicable products before activation.
- Promotion can only belong to one of two types (Fixed price or Percent discount)

Orders related:

- All orders require valid shipping address and payment method (include cash or online transaction)
- Order status include “Rejected by vendor” ,” Approved by vendor”, “Unresolved”, “Complaint filed” and “Complaint resolved”
- Action that changes the order status will be notified to the other parties
- Both shipping and payment process are handled by vendors
- Vendors must update shipping status manually within the system.
- Shipping status include: “Not shipped”, ”Shipped”, "Received"
- All shipping status require a timestamp on the time it was updated
- Shipping status automatically move to “Recieved” status 14 days after Shipping status is set to “Shipped”.
- When order first enter the system, shipping status will be “Not shipped”

Complaints related:

- Complaints must belong to an order and include a comment

Review related:

- Customers can only leave review if they have purchased the product.
- Reviews must include a rating (1–5 stars) and an optional comment.

Reports related:

- Reports must specify a category from a list of valid categories and include a comment, sender and vendors that are being reported.
- All complaints will have its status be “unresolved” when first created
- Refunds or returns can only be initiated through the complaint process.

Notification related:

- All notification must include a type from a list of valid types and content

1.6. Application Functionalities

1.6.1. Admin features

Function	Manage User Accounts
Description	Allows the admin to edit account details or change account status of users

Constraint	<ul style="list-style-type: none"> - Admin cannot change their own privilege level or deactivate their own account. - Validation: All input fields must pass validation checks (e.g., valid email format, recognized privilege level).
Inputs	<ul style="list-style-type: none"> - UserID (to find, edit, or deactivate) - User Details (name, email, Privilege) - Account Status (active, suspended, banned)
Outputs	The system displays the updated list of users or a confirmation message of the change.

Function	Manage Categories
Description	Allows Admin to create, edit, delete, and organize product categories.
Constant	<ul style="list-style-type: none"> - Uniqueness: Category names must be unique within the platform. - Deletion Check: A category cannot be deleted if there are existing active products currently associated with it. - Input Length: Category Name must adhere to a defined character limit (e.g., max 50 characters).
Inputs	<ul style="list-style-type: none"> - CategoryID (to find, edit, or deactivate) - Category Name
Outputs	Category structure is updated.

Function	Moderate Seller Products
Description	Allows the admin to review, approve, or reject new or edited products submitted by sellers to ensure they meet platform guidelines.
Constraint	<ul style="list-style-type: none"> - Guideline Compliance: Approval/Rejection decision must strictly follow the platform's Content and Product Listing Guidelines. - Mandatory Reason: A Rejection Reason must be provided if the product status is set to "Rejected" or "Removed." - Notification Trigger: Notification to the seller must be triggered immediately upon status update.
Inputs	<ul style="list-style-type: none"> - ProductID (to review) - Status (active, pending approval, removed) - Rejection Reason
Outputs	The product's visibility status is updated. System sends a notification to the seller regarding the decision.

Function	Manage Stores
Description	Allows the admin to manage stores
Constraint	<ul style="list-style-type: none"> - Status Dependency: A Store's status (e.g., 'Suspended', 'Banned') must automatically change the status of all associated products to 'Removed'. - Input Validation: Store Details update requires re-validation of critical information (e.g., Tax ID, Business License validity).
Inputs	<ul style="list-style-type: none"> - StoreID (to find, edit, or deactivate) - Store Details - Store Status
Outputs	System stores the change and display it

Function	Handle reports
Description	Allows the admin to read reports from user, handles it then update its status to “resolved”
Constraint	<ul style="list-style-type: none"> - Access Control: Admin can only view and handle reports where the status is 'Unresolved'. - Audit Trail: Every status update (e.g., to 'Resolved') must be logged with the Admin ID. - Mandatory Action: The Admin must enter a handling note/resolution summary before updating the status to 'Resolved'.
Inputs	<ul style="list-style-type: none"> - ReportID - Status
Outputs	System display the report which includes: <ul style="list-style-type: none"> - Category - Vendor reported on - Date - Status (resolved, unresolved) - Comment - Sender - Admin who resolved the report - Admin note

Function	Send Notification
Description	Allows the send user notifications
Constraint	<ul style="list-style-type: none"> - Target Validation: UserID must be an existing, active user in the database. - Message Length: Message content must not exceed a predefined character limit (e.g., 500 characters).
Inputs	<ul style="list-style-type: none"> - UserID - Notification type - Message
Outputs	The notification is stored in the db then sent to the user

1.6.2. Customer features

Function	Browse and Search Products
Description	Allows customers to browse through available products or search for specific items using keywords or filters such as category, price range.
Inputs	<ul style="list-style-type: none"> - Keywords (optional) - CategoryID, Price Range, Filter Options
Outputs	System displays a list of matching products with images, prices, and basic details.

Function	View Product Details
Description	Allows customers to view detailed information about a specific product, including price, stock availability, vendor name, and customer reviews.
Inputs	ProductID
Outputs	System displays full product details includes: <ul style="list-style-type: none"> - Product Name - Category - Variant Types (eg. color, size) - Price - Stock - Description - Product Images - Total Rating

Function	Add Product to Cart
Description	Allows customers to add selected products to their shopping cart.
Inputs	<ul style="list-style-type: none"> - ProductID - Quantity - CustomerID
Outputs	System updates the shopping cart and displays a confirmation message.

Function	Manage Shopping Cart
Description	Allows customers to update quantities or remove items from their shopping cart before checkout.
Inputs	<ul style="list-style-type: none"> - CustomerID - ProductID - New Quantity or Delete Action
Outputs	System displays the updated shopping cart and calculated total amount.

Function	Place Order
Constraint	Customer needs to log in, provide a valid shipping address and payment method to place an order
Description	Allows customers to confirm and complete their purchase by providing delivery information and payment details.
Inputs	<ul style="list-style-type: none"> - CustomerID - Cart Details (ProductID, Quantity, Price) - Shipping Address - Payment Method
Outputs	Order record is created and confirmation message with OrderID is displayed.

Function	Track Order Status
Description	Allows customers to view the current status of their orders (e.g., Processing, Shipped, Delivered).
Inputs	<ul style="list-style-type: none"> - CustomerID - OrderID
Outputs	System displays real-time order status.

Function	View Order History
Description	Allows customers to view their past purchase history and related details.
Inputs	CustomerID
Outputs	System displays list of previous orders, including products, order date, total amount, and delivery status.

Function	Leave Product Review
Constraint	Customer can only leave a review if they have purchased the product before
Description	Allows customers to write reviews and rate the products they have purchased.
Inputs	<ul style="list-style-type: none"> - ProductID - CustomerID - Rating (1–5 stars) - Review Comment
Outputs	Review is stored and displayed under the product’s feedback section.

Function	Create Account
Description	Allows new users to create an account on the platform to access shopping and order features.
Inputs	<ul style="list-style-type: none"> - Full Name - Email Address - Password - Phone Number - Shipping Address
Outputs	<ul style="list-style-type: none"> - A new account is created. - System displays confirmation message and redirects user to the login or homepage. - An UserID is generated

Function	Leave Complaints
Description	Allows users to leave a complaints on an order regarding products/purchases
Inputs	<ul style="list-style-type: none"> - Related order - Comment
Outputs	<ul style="list-style-type: none"> - The order status is updated to “Complaint filed” and the complained is sent and notified to the vendor

Function	Report a vendor
Description	Allows users to report a vendor for more serious violations
Inputs	<ul style="list-style-type: none"> - Category - Vendor reported on - Comment
Outputs	<ul style="list-style-type: none"> - The order report is forwarded to the admin

1.6.3. Vendor features

Function	Create and Manage Shop
Constraint	Vendor must provide valid licensing to create a store
Description	Allow vendors to create their store page to sell their products
Inputs	<ul style="list-style-type: none"> - UserID - Store Name - Description - Logo - Contact - Pickup Address - CIC or Business License - Bank Account - Tax ID
Outputs	<ul style="list-style-type: none"> - Record is created and confirmation is shown.

Function	Manage Product
Description	This function allows sellers to add, update, or remove products from the catalog.
Inputs	<ul style="list-style-type: none"> - ProductID

	<ul style="list-style-type: none"> - Product Name - Category - Variant Type (e.g., Size, Color) - Price - Stock(y/n) - Description - Product Images
Outputs	<ul style="list-style-type: none"> - Record is created, updated, or removed and confirmation is shown.

Function	Manage Orders
Description	Allows vendors to view order details and update its status
Inputs	<ul style="list-style-type: none"> - OrderID - Order Status (“Rejected by vendor” ,” Approved by vendor”, “Unresolved”, “Complaint resolved”)
Outputs	<ul style="list-style-type: none"> - System displays updated order status and order details and notify the customer of changes in the order status.

Function	View complaints
Description	Allows vendors to view complaints related to an order
Inputs	<ul style="list-style-type: none"> - OrderID
Outputs	<ul style="list-style-type: none"> - System displays the complaint include: - Sender - Message

Function	Manage Shipment
Description	Allows vendors to arrange delivery and generate a shipment.
Inputs	<ul style="list-style-type: none"> - OrderID - Shipping Status - Timestamp
Outputs	<ul style="list-style-type: none"> - Seller updates order tracking status visible to customers in real time.

Function	Manage Product Review
Description	Allows vendors to reply to customer reviews publicly.
Inputs	<ul style="list-style-type: none"> - ProductID - CustomerID - Message
Outputs	<ul style="list-style-type: none"> - System displays seller reply under the review.

Function	View Sales Reports
Description	Allows vendors to view sales data and revenue trends over time.
Inputs	<ul style="list-style-type: none"> - VendorID - Date Range
Outputs	<ul style="list-style-type: none"> - System displays total revenue, number of orders, top products,..

Function	Manage Promotions
Description	Allows vendors to create promotional campaigns such as discounts, bundles, or buy-one-get-one offers
Inputs	<ul style="list-style-type: none"> - PromotionID - Promotion Name - Discount Type - Applied Product - Time
Outputs	<ul style="list-style-type: none"> - Record is created or updated in the promotion list. System displays confirmation and active promotion details.

1.6.4. Common features

Read Notification

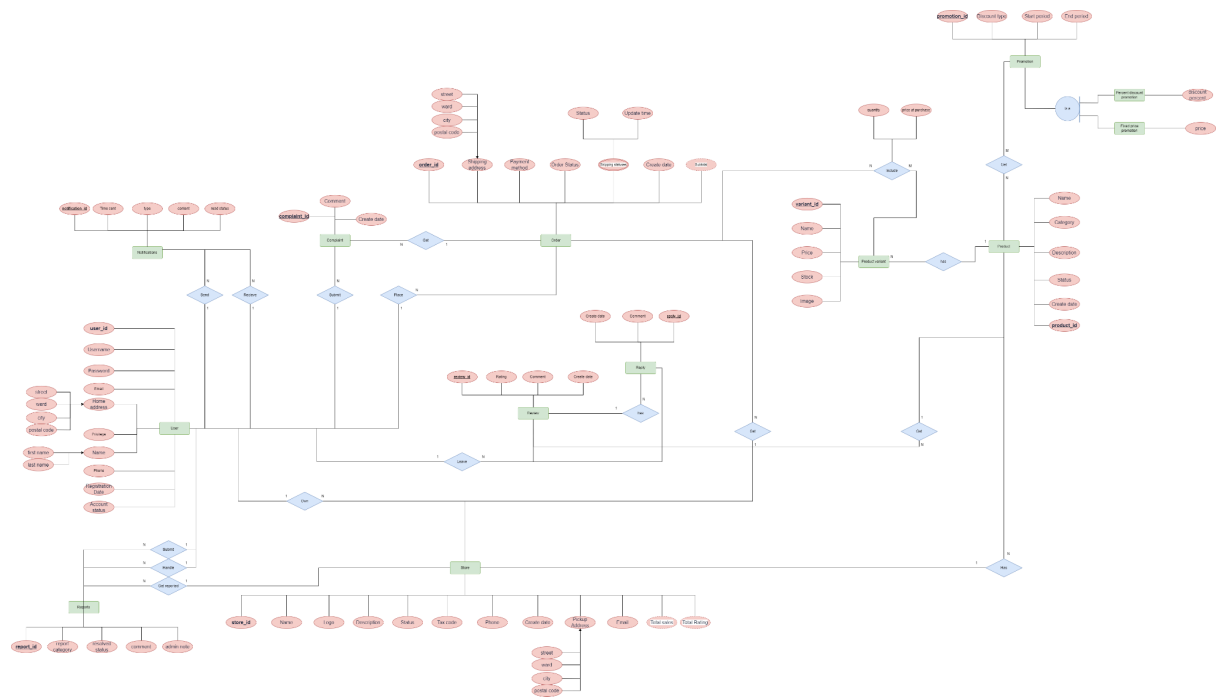
User	Customers, Vendor, Admin
Description	Allow all users to see notifications regarding various aspect of the business process
Inputs	<ul style="list-style-type: none"> - UserID - NotificationID
Outputs	<ul style="list-style-type: none"> - System displays a list of notifications includes: - Sender

- | | |
|--|---|
| | <ul style="list-style-type: none"> - Timestamp - Type - Content - Read status |
|--|---|

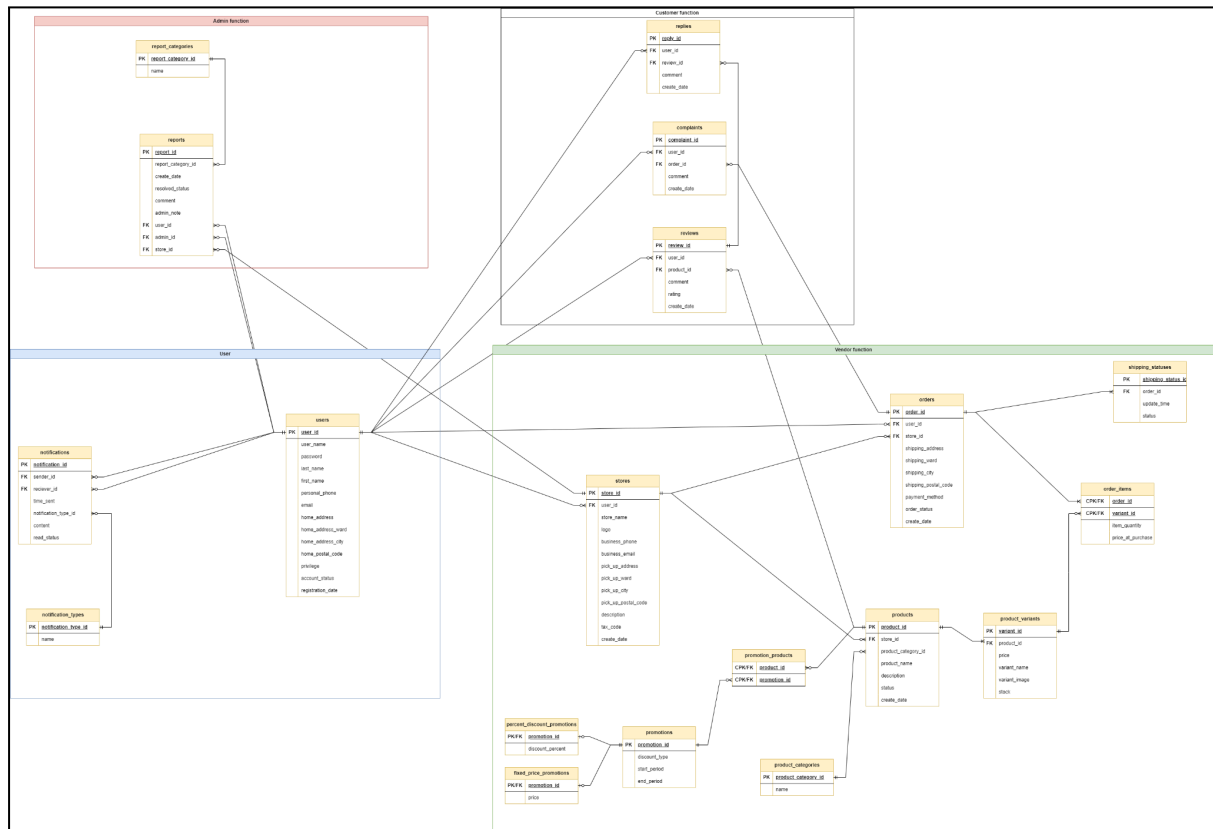
2. Database Design

2.1 Entity–Relationship Diagram (ERD)

2.1.1 Chen's ERD



2.1.2 ERD



2.1.3 Relation Description

- **Users – Stores:** one user can own many stores, but each store belongs to exactly one user.
- **Stores – Products:** each store can sell many products, while each product belongs to one store. This is a one-to-many relationship.
- **Product_categories – Products:** one category can contain many products, but each product belongs to one category. This is a one-to-many relationship.
- **Products – Product_variants:** one product can have multiple variants, while each variant belongs to one product. This is a one-to-many relationship.
- **Users – Orders:** one user can place many orders, but each order is placed by exactly one user. This is a one-to-many relationship.
- **Stores – Orders:** one store can receive many orders, but each order belongs to one store. This is a one-to-many relationship.
- **Orders – Product_variants:** one order can contain many product variants, one product variant can get many orders. This is a many-many relationship

- **Orders – Shipping_statuses**: one order can have multiple shipping status updates, but each shipping status record belongs to one order. This is a one-to-many relationship.
- **Users – Reviews**: one user can write many reviews, but each review is written by one user. This is a one-to-many relationship.
- **Products – Reviews**: one product can have many reviews, but each review refers to one product. This is a one-to-many relationship.
- **Reviews – Replies**: one review can have multiple replies, but each reply belongs to one review. This is a one-to-many relationship.
- **Users – Complaints**: one user can submit many complaints, but each complaint is created by one user. This is a one-to-many relationship.
- **Orders – Complaints**: one order can have multiple complaints, but each complaint belongs to one order. This is a one-to-many relationship.
- **Report_categories – Reports**: one report category can be used by many reports. This is a one-to-many relationship.
- **Users – Reports**: one user can submit many reports, but each report is submitted by one user. This is a one-to-many relationship.
- **Promotions – Promotion_products – Products**: One product can have many promotions applied to it throughout its time in the database, even expired promotions. Promotions and products have a many-to-many relationship, resolved through the promotion_products table.
- **Users – Notifications**: one user can receive many notifications. This is a one-to-many relationship.
- **Notification_types – Notifications**: one notification type can be used by many notifications. This is a one-to-many relationship.

2.2 Table & Field Specifications

2.2.1. Tables overview

Table Name	Fields
users	<u>user_id</u> , user_name, password, last_name, first_name, personal_phone, email, home_address, home_address_ward, home_address_city, home_postal_code, privilege, account_status, registration_date
notification_types	<u>notification_type_id</u> , name

notifications	<u>notification_id</u> , <i>sender_id</i> , <i>receiver_id</i> , <i>time_sent</i> , <i>notification_type_id</i> , <i>content</i> , <i>read_status</i>
report_categories	<u>report_category_id</u> , <i>name</i>
stores	<u>store_id</u> , <i>user_id</i> , <i>store_name</i> , <i>logo</i> , <i>business_phone</i> , <i>business_email</i> , <i>pickup_address</i> , <i>pickup_address_ward</i> , <i>pickup_address_city</i> , <i>postal_code</i> , <i>tax_code</i> , <i>description</i> , <i>create_date</i>
reports	<u>report_id</u> , <i>report_category_id</i> , <i>create_date</i> , <i>resolved_status</i> , <i>comment</i> , <i>admin_note</i> , <i>user_id</i> , <i>admin_id</i> , <i>store_id</i>
product_categories	<u>product_category_id</u> , <i>name</i>
products	<u>product_id</u> , <i>store_id</i> , <i>product_name</i> , <i>product_category_id</i> , <i>description</i> , <i>status</i> , <i>create_date</i>
product_variants	<u>variant_id</u> , <i>product_id</i> , <i>variant_name</i> , <i>price</i> , <i>stock</i> , <i>variant_image</i>
promotions	<u>promotion_id</u> , <i>discount_type</i> , <i>start_period</i> , <i>end_period</i>
fixed_price_promotions	<u>promotion_id</u> , <i>price</i>
percent_discount_promotions	<u>promotion_id</u> , <i>discount_percent</i>
promotion_products	<u>promotion_id</u> , <u>product_id</u>
orders	<u>order_id</u> , <i>user_id</i> , <i>store_id</i> , <i>shipping_address</i> , <i>shipping_ward</i> , <i>shipping_city</i> , <i>shipping_postal_code</i> , <i>payment_method</i> , <i>order_status</i> , <i>create_date</i>
order_items	<u>order_id</u> , <u>variant_id</u> , <i>item_quantity</i> , <i>price_at_purchase</i>
shipping_statuses	<u>shipping_status_id</u> , <i>order_id</i> , <i>update_time</i> , <i>status</i>
complaints	<u>complaint_id</u> , <i>user_id</i> , <i>order_id</i> , <i>comment</i> , <i>create_date</i>
reviews	<u>review_id</u> , <i>user_id</i> , <i>product_id</i> , <i>comment</i> , <i>rating</i> , <i>create_date</i>
replies	<u>reply_id</u> , <i>user_id</i> , <i>review_id</i> , <i>comment</i> , <i>create_date</i>

2.2.2. Table descriptions and field specifications

Note: due to the size of our table description and field specifications, it will be separated into another document. Please refer to ‘G04_TableDescriptionsAndFieldSpecifications.pdf’ for more details

2.2.3. Explanation for each specification detail

This section explains how each **field specification attribute** should be interpreted in our data dictionary, so readers understand the meaning and purpose of every field in the specification.

A) General metadata

- **Specification Type:** Indicates whether the field is defined in its *original table* (**Unique**) or reused from another table (**Replica**).
 1. *Unique*: The source/original definition of a field (e.g., `users.user_id`).
 2. *Replica*: A field copied/implemented in another table to reference the source (e.g., `orders.user_id`).
- **Source specification:** Only applies to **Replica** fields. It specifies where the field originates from (e.g., `orders.user_id` → source is `users.user_id`).
- **Alias:** An alternative/common name used in the system for the same concept (helps readers map different naming styles).
- **Parent table:** The table where the field currently belongs.
- **Description:** A short business description of the field. Recommended format:
 1. define what the field stores,
 2. why it matters to the business process.

B) Physical elements (how data is stored)

- **Data type:** The data category used for storage (e.g., *Alphanumeric*, *Numeric*, *Datetime*). This helps validate input and choose indexes.
- **Length:** Maximum storage length of the field (e.g., 255 characters for text).
- **Decimal places:** Number of digits after the decimal point for numeric fields (e.g., price might have 2).
- **Character support:** Which character sets are allowed in this field, such as:
 - Letters (A–Z)
 - Numbers (0–9)
 - Keyboard symbols (., / \$ # %)
 - Special symbols (© ® ™ Σ π)

C) Logical elements (how data behaves in the schema)

- **Key type:** The role of the field in relationships:
 - *Primary key (PK)*: Uniquely identifies rows in a table
 - *Foreign key (FK)*: References a PK in another table
 - *Non-key*: Regular attribute
- **Key structure:** Whether the key is:
 - *Simple*: single column key
 - *Composite*: multiple columns form the key (common in bridge tables like many-to-many)

- **Uniqueness:** Whether duplicate values are allowed:
 - *Unique:* no duplicates allowed
 - *Non-unique:* duplicates allowed
- **Null support:** Whether the field can be **NULL** (missing/unknown).
- **Required value:** Whether the field must have a value when inserting a record (business requirement).
(Note: “Required value” is business-level; “Null support” is schema-level — they often align but are conceptually different.)
- **Range of values:** Valid constraints for the field values (e.g., rating 1–5, status in a fixed list).
- **Value entered by:** Who provides the value:
 - *System:* auto-generated/defaulted by database/app logic (e.g., **create_date**, IDs)
 - *User:* provided by the user or external actor via the application

3. Built in Triggers and Function

3.1. Function to calculate product rating based on average of all customer reviews

```
CREATE OR REPLACE FUNCTION calculate_product_rating(p_product_id INT)
RETURNS NUMERIC(3,2)
AS $$
DECLARE
    avg_rating NUMERIC(3,2);
BEGIN
    SELECT AVG(rating)
    INTO avg_rating
    FROM reviews
    WHERE product_id = p_product_id;

    -- If there is no review, return 0
    RETURN COALESCE(avg_rating, 0);
END;
$$ LANGUAGE plpgsql;
```

3.2. Trigger on update for order_status in orders, send a notification of type ‘Order update’ to opposing party,

- **Rejected by vendor” ,” Approved by vendor” and “Complaint resolved” should send a notification to the customer**

- **“Unresolved” and “Complaint filed” should send a notification to the store owner**

```
CREATE OR REPLACE FUNCTION trg_fn_notify_order_status_update()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    v_type_id INT;
    v_store_owner_id INT;
    v_sender_id INT;
    v_receiver_id INT;
    v_message_content VARCHAR(500);
BEGIN

    IF (TG_OP = 'UPDATE') AND (NEW.order_status IS NOT DISTINCT FROM
OLD.order_status) THEN
        RETURN NEW;
    END IF;

    SELECT nt.notification_type_id
        INTO v_type_id
    FROM notification_types nt
    WHERE nt.name = 'Order update'
    LIMIT 1;

    IF v_type_id IS NULL THEN
        RAISE EXCEPTION notification_types.name = Order update not
found;
    END IF;

    SELECT s.user_id
        INTO v_store_owner_id
    FROM stores s
    WHERE s.store_id = NEW.store_id;

    IF v_store_owner_id IS NULL THEN
        RAISE EXCEPTION 'Store owner not found for store_id=%',
NEW.store_id;
    END IF;

    IF NEW.order_status IN ('Rejected by vendor', 'Approved by vendor',
'Complaint resolved') THEN
        v_sender_id := v_store_owner_id;
        v_receiver_id := NEW.user_id;
    ELSIF NEW.order_status IN ('Unresolved', 'Complaint filed') THEN
```

```

        v_sender_id    := NEW.user_id;
        v_receiver_id := v_store_owner_id;
ELSE

    RETURN NEW;
END IF;

IF (TG_OP = 'INSERT') THEN
    v_message_content := 'New Order #' || NEW.order_id || ' placed.
Status: ' || NEW.order_status;
ELSE
    v_message_content := 'Order #' || NEW.order_id || ' status
updated: ' || NEW.order_status;
END IF;

INSERT INTO notifications (
    sender_id,
    receiver_id,
    notification_type_id,
    content
)
VALUES (
    v_sender_id,
    v_receiver_id,
    v_type_id,
    v_message_content
);

RETURN NEW;
END;
$$;

```

```

DROP TRIGGER IF EXISTS trg_notify_order_status_update ON orders;

```

```

CREATE TRIGGER trg_notify_order_status_update
AFTER INSERT OR UPDATE OF order_status
ON orders
FOR EACH ROW
EXECUTE FUNCTION trg_fn_notify_order_status_update();

```

3.3. Trigger to insert into shipping_status with status “Not shipped” when an order is created

```

CREATE OR REPLACE FUNCTION set_default_shipping_status()
RETURNS TRIGGER AS $$
BEGIN

```

```

        INSERT INTO shipping_status (order_id, status, created_at)
        VALUES (NEW.id, 'Not shipped', NOW());
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS trg_create_shipping_status ON orders;
CREATE TRIGGER trg_create_shipping_status
AFTER INSERT
ON orders
FOR EACH ROW
EXECUTE FUNCTION set_default_shipping_status();

```

3.4. Function to move insert into shipping status "Received" status 14 days after Shipping status of the order is set to "Shipped"

```

CREATE OR REPLACE PROCEDURE auto_complete_shipping()
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO shipping_status (order_id, status, created_at)
    SELECT
        s.order_id,
        'Received',
        NOW()
    FROM shipping_status s
    WHERE s.status = 'Shipped'
        -- Check if the shipped status is older than 14 days
        AND s.created_at < NOW() - INTERVAL '14 days'
        -- EXCLUSION: Ensure this order doesn't already have a 'Received'
entry
        AND NOT EXISTS (
            SELECT 1
            FROM shipping_status existing
            WHERE existing.order_id = s.order_id
            AND existing.status = 'Received'
        );
END;
$$;

```

3.5. Trigger to check for 'Customers can only leave review if they have purchased the product.' RAISE EXCEPTION when an this rule is broken.

```

CREATE OR REPLACE FUNCTION check_purchase_before_review()
RETURNS TRIGGER AS $$

```

```

BEGIN
    IF NOT EXISTS (
        SELECT 1
        FROM orders o
        JOIN order_items oi ON o.order_id = oi.order_id
        WHERE o.user_id = NEW.user_id
            AND o.order_status IN ('Delivered', 'Completed')
            AND oi.variant_id IN (
                SELECT variant_id
                FROM product_variants
                WHERE product_id = NEW.product_id
            )
    ) THEN
        RAISE EXCEPTION 'You can only review products you have purchased
and received.';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER trg_check_purchase_before_review
BEFORE INSERT ON reviews
FOR EACH ROW
EXECUTE FUNCTION check_purchase_before_review();

```

3.6. Trigger to automatically register price_at_purchase when an order_item is placed. This need to account for promotion aswell.

```

CREATE OR REPLACE FUNCTION set_order_item_price()
RETURNS TRIGGER AS $$
DECLARE
    rec RECORD;
    v_price_after_percent DECIMAL(10, 2);
BEGIN
    SELECT
        pv.price AS base_price,
        MIN(fpp.price) AS best_fixed_price,
        MAX(pdp.discount_percent) AS best_percent
    INTO rec
    FROM product_variants pv
    LEFT JOIN promotion_products pp ON pv.product_id = pp.product_id
    LEFT JOIN promotions p ON pp.promotion_id = p.promotion_id
    AND CURRENT_TIMESTAMP BETWEEN p.start_period AND p.end_period
    LEFT JOIN fixed_price_promotions fpp ON p.promotion_id =

```

```

fpp.promotion_id
    LEFT JOIN percent_discount_promotions pdp ON p.promotion_id =
pdp.promotion_id
    WHERE pv.variant_id = NEW.variant_id
    GROUP BY pv.price;
    IF rec.best_percent IS NOT NULL THEN
        v_price_after_percent := rec.base_price * (1.0 -
(rec.best_percent / 100.0));
    END IF;
    NEW.price_at_purchase := LEAST(
        rec.base_price,
        rec.best_fixed_price,
        v_price_after_percent
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_set_order_item_price
BEFORE INSERT ON order_items
FOR EACH ROW
EXECUTE FUNCTION set_order_item_price();

```

3.7. Trigger to check if admin_id in reports table is someone with admin privilege

```

CREATE OR REPLACE FUNCTION check_report_admin_privilege()
RETURNS TRIGGER AS $$
BEGIN

    IF NEW.admin_id IS NOT NULL THEN

        IF NOT EXISTS (
            SELECT 1
            FROM users
            WHERE user_id = NEW.admin_id
            AND privilege = 'Admin'
        ) THEN
            RAISE EXCEPTION 'Violation: The user assigned to Report #%(
User ID %) does not have Admin privileges.', NEW.report_id,
NEW.admin_id;
        END IF;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```
DROP TRIGGER IF EXISTS trg_check_report_admin ON reports;
```

```
CREATE TRIGGER trg_check_report_admin  
BEFORE INSERT OR UPDATE OF admin_id  
ON reports  
FOR EACH ROW  
EXECUTE FUNCTION check_report_admin_privilege();
```

4. Query and optimization

This part includes the queries written by each member. All performance is evaluated using execution time of EXPLAIN ANALYZE as optimization success. This number is recorded after a warm up run to eliminate caching variability

Hà Hiểu Minh 20215223's query:

1. Query to search for a product from their name or description. In this example i chose the keyword to be 'LED light'

```
SELECT product_id  
FROM products  
WHERE product_name LIKE '%LED light%'  
OR description LIKE '%LED light%';
```

Initial performance: Execution Time: 0.112 ms - Planning Time: 0.076 ms

Optimized query:

```
SELECT product_id  
FROM products  
WHERE to_tsvector('english', product_name || ' ' || description)  
@@ plainto_tsquery('english', 'LED light');
```

Optimized performance: Execution Time: 0.033 ms Planning Time: 0.112 ms

Optimization method: GIN index is created on product_name and description in order to support full text search

2. Get the notifications list that belongs to a user to show them. Useful for application display. For this example i used user_id = 2

```
SELECT *  
FROM notifications  
WHERE receiver_id = 2
```

Initial performance: Execution Time: 0.131 ms Planning Time: 0.044 ms

Optimized query:

```
SELECT sender_id, time_sent, notification_type_id, content, read_status  
FROM notifications  
WHERE receiver_id = 2  
ORDER BY time_sent  
LIMIT 10
```

Optimized performance: Execution Time: 0.041 ms Planning Time: 0.074 ms

Optimization method:

- Select * is removed
- BTREE composite index is placed in 2 columns (receiver_id, time_sent) in this order
- LIMIT and ORDER is used to simulate application usually only display the top most recents notifications

3. Get a list of customers that have a buying streak of 3 days in a row. Good for getting data on high quality customer

```
WITH unique_orders AS (  
    SELECT DISTINCT user_id, create_date  
    FROM orders  
)  
tmp AS (  
    SELECT user_id, create_date::date - ROW_NUMBER() OVER(PARTITION BY user_id ORDER BY create_date)::int AS streak_start_id  
    FROM unique_orders  
)  
SELECT DISTINCT user_id  
FROM tmp  
GROUP BY user_id, streak_start_id  
HAVING COUNT(user_id) >=3
```

Initial performance: Execution Time: 2.288 ms Planning Time: 0.086 ms

Optimization method: An index on user_id could be created on orders table, but it would cause strains on the system since orders table is updated regularly

4. Quantity sold and revenue of each category of products

```
SELECT name, SUM(item_quantity), SUM(price_at_purchase)
FROM order_items
JOIN orders USING(order_id)
JOIN shipping_statuses ss USING(order_id)
JOIN product_variants USING(variant_id)
JOIN products USING (product_id)
JOIN product_categories USING(product_category_id)
WHERE ss.status = 'received'
GROUP BY 1
```

Initial performance: Execution Time: 2.167 ms Planning Time: 0.625 ms

Optimized query:

```
SELECT name, SUM(item_quantity), SUM(price_at_purchase)
FROM order_items
JOIN orders o USING(order_id)
JOIN product_variants USING(variant_id)
JOIN products USING (product_id)
JOIN product_categories USING(product_category_id)
WHERE EXISTS (
    SELECT 1
    FROM shipping_statuses ss
    WHERE ss.order_id = o.order_id
    AND ss.status = 'received'
)
GROUP BY 1
```

Optimized performance: Execution Time: 1.838 ms Planning Time: 0.850 ms

Optimization method:

- Removed join with shipping_statuses to avoid explosive join
- A partial index is added on the order_id of shipping_statuses, but only on row with status = 'received', because 'received' only take up a small portion of this table

5. Query to get list of unapproved product. Useful for admin users whos job is to manually review these

```
SELECT p.product_name, s.store_name, p.create_date
FROM products p
JOIN stores s ON p.store_id = s.store_id
WHERE p.status = 'pending approval'
ORDER BY p.create_date ASC;
```

Initial performance: Execution Time: 0.065 ms Planning Time: 0.116 ms

Optimized performance: Execution Time: 1.838 ms Planning Time: 0.850 ms

Optimization method:

- A partial index was created on status of the products table but only for where status = 'pending approval' . With the assumption is that most products will have its status be 'active' so searching through the entire table would be wasteful

6. Customers who registered more than a year ago but never purchased anything

```
SELECT user_id FROM users
WHERE user_id NOT IN (SELECT user_id FROM orders)
AND registration_date < NOW() - INTERVAL '1 year';
```

Initial performance: Execution Time: 0.504 ms Planning Time: 0.105 ms

Optimized query:

```
SELECT user_id FROM users u
WHERE NOT EXISTS (SELECT 1 FROM orders o WHERE o.user_id = u.user_id)
AND registration_date < NOW() - INTERVAL '1 year';
```

Optimized performance: Execution Time: 0.016 ms Planning Time: 0.181 ms

Optimization method:

- Instead of using NOT IN, i used NOT EXISTS which massively improved the performance
- An index is also placed on user_id of orders table and another one on registration_date of users table

7. Query to display the list of order with order_status = 'Complain filed'. Useful for quick navigation for store owners

```
SELECT o.order_id, c.comment
FROM orders o
JOIN complaints c ON o.order_id = c.order_id
JOIN stores s ON o.store_id = s.store_id
WHERE o.order_status = 'Complaint filed' AND o.store_id = 2;
```

Initial performance: Execution Time: 0.332 ms Planning Time: 0.157 ms

Optimized performance: Execution Time: 0.126 ms Planning Time: 0.189 ms

Optimization method:

- Partial index created on order_status where order_status = 'Complain filed'

8. Calculate the average time difference between an order being created and it getting shipped

```
SELECT AVG(ss.update_time - o.create_date)
FROM orders o
JOIN shipping_statuses ss ON o.order_id = ss.order_id
WHERE o.store_id = 2
AND ss.status = 'shipped';
```

Initial performance: Execution Time: 0.524 ms Planning Time: 0.141 ms

Optimized performance: Execution Time: 0.200 ms Planning Time: 0.181 ms

Optimization method:

- Added partial composite index (order_id, update_time) on status of the shipping_statuses table. Reason is order_id and update_time are usually query together so it is more efficient

9. Filter for review with low rating. Help stores handle situation where a product flops hard and damage control is needed

```
SELECT * FROM reviews
WHERE rating = 1 AND create_date > NOW() - INTERVAL '1 day';
```

Initial performance: Execution Time: 0.717 ms Planning Time: 0.065 ms

Optimized performance: Execution Time: 0.126 ms Planning Time: 0.078 ms

Optimization method:

- Created an index on reviews(create_date) but only where rating <=2

10. Trigger to check if field admin_id in reports table is someone with admin privilege

```
CREATE OR REPLACE FUNCTION check_report_admin_privilege()
RETURNS TRIGGER AS $$
BEGIN

    IF NEW.admin_id IS NOT NULL THEN

        IF NOT EXISTS (
            SELECT 1
            FROM users
            WHERE user_id = NEW.admin_id
            AND privilege = 'Admin'
        ) THEN
            RAISE EXCEPTION 'The user assigned to Report #%(User ID %) does not have Admin privileges.',
                NEW.report_id, NEW.admin_id;
        END IF;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS trg_check_report_admin ON reports;

CREATE TRIGGER trg_check_report_admin
BEFORE INSERT OR UPDATE OF admin_id
ON reports
FOR EACH ROW
EXECUTE FUNCTION check_report_admin_privilege();
```

Phạm Thu Quỳnh - 202417189

1. Calculate the total stock quantity of each product.

Initial query

```
SELECT p.product_id, p.product_name, sum(stock) as total_stock
FROM products p
LEFT JOIN product_variants USING(product_id)
GROUP BY product_id;
```

Performance: Planning Time: 0.215 ms Execution Time: 1.124 ms

Optimized query

Method: Create index on product_id.

```
CREATE INDEX idx_variants_product_stock ON
product_variants(product_id) INCLUDE (stock);
```

Performance: Planning Time: 0.157 m Execution Time: 0.545 ms

2. Look for products with an average rating below 3.0 stars.

```
SELECT p.product_id, p.product_name, AVG(r.rating) as average_rating
FROM products p
JOIN reviews r USING(product_id)
GROUP BY p.product_id
HAVING AVG(r.rating) < 3.0;
```

3. List the products in the Electronics category and the names of the stores that sell them.

```
SELECT p.product_id, p.product_name, s.store_name
FROM products p
JOIN stores s USING(store_id)
JOIN product_categories pc USING(product_category_id)
WHERE pc.name = 'Electronics';
```

4. Find the top 10 stores with the highest revenue in the system. Revenue is calculated based on orders that have been 'Approved by vendor' or 'Received'.

```
SELECT
    s.store_id,
    s.store_name,
    SUM(oi.item_quantity * oi.price_at_purchase) as total_revenue
FROM stores s
JOIN orders o USING(store_id)
JOIN order_items oi USING(order_id)
WHERE o.order_status IN ('Approved by vendor', 'Received')
GROUP BY s.store_id, s.store_name
ORDER BY total_revenue DESC
LIMIT 10;
```

5. Statistics on the number of orders by status .

```
SELECT o.order_status, COUNT(o.order_id) as order_count
FROM orders o
GROUP BY o.order_status;
```

6. Calculate the total sales revenue for each month.

```
SELECT EXTRACT('year' FROM o.create_date) as order_year,
    EXTRACT('month' FROM o.create_date) as order_month,
    SUM(oi.item_quantity * oi.price_at_purchase) as total_sales
FROM orders o
```

```

JOIN order_items oi USING(order_id)
JOIN product_variants USING(variant_id)
WHERE o.order_status IN ('Received', 'Complaint resolved', 'Approved by
vendor')
GROUP BY order_month, order_year
ORDER BY order_year, order_month;

```

7. Report the number of canceled orders (Status: 'Rejected by vendor') for each store in the current month.

Initial query:

```

SELECT s.store_id, s.store_name, COUNT(o.order_id) as canceled_orders
FROM stores s
JOIN orders o USING(store_id) -- Direct JOIN
WHERE o.order_status = 'Rejected by vendor'
      AND EXTRACT('month' FROM o.create_date) = EXTRACT('month' FROM
CURRENT_DATE)
      AND EXTRACT('year' FROM o.create_date) = EXTRACT('year' FROM
CURRENT_DATE)
GROUP BY s.store_id, s.store_name;

```

Performance:

Planning Time: 0.688 ms

Execution Time: 0.304 ms

Optimized query:

Method: Create a composite index on order_status and create_date and use DATE_TRUNC instead of EXTRACT.

```

CREATE INDEX idx_orders_status_createdate
ON orders(order_status, create_date);

SELECT s.store_id, s.store_name, COUNT(o.order_id)
FROM stores s
JOIN orders o USING(store_id)
WHERE o.order_status = 'Rejected by vendor'
      AND o.create_date >= DATE_TRUNC('month', CURRENT_DATE)
      AND o.create_date < DATE_TRUNC('month', CURRENT_DATE) + INTERVAL '1
month'
GROUP BY s.store_id, s.store_name;

```

Performance:

Planning Time: 0.308 ms

Execution Time: 0.087 ms

8. Calculate the total number of orders and the total spending per customer.

Initial query:

```
SELECT
    u.user_id,
    u.user_name,
    COUNT(DISTINCT o.order_id) as total_orders,
    SUM(oi.item_quantity * oi.price_at_purchase) as total_spent
FROM users u
LEFT JOIN orders o USING(user_id)
LEFT JOIN order_items oi USING(order_id)
GROUP BY u.user_id, u.user_name;
```

Performance: Planning Time: 0.666 ms Execution Time: 8.850 ms

Optimized query:

Method: Instead of JOIN, perform calculations on the sub-tables first to completely eliminate the need to COUNT(DISTINCT).

```
WITH order_totals AS (
    SELECT order_id,
        SUM(item_quantity * price_at_purchase) as order_value
    FROM order_items
    GROUP BY order_id
),
user_stats AS (
    SELECT o.user_id, COUNT(o.order_id) as total_orders,
        SUM(ot.order_value) as total_spent
    FROM orders o
    LEFT JOIN order_totals ot USING(order_id)
    GROUP BY o.user_id
)

SELECT u.user_id, u.user_name,
    COALESCE(us.total_orders, 0) as total_orders,
    COALESCE(us.total_spent, 0) as total_spent
FROM users u
LEFT JOIN user_stats us USING(user_id);
```

Performance: Planning Time: 0.476 ms Execution Time: 4.484 ms

9. Identify "VIP Customers" - those whose total spending exceeds the average spending of all customers in the system.

Initial query:

```
WITH customer_spending AS (  
    SELECT u.user_id, u.user_name, SUM(oi.item_quantity *  
oi.price_at_purchase) as total_spent  
    FROM users u  
    JOIN orders o USING(user_id)  
    JOIN order_items oi USING(order_id)  
    GROUP BY u.user_id  
) , average_spending AS (  
    SELECT AVG(total_spent) as avg_spent  
    FROM customer_spending  
)  
  
SELECT cs.user_id, cs.user_name, cs.total_spent  
FROM customer_spending cs, average_spending avg  
WHERE cs.total_spent > avg.avg_spent  
ORDER BY cs.total_spent DESC;
```

Performance:

Planning Time: 1.292 ms

Execution Time: 20.194 ms

Optimized query:

Method: Create materialized view and index to store customer's total spent.

```
CREATE MATERIALIZED VIEW mv_customer_spending AS  
SELECT u.user_id, u.user_name,  
    SUM(oi.item_quantity * oi.price_at_purchase) as total_spent  
FROM users u  
JOIN orders o USING(user_id)  
JOIN order_items oi USING(order_id)  
WHERE o.order_status IN ('Approved by vendor', 'Received', 'Complaint  
resolved')  
GROUP BY u.user_id, u.user_name;  
  
CREATE INDEX idx_mv_spending_userid ON mv_customer_spending(user_id);  
CREATE INDEX idx_mv_spending_total ON mv_customer_spending(total_spent  
DESC);  
  
SELECT user_id, user_name, total_spent  
FROM mv_customer_spending  
WHERE total_spent > (SELECT AVG(total_spent) FROM mv_customer_spending)
```

```
ORDER BY total_spent DESC;
```

Performance: Planning Time: 0.331 ms Execution Time: 1.385 ms

10. Find customers who have placed more than 5 orders, and whose order percentage with complaints (recorded in the complaints table) exceeds 20% of their total orders.

Initial query:

```
SELECT
    u.user_id,
    u.user_name,
    COUNT(DISTINCT o.order_id) as total_orders,
    COUNT(DISTINCT c.complaint_id) as complaint_orders,
    (COUNT(DISTINCT c.complaint_id)::FLOAT / COUNT(DISTINCT o.order_id))
* 100 as complaint_percentage
FROM users u
JOIN orders o USING(user_id)
LEFT JOIN complaints c USING(order_id)
GROUP BY u.user_id, u.user_name
HAVING COUNT(DISTINCT o.order_id) > 5
    AND (COUNT(DISTINCT c.complaint_id)::FLOAT / COUNT(DISTINCT
o.order_id)) > 0.2;
```

Performance: Planning Time: 1.410 ms Execution Time: 12.892 ms

Optimized query:

Method: Using WITH instead of JOIN directly to completely eliminate the need to COUNT(DISTINCT).

```
WITH customer_orders AS (
    SELECT u.user_id, u.user_name, COUNT(o.order_id) as total_orders
    FROM users u
    JOIN orders o USING(user_id)
    GROUP BY u.user_id
),
customer_complaints AS (
    SELECT o.user_id, COUNT(DISTINCT c.complaint_id) as complaint_orders
    FROM orders o
    JOIN complaints c USING(order_id)
    GROUP BY o.user_id
)
SELECT co.user_id, co.user_name, co.total_orders, cc.complaint_orders,
```

```

        (cc.complaint_orders::FLOAT / co.total_orders) * 100 as
complaint_percentage
FROM customer_orders co
LEFT JOIN customer_complaints cc ON co.user_id = cc.user_id
WHERE co.total_orders > 5
      AND (cc.complaint_orders::FLOAT / co.total_orders) > 0.2;

```

Performance:

Planning Time: 1.464 ms

Execution Time: 9.179 ms

Nguyễn Hoàng Trung Kiên - 20235960

1. Shipping overdue: shipped for >14 days but not received.

Initial query

```

WITH last_shipped AS (
    SELECT DISTINCT ON (ss.order_id)
        ss.order_id,
        ss.update_time AS shipped_time
    FROM shipping_statuses ss
    WHERE ss.status = 'shipped'
    ORDER BY ss.order_id, ss.update_time DESC
)
SELECT
    o.order_id,
    o.create_date,
    s.store_name,
    u.user_name AS customer_username,
    'shipped' AS last_shipping_status,
    ls.shipped_time AS last_shipping_time
FROM last_shipped ls
JOIN orders o ON o.order_id = ls.order_id
JOIN users u ON u.user_id = o.user_id
JOIN stores s ON s.store_id = o.store_id
WHERE ls.shipped_time < CURRENT_TIMESTAMP - INTERVAL '14 days'
      AND NOT EXISTS (
        SELECT 1
        FROM shipping_statuses ss2
        WHERE ss2.order_id = ls.order_id
              AND ss2.status = 'received'
              AND ss2.update_time > ls.shipped_time
      )
ORDER BY ls.shipped_time ASC;

```

Performance:

Planning Time: 0.325 ms

Execution Time: 1.733 ms

Optimized query

Method: Filter early on status='shipped' inside the CTE to leverage a partial index, and add an anti-join (NOT EXISTS) to exclude orders that already have a later received status..

```
CREATE INDEX IF NOT EXISTS idx_ship_shipped_latest
ON shipping_statuses (order_id, update_time DESC)
WHERE status = 'shipped';
```

```
CREATE INDEX IF NOT EXISTS idx_ship_received_latest
ON shipping_statuses (order_id, update_time DESC)
WHERE status = 'received';
```

Performance:

Planning Time:0.328 ms

Execution Time: 1.479 ms

2.Top 10 stores by revenue

```
SELECT
  s.store_id,
  s.store_name,
  SUM(oi.item_quantity * oi.price_at_purchase) AS total_revenue
FROM stores s
JOIN orders o ON o.store_id = s.store_id
JOIN order_items oi ON oi.order_id = o.order_id
WHERE o.order_status IN ('Approved by vendor', 'Complaint resolved')
GROUP BY s.store_id, s.store_name
ORDER BY total_revenue DESC
LIMIT 10
```

Performance:

Planning Time: 0.375 ms

Execution Time: 2.543 ms

Optimized query

A partial index was used to index only orders with relevant statuses, reducing scanned rows and improving execution performance.

```
CREATE INDEX idx_orders_revenue_partial
ON orders (store_id, order_id)
WHERE order_status IN ('Approved by vendor', 'Complaint
resolved');
```

Performance:

Planning Time: 0.399 ms

Execution Time: 2.199 ms

3. Top 10 best-selling products (units sold)

```
SELECT
  p.product_id,
  p.product_name,
  SUM(oi.item_quantity) AS units_sold
FROM products p
JOIN product_variants pv ON pv.product_id = p.product_id
JOIN order_items oi ON oi.variant_id = pv.variant_id
JOIN orders o ON o.order_id = oi.order_id
WHERE o.order_status IN ('Approved by vendor', 'Complaint resolved')
GROUP BY p.product_id, p.product_name
ORDER BY units_sold DESC
LIMIT 10;
```

Performance: Planning Time: 0.531 ms Execution Time: 2.555 ms

Optimized query

Applied partial index on orders to optimize frequent status filtering.

```
CREATE INDEX idx_orders_valid_status
ON orders(order_id)
WHERE order_status IN ('Approved by vendor', 'Complaint resolved');
```

Performance: Planning Time: 0.481 ms Execution Time: 2.143 ms

4. Stores with highest average product rating (minimum 5 reviews)

```
SELECT
  s.store_id,
  s.store_name,
  AVG(r.rating)::numeric(3,2) AS avg_store_rating,
  COUNT(r.review_id) AS total_reviews
FROM stores s
JOIN products p ON p.store_id = s.store_id
JOIN reviews r ON r.product_id = p.product_id
GROUP BY s.store_id, s.store_name
HAVING COUNT(r.review_id) >= 5
```

```
ORDER BY avg_store_rating DESC;
```

Performance: Planning Time: 0.318 ms Execution Time: 2.123 ms

Optimized query

Create B-tree index on products(store_id, product_id) to speed up join stores → products, reducing scanned rows and improving aggregation performance.

```
CREATE INDEX IF NOT EXISTS idx_products_store_id
ON products(store_id, product_id);
```

```
CREATE INDEX IF NOT EXISTS idx_reviews_product_id
ON reviews(product_id);
```

Performance: Planning Time: 1.690 ms Execution Time: 0.274 ms

5. Summarizes the admin queue by counting unresolved reports per report category.

```
SELECT
    rc.report_category_id,
    rc.name AS category_name,
    COUNT(r.report_id) AS unresolved_reports
FROM report_categories rc
LEFT JOIN reports r
    ON r.report_category_id = rc.report_category_id
    AND r.resolved_status = 'Unresolved'
GROUP BY rc.report_category_id, rc.name
ORDER BY unresolved_reports DESC;
```

Performance: Planning Time: 0.170 ms Execution Time: 0.101 ms

Optimized query

Created a partial index on reports(report_category_id) where resolved_status = 'Unresolved' to reduce scanned rows and speed up aggregation of unresolved reports per category.

```
CREATE INDEX IF NOT EXISTS idx_reports_unresolved_by_category
ON reports(report_category_id)
WHERE resolved_status = 'Unresolved';
```

Performance: Planning Time: 0.191 ms Execution Time: 0.089 ms

6. Selects users who registered more than one year ago and whose user_id does not appear in the orders table, indicating that they have never placed an order.

```
SELECT u.user_id, u.user_name
FROM users u
WHERE u.user_id NOT IN (
    SELECT o.user_id
    FROM orders o
)
AND u.registration_date < NOW() - INTERVAL '1 year';
```

Performance:

Planning Time: 0.122 ms

Execution Time: 0.575 ms

Optimized query

Replaced NOT IN with NOT EXISTS, allowing the database to perform a more efficient anti-join and significantly improve query performance.

```
SELECT u.user_id, u.user_name
FROM users u
WHERE NOT EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.user_id = u.user_id
)
AND u.registration_date < NOW() - INTERVAL '1 year';
```

Performance:

Planning Time: 0.195 ms

Execution Time: 0.439 ms

7. Display store orders with customer information and total order values..

Initial query:

```
SELECT
    o.order_id,
    o.create_date,
    o.order_status,
    u.user_id AS customer_id,
    u.user_name AS customer_username,
    u.email AS customer_email,
    u.personal_phone AS customer_phone,
    SUM(oi.item_quantity * oi.price_at_purchase) AS order_total
FROM orders o
JOIN users u ON u.user_id = o.user_id
JOIN order_items oi ON oi.order_id = o.order_id
WHERE o.store_id = :store_id
```

```

    AND (:status_filter IS NULL OR o.order_status = :status_filter)
GROUP BY
    o.order_id, o.create_date, o.order_status,
    u.user_id, u.user_name, u.email, u.personal_phone
ORDER BY o.create_date DESC;

```

8. View customer order history with shipping status.

```

WITH latest_shipping AS (
    SELECT DISTINCT ON (ss.order_id)
        ss.order_id,
        ss.status AS latest_status,
        ss.update_time AS latest_time
    FROM shipping_statuses ss
    ORDER BY ss.order_id, ss.update_time DESC
)
SELECT
    o.order_id,
    o.create_date,
    o.order_status,
    o.payment_method,
    s.store_name,
    COALESCE(ls.latest_status, 'Not shipped') AS shipping_status,
    COALESCE(ls.latest_time, o.create_date) AS shipping_updated_at,
    SUM(oi.item_quantity * oi.price_at_purchase) AS subtotal
FROM orders o
JOIN stores s ON s.store_id = o.store_id
JOIN order_items oi ON oi.order_id = o.order_id
LEFT JOIN latest_shipping ls ON ls.order_id = o.order_id
WHERE o.user_id = :customer_id
GROUP BY
    o.order_id, o.create_date, o.order_status, o.payment_method,
    s.store_name, ls.latest_status, ls.latest_time
ORDER BY o.create_date DESC;

```

9. Retrieve orders that have been approved or unresolved for more than three days and have not been shipped yet.

```

SELECT
    o.order_id, o.store_id, o.user_id, o.create_date
FROM orders o
WHERE o.order_status IN ('Approved by vendor', 'Unresolved')
    AND o.create_date < NOW() - INTERVAL '3 days'

```

```

AND NOT EXISTS (
    SELECT 1
    FROM shipping_statuses ss
    WHERE ss.order_id = o.order_id
          AND ss.status = 'shipped'
)
ORDER BY o.create_date ASC;

```

10. Identify active products with high stock levels and low sales performance.

```

WITH product_stock AS (
    SELECT pv.product_id, SUM(pv.stock) AS total_stock
    FROM product_variants pv
    GROUP BY pv.product_id
),
product_sales AS (
    SELECT pv.product_id, COALESCE(SUM(oi.item_quantity), 0) AS units_sold
    FROM product_variants pv
    LEFT JOIN order_items oi ON oi.variant_id = pv.variant_id
    LEFT JOIN orders o ON o.order_id = oi.order_id
                      AND o.order_status IN ('Approved by vendor', 'Complaint
resolved')
    GROUP BY pv.product_id
)
SELECT
    p.product_id,
    p.product_name,
    ps.total_stock,
    sa.units_sold
FROM products p
JOIN product_stock ps ON ps.product_id = p.product_id
JOIN product_sales sa ON sa.product_id = p.product_id
WHERE p.status = 'active'
      AND ps.total_stock >= :stock_threshold -- e.g. 200
      AND sa.units_sold <= :sales_threshold  -- e.g. 5
ORDER BY ps.total_stock DESC, sa.units_sold ASC;

```

Ngô Đức Huy - 20235946

1. The review response rate for each store

Initial code:

```

SELECT
    s.store_id,
    s.store_name,
    COUNT(DISTINCT r.review_id) AS total_reviews,
    COUNT(rep.reply_id) AS total_replies,
    ROUND(
        (COUNT(rep.reply_id)::NUMERIC / NULLIF(COUNT(DISTINCT
r.review_id), 0)) * 100, 2) AS reply_percentage
FROM stores s
JOIN products p ON s.store_id = p.store_id
JOIN reviews r ON p.product_id = r.product_id
LEFT JOIN replies rep ON r.review_id = rep.review_id
GROUP BY s.store_id, s.store_name
HAVING COUNT(DISTINCT r.review_id) > 0
ORDER BY reply_percentage ASC;

```

Performance: Planning Time: 1.161 ms Execution Time: 13.111 ms

Optimized code:

Method: Using CTE instead of creating a massive table, avoiding using Distinct, and using INDEX

```

WITH ReplyAgg AS (
    SELECT review_id, COUNT(*) AS reply_count
    FROM replies
    GROUP BY review_id
),
StoreStats AS (
    SELECT p.store_id, COUNT(r.review_id) AS total_reviews,
           COALESCE(SUM(ra.reply_count), 0) AS total_replies
    FROM products p
    JOIN reviews r ON p.product_id = r.product_id
    LEFT JOIN ReplyAgg ra ON r.review_id = ra.review_id
    GROUP BY p.store_id
)
SELECT s.store_id, s.store_name,
       COALESCE(ss.total_reviews, 0) AS total_reviews,
       COALESCE(ss.total_replies, 0) AS total_replies,
       COALESCE(
           ROUND((ss.total_replies::NUMERIC / NULLIF(ss.total_reviews, 0)) * 100,
2),
           0.00
       ) AS reply_percentage
FROM stores s
JOIN StoreStats ss ON s.store_id = ss.store_id
ORDER BY reply_percentage ASC;
CREATE INDEX IF NOT EXISTS idx_replies_review_id

```

```

ON replies(review_id);
CREATE INDEX IF NOT EXISTS idx_reviews_product_store_opt
ON reviews(product_id) INCLUDE (review_id);
CREATE INDEX IF NOT EXISTS idx_products_store_id
ON products(store_id, product_id);

```

Performance: Planning Time: 1.014 ms Execution Time: 7.770 ms

2. Top 10 best-selling products

Initial code:

```

SELECT
    p.product_id,
    p.product_name,
    SUM(oi.item_quantity) AS total_quantity_sold
FROM products p
JOIN product_variants pv ON p.product_id = pv.product_id
JOIN order_items oi ON pv.variant_id = oi.variant_id
JOIN orders o ON oi.order_id = o.order_id
WHERE o.order_status IN ('Approved by vendor', 'Complaint resolved')
GROUP BY p.product_id, p.product_name
ORDER BY total_quantity_sold DESC
LIMIT 10;

```

Performance: Planning Time: 1.115 ms Execution Time: 7.934 ms

Optimized code:

Method: Pre-computation with MATERIALIZED VIEW and using Indexing

```

CREATE MATERIALIZED VIEW mv_top_selling_products AS
WITH AggregatedProduct AS (
    SELECT
        pv.product_id,
        SUM(oi.item_quantity) AS total_quantity_sold
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    JOIN product_variants pv ON oi.variant_id = pv.variant_id
    WHERE o.order_status IN ('Approved by vendor', 'Complaint resolved')
    GROUP BY pv.product_id
)
SELECT
    p.product_id,
    p.product_name,
    COALESCE(ap.total_quantity_sold, 0) AS total_quantity_sold
FROM products p
JOIN AggregatedProduct ap ON p.product_id = ap.product_id;

```

```

SELECT
    product_id,
    product_name,
    total_quantity_sold
FROM mv_top_selling_products
ORDER BY total_quantity_sold DESC
LIMIT 10;

CREATE INDEX idx_mv_products_total_sold_desc ON mv_top_selling_products
(total_quantity_sold DESC);
CREATE INDEX IF NOT EXISTS idx_orders_status_composite
ON orders (order_status) INCLUDE (order_id);
CREATE INDEX IF NOT EXISTS idx_order_items_order_id
ON order_items(order_id);
CREATE INDEX IF NOT EXISTS idx_order_items_variant_id
ON order_items(variant_id);
CREATE INDEX IF NOT EXISTS idx_product_variants_product_id
ON product_variants(product_id);

```

Performance: Planning Time: 0.207 ms Execution Time: 0.120 ms

3. Average order value per store

Initial code:

```

SELECT
    s.store_name,
    AVG(sub.order_total) AS avg_order_value
FROM (
    SELECT
        o.store_id,
        o.order_id,
        SUM(oi.item_quantity * oi.price_at_purchase) AS order_total
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    WHERE o.order_status IN ('Approved by vendor', 'Complaint resolved')
    GROUP BY o.store_id, o.order_id
) sub
JOIN stores s ON sub.store_id = s.store_id
GROUP BY s.store_id, s.store_name
ORDER BY avg_order_value DESC;

```

Performance: Planning Time: 0.776 ms Execution Time: 7.992 ms

Optimized code:

Method: Create MATERIALIZED VIEW (The average order value (AOV) for each shop is pre-calculate) and index.

```

CREATE MATERIALIZED VIEW mv_store_avg_order_value AS
SELECT
    o.store_id,
    SUM(oi.item_quantity * oi.price_at_purchase) / NULLIF(COUNT(DISTINCT
o.order_id), 0) AS avg_order_value
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
WHERE o.order_status IN ('Approved by vendor', 'Complaint resolved')
GROUP BY o.store_id;

SELECT
    s.store_name,
    mv.avg_order_value
FROM stores s
JOIN mv_store_avg_order_value mv ON s.store_id = mv.store_id
ORDER BY mv.avg_order_value DESC;

CREATE UNIQUE INDEX idx_mv_store_id ON mv_store_avg_order_value(store_id);
CREATE INDEX idx_mv_avg_value ON mv_store_avg_order_value(avg_order_value DESC);

```

Performance: Planning Time: 0.479 ms Execution Time: 0.438 ms

4. Orders that have been pending for more than 3 days

Initial code:

```

SELECT
    o.order_id,
    o.create_date,
    u.user_name AS customer_name,
    s.store_name
FROM orders o
JOIN users u ON o.user_id = u.user_id
JOIN stores s ON o.store_id = s.store_id
WHERE o.order_status = 'Unresolved'
AND o.create_date < CURRENT_TIMESTAMP - INTERVAL '3 days'
ORDER BY o.create_date ASC;

```

Performance: Planning Time: 0.660 ms Execution Time: 2.257 ms

Optimized code:

Method: Create INDEX to search faster

```

CREATE INDEX idx_orders_unresolved_date ON orders(create_date) WHERE
order_status = 'Unresolved';

```

Performance: Planning Time: 0.838 ms Execution Time: 1.642 ms

5. Track revenue fluctuations over time to assess the growth of the e-commerce platform.

Initial code:

```
SELECT
    TO_CHAR(o.create_date, 'YYYY-MM') AS month_year,
    SUM(oi.item_quantity * oi.price_at_purchase) AS revenue
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
WHERE o.order_status IN ('Approved by vendor', 'Complaint resolved')
GROUP BY month_year
ORDER BY month_year DESC;
```

6. List of products currently on promotion with valid discounts.

Initial code:

```
SELECT
    p.product_id,
    p.product_name,
    prom.discount_type,
    prom.start_period,
    prom.end_period
FROM products p
JOIN promotion_products pp ON p.product_id = pp.product_id
JOIN promotions prom ON pp.promotion_id = prom.promotion_id
WHERE CURRENT_TIMESTAMP BETWEEN prom.start_period AND prom.end_period;
```

7. Identify items that are almost out of stock

Initial code:

```
SELECT
    s.store_name,
    p.product_name,
    pv.variant_name,
    pv.stock
FROM product_variants pv
JOIN products p ON pv.product_id = p.product_id
JOIN stores s ON p.store_id = s.store_id
WHERE pv.stock < 10
ORDER BY pv.stock ASC;
```

8. Complaint rate per total order for each shop (Only shops with more than 10 orders are included for the data to be meaningful).

Initial code:

```

SELECT
    s.store_name,
    COUNT(DISTINCT o.order_id) AS total_orders,
    COUNT(DISTINCT c.complaint_id) AS total_complaints,
    (COUNT(DISTINCT c.complaint_id)::FLOAT / COUNT(DISTINCT o.order_id)) * 100
AS complaint_rate_percent
FROM stores s
JOIN orders o ON s.store_id = o.store_id
LEFT JOIN complaints c ON o.order_id = c.order_id
GROUP BY s.store_id, s.store_name
HAVING COUNT(DISTINCT o.order_id) > 10
ORDER BY complaint_rate_percent DESC;

```

9. Trigger to automatically register price_at_purchase when an order_item is placed. This need to account for promotion aswell.

Initial code:

```

CREATE OR REPLACE FUNCTION set_order_item_price()
RETURNS TRIGGER AS $$
DECLARE
    rec RECORD;
    v_price_after_percent DECIMAL(10, 2);
BEGIN
    SELECT
        pv.price AS base_price,
        MIN(fpp.price) AS best_fixed_price,
        MAX(pdp.discount_percent) AS best_percent
    INTO rec
    FROM product_variants pv
    LEFT JOIN promotion_products pp ON pv.product_id = pp.product_id
    LEFT JOIN promotions p ON pp.promotion_id = p.promotion_id
    AND CURRENT_TIMESTAMP BETWEEN p.start_period AND p.end_period
    LEFT JOIN fixed_price_promotions fpp ON p.promotion_id = fpp.promotion_id
    LEFT JOIN percent_discount_promotions pdp ON p.promotion_id =
pdp.promotion_id
    WHERE pv.variant_id = NEW.variant_id
    GROUP BY pv.price;
    IF rec.best_percent IS NOT NULL THEN
        v_price_after_percent := rec.base_price * (1.0 - (rec.best_percent /
100.0));
    END IF;
    NEW.price_at_purchase := LEAST(
        rec.base_price,
        rec.best_fixed_price,
        v_price_after_percent
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE INDEX IF NOT EXISTS idx_promo_products_prod_id
ON promotion_products(product_id, promotion_id);
CREATE INDEX IF NOT EXISTS idx_promotions_period
ON promotions(start_period, end_period) INCLUDE (promotion_id);
CREATE INDEX IF NOT EXISTS idx_fixed_promo_id ON
fixed_price_promotions(promotion_id);
CREATE INDEX IF NOT EXISTS idx_percent_promo_id ON
percent_discount_promotions(promotion_id);
CREATE TRIGGER trg_set_order_item_price
BEFORE INSERT ON order_items
FOR EACH ROW
EXECUTE FUNCTION set_order_item_price();

```

10. Trigger to check for ‘Customers can only leave reviews if they have purchased the product.’ RAISE EXCEPTION when an this rule is broken

Initial code:

```

CREATE OR REPLACE FUNCTION check_purchase_before_review()
RETURNS TRIGGER AS $$
BEGIN
    IF NOT EXISTS (
        SELECT 1
        FROM orders o
        JOIN order_items oi ON o.order_id = oi.order_id
        WHERE o.user_id = NEW.user_id
        AND o.order_status IN ('Delivered', 'Completed')
        AND oi.variant_id IN (
            SELECT variant_id
            FROM product_variants
            WHERE product_id = NEW.product_id
        )
    ) THEN
        RAISE EXCEPTION 'You can only review products you have purchased and
received.';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER trg_check_purchase_before_review
BEFORE INSERT ON reviews
FOR EACH ROW
EXECUTE FUNCTION check_purchase_before_review();

```

```

CREATE INDEX IF NOT EXISTS idx_orders_user_status ON orders(user_id,
order_status);
CREATE INDEX IF NOT EXISTS idx_order_items_order_variant ON

```

```
order_items(order_id, variant_id);
CREATE INDEX IF NOT EXISTS idx_product_variants_product_id ON
product_variants(product_id);
```

5. Result and Conclusion

5.1 Difficulties during Project Execution

A fully realistic e-commerce platform requires integration with external payment gateways and banking systems. Implementing a secure, live payment processing layer was deemed too complex and out of scope for a database design course project. Furthermore, Real-world shipping requires dynamic calculations and real-time API calls to generate shipping labels and tracking numbers. Attempting to model would require excessive external API integration that distracts from the core database design goals

The Solution:

- Instead of integrating a live gateway, we designed the system to record the *Payment Method* and the *Order Status*.
- We shifted the confirmation logic to the Vendor. For online transactions, the system assumes the payment is processed externally, allowing the Vendor to mark an order as "Approved" once they verify funds, effectively removing the need for the database to handle banking protocols.
- We designed the `shipping_statuses` tables to support the *tracking* of a package without automating the *logistics* of it.

5.2 Evaluation

Pros:

- Successfully simulated some aspects of an ecommerce application such as: Placing orders, promotions, user experience and user management

Cons:

- Lacking method to calculate shipping fees in the system
- Current shipping tracking system is flawed and cant accurately track shipper's action

6. Group work distribution

Student name	Class	Email	Work done
--------------	-------	-------	-----------

Hà Hiếu Minh 20215223 (leader)	ICT 01-K66	minh.hh 215223 @sis.hust.edu.vn	<ul style="list-style-type: none"> • Topic proposal • ERD and table design from ERD • Table description and field specification • 10 queries • Slide + report
Ngô Đức Huy 20235946	ICT 02-K68	huy.nd23 5946@sis.hust.edu.vn	<ul style="list-style-type: none"> • Topic proposal • Table description and field specification • Built in trigger and functions • 10 queries • Slide
Phạm Thu Quỳnh 202417189	ICT 01-K69	quynh.pt 2417189 @sis.hust.edu.vn	<ul style="list-style-type: none"> • Topic proposal • Table description and field specification • Built in trigger and functions • 10 queries • Slide + report
Nguyễn Hoàng Trung Kiên 20235960	ICT 02-K68	kien.nht2 35960@sis.hust.edu.vn	<ul style="list-style-type: none"> • Topic proposal • Table description and field specification • Built in trigger and functions • 10 queries • Report