Kuick Commerce

CIS 761: Term Project Report

Section 1: Introduction

1.1 Overview

This project involves designing and developing a complete Quick Commerce (Q-Commerce) system, inspired by Zepto, and built collaboratively by a team of three members. Q-Commerce focuses on rapid delivery of daily essentials—like groceries and household items—within minutes.

The system includes a responsive frontend website and a fully normalized backend database. Key features include user registration, product browsing by category, cart management, order placement, and customer feedback. The backend, built using relational modeling up to BCNF, ensures data integrity and efficiency, with real-time operations handled through SQL queries, stored functions, and triggers on the Supabase platform.

While administrative tools are not currently included, the system is designed for future expansion to support roles like inventory managers and delivery agents.

1.2 Function of the Project

The key functions of the project are:

- **User Registration and Authentication:**
  Users can sign up and log in through the website interface. Authentication is handled securely using Supabase Auth, and user details are stored in the users table.
- **Product Browsing and Categorization:**
  Products are organized into categories (e.g., snacks, beverages, essentials) and displayed through the web interface. Users can explore available items before making a purchase.
- **Cart Management:**
  Users can add, remove, and update products in their cart through the frontend. These actions are reflected in the cart and cart_items tables in real-time.
- **Order Placement and Tracking:**
  Upon checkout, the cart is converted into an order. The system generates a new order_id, stores the order details in the database (orders, hasorderitems), and updates product inventory automatically.
- **Feedback System:**
  After order completion, users can rate their experience and leave a comment. Feedback is stored in the givesfeedback table and linked to both the user and the corresponding order.

- **Frontend Website Integration:**
  A responsive website built using modern web technologies allows users to perform all actions through a user-friendly interface, with backend operations handled seamlessly via API calls to the Supabase backend.

1.3 Potential Users

1. Customers (Primary Users)

- Customers are the main users of the platform.
- They interact through the website to:
  - Register and log in
  - Browse and search for products
  - Add items to their cart
  - Place orders and make payments
  - Submit ratings and feedback on completed orders

2. System Developers

- Developers are responsible for building and maintaining both the frontend and backend components.
- They manage:
  - Database schema design (tables, relationships, constraints)
  - Implementation of stored procedures, triggers, and functions
  - Integration between the website frontend and Supabase backend
  - Deployment and debugging of the system

3. Future Roles (Planned or Extendable Users)

- The system is structured to support additional roles in future phases, such as:
  - **Delivery Agents** – to track and manage deliveries
  - **Inventory Managers** – to monitor stock levels and restock products
  - **Administrators** – to manage users, orders, and feedback across the platform

Section 2: SQL Implementation Review

2.1 Technical Description

The Quick Commerce system uses a PostgreSQL database hosted on Supabase, which provides backend-as-a-service (BaaS) features like authentication, storage, and real-time APIs. Supabase allows secure interaction with the database through RESTful and real-time APIs, removing the need for manual backend setup.

The database schema was first developed and tested locally for structure and performance, then migrated to Supabase for deployment. It models key entities such as users, products, carts, orders,

and feedback, and is fully normalized to **BCNF** to ensure data integrity and minimize redundancy. The frontend connects to the live database via Supabase's client SDK and API, enabling real-time user interactions.

Each table includes:

- A **primary key** to uniquely identifying records
- **Foreign keys** to enforce relationships and integrity constraints
- **Constraints** such as NOT NULL, UNIQUE, and CHECK for validation

To enhance functionality and maintain data accuracy, the backend uses:

- **SQL Functions** for reusable logic like processing orders or retrieving user order history
- **Triggers** for automating actions such as reducing product stock after an order is placed
- **Indexes** to optimize performance for key queries

2.2 Criteria for Design

Key criteria and assumptions include:

- **One Cart per User:** Each user is assigned a single active cart to manage product selection before placing an order.
- **Products Categorized by Type:** All products are linked to specific categories (e.g., snacks, groceries), enabling structured browsing and filtering.
- **Orders Generated from Carts:** When an order is placed, the contents of the cart are moved to the orders and hasorderitems tables, capturing order details with timestamps and total cost.
- **Feedback Linked to Orders:** Each user can submit a single feedback entry per order, capturing a rating and comment.
- **Inventory Management:** Product stock levels are automatically reduced after a successful order using triggers to prevent overselling.
- **Scalable Integration:** The design supports integration with a frontend and potential role expansion (e.g., admin, delivery agents) without schema restructuring.

2.3 E/R Implementation

The E/R model consists of the following main entities and relationships:

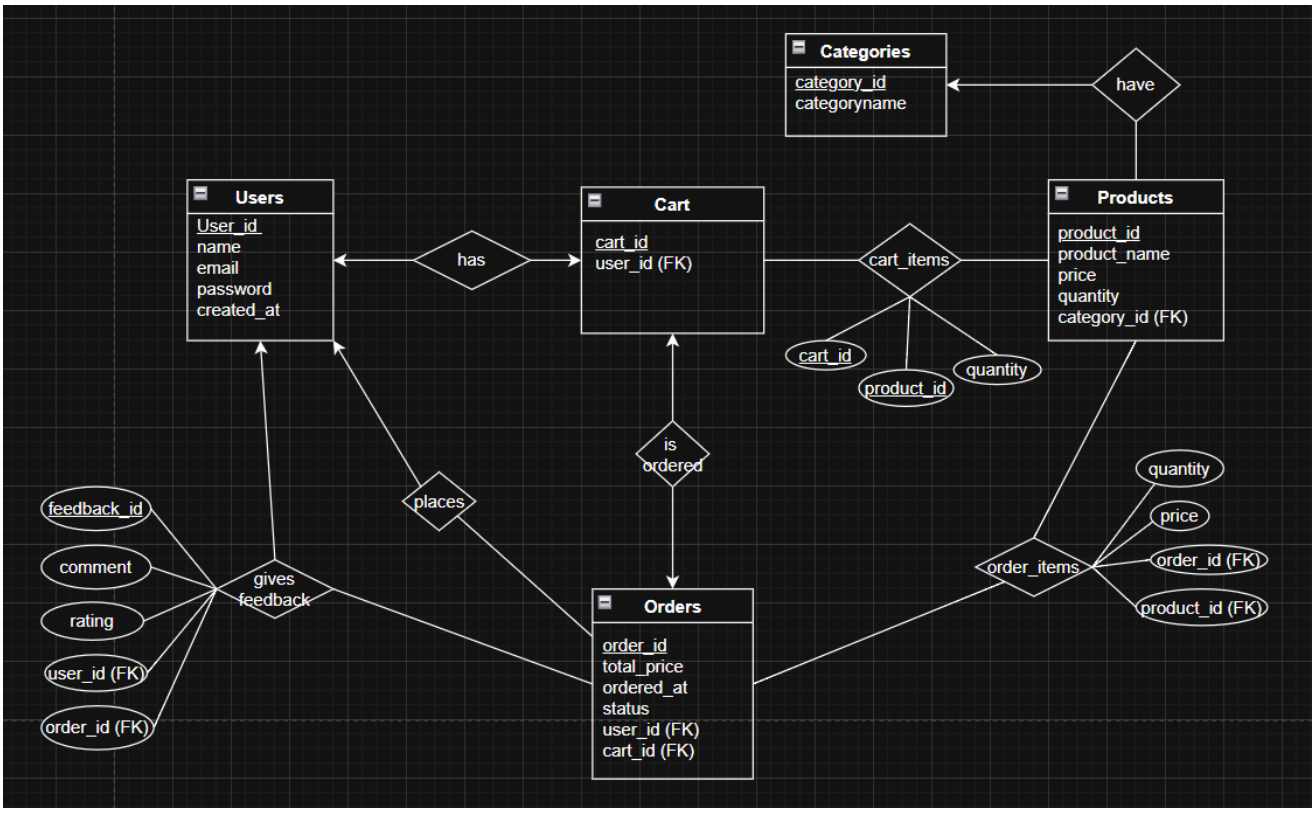
Entities

- **Users:** Stores customer information such as **user ID, name, and email.** Each user is associated with one cart and can place multiple orders and provide feedback.
- **Categories:** Represents different **product classifications** (e.g., groceries, beverages) and links to products.
- **Products:** Contains details like **product ID, name, price, quantity, and category ID.**

- **Cart:** Represents a temporary **collection of products selected by the user**. Each cart is tied to a single user.
- **Cart_Items:** A junction table that **links products to a cart**, along with quantity selected.
- **Orders:** Stores finalized transactions including the **user ID, cart ID, timestamp, total cost, and order status.**
- **HasOrderItems:** A junction table that **links each order to multiple products**, capturing the quantity and price of each item.
- **GivesFeedback:** Captures a **user's rating and comments** for a specific order after completion.

Relationships

- **Users → Cart:** One-to-one (each user has one active cart).
- **Users → Orders:** One-to-many (a user can place multiple orders).
- **Cart → Cart_Items → Products:** Many-to-many (a cart can contain multiple products, and products can be in multiple carts).
- **Orders → HasOrderItems → Products:** Many-to-many (each order contains multiple products; each product can be in many orders).
- **Users → GivesFeedback → Orders:** One-to-one per order (each feedback is linked to a specific user and order).

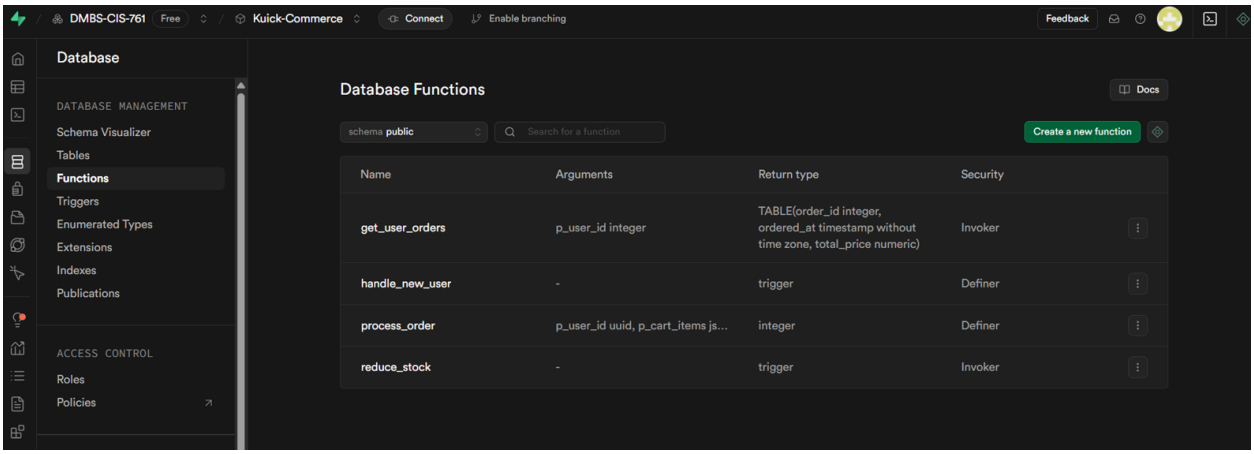


2.4 SQL Queries, Functions, and Triggers

A. SQL Functions

Custom SQL functions were created to encapsulate business logic and simplify interactions between the frontend and backend. These include:

- **get_user_orders(user_id)**
  Retrieves a complete list of orders placed by a given user, including order ID, timestamp, and total price.
- **process_order(user_id, cart_items)**
  Accepts a user's cart content and finalizes the order. It moves items from the cart to the hasorderitems table, generates a new order record, and calculates the total.
- **reduce_stock()**
  Automatically decreases product quantity in the inventory after each order is placed. It ensures that the stock remains accurate and prevents overselling.
- **handle_new_user()**
  Triggered during user sign-up. Prepares default records or actions for new users (e.g., initializing a cart).

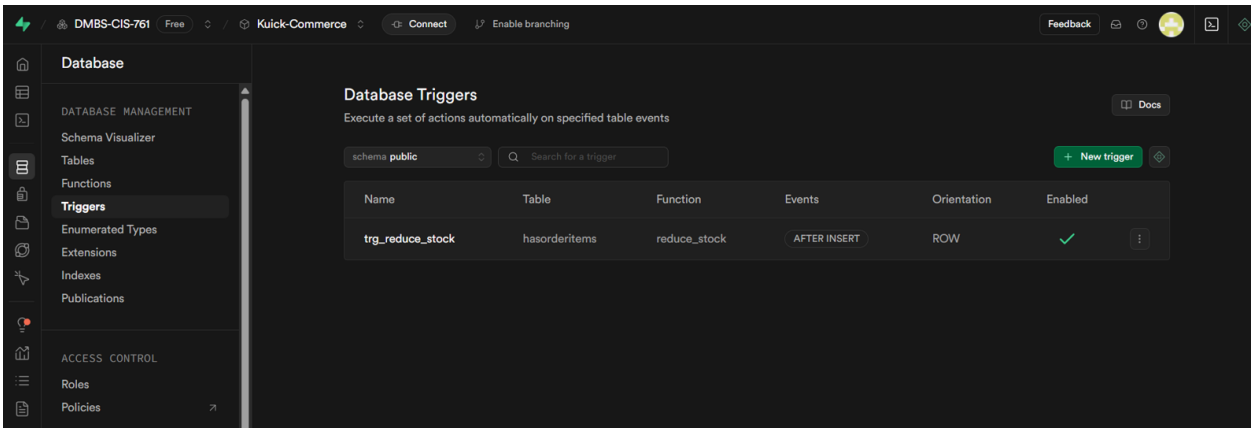


B. SQL Triggers

Triggers were implemented to automate database responses to specific events:

- **trg_reduce_stock**
  Triggered AFTER INSERT on the hasorderitems table. It calls the reduce_stock() function to deduct item quantities based on order data.

This helps maintain real-time accuracy in inventory levels and ensures data consistency without requiring manual stock management.

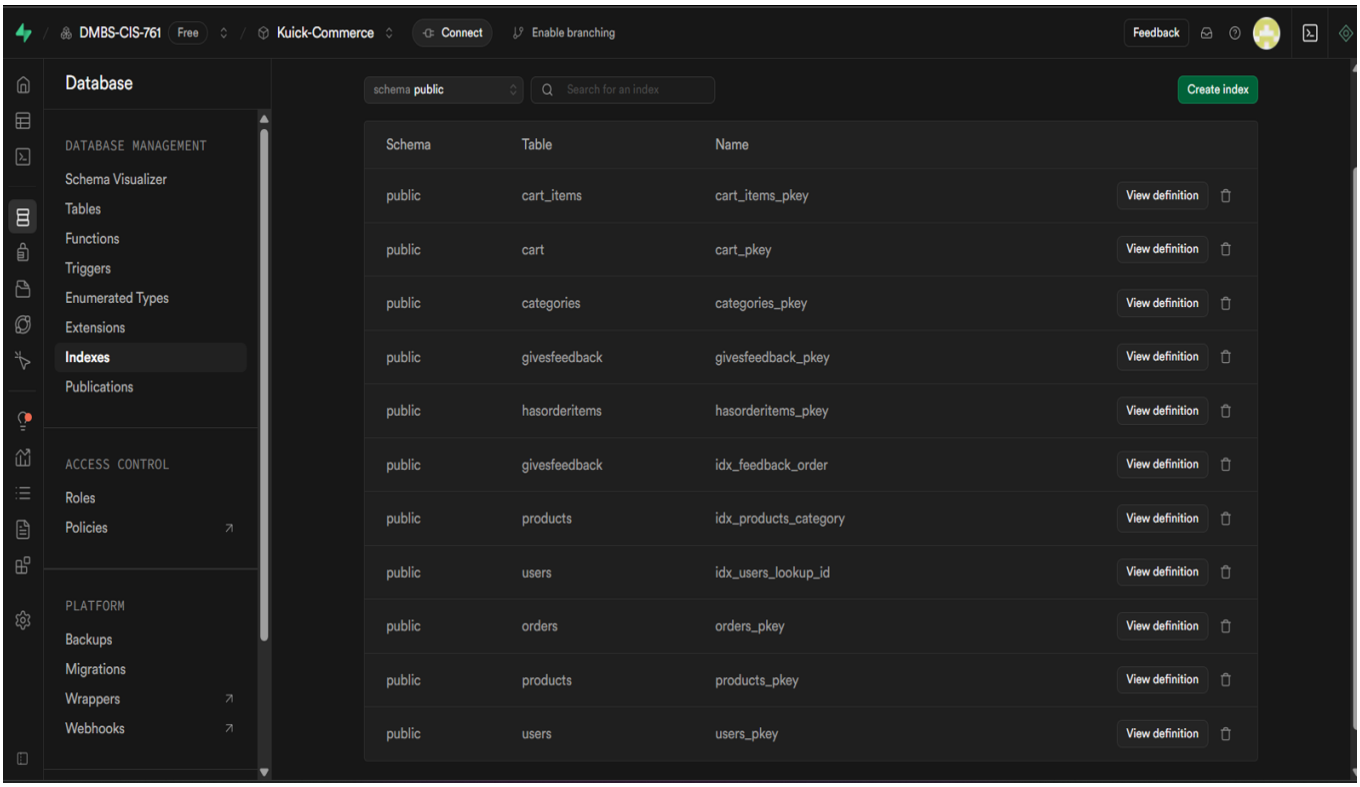


C. Indexes and Performance Optimization

To support efficient querying, especially for JOIN operations and search filters, indexes were added to:

- **Primary keys** (e.g., user_id, order_id, product_id)
- **Foreign keys** (e.g., cart_id, category_id, order_id in child tables)

While indexing has not been fully implemented in the current version, we plan to introduce it in the future to significantly reduce lookup times, particularly for operations like loading a user's cart, retrieving past orders, and filtering product listings.



D. Sample SQL Queries

Below are examples of SQL queries used to support key features of the system:

Query 1: Find the product(s) with the highest price.

```
SELECT
  product_id,
  product_name,
  price
FROM Products
WHERE price = (SELECT MAX(price) FROM Products);
```

Data Output | Messages | Graph Visualiser | Notifications

	product_id [PK] integer	product_name character varying (255)	price numeric (10,2)
1	13	Bluetooth Headphones	79.99

Total rows: 1 | Query complete 00:00:00.157

Query 2: Find products that have never been ordered.

```
SELECT p.product_id, p.product_name, p.price
FROM Products p
LEFT JOIN hasorderitems ho ON p.product_id = ho.product_id
WHERE ho.order_id IS NULL;
```

Data Output | Messages | Graph Visualiser | Notifications

	product_id [PK] integer	product_name character varying (255)	price numeric (10,2)
1	20	Bottled Water 24-Pack	4.99
2	27	Stapler - Standard	8.99
3	17	Laptop Stand - Aluminum	29.99
4	10	Cookies - Chocolate Chip	3.79
5	18	Cola Can 12-Pack	6.99
6	8	Pretzels Bag	2.99
7	28	Paper Clips - Box of 100	1.20
8	30	Scissors - Office	5.50
9	6	Potato Chips - Salted	3.29
10	29	Binder - 1 inch	3.50
11	22	Coffee Beans - 1kg Bag	14.99
12	3	Ice Cream Tub - Vanilla	5.50
13	14	Smartphone Screen Protector	15.00
14	9	Trail Mix - Nuts & Fruit	4.50

Total rows: 14 | Query complete 00:00:00.100

Query 3: Show the highest-priced product in each category

```
SELECT
  c.category_name,
  MAX(p.price) AS max_price_in_category
FROM Categories c
JOIN Products p ON c.category_id = p.category_id
GROUP BY c.category_name;
```

Data Output | Messages | Graph Visualiser | Notifications

	category_name character varying (100)	max_price_in_category numeric
1	Snacks	4.50
2	Electronics	79.99
3	Stationery	8.99
4	Frozen Food	8.99
5	Beverages	14.99

Query 4: Sales Performance by Category Report

```
SELECT
  c.category_name,
  SUM(ho.quantity) AS total_quantity_sold,
  SUM(ho.quantity * ho.price) AS total_revenue
FROM Categories c
JOIN Products p ON c.category_id = p.category_id
JOIN hasorderitems ho ON p.product_id = ho.product_id
GROUP BY c.category_name
ORDER BY total_revenue DESC;
```

Data Output | Messages | Graph Visualiser | Notifications

	category_name character varying (100)	total_quantity_sold bigint	total_revenue numeric
1	Electronics	6	216.47
2	Frozen Food	6	28.44
3	Beverages	2	18.48
4	Stationery	5	14.48
5	Snacks	1	1.99

This combination of queries, stored procedures, and triggers ensures the backend remains responsive, efficient, and fully aligned with user interactions on the frontend website.

Section 3: System Implementation

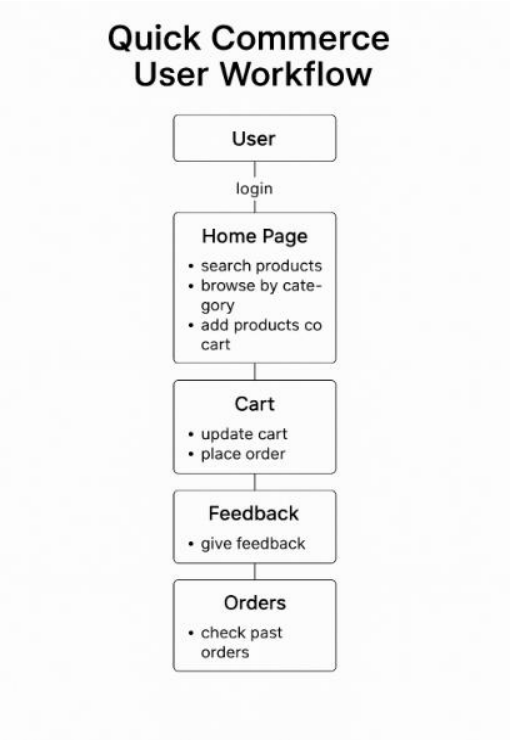
3.1 Architecture Overview

- **Frontend:** Built using **HTML, CSS, and Javascript**. It includes pages for login, home (product search/category display), cart, order history, and feedback.
- **Backend: PostgreSQL** database hosted on **Supabase**. It handles relational data storage, business logic, constraints, and automation through triggers and functions.

3.2 Data Flow and Task Execution

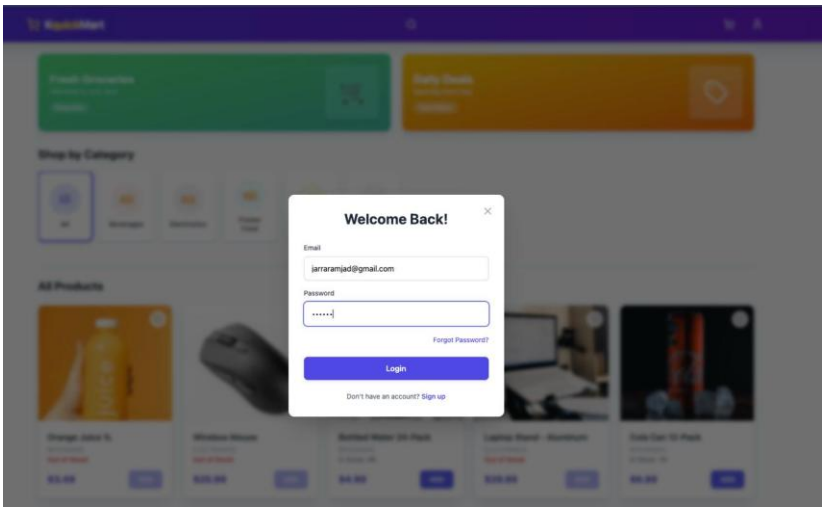
- **User Sign-Up & Authentication:**
  - Users sign up using the Supabase Auth system.
  - A trigger `handle_new_user` listens for new user insertions in `auth.users` and automatically adds corresponding entries into our `users` table.
- **Product Browsing and Search:**
  - Categories are fetched and displayed using Supabase queries.
  - Users can search products using a search bar that performs SQL `ILIKE` queries.
  - Products are retrieved from the `products` table and displayed dynamically.
- **Cart Management:**
  - Users can increment/decrement product quantities.
  - Cart updates reflect in `cart_items` table tied to the user's `cart_id`.
- **Order Placement:**
  - process_order(user_id, cart_items)` function validates input, checks stock, inserts data in `orders` and `hasorderitems`, and returns `order_id`.
- **Inventory Management:**
  - Trigger `trg_reduce_stock` is fired AFTER INSERT on `hasorderitems`, calling `reduce_stock()` to adjust product quantities.
- **Feedback Capture:**
  - Users submit comments and ratings post-order into `givesfeedback` table.
- **Order History:**
  - get_user_orders(user_id)` function retrieves past orders by joining `orders` and hasorderitems`

3.3 Work flow

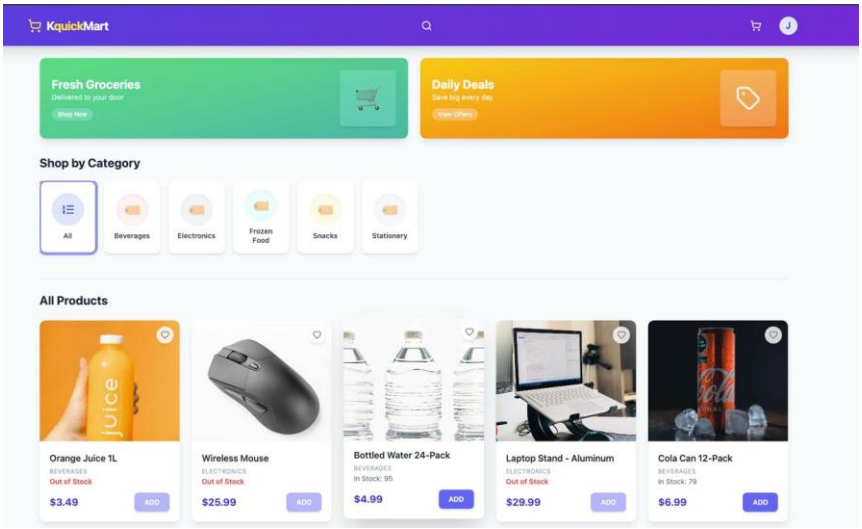


3.4 Screenshots of User Interface.

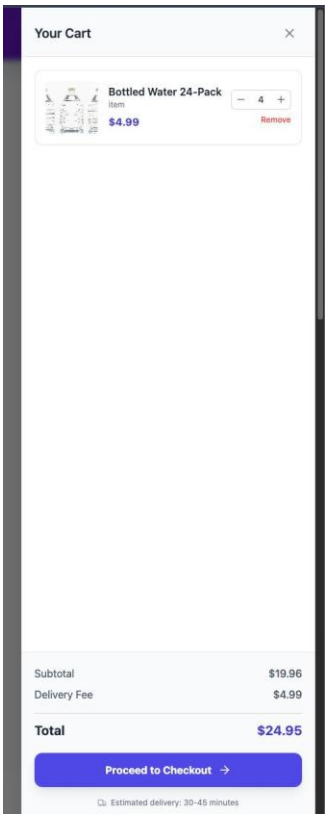
Login.



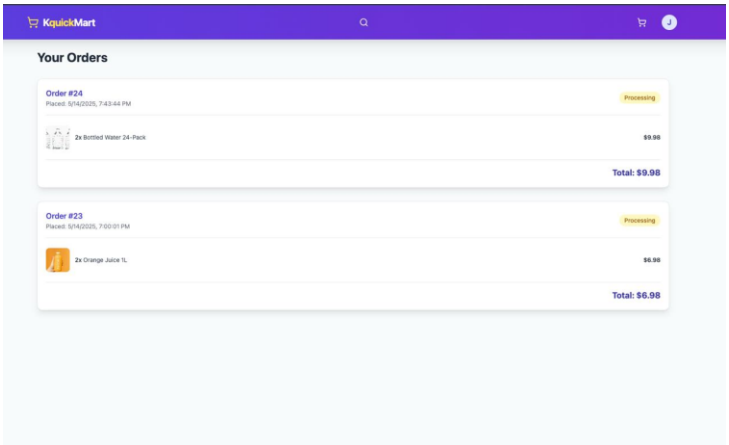
Home-Page



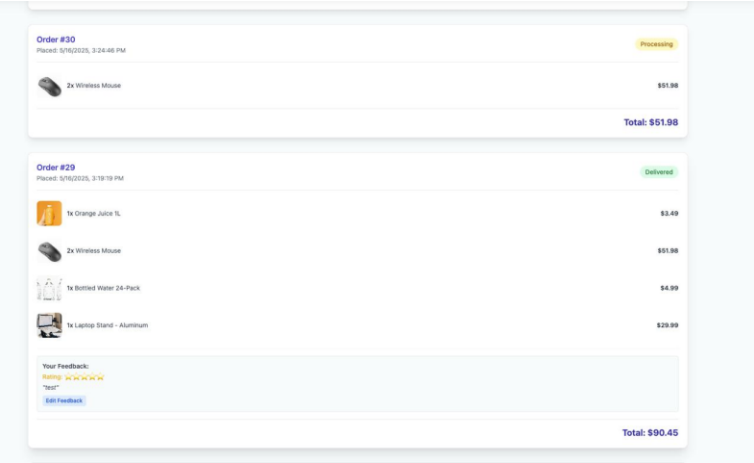
Cart



Orders



Feedback



3.5 Summary of Logic Layer Integration

The Kuick-Commerce system employs several backend components to manage logic and data flow efficiently:

- Supabase Auth handles all user authentication and authorization processes, ensuring only valid users can access the system.

- The handle_new_user trigger is invoked upon user signup and automatically inserts user metadata into the users table to keep the database synchronized with the Supabase Auth system.
- The process_order function encapsulates the entire order placement workflow. It validates user input, checks stock availability, inserts the order and related items, and ensures transactional atomicity — rolling back all changes if any step fails.
- The trg_reduce_stock trigger is fired after an item is inserted into the hasorderitems table. It calls the reduce_stock function, which automatically adjusts product quantities in the inventory, maintaining consistency.
- The get_user_orders function retrieves all previous orders made by a specific user, allowing seamless integration with the frontend order history view.

Section 4: System Features and Usage

4.1 User Interaction

The Kuick-Commerce system is designed for simplicity and user-friendliness. The user journey begins with a login page powered by Supabase authentication. After logging in, users are directed to the homepage, where they can browse products by category or use a search bar to locate specific items. Each product listing includes an option to increase or decrease quantity and add items to the cart.

Users can access the cart page to review selected items, adjust quantities, and proceed to checkout. Upon confirming an order, the system triggers backend functions that validate stock, calculate totals, and process the order securely. Once an order is placed, the user is prompted to leave feedback on their shopping experience. They can also navigate to the order history page to view past purchases.

4.2 Strengths and Limitations

Strengths:

- Seamless integration between frontend and backend using Supabase
- Real-time data updates through Supabase SDK
- Use of database functions and triggers to ensure data consistency
- Search and category filter functionalities for enhanced user experience
- Secure and scalable PostgreSQL backend hosted in the cloud
- Modular, testable backend logic using SQL functions and transaction safety

Limitations:

- Basic UI layout; lacks modern design aesthetics or animations
- Does not support multiple active carts or saved cart sessions
- No admin interface to add/update products or manage orders
- Limited error handling on the frontend
- Currently supports only single-user cart without multi-device sync

Section 5: Evaluation

To evaluate the system's performance and usability, several techniques and database optimizations were implemented.

5.1 Indexes

The system uses multiple B-tree indexes on foreign keys and lookup fields such as:

- idx_products_category` (for filtering products by category),
- idx_feedback_order`, `idx_users_lookup_id`, etc.

These indexes improve query performance, especially for join-heavy and filter-heavy queries. While their full impact isn't visible with limited test data, they will offer noticeable speed benefits as data scales.

5.2 Function-Based Processing

PostgreSQL functions like `process_order` and `get_user_orders` encapsulate logic in reusable and secure database-side procedures. This reduces frontend complexity and offloads processing to the DBMS, increasing performance.

5.3 Error Handling and Validation

Transactional integrity is preserved through validation checks and exception handling in `process_order`. For example, if a product is out of stock, the order is rolled back with a meaningful error message.

5.4 Triggers

The system leverages triggers like `handle_new_user` and `trg_reduce_stock` to ensure automatic data propagation and consistency without requiring manual intervention in the frontend logic.

Section 6: Technical Details and Justification of Technology Choices

We selected Supabase as our backend because it offers a powerful PostgreSQL database along with built-in authentication, real-time updates, and RESTful API support. Supabase made it easy to define tables, write SQL functions, and implement triggers, all through a web interface. This

allowed us to focus on database logic without setting up a separate backend server, while also ensuring data integrity and scalability.

For the frontend, we used React combined with HTML and CSS. React was chosen for its component-based architecture, which helped us manage UI elements like product cards, search filters, and the dynamic cart. HTML and CSS were used to style the interface and make it clean and responsive. Together, these technologies enabled us to build an intuitive, interactive user experience that seamlessly connects with our Supabase backend.

Section 7: Summary and Discussion

This project aimed to design and implement a simplified quick-commerce platform named Kuick-Commerce, allowing users to browse products, manage their cart, place orders, and provide feedback. The system was built using a cloud-hosted PostgreSQL database via Supabase, and a React-based frontend with HTML and CSS for a clean, user-friendly interface.

Throughout the project, we applied core database concepts such as relational schema design, normalization, stored procedures, triggers, and transactions. Features like stock validation, order processing, and feedback storage were implemented using SQL functions and Supabase API integration.

What We Learned

We gained hands-on experience with:

- Real-world database modeling
- Writing complex SQL functions and triggers
- Handling transactional consistency and data integrity
- Connecting frontend to a live database via Supabase

What We'd Change

If we were to redo the project:

- We would modularize the frontend more cleanly using a full React app with routing.
- We'd explore using Supabase Row Level Security (RLS) rules for stricter data access control.
- We'd also automate schema migrations and test insertions using SQL scripts or migration tools.

Ideas for Future Work

- Adding an admin dashboard to manage inventory and view analytics
- Integrating payment simulation or gateway

- Improving UI styling and responsiveness
- Adding real-time stock updates using Supabase's subscriptions

Section 8: Teamwork Division and Team Experience

Each team member took on these roles at different stages, which added depth to our learning and made the process engaging. We used GitHub for version control and shared resources like SQL files and frontend code. Regular discussions helped resolve technical roadblocks and align progress.

Overall, the teamwork experience was smooth and productive. The project strengthened our understanding of database-driven applications and improved our skills in working collaboratively toward a technical solution.