GROUP 13 - Optimizing Online Sports Retail Revenue

Panshul Jindal (AI23BTECH11018)
Yasaswi(AI23BTECH11005)
Shreeja(AI23BTECH11026)

Contents

1 Report

1.1 Weekly Division

- **Week 1-2 (20 Feb - 7 March):** We worked on schema understanding, business requirements, and identified functional dependencies. We checked our schema for normalization.
- **Week 3 (8 March - 16 March):** We focused on learning web development, though the progress was limited.
- **Week 4 (17 March - 23 March):** Due to exams, we were unable to meet and make progress.
- **Week 5 (24 March - 28 March):** Completed the implementation part using React for the frontend and Express for the backend.

1.2 Work Division

- **Yasaswi:** Schema planning, normalization checking, gathering data, creating procedures, triggers, and queries.
- **Shreeja:** Schema planning, normalization checking, gathering data, creating procedures, triggers, and queries.
- **Panshul:** Schema planning, normalization checking, web development.

1.3 Link for ER Diagram

ER Diagram Link

1.4 Assumptions and Design Choices

- Every user has a unique email id and each user has only one role.
- Each user can review a product only once.
- `last_updated` in inventory changes every time we update it.
- Orders status is just `completed` or `returned` or `replaced`.

1.5 Functional Dependencies

Users

`user_id`	→ name, email, password_hash, role
email	→ user_id (since email is unique)

Suppliers

`supplier_id`	→ name, contact_number, email, category_id, brand_id
email	→ supplier_id (if supplier emails are unique)
contact_number	→ supplier_id (if contact numbers are unique)

Products

`product_id`	→ name, brand_id, category_id, cost_price, selling_price, mrp, revenue_per_unit, average_r
(brand_id, name)	→ product_id (if each product name is unique within a brand)

Categories
`category_id` → name, parent_category_id, gender, description
name → category_id (if category names are unique)

Brand
`brand_id` → name
name → brand_id (if brand names are unique)

Customer_Reviews
(user_id, product_id) → rating, review_text, review_date
product_id → average_rating (if calculated as an aggregate of all ratings)

Inventory
`product_id` → current_stock, low_stock_threshold, last_updated

Vendor_Orders
(id, product_id, vendor_id) → quantity, cost_price, amount, date
(quantity, cost_price) → amount (Derived attribute)

Orders
`order_id` → user_id, order_date, total_amount, status

Order_Items
(order_id, product_id) → quantity, selling_price

Customer_Support
`support_id` → user_id, issue_category, issue_description, status, created_at, resolved_at

Notifications
`notification_id` → user_id, message, date_sent

Price_Change_Strategy
`strategy_id` → product_id, old_price, new_price, change_date, reason
(product_id, change_date) → old_price, new_price, reason

Returns
`return_id` → order_id, user_id, status, reason, return_date
order_id → user_id (since each order belongs to a single user)

Stock_Strategy_Suggestions
`suggestion_id` → product_id, suggested_quantity, reason, suggested_date

Wishlist
(user_id, product_id) → (exists in Wishlist)

1.6 Normalisation

- Our schema satisfies **1NF** (there are no indivisible attributes).
- Our schema satisfies **3NF/BCNF** since all functional dependencies are trivial.
- Our schema satisfies **4NF** because there are no multivalued attributes.

1.7 MVC Framework

Database (Model):

- PostgreSQL is used for storing all data.
- Relations are in at least 3NF with a carefully planned ER Diagram.
- Indexing: No indexing implemented.
- Stored procedures and functions implement automatic pricing strategies and stock inventory updates.

Backend (Controller):

- Built with Node.js & Express.
- Routes define endpoints.

Frontend (View):

- Implemented using React.

1.8 Main Logic

- **Price Optimization Based on Sales**
  - Increase the price of best-selling products to maximize revenue.
  - Decrease the price of products with lower sales to boost demand.
- **Inventory Optimization Based on Sales**
  - If sales in a fixed time period (e.g., 7 days) are less than 10% of stock holdings, reduce the price to improve turnover.

1.9 Actual Schema

Users:

```
CREATE TABLE Users (
    user_id SERIAL PRIMARY KEY,
    name VARCHAR(255),
    email VARCHAR(255) UNIQUE,
    password_hash TEXT,
    role VARCHAR(50)
);
```

Suppliers:

```
CREATE TABLE Suppliers (
    supplier_id INTEGER PRIMARY KEY,
    name VARCHAR(10),
    contact_number VARCHAR(20),
    email VARCHAR(255),
    category_id INTEGER REFERENCES Categories(category_id),
    brand_id INTEGER REFERENCES Brand(brand_id)
);
```

Products:

```sql
CREATE TABLE Products (
    product_id INTEGER PRIMARY KEY,
    name VARCHAR(255),
    brand_id INTEGER REFERENCES Brand(brand_id),
    category_id INTEGER REFERENCES Categories(category_id),
    cost_price DECIMAL(10,2),
    selling_price DECIMAL(10,2),
    mrp DECIMAL(10,2),
    revenue_per_unit DECIMAL(10,2),
    average_rating DECIMAL(4,2),
    supplier_id INTEGER REFERENCES Suppliers(supplier_id)
);
```

Categories:

```sql
CREATE TABLE Categories (
    category_id INTEGER PRIMARY KEY,
    name VARCHAR(255) UNIQUE,
    parent_category_id INTEGER REFERENCES Categories(category_id),
    gender VARCHAR(20),
    description TEXT
);
```

Brand:

```sql
CREATE TABLE Brand (
    brand_id INTEGER PRIMARY KEY,
    name VARCHAR(255)
);
```

Customer_Reviews:

```sql
CREATE TABLE Customer_Reviews (
    user_id INTEGER REFERENCES Users(user_id),
    product_id INTEGER REFERENCES Products(product_id),
    rating INTEGER,
    review_text TEXT,
    review_date TIMESTAMP,
    PRIMARY KEY (user_id, product_id)
);
```

Inventory:

```sql
CREATE TABLE Inventory (
    product_id INTEGER PRIMARY KEY REFERENCES Products(product_id),
    current_stock INTEGER,
    low_stock_threshold INTEGER,
    last_updated TIMESTAMP
);
```

Vendor_Orders:

```sql
CREATE TABLE Vendor_Orders (
    id INTEGER,
    product_id INTEGER REFERENCES Products(product_id),
    vendor_id INTEGER REFERENCES Suppliers(supplier_id),
    quantity INTEGER,
    cost_price DECIMAL(10,2),
    amount DECIMAL(10,2),
    date TIMESTAMP,
    PRIMARY KEY (id, product_id, vendor_id)
);
```

Orders:

```sql
CREATE TABLE Orders (
    order_id INTEGER PRIMARY KEY,
    user_id INTEGER REFERENCES Users(user_id),
    order_date TIMESTAMP,
    total_amount DECIMAL(10,2),
    status VARCHAR(50)
);
```

Order_Items:

```sql
CREATE TABLE Order_Items (
    order_id INTEGER REFERENCES Orders(order_id),
    product_id INTEGER REFERENCES Products(product_id),
    quantity INTEGER,
    selling_price DECIMAL(10,2),
    PRIMARY KEY (order_id, product_id)
);
```

Customer_Support:

```sql
CREATE TABLE Customer_Support (
    support_id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES Users(user_id),
    issue_category VARCHAR(50),
    issue_description TEXT,
    status VARCHAR(20),
    created_at TIMESTAMP,
    resolved_at TIMESTAMP
);
```

Price_Change_Strategy: (First Definition)

```sql
CREATE TABLE Price_Change_Strategy (
    strategy_id SERIAL PRIMARY KEY,
    product_id INTEGER REFERENCES Products(product_id),
    old_price DECIMAL(10,2),
    new_price DECIMAL(10,2),
    change_date TIMESTAMP,
    reason TEXT
);
```

Returns:

```sql
CREATE TABLE Returns (
    return_id SERIAL PRIMARY KEY,
    order_id INTEGER REFERENCES Orders(order_id),
    user_id INTEGER REFERENCES Users(user_id),
    status VARCHAR(10),
    reason TEXT,
    return_date TIMESTAMP
);
```

Stock_Strategy_Suggestions:

```sql
CREATE TABLE Stock_Strategy_Suggestions (
  suggestion_id SERIAL PRIMARY KEY,
  product_id INT NOT NULL REFERENCES Products(product_id),
  stock_change_reason TEXT,
  suggested_percentage_change INT NOT NULL,
  suggested_start_date TIMESTAMP NOT NULL,
  suggested_end_date TIMESTAMP NOT NULL CHECK(suggested_end_date >
      suggested_start_date),
  admin_decision VARCHAR(10) DEFAULT 'Pending' CHECK (admin_decision IN ('
      Accepted', 'Rejected', 'Pending')),
```

```
  decision_date TIMESTAMP
);
```

Price_Change_Strategy: (Second Definition)

```
CREATE TABLE Price_Change_Strategy (
    request_id SERIAL PRIMARY KEY,
    product_id INT REFERENCES Products(product_id) ON DELETE CASCADE,
    reason TEXT,  -- 'Best-Seller' or 'Low-Sales'
    suggested_percentage DECIMAL(5,2),  -- Suggested % change
    suggested_time_period INTERVAL,  -- Suggested time (e.g., '7 days')
    admin_approved_percentage DECIMAL(5,2),  -- Final % set by admin
    admin_approved_time_period INTERVAL,  -- Final time set by admin
    admin_action VARCHAR(20) CHECK (admin_action IN ('pending', 'approved', '
        rejected', 'reverted')),
    revert_at TIMESTAMP,  -- Time when price should revert
    original_price DECIMAL(10,2),  -- Store the original price before changing
    created_at TIMESTAMP DEFAULT NOW()
);
```

Wishlist:

```
CREATE TABLE Wishlist (
    user_id INTEGER REFERENCES Users(user_id),
    product_id INTEGER REFERENCES Products(product_id),
    PRIMARY KEY (user_id, product_id)
);
```

1.10 Functions and Stored Procedures

1. **Function: insert_order**
   Inserts a new order into the `Orders` table by generating a new order_id dynamically. It also supports applying a coupon code if provided.

2. **Function: get_new_product_id**
   Generates a new product_id dynamically by selecting the maximum existing product_id and incrementing it.

3. **Stored Procedure: Generate_Stock_Strategies**
   Scans each product (via joining `Products` and `Inventory`), calculates recent sales (last 7 days), and checks if the current stock is below a predefined low threshold. Based on the analysis, it suggests increasing the stock by 20% or decreasing it by 15% and logs a strategy suggestion in `Stock_Strategy_Suggestions` (valid for the next 7 days).

4. **Stored Procedure: Apply_Accepted_Strategies**
   Identifies an admin user, updates the `Inventory` for products with an accepted stock strategy suggestion (within the active date range and not yet applied), adjusts the current stock, marks the suggestions as "Applied", and notifies the admin.

5. **Stored Procedure: Revert_Stock_Strategies**
   Reverts previously applied stock adjustments once the suggestion's validity period expires. It updates the inventory to reverse the earlier adjustment and marks the suggestion as "Reverted", notifying the admin.

1.11 Triggers

1. **Trigger: Stock_Strategy_Notification**
   Fires after a new record is inserted into the `Stock_Strategy_Suggestions` table and calls the function `Notify_Admin_Stock_Strategy` to notify the admin.

2. **Trigger: trigger_detect_product_performance**
   Activated after inserting a new order into the `Orders` table. The associated function `detect_product_performance` calculates total sales over the past 30 days, categorizes the product as a best-seller or low-seller, and logs a corresponding suggestion in the `Price_Change_Strategy` table with a pending admin action.

3. **Trigger: trigger_apply_price_change**
   Listens for updates on the `Price_Change_Strategy` table where the admin action is updated to 'approved'. It then updates the product's selling price accordingly and sets a revert time, ensuring admin-approved price changes are immediately applied with a mechanism to revert them later.

2 Conclusion

This report details the design, normalization, and implementation aspects of our online sports retail revenue optimizer. The schema, functional dependencies, and normalization checks ensure data integrity, while our MVC-based architecture using PostgreSQL, Express, and React facilitates effective data handling and a responsive frontend. The inclusion of stored procedures, functions, and triggers automates key business processes such as pricing strategy adjustments and stock management.