Online Shopping System
Team Name: Data Driven Shoppers

Mounika Pasupuleti
Engineering Science Data Science
University at Buffalo
UBITName: mpasupul
mpasupul@buffalo.edu

Sahithya Arveti Nagaraju
Engineering Science Data Science
University at Buffalo
UBITName: sarvetin
sarvetin@buffalo.edu

Sushmitha Manjunatha
Engineering Science Data Science
University at Buffalo
UBITName: smanjuna
smanjuna@buffalo.edu

***Abstract*—This project focuses on developing a complete database system for online shopping that discharges the complexity of contemporary electronic commerce. With the ever-growing need for easy to access and effortless shopping experience, online shopping has become very popular. But, whilst online marketplaces have become dependent on vast amounts of data for user and product maintenance, and transactions and 'data junkies' exist within, most companies utilize a simple database that does not fulfill the volume of data. The goal of this project, therefore, is to solve the problems of online traders through the development of a safe, flexible and equally effective database management system that ensures proper performance, maximization of customer satisfaction, and achievement of business goals.**

I. Introduction

Need of database instead of excel: A database is more suitable than Excel for managing an online shopping platform due to its scalability, ability to handle large datasets, and ensure data integrity. It supports real-time multi-user access, automates key processes like stock management and order tracking, and offers enhanced security features such as encryption. Databases also provide efficient data querying, reporting, and error handling, making them ideal for managing complex e-commerce operations compared to Excel's limitations.

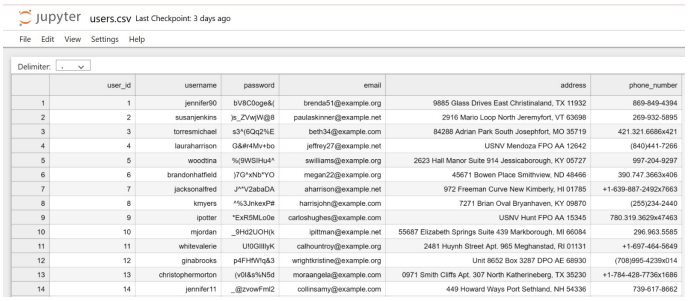
Shopping from marketplaces has transformed the way retail operates, with customers able to easily view and purchase items without leaving their homes. However, the growth of an online shopping site will largely depend on the database management system responsible for handling diverse information ranging from product inventory to related transactions. Without a structured database system, businesses are bound to encounter problems such as mismanagement of stock, inefficient order processing, and lack of security of company and client data, which will negatively affect customer relation and revenue.

Database issues can cause problems in managing user data, tracking orders, and controlling stock levels, while weak security increases the risk of data loss. A specialized, secure database is needed for efficient e-commerce operations.

In this project, we utilized the Faker Python module [1] to generate data for our tables. This allowed us to create realistic datasets for user accounts, items for sale, order fulfillment, and payments, ensuring a comprehensive simulation of an online shopping experience.

```python
# Initialize Faker
fake = Faker()

# Step 1: Generate Users
num_users = 3500
users = []

for user_id in range(1, num_users + 1):
    username = fake.user_name()
    password = fake.password()
    email = fake.email()
    address = fake.address()
    phone_number = fake.phone_number()
    users.append({
        'user_id': user_id,
        'username': username,
        'password': password,
        'email': email,
        'address': address,
        'phone_number': phone_number
    })

users_df = pd.DataFrame(users)
users_df.to_csv("users.csv", index=False)
```

Fig. 1. Tables creation using faker



	user_id	username	password	email	address	phone_number
1	1	jennifer90	bV8C0oge&(	brenda51@example.org	9885 Glass Drives East Christinaland, TX 11932	869-849-4394
2	2	susanjenkins	)s_ZVwjW@8	paulaskinner@example.net	2916 Mario Loop North Jeremyfort, VT 63698	269-932-5895
3	3	torresmichael	s3^(6Qq2%E	beth34@example.com	84288 Adrian Park South Josephfort, MO 35719	421.321.6686x421
4	4	lauraharrison	G&#r4Mv+bo	jeffrey27@example.net	USNV Mendoza FPO AA 12642	(840)441-7266
5	5	woodtina	%(9WSIlHu4^	swilliams@example.org	2623 Hall Manor Suite 914 Jessicaborough, KY 05727	997-204-9297
6	6	brandonhatfield	)7G*xNb*YO	megan22@example.org	45671 Bowen Place Smithview, ND 48466	390.747.3663x406
7	7	jacksonalfred	J^*V2abaDA	aharrison@example.net	972 Freeman Curve New Kimberly, HI 01785	+1-639-887-2492x7663
8	8	kmyers	^%3JnkexP#	harrisjohn@example.com	7271 Brian Oval Bryanhaven, KY 09870	(255)234-2440
9	9	ipotter	*ExR5MLc0e	carloshughes@example.com	USNV Hunt FPO AA 15345	780.319.3629x47463
10	10	mjordan	_9Ht2UOH)k	ipittman@example.net	55687 Elizabeth Springs Suite 439 Markborough, MI 66084	296.963.5585
11	11	whitevalerie	UI0GillIyK	calhountroy@example.org	2481 Huynh Street Apt. 965 Meghanstad, RI 01131	+1-697-464-5649
12	12	ginabrooks	p4FHfWfq&3	wrightkristine@example.org	Unit 8652 Box 3287 DPO AE 68930	(708)995-4239x014
13	13	christophermorton	(v0l&s%N5d	moraangela@example.com	0971 Smith Cliffs Apt. 307 North Katherineberg, TX 35230	+1-784-428-7736x1686
14	14	jennifer11	_@zvowFml2	collinsamy@example.com	449 Howard Ways Port Sethland, NH 54336	739-617-8662

Fig. 2. Output

II. Background

This project also aims to reinforce the need for having an organized and accessible database particularly in the area of e-shops that are continually growing. e commerce is on the rise largely due to the proliferation of internet and mobile phone hence the need for greater backend systems has deepened. Merchants who operate online must maintain product information, manage customer interaction and transactions, as well as offer their customers user satisfactions.

With an increase in the number of products and the volume of the transactions, the e-business cannot do without a formidable database management system. The aim of the project is to develop a uniform and distributed architecture which is able to operate with different kinds of e-commerce information such as product specifications, orders, customer reviews, as well as payment details. This will include keeping record of the products, order management, customer reviews business, and payments management. Due to the elimination of the inefficiency factor from the use of the dataset by implementing a relational database system in this case, the project seeks to achieve the goal from the structure that will be adaptable all round as far as E-commerce is concerned.

Significance of the Problem: The lack of a well-defined database system in businesses leads to inefficiency in electronic transactions for the business, high chances of data duplication, and lots of security risks. The limitation on the amount of data support becomes an issue and response time and system performance on the other hand suffer and users aggravate. The construction of a comprehensive database management system assists in the maintenance of proper data integrity, increased storage space, and data safety which are some of the most crucial elements of an e-commerce system.

III. Scope

The project goals in terms of the design and the development of the "Online Shopping Cart System" database cover the three levels of a database - a multi user relational database suitable for online shopping centers. The main purpose of the project is to create a database that can smoothly resolve various e-commerce challenges such as data integrity, scalability, security, performance, etc.

The database will include important elements such as Users, Products, Categories, Shopping Carts, Orders, Payments, and Reviews taking from the data set. There will be prospects of user account management, tracking the availability of the products, shoppers' cart actions, and order management among others. Besides, the database is going to be developed in such a way that complex features such as how to make payments online to prevent fraud, real-time information, or working with other companies' applications is possible. It will ensure that optimization works on the target format so that there is very little duplication in the database design, there is efficient retrieval of information from the database and also there is provision for other functions e.g. payment, review and order tracking penetration. The database of the project will be designed to ensure that the growing number of users and constant change of the business will be adequately supported by the development of appropriate database management systems. This is important because it is the main part supporting a good and easy to use electronically based marketing.

IV. Target User

The Shopper's database system is intended to serve the shoppers who go to the online shopping web site to seek products, buy them and check the status of their orders. These customers use the database to create and enforce their profiles, provide true product information, and handle the payment settlements. Also, Managers and Administrators will have access to the database for management purposes to keep track of activities, customer activities, stock control and compile sales analysis reports using the system. System will be used for catalog marketing, catalog control, price updates and promotion management.

On this type of database architecture, an online vendor such as Amazon or Flipkart would implement in order to handle millions of users and products. The online store platform has a hierarchical structure, based on the database. Online customers can view products, order goods, and give feedback, whilst offline users maintain the sales database and analyze the sales data to improve stock levels. This project is envisaged to enhance the very base of the business systems that will help manage the databases to ensure quality, customer satisfaction, and the business operating in a viable arena.

V. E/R Diagram

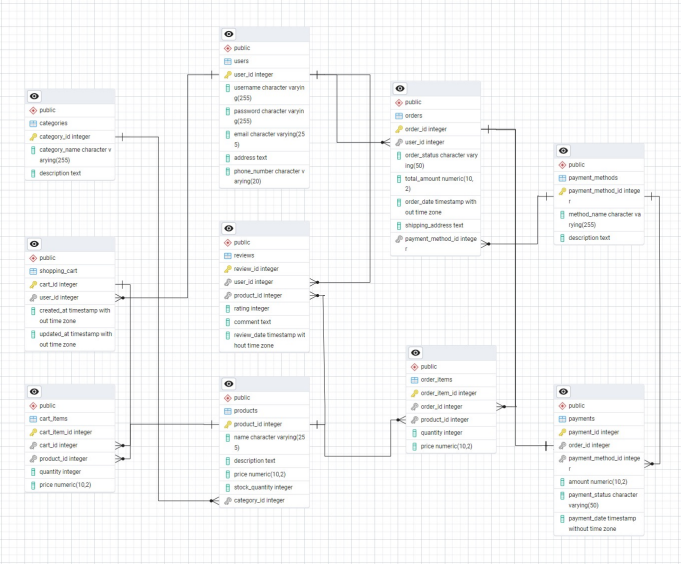


Fig. 3. ER diagram for Online Shopping System

1) **Users to Shopping Cart:**
   - One-to-Many: One user can have multiple shopping carts, but each shopping cart belongs to only one user.
   - The user_id in the shopping_cart table references user_id in the users table.

2) **Users to Orders:**
   - One-to-Many: A user can place multiple orders (one-to-many), and an order is placed by one user (many-to-one).
   - The user_id in the orders table references user_id in the users table.

3) **Users to Reviews:**
   - One-to-Many: A user can write multiple reviews.
   - o The user_id in the reviews table references user_id in the users table.

4) **Products to Categories:**
   - Many-to-One: Each product belongs to a single category, but a category can have multiple products.
   - The category_id in the products table references category_id in the categories table.

5) **Shopping Cart to Cart Items:**
   - One-to-Many: A shopping cart can contain multiple cart items.
   - The cart_id in the cart_items table references cart_id in the shopping_cart table.

6) **Cart Items to Products:**
   - Many-to-One: Each cart item is associated with a single product, but a product can appear in multiple cart items.
   - The product_id in the cart_items table references product_id in the products table.

7) **Orders to Order Items:**
   - One-to-Many: An order can have multiple order items.
   - The order_id in the order_items table references order_id in the orders table.

8) **Order Items to Products:**
   - Many-to-One: Each order item is associated with one product, but a product can appear in multiple order items.
   - The product_id in the order_items table references product_id in the products table.

9) **Orders to Payments:**
   - One-to-One and One-to-Many: Typically one-to-one in many cases. Once an order is placed, it is associated with a single payment.
   - There may be few cases when an order can be paid in installments.

10) **Payments to Payment Methods:**
   - Many-to-One: A payment is associated with one payment method, but a payment method can be used for multiple payments.
   - The payment_method_id in the payments table references payment_method_id in the payment_methods table.

11) **Products to Reviews:**
   - One-to-Many: A product can have multiple reviews.
   - The product_id in the reviews table references product_id in the products table.

VI. BCNF

To ensure the accuracy and efficiency of our database structure, it's crucial to transform the schema into Boyce-Codd Normal Form (BCNF). This process involves analyzing the functional dependencies within each relation and decomposing them where necessary. The primary goal is to eliminate anomalies and redundancies while preserving data integrity. We'll begin by identifying the dependencies in each table and then modify them to comply with the rules of BCNF.

1) **Users Table:**
   - Functional Dependencies:
     - `user_id` (Candidate key) → `username, password, email, address, phone_number`

     Using `user_id` we can uniquely determine all the other attributes and there are no partial dependencies or transitive dependencies.
     Conclusion: This table is already in BCNF.

2) **Categories Table:**
   - Functional Dependencies:
     - `category_id` (Candidate key) → `category_name, description`

     Using `category_id`,we can uniquely determine all the other attributes and there are no partial dependencies or transitive dependencies.
     Conclusion: This table is already in BCNF.

3) **Products Table:**
   - Functional Dependencies:
     - `product_id` (Candidate key) → `name, description, price, stock_quantity, category_id`

     `product_id` is the Candidate key, and it determines all other attributes. Each product has only one category and there are no partial dependencies or transitive dependencies.
     Conclusion: This table is already in BCNF.

4) **Reviews Table:**
   - Functional Dependencies:
     - `review_id` (Candidate key) → `user_id, product_id, rating, comment, review_date`

     `review_id` is the Candidate key and it determines all other attributes. One review_id can have only one rating or comment for one product by the user and there are no partial dependencies or transitive dependencies.
     Conclusion: This table is already in BCNF.

5) **Shopping Cart Table:**
   - Functional Dependencies:
     - `cart_id` (Candidate key) → `user_id, created_at, updated_at`

     `cart_id` is the Candidate key and it determines all other attributes, which means each cart belongs to one user and has timestamps and there are no partial dependencies or transitive dependencies.
     Conclusion: This table is already in BCNF.
6) **Cart Items Table:**
   - Functional Dependencies:
     - `cart_item_id` (Candidate key) → {`cart_id, product_id, quantity, price`}
       Using `cart_item_id` we can determines all other attributes and there are no partial dependencies or transitive dependencies..

     Conclusion: This table is already in BCNF.
7) **Orders Table:**
   - Functional Dependencies:
     - `order_id` (Candidate key) → {`user_id, order_status, total_amount, order_date, shipping_address, payment_method_id`}

     Using `order_id` as the Candidate key we can determine all other attributes. Each order belongs to one user and there are no partial dependencies or transitive dependencies.
     Conclusion: This table is already in BCNF.
8) **Order Items Table:**
   - Functional Dependencies:
     - `order_item_id` (Candidate key)→ {`order_id, product_id, quantity, price`}

     `order_item_id` is the Candidate key, and it determines all other attributes. Each Order Item ID has only one Price and there are no partial dependencies or transitive dependencies.
     Conclusion: This table is already in BCNF.
9) **Payments Table:**
   - Functional Dependencies:
     - `payment_id` (Candidate key)→ {`order_id, payment_method_id, amount, payment_status, payment_date`}

     `payment_id` is the Candidate key, and it determines all other attributes. Each payment_id can have only one payment_method and there are no partial dependencies or transitive dependencies.
     Conclusion: This table is already in BCNF.
10) **Payment Methods Table:**
    - Functional Dependencies:
      - `payment_method_id` (Candidate key) → `method_name, description`

      `payment_method_id` is the Candidate key and determines all other attributes and there are no partial dependencies or transitive dependencies.
      Conclusion: This table is already in BCNF.

VII. Table Relations

1) **USERS**
   - Primary Key: user_id
   - Foreign Keys: None
   - Attributes:
     - user_id (integer, not NULL): The unique identifier for each user. Serves as the primary key.
     - username (varchar, not NULL): User's display name for account purposes.
     - email (varchar, not NULL): Email address used for communication.
     - password (varchar, not NULL): Password for authentication.
     - address (varchar, NULL): User's physical address for deliveries.
     - phone_number (varchar, NULL): User's contact number for communication.
   - Purpose: The "users" table holds the data related to users. It stores unique data for each user like user_id, username, email, password, and optional fields like address and phone number for communication.
   - Constraints: user_id, username, email, and password cannot be NULL as they are essential for identifying and communicating with the user.
   - Actions on Foreign Keys: No action, as no foreign key references this table.

2) **CATEGORIES**
   - Primary Key: category_id
   - Foreign Keys: None
   - Attributes:
     - category_id (integer, not NULL): Unique identifier for each category.
     - category_name (varchar, not NULL): Name of the product category.
     - description (text, NULL): Description of the category.
   - Purpose: The "categories" table classifies the types of products available in the system.
   - Constraints: category_id and category_name cannot be NULL as every product must belong to a category.
   - Actions on Foreign Keys: No action, as no foreign key references this table.

3) **PRODUCTS**
   - Primary Key: product_id
   - Foreign Keys: category_id (references Categories.category_id)

- Attributes:
  - product_id (integer, not NULL): Unique identifier for each product.
  - name (varchar, not NULL): Name of the product.
  - description (text, NULL): Detailed description of the product.
  - price (decimal, not NULL): The selling price of the product.
  - stock_quantity (integer, not NULL): Available stock for the product.
  - category_id (integer, not NULL): Refers to the category to which the product belongs.
- Purpose: The "products" table holds information about the items available for purchase. Each product has a unique identifier and attributes like price, stock, and category.
- Constraints: product_id, price, stock_quantity, and category_id cannot be NULL because they are essential for product sales and categorization.
- Actions on Foreign Keys: If category_id (referenced category) is deleted, associated products can be deleted (ON DELETE CASCADE) or set to NULL.

4) **REVIEWS:**
- Primary Key: review_id
  - user_id (references Users.user_id)
  - product_id (references Products.product_id)
- Foreign Keys: None
- Attributes:
  - review_id (integer, not NULL): Unique identifier for each review.
  - user_id (integer, not NULL): Refers to the user writing the review.
  - product_id (integer, not NULL): Refers to the product being reviewed.
  - rating (integer, not NULL): Rating given to the product (1 to 5).
  - comment (text, NULL): Optional comment about the product.
  - review_date (timestamp, not NULL): Date the review was submitted.
- Purpose: The "reviews" table stores product reviews from users, including the rating and optional comment.
- Constraints: review_id, user_id, product_id, and rating cannot be NULL.
- Actions on Foreign Keys:
  - If user_id is deleted, the reviews may either be deleted or anonymized.
  - If product_id is deleted, the reviews may either be deleted or retained without product reference.

5) **SHOPPING CART:**
- Primary Key: cart_id
- Foreign Keys: user_id (references Users.user_id)
- Attributes:
  - cart_id (integer, not NULL): Unique identifier for each cart.
  - user_id (integer, not NULL): Refers to the user who owns the cart.
  - created_at (timestamp, not NULL): Timestamp when the cart was created.
  - updated_at (timestamp, NULL): Timestamp when the cart was last updated.
- Purpose: The "shopping_cart" table stores details about each user's cart, including the user and time-related data.
- Constraints: cart_id and user_id cannot be NULL, as they are needed to track the cart's owner and lifecycle.
- Actions on Foreign Keys: When user_id is deleted, the related cart will be deleted (ON DELETE CASCADE).

6) **CART ITEMS:**
- Primary Key: cart_item_id
- Foreign Keys: user_id (references Shopping Cart.cart_id)
  - product_id (references Products.product_id)
- Attributes:
  - cart_item_id (integer, not NULL): Unique identifier for each item in a cart.
  - cart_id (integer, not NULL): Refers to the cart this item belongs to.
  - product_id (integer, not NULL): Refers to the product added to the cart.
  - quantity (integer, not NULL): Quantity of the product in the cart.
  - price (decimal, not NULL): Price of the product at the time of adding to the cart.
- Purpose: The "cart_items" table stores information about products added to the shopping cart, including product details and quantity.
- Constraints: cart_id, product_id, quantity, and price cannot be NULL since each item must be linked to a product in a cart.
- Actions on Foreign Keys:
  - If cart_id is deleted, the associated items will also be deleted (ON DELETE CASCADE).
  - If product_id is deleted, the cart items may either be deleted or set to NULL.

7) **ORDERS:**
- Primary Key: order_id
- Foreign Keys: user_id (references Users.user_id)
  - payment_method_id (references Payment Methods.payment_method_id)
- Attributes:

  - order_id (integer, not NULL): Unique identifier for each order.
  - user_id (integer, not NULL): Refers to the user placing the order.
  - order_status (varchar, not NULL): Status of the order (e.g., pending, shipped, delivered).
  - total_amount (decimal, not NULL): Total amount for the order.
  - order_date (timestamp, not NULL): Date the order was placed.
  - shipping_address (varchar, not NULL): Address for delivering the order.
  - payment_method_id (integer, not NULL): Refers to the method used for payment.
- Purpose: The "orders" table stores details of user purchases, including payment and shipping information.
- Constraints: order_id, user_id, payment_method_id, and total_amount cannot be NULL.
- Actions on Foreign Keys:
  - If user_id is deleted, the order may either be deleted or set to NULL.
  - If payment_method_id is deleted, the order can either reference another method or be set to NULL.

8) **ORDER ITEMS:**
- Primary Key: order_item_id
- Foreign Keys:order_id (references Orders.order_id)
  - product_id (references Products.product_id)
- Attributes:
  - order_item_id (integer, not NULL): Unique identifier for each item in the order.
  - order_id (integer, not NULL): Refers to the order this item belongs to.
  - product_id (integer, not NULL): Refers to the product in the order.
  - quantity (integer, not NULL): Quantity of the product ordered.
  - price (decimal, not NULL): Price of the product at the time of the order.
- Purpose: The "order_items" table stores information about the products in an order, including quantity and price.
- Constraints: order_item_id, order_id, product_id, and quantity cannot be NULL.
- Actions on Foreign Keys:
  - If order_id is deleted, the corresponding order items are deleted (ON DELETE CASCADE).
  - If product_id is deleted, the corresponding order items may either be deleted or set to NULL.

9) **PAYMENTS:**
- Primary Key: payment_id
- Foreign Keys:
  - order_id (references Orders.order_id)
  - payment_method_id (references Payment Methods.payment_method_id)
- Attributes:
  - payment_id (integer, not NULL): Unique identifier for each payment.
  - order_id (integer, not NULL): Refers to the associated order.
  - payment_method_id (integer, not NULL): Refers to the method used for payment.
  - amount (decimal, not NULL): Amount paid.
  - payment_status (varchar, not NULL): Status of the payment (e.g., pending, completed).
  - payment_date (timestamp, not NULL): Date the payment was made.
- Purpose: The "payments" table stores information about payments made for orders, including the amount and payment method used.
- Constraints:payment_id, order_id, payment_method_id, and amount cannot be NULL.
- Actions on Foreign Keys:
  - If order_id is deleted, the payment may be deleted or set to NULL.
  - If payment_method_id is deleted, the payment may reference another method or be set to NULL.

10) **PAYMENT METHODS:**
- Primary Key: payment_method_id
- Foreign Keys: None
- Attributes:
  - payment_method_id (integer, not NULL): Unique identifier for each payment method.
  - method_name (varchar, not NULL): Name of the payment method (e.g., credit card, debit card).
  - description (text, NULL): Description of the payment method.
- Purpose: The "payment_methods" table stores available payment options, such as credit cards, debit cards, etc.
- Constraints: payment_method_id and method_name cannot be NULL.
- Actions on Foreign Keys:
  - No action, as no foreign key references this table.

VIII. Query Execution

1) **INSERTING DATA INTO TABLES:** The following SQL query was executed to insert sample data into the categories table:
Two hundred categories were inserted, each with a unique category_id and a name and description of the category.

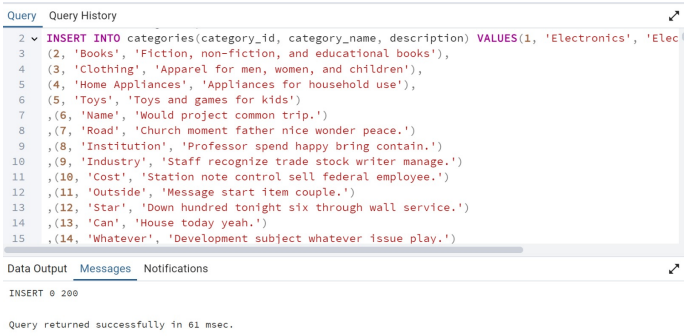
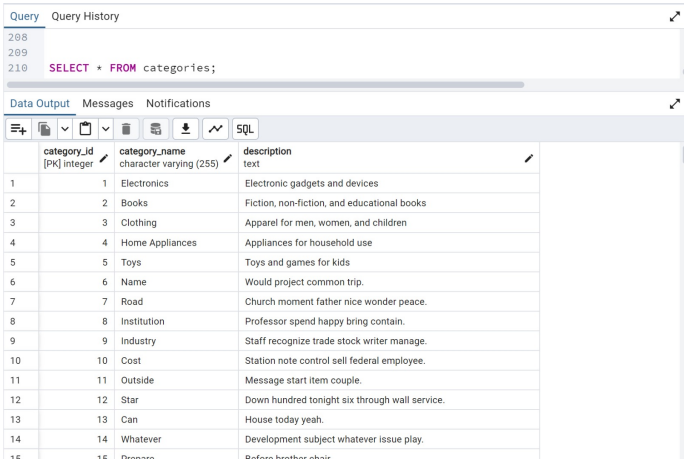


Fig. 4. Insert into categories

To verify the insertion, the following query was executed which returns all rows from the categories table, confirming that the data was successfully inserted. Below is the output:



	category_id [PK] integer	category_name character varying (255)	description text
1	1	Electronics	Electronic gadgets and devices
2	2	Books	Fiction, non-fiction, and educational books
3	3	Clothing	Apparel for men, women, and children
4	4	Home Appliances	Appliances for household use
5	5	Toys	Toys and games for kids
6	6	Name	Would project common trip.
7	7	Road	Church moment father nice wonder peace.
8	8	Institution	Professor spend happy bring contain.
9	9	Industry	Staff recognize trade stock writer manage.
10	10	Cost	Station note control sell federal employee.
11	11	Outside	Message start item couple.
12	12	Star	Down hundred tonight six through wall service.
13	13	Can	House today yeah.
14	14	Whatever	Development subject whatever issue play.

Fig. 5. Displaying the categories table data

The following SQL query was executed to insert sample data into the products table:
Thousand products were inserted, each with a unique product_id and the information about the items available for purchase, including their pricing, stock levels, and associated categories.

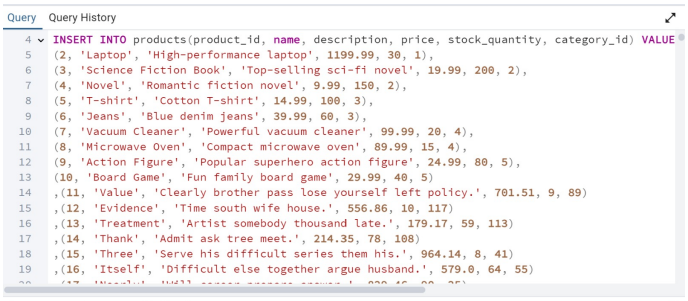
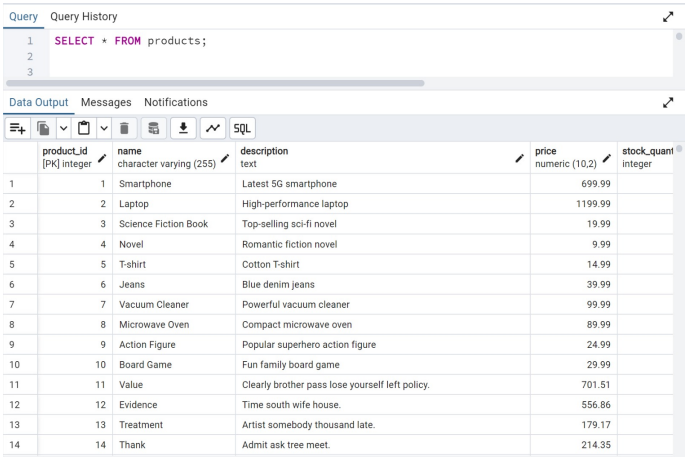


Fig. 6. Inserting into products

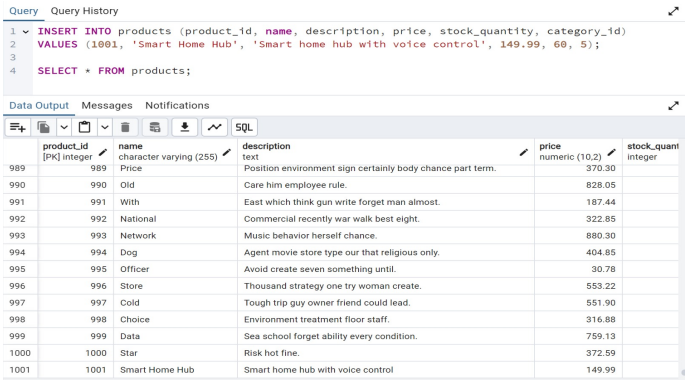
To verify the insertion, the following query was executed which returns all rows from the products table, confirming that the data was successfully inserted. Below is the output:



	product_id [PK] integer	name character varying (255)	description text	price numeric (10,2)	stock_quant integer
1	1	Smartphone	Latest 5G smartphone	699.99	
2	2	Laptop	High-performance laptop	1199.99	
3	3	Science Fiction Book	Top-selling sci-fi novel	19.99	
4	4	Novel	Romantic fiction novel	9.99	
5	5	T-shirt	Cotton T-shirt	14.99	
6	6	Jeans	Blue denim jeans	39.99	
7	7	Vacuum Cleaner	Powerful vacuum cleaner	99.99	
8	8	Microwave Oven	Compact microwave oven	89.99	
9	9	Action Figure	Popular superhero action figure	24.99	
10	10	Board Game	Fun family board game	29.99	
11	11	Value	Clearly brother pass lose yourself left policy.	701.51	
12	12	Evidence	Time south wife house.	556.86	
13	13	Treatment	Artist somebody thousand late.	179.17	
14	14	Thank	Admit ask tree meet.	214.35	

Fig. 7. Displaying products table data

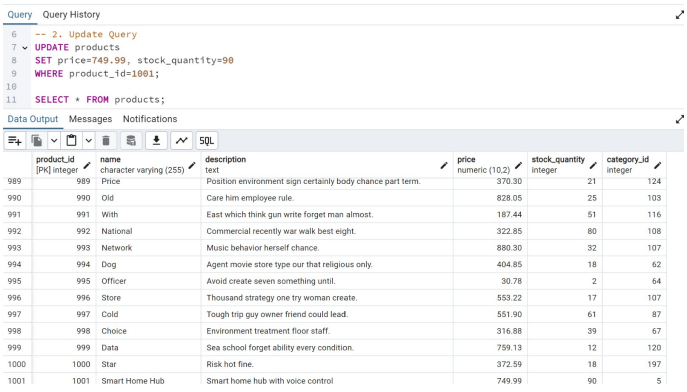
2) **INSERTING NEW ROW INTO EXISTING TABLE:** The query returned all rows from the products table, including the newly added product. Below is the representation of the new entry:



	product_id [PK] integer	name character varying (255)	description text	price numeric (10,2)	stock_quant integer
989	989	Price	Position environment sign certainly body chance part term.	370.30	
990	990	Old	Care him employee rule.	828.05	
991	991	With	East which think gun write forget man almost.	187.44	
992	992	National	Commercial recently war walk best eight.	322.85	
993	993	Network	Music behavior herself chance.	880.30	
994	994	Dog	Agent movie store type our that religious only.	404.85	
995	995	Officer	Avoid create seven something until.	30.78	
996	996	Store	Thousand strategy one try woman create.	553.22	
997	997	Cold	Tough trip guy owner friend could lead.	551.90	
998	998	Choice	Environment treatment floor staff.	316.88	
999	999	Data	Sea school forget ability every condition.	759.13	
1000	1000	Star	Risk hot fine.	372.59	
1001	1001	Smart Home Hub	Smart home hub with voice control	149.99	

Fig. 8. Inserting new row to the existing product table

3) **UPDATE QUERY:** The following SQL query was executed to update the price and stock quantity of the product with product_id = 1001 and verifying the update. The output shows that the product, Smart Home Hub, was successfully updated with a new price of 749.99 and stock_quantity = 90.



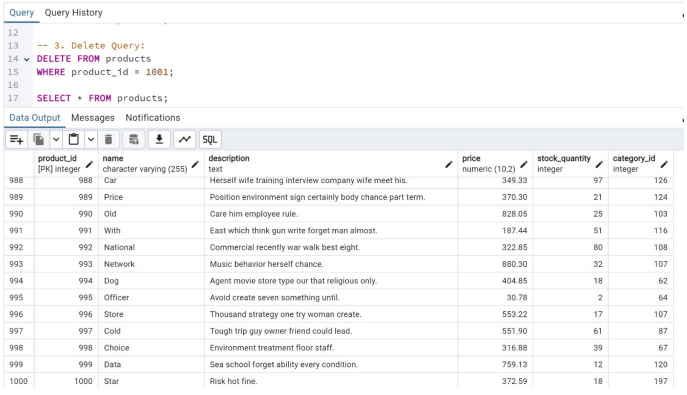
```
-- 2. Update Query
UPDATE products
SET price=749.99, stock_quantity=90
WHERE product_id=1001;

SELECT * FROM products;
```

Data Output | Messages | Notifications

	product_id [PK] integer	name character varying (255)	description text	price numeric (10,2)	stock_quantity integer	category_id integer
989	989	Price	Position environment sign certainly body chance part term.	370.30	21	124
990	990	Old	Care him employee rule.	828.05	25	103
991	991	With	East which think gun write forget man almost.	187.44	51	116
992	992	National	Commercial recently war walk best eight.	322.85	80	108
993	993	Network	Music behavior herself chance.	880.30	32	107
994	994	Dog	Agent movie store type our that religious only.	404.85	18	62
995	995	Officer	Avoid create seven something until.	30.78	2	64
996	996	Store	Thousand strategy one try woman create.	553.22	17	107
997	997	Cold	Tough trip guy owner friend could lead.	551.90	61	87
998	998	Choice	Environment treatment floor staff.	316.88	39	67
999	999	Data	Sea school forget ability every condition.	759.13	12	120
1000	1000	Star	Risk hot fine.	372.59	18	197
1001	1001	Smart Home Hub	Smart home hub with voice control	749.99	90	5

Fig. 9. Update Query on products table

4) **DELETE QUERY:** The following SQL query was executed to delete the product with product_id = 1001.The WHERE clause specifies the product to be deleted by using (product_id = 1001).
In this case, the product Smart Home Hub is removed from the database. So, the output of the query is the old table before an update.



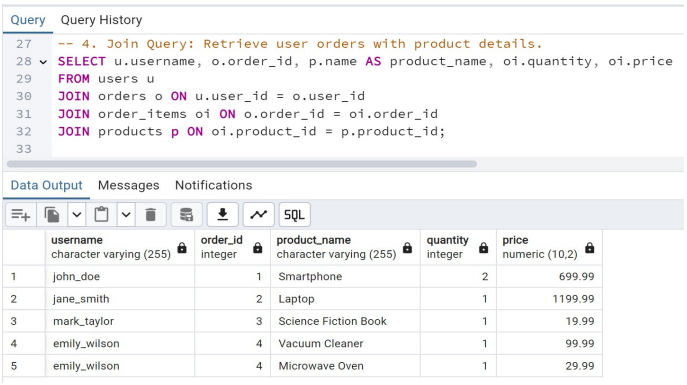
```
-- 3. Delete Query:
DELETE FROM products
WHERE product_id = 1001;

SELECT * FROM products;
```

Data Output | Messages | Notifications

	product_id [PK] integer	name character varying (255)	description text	price numeric (10,2)	stock_quantity integer	category_id integer
988	988	Car	Herself wife training interview company wife meet his.	349.33	97	126
989	989	Price	Position environment sign certainly body chance part term.	370.30	21	124
990	990	Old	Care him employee rule.	828.05	25	103
991	991	With	East which think gun write forget man almost.	187.44	51	116
992	992	National	Commercial recently war walk best eight.	322.85	80	108
993	993	Network	Music behavior herself chance.	880.30	32	107
994	994	Dog	Agent movie store type our that religious only.	404.85	18	62
995	995	Officer	Avoid create seven something until.	30.78	2	64
996	996	Store	Thousand strategy one try woman create.	553.22	17	107
997	997	Cold	Tough trip guy owner friend could lead.	551.90	61	87
998	998	Choice	Environment treatment floor staff.	316.88	39	67
999	999	Data	Sea school forget ability every condition.	759.13	12	120
1000	1000	Star	Risk hot fine.	372.59	18	197

Fig. 10. Delete Query

5) **JOIN QUERY:** The following SQL query was executed to retrieve user orders with product details:
This query fetches detailed user order information using a JOIN of the users, orders, order_items, and products tables. It selects the username, order ID, product name, quantity, and price. Linking the users with their respective orders, orders to items, and items to products, this query provides a detailed summary of the orders by the users along with the product details, which might be useful in order management.



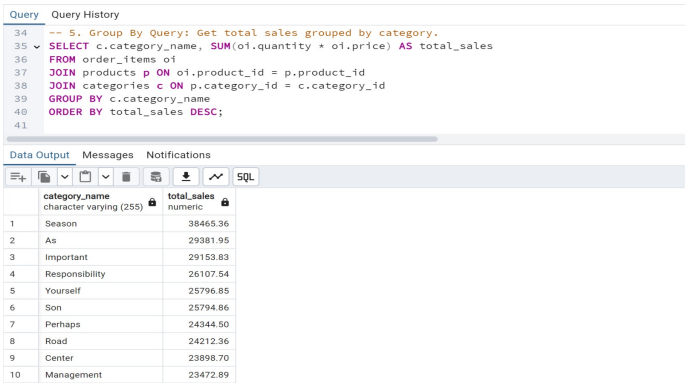
```
-- 4. Join Query: Retrieve user orders with product details.
SELECT u.username, o.order_id, p.name AS product_name, oi.quantity, oi.price
FROM users u
JOIN orders o ON u.user_id = o.user_id
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id;
```

Data Output | Messages | Notifications

	username character varying (255)	order_id integer	product_name character varying (255)	quantity integer	price numeric (10,2)
1	john_doe	1	Smartphone	2	699.99
2	jane_smith	2	Laptop	1	1199.99
3	mark_taylor	3	Science Fiction Book	1	19.99
4	emily_wilson	4	Vacuum Cleaner	1	99.99
5	emily_wilson	4	Microwave Oven	1	29.99

Fig. 11. Join Query: Retrieve user orders with product details

6) **GROUP BY QUERY:** The following SQL query was executed to compute total sales grouped by category:
It calculates total sales per category using SUM(oi.quantity * oi.price), groups the results by category_name, and sorts them in descending order of total sales.



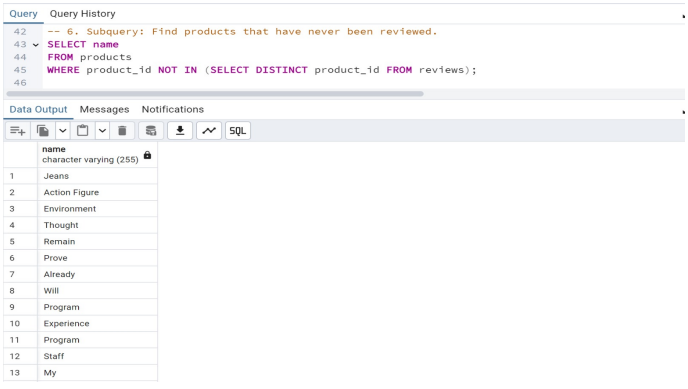
```
-- 5. Group By Query: Get total sales grouped by category.
SELECT c.category_name, SUM(oi.quantity * oi.price) AS total_sales
FROM order_items oi
JOIN products p ON oi.product_id = p.product_id
JOIN categories c ON p.category_id = c.category_id
GROUP BY c.category_name
ORDER BY total_sales DESC;
```

Data Output | Messages | Notifications

	category_name character varying (255)	total_sales numeric
1	Season	38465.36
2	As	29381.95
3	Important	29153.83
4	Responsibility	26107.54
5	Yourself	25796.85
6	Son	25794.86
7	Perhaps	24344.50
8	Road	24212.36
9	Center	23898.70
10	Management	23472.89

Fig. 12. Group By Query: Get total sales grouped by category

7) **SUBQUERY:** The following SQL query performs query inside another query operation which is called subquery.The purpose of this query is to identify products in the products table that have never been reviewed by customers.



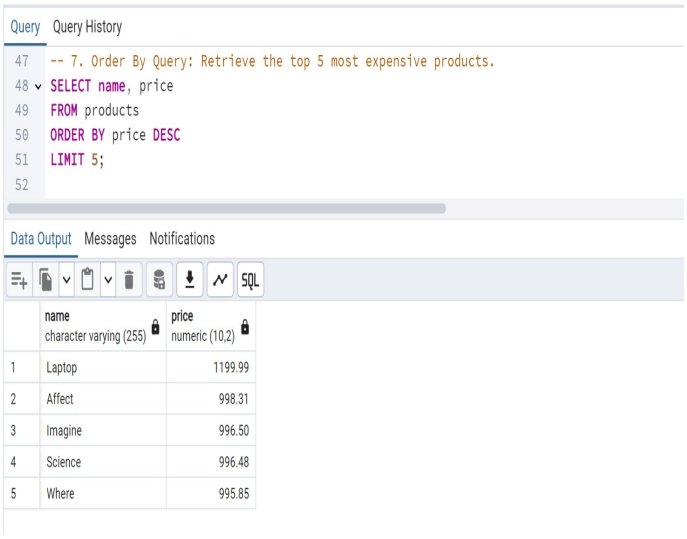
```
-- 6. Subquery: Find products that have never been reviewed.
SELECT name
FROM products
WHERE product_id NOT IN (SELECT DISTINCT product_id FROM reviews);
```

Data Output | Messages | Notifications

	name character varying (255)
1	Jeans
2	Action Figure
3	Environment
4	Thought
5	Remain
6	Prove
7	Already
8	Will
9	Program
10	Experience
11	Program
12	Staff
13	My

Fig. 13. Subquery: Find products that have never been reviewed

8) **ORDER BY:** It retrieves from the products table 5 of the most expensive products. It selects those columns: name and price. The rows are ordered in descending manner depending on the price column (ORDER BY price DESC) and the results will only show the first 5 rows because of LIMIT 5. This query is useful to identify high-value products in a database.

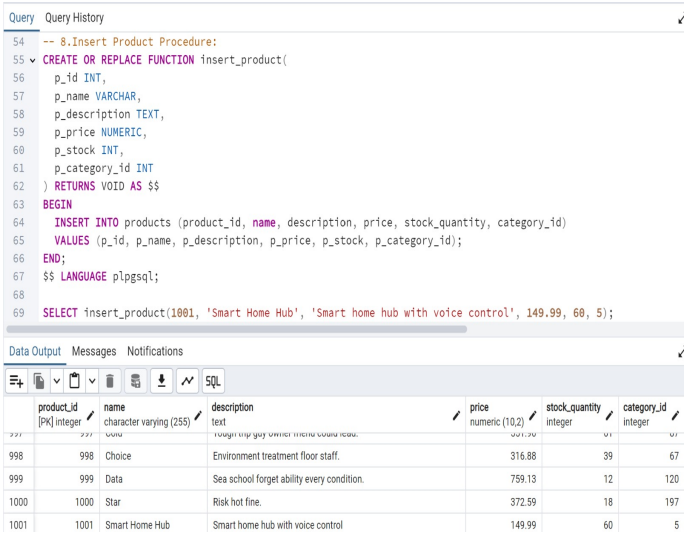


```
-- 7. Order By Query: Retrieve the top 5 most expensive products.
SELECT name, price
FROM products
ORDER BY price DESC
LIMIT 5;
```

	name character varying (255)	price numeric (10,2)
1	Laptop	1199.99
2	Affect	998.31
3	Imagine	996.50
4	Science	996.48
5	Where	995.85

Fig. 14. Order by: Retrieve the top 5 most expensive products

9) **PROCEDURES (insertion and deletion):** The below code create and execute a stored procedure to insert a new product into the products table. This simplifies repetitive insertion operations and demonstrates the use of procedural logic in the database.

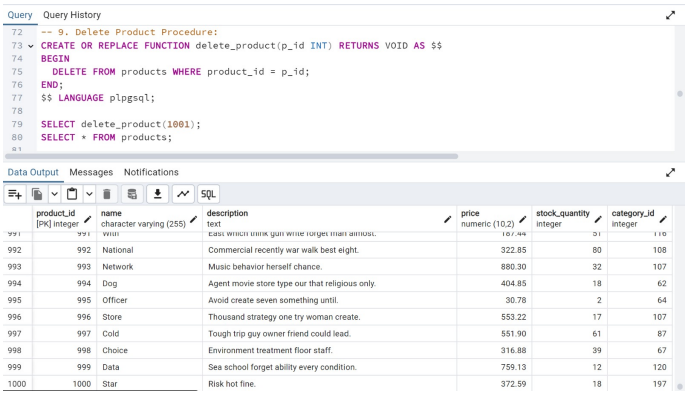


```
-- 8.Insert Product Procedure:
CREATE OR REPLACE FUNCTION insert_product(
  p_id INT,
  p_name VARCHAR,
  p_description TEXT,
  p_price NUMERIC,
  p_stock INT,
  p_category_id INT
) RETURNS VOID AS $$
BEGIN
  INSERT INTO products (product_id, name, description, price, stock_quantity, category_id)
  VALUES (p_id, p_name, p_description, p_price, p_stock, p_category_id);
END;
$$ LANGUAGE plpgsql;

SELECT insert_product(1001, 'Smart Home Hub', 'Smart home hub with voice control', 149.99, 60, 5);
```

product_id [PK] integer	name character varying (255)	description text	price numeric (10,2)	stock_quantity integer	category_id integer
998	998	Choice	Environment treatment floor staff.	316.88	39	67
999	999	Data	Sea school forget ability every condition.	759.13	12	120
1000	1000	Star	Risk hot fine.	372.59	18	197
1001	1001	Smart Home Hub	Smart home hub with voice control	149.99	60	5

Fig. 15. Insert Product Procedure

The below code create and execute a stored procedure to delete a product from the products table. This simplifies repetitive delete operations and demonstrates the use of procedural logic in the database.



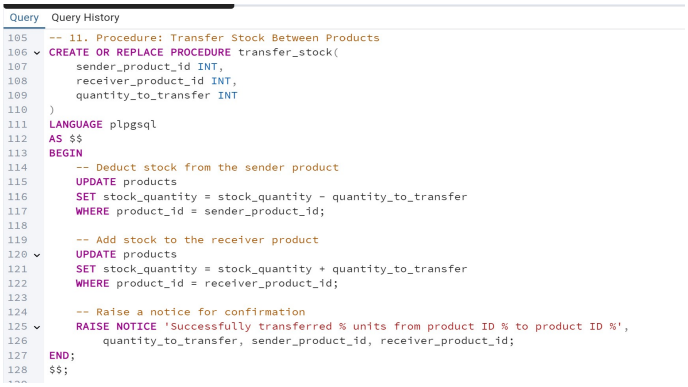
```
-- 9. Delete Product Procedure:
CREATE OR REPLACE FUNCTION delete_product(p_id INT) RETURNS VOID AS $$
BEGIN
  DELETE FROM products WHERE product_id = p_id;
END;
$$ LANGUAGE plpgsql;

SELECT delete_product(1001);
SELECT * FROM products;
```

product_id [PK] integer	name character varying (255)	description text	price numeric (10,2)	stock_quantity integer	category_id integer
992	992	National	Commercial recently war walk best eight.	322.85	80	108
993	993	Network	Music behavior herself chance.	880.30	32	107
994	994	Dog	Agent movie store type our that religious only.	404.85	18	62
995	995	Officer	Avoid create seven something until.	30.78	2	64
996	996	Store	Thousand strategy one try woman create.	553.22	17	107
997	997	Cold	Tough trip guy owner friend could lead.	551.90	61	87
998	998	Choice	Environment treatment floor staff.	316.88	39	67
999	999	Data	Sea school forget ability every condition.	759.13	12	120
1000	1000	Star	Risk hot fine.	372.59	18	197

Fig. 16. Delete Product Procedure

These functionalities make the management of products easier by encapsulating common queries into re-usable procedures. They help maintain cleaner application code while ensuring data manipulation remains efficient and secure.

10) **PROCEDURES (selection):** This procedure, named transfer_stock, moves a specified amount of stock from one product to another in the database. It accepts three parameters: the ID of the sender product, the ID of the receiver product, and the quantity to transfer. The procedure decreases the sender product's stock by the given quantity and then increases the stock of the receiver product by the same amount. It then creates a confirmation message showing the amount transferred and the IDs of the products involved. This process helps in handling stock transfers among products in the database.



```
-- 11. Procedure: Transfer Stock Between Products
CREATE OR REPLACE PROCEDURE transfer_stock(
    sender_product_id INT,
    receiver_product_id INT,
    quantity_to_transfer INT
)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Deduct stock from the sender product
    UPDATE products
    SET stock_quantity = stock_quantity - quantity_to_transfer
    WHERE product_id = sender_product_id;

    -- Add stock to the receiver product
    UPDATE products
    SET stock_quantity = stock_quantity + quantity_to_transfer
    WHERE product_id = receiver_product_id;

    -- Raise a notice for confirmation
    RAISE NOTICE 'Successfully transferred % units from product ID % to product ID %',
        quantity_to_transfer, sender_product_id, receiver_product_id;
END;
$$;
```

Fig. 17. Procedure: Transfer Stock Between Products

Below invokes the transfer_stock procedure, transferring 2 units of stock from product 1 to product 2. It reduces the stock of product 1 by 2 and increases the stock of product 2 by 2, as per the logic defined in the procedure.

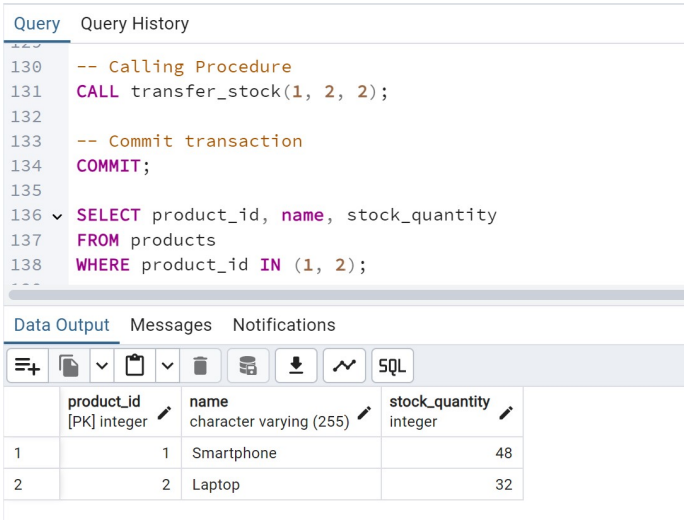


Fig. 18. Procedure Output: Transfer Stock Between Products

11) **FUNCTIONS:** The following function, get_product_revenue creates and executes a function that calculates the total revenue generated by a specific product. The function is used to analyze product performance in terms of sales.Products generating no revenue are handled by returning 0 instead of null values, ensuring completeness in the output.

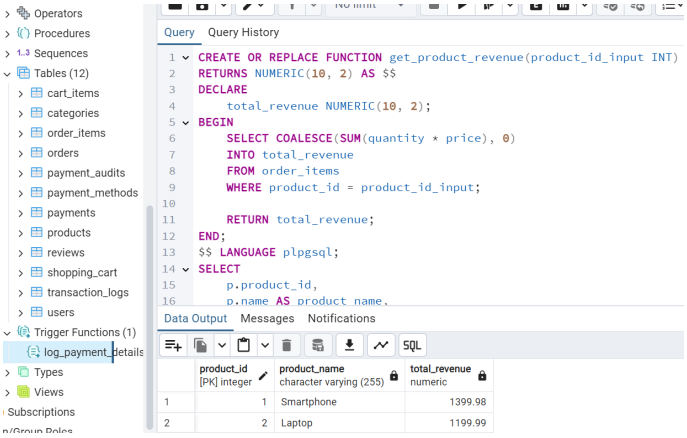


Fig. 19. Function: Calculates the total revenue generated by a specific product

IX. Transaction & Triggers

1) **Creating Trigger Function:** The check_insufficient_stock function is a trigger function that runs whenever there is an update to the stock_quantity in the products table. It ensures that the stock cannot be reduced below zero. If the NEW.stock_quantity is less than zero (indicating insufficient stock), it raises an exception with an error message and aborts the transaction, ensuring that invalid stock levels are never recorded in the table.

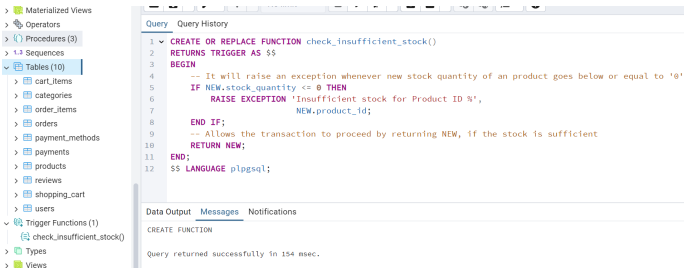


Fig. 20. Trigger Function

2) **Trigger Creation:** Before any update to the stock_quantity column in the products table, the prevent_insufficient_stock trigger is executed. It checks if the updated value (NEW.stock_quantity) is less than or equal to 0 for each row being updated. If this condition is true, it calls the trigger function check_insufficient_stock, which raises an exception to abort the update. This triggers an event and displays an error message, alerting the admin about the issue.

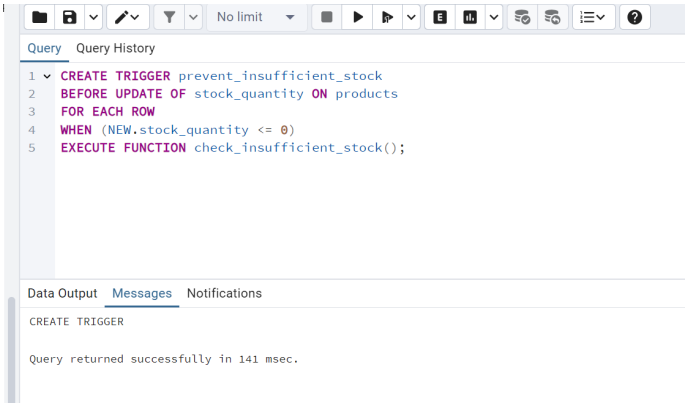


Fig. 21. Creation of Trigger

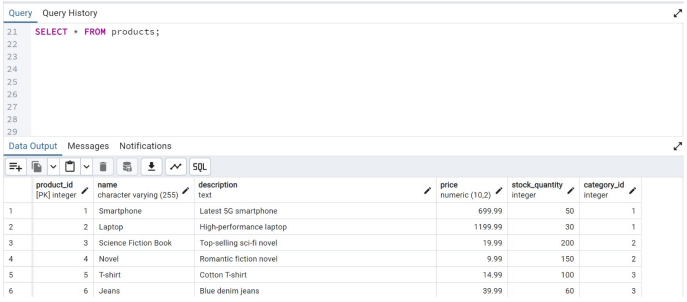


Fig. 22. Initial product stock_quantity details

3) **Updating stock_quantity:** This query reduces the stock_quantity of product_id = 1 by 30, successfully updating the value. The stock_quantity changes from 50 to 20 as the transaction is valid and does not violate any constraints.

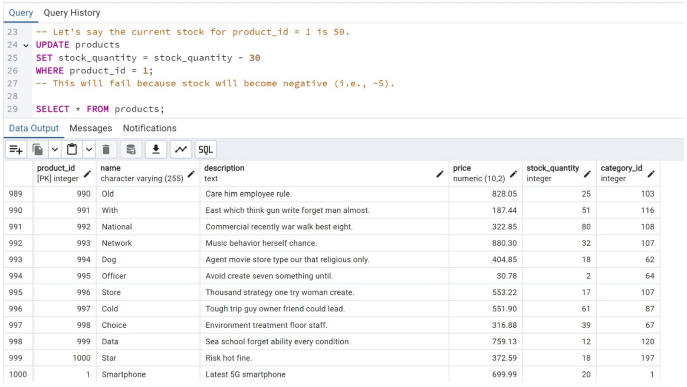


Fig. 23. Updating the product stock_quantity details

4) **Unsuccessful transaction:**
Here we are trying to update the stock_quantity for product_id = 1 by 30, which would result the stock_quantity by negative value. Since the new stock is less than 0, the prevent_insufficient_stock trigger is activated, and the check_insufficient_stock function raises an exception. So, the transaction is aborted, and the stock_quantity remains unchanged at 20 only.

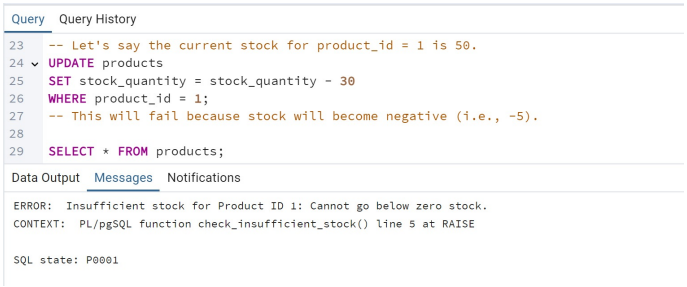


Fig. 24. Updating stock_quantity on unsuccessful transaction

We can see in the below attached image that for product_id = 1, the stock_quantity remains unchanged, the attempted update violated the rule against negative stock, so the transaction was aborted to assure data integrity and consistenty.

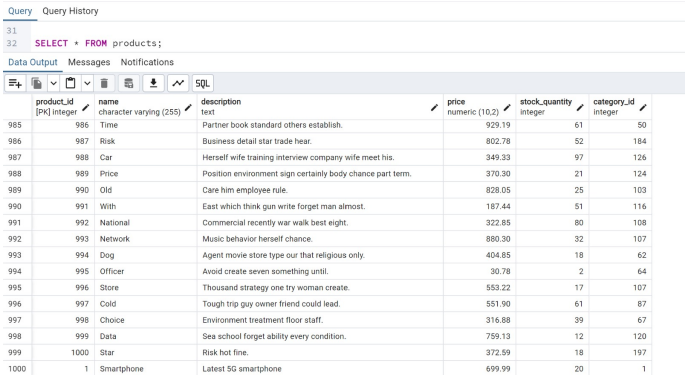


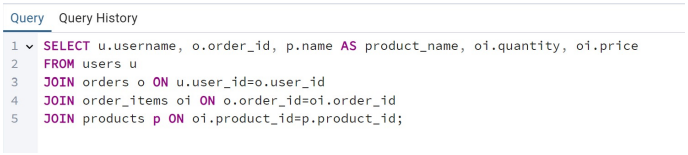
Fig. 25. Displaying unsuccessful transaction

When a transaction is aborted due to failure, all the changes made so far during the transaction are automatically rolled back, ensuring that the database is restored to its original state before the transaction began. This rollback mechanism prevents partial updates, preserving the integrity and consistency of the database and the trigger is executed to validate the operation and provide immediate feedback to the admin. This trigger raises an event notifying the admin as "Insufficient stock for Product ID 1.

X. Indexing and Query Execution Analysis

Query 1

1) **Join Query with Multiple Tables**



2) **Performance Issue**

- **Sequential Scans:** Without indexes on `users.user_id`, `orders.user_id`, `orders.order_id`, `order_items.order_id`, and `order_items.product_id`, PostgreSQL performs sequential scans, significantly slowing down the query execution.
- **Join Processing:** Each join operation (`users.user_id = orders.user_id`, `orders.order_id = order_items.order_id`, `order_items.product_id = products.product_id`) involves scanning the tables multiple times, which is computationally inefficient .
- **Lack of Indexes:** Without indexes on the join columns (`user_id`, `order_id`, `product_id`),

PostgreSQL cannot quickly locate matching rows, , leading to slower execution .

- **Table Size Impact:** As the size of the `users`, `orders`, `order_items`, and `products` tables grows, the performance degradation due to sequential scans becomes more pronounced.
- **Optimization:** Adding indexes on the columns `users.user_id`, `orders.user_id`, `orders.order_id`, `order_items.order_id`, and `order_items.product_id` enables faster lookups and significantly improves query performance.

3) **Execution Plan Analysis**
The execution plan of the query can be analysed using the `EXPLAIN ANALYZE` command. The observed issues include:

Query | Query History

```
1  EXPLAIN ANALYZE
2  SELECT u.username, o.order_id, p.name AS product_name, oi.quantity, oi.price
3  FROM users u
4  JOIN orders o ON u.user_id=o.user_id
5  JOIN order_items oi ON o.order_id=oi.order_id
6  JOIN products p ON oi.product_id=p.product_id;
```

4) **Analysis of the Execution Plan (Before Indexing)**

Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Join (cost=210.17..258.88 rows=1994 width=30) (actual time=1.756..3.486 rows=1994 loops=1)
2	Hash Cond: (oi.product_id = p.product_id)
3	-> Hash Join (cost=175.67..219.12 rows=1994 width=28) (actual time=1.419..2.659 rows=1994 loops=1)
4	Hash Cond: (o.user_id = u.user_id)
5	-> Hash Join (cost=34.92..73.13 rows=1994 width=22) (actual time=0.235..1.018 rows=1994 loops=1)
6	Hash Cond: (oi.order_id = o.order_id)
7	-> Seq Scan on order_items oi (cost=0.00..32.94 rows=1994 width=18) (actual time=0.013..0.203 rows=1994 loop...
8	-> Hash (cost=26.63..26.63 rows=663 width=8) (actual time=0.208..0.208 rows=663 loops=1)
9	Buckets: 1024 Batches: 1 Memory Usage: 34kB
10	-> Seq Scan on orders o (cost=0.00..26.63 rows=663 width=8) (actual time=0.033..0.135 rows=663 loops=1)
11	-> Hash (cost=97.00..97.00 rows=3500 width=14) (actual time=1.154..1.154 rows=3500 loops=1)
12	Buckets: 4096 Batches: 1 Memory Usage: 198kB
13	-> Seq Scan on users u (cost=0.00..97.00 rows=3500 width=14) (actual time=0.012..0.655 rows=3500 loops=1)
14	-> Hash (cost=22.00..22.00 rows=1000 width=10) (actual time=0.322..0.322 rows=1000 loops=1)
15	Buckets: 1024 Batches: 1 Memory Usage: 52kB
16	-> Seq Scan on products p (cost=0.00..22.00 rows=1000 width=10) (actual time=0.025..0.179 rows=1000 loops=1)
17	Planning Time: 6.605 ms
18	Execution Time: 3.714 ms

Fig. 26. Execution plan before indexing

The execution plan revealed several inefficiencies due to the lack of indexing and optimizations:

- **Hash Join Operations:**
  - The query uses multiple hash joins to combine tables based on various join conditions:
    - **Join 1:** `oi.product_id = p.product_id` with a cost range of `210.17..258.88`, processing 1994 rows in `1.756..3.486 ms`.
    - **Join 2:** `o.user_id = u.user_id` with a cost range of `175.67..219.12`, processing 1994 rows in `1.419..2.659 ms`.
    - **Join 3:** `oi.order_id = o.order_id` with a cost range of `34.92..73.13`, processing 1994 rows in `0.235..1.018 ms`.
  - The memory usage for the hash joins varies, with `34kB` and `198kB` buckets observed.
- **Sequential Scans:**
  - The query performs sequential scans on all involved tables:
    - `order_items (oi)`: Processes 1994 rows with a cost of `0.00..32.94` and execution time of `0.013..0.203 ms`.
    - `orders (o)`: Processes 663 rows with a cost of `0.00..26.63` and execution time of `0.033..0.135 ms`.
    - `users (u)`: Processes 3500 rows with a cost of `0.00..97.00` and execution time of `0.012..0.655 ms`.
    - `products (p)`: Processes 1000 rows with a cost of `0.00..22.00` and execution time of `0.025..0.179 ms`.
  - These scans indicate full table reads, which are suboptimal for larger datasets.
- **Hash Operations:**
  - The query builds hash tables for join operations, with memory usage ranging from `34kB` to `198kB`.
  - Hash tables processed rows as follows:
    - `663 rows` for `orders`.
    - `3500 rows` for `users`.
    - `1000 rows` for `products`.
- **Planning and Execution Times:**
  - The query planning time is `6.605 ms`, and the execution time is `3.714 ms`.
  - The execution times are efficient for this dataset size but may become a bottleneck as data grows.

5) **Proposed Solution**
To address the performance issues, Create indexes on foreign key columns to speed up joins:

Query | Query History

```
1  CREATE INDEX idx_orders_user_id ON orders(user_id);
2  CREATE INDEX idx_order_items_order_id ON order_items(order_id);
3  CREATE INDEX idx_order_items_product_id ON order_items(product_id);
4  CREATE INDEX idx_products_product_id ON products(product_id);
```

6) **Analysis of the Execution Plan (After Indexing)**
Since our dataset is sparse, we don't see any improvements in performance and thus this is not a problematic query for now, but as the dataset grows below are the common scenarios Where Queries Could Lead to Poor Performance

Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Join (cost=210.17..258.88 rows=1994 width=30) (actual time=1.897..3.264 rows=1994 loops=1)
2	Hash Cond: (oi.product_id = p.product_id)
3	-> Hash Join (cost=175.67..219.12 rows=1994 width=28) (actual time=1.534..2.487 rows=1994 loops=1)
4	Hash Cond: (o.user_id = u.user_id)
5	-> Hash Join (cost=34.92..73.13 rows=1994 width=22) (actual time=0.266..0.847 rows=1994 loops=1)
6	Hash Cond: (oi.order_id = o.order_id)
7	-> Seq Scan on order_items oi (cost=0.00..32.94 rows=1994 width=18) (actual time=0.010..0.139 rows=1994 loop...
8	-> Hash (cost=26.63..26.63 rows=663 width=8) (actual time=0.241..0.242 rows=663 loops=1)
9	Buckets: 1024 Batches: 1 Memory Usage: 34kB
10	-> Seq Scan on orders o (cost=0.00..26.63 rows=663 width=8) (actual time=0.015..0.130 rows=663 loops=1)
11	-> Hash (cost=97.00..97.00 rows=3500 width=14) (actual time=1.244..1.244 rows=3500 loops=1)
12	Buckets: 4096 Batches: 1 Memory Usage: 198kB
13	-> Seq Scan on users u (cost=0.00..97.00 rows=3500 width=14) (actual time=0.013..0.586 rows=3500 loops=1)
14	-> Hash (cost=22.00..22.00 rows=1000 width=10) (actual time=0.353..0.354 rows=1000 loops=1)
15	Buckets: 1024 Batches: 1 Memory Usage: 52kB
16	-> Seq Scan on products p (cost=0.00..22.00 rows=1000 width=10) (actual time=0.017..0.188 rows=1000 loops=1)
17	Planning Time: 6.296 ms
18	Execution Time: 3.427 ms

Fig. 27. Execution plan after indexing

7) **Common Scenario Where Queries Could Lead to Poor Performance**
   - **Larger Tables:**
     - As the `users`, `orders`, `order_items`, and `products` tables grow significantly (e.g., `order_items` reaching millions of rows), sequential scans on these tables will increase query execution time.
     - Joining large tables without indexing on join keys (e.g., `user_id`, `order_id`, and `product_id`) will become increasingly expensive.
   - **Joins Over Large Tables:**
     - The query performs multiple hash joins, which are efficient for small datasets but consume memory and degrade in performance as table sizes increase.
     - For example, joining `order_items` with `products` or `orders` will require scanning and hashing larger datasets, resulting in higher costs.
   - **Increased Memory Usage:**
     - Hash joins require building in-memory hash tables. As observed in the current execution plan, hash table memory usage ranges from `34kB` to `198kB`, but this will grow linearly with the number of rows in the joined tables.
     - Insufficient memory for hash operations could result in disk-based operations, significantly slowing the query.
   - **I/O Bottlenecks:**
     - Sequential scans on large tables (e.g., `users`, `products`) will result in higher I/O costs, especially if the dataset cannot fit into memory.
     - These I/O operations will degrade performance, particularly in scenarios with high query concurrency.
   - **Frequent Aggregations and Sorting:**
     - While this query does not include `GROUP BY` or `ORDER BY` clauses, similar queries that aggregate or sort results will require significant memory and computation time for larger datasets.
     - Sorting operations on unsorted or unindexed columns (e.g., `username` or `product_name`) will further degrade performance.
   - **Frequent Query Execution:**
     - If this query or similar joins are executed frequently, it may strain system resources due to repeated full table scans and hash join operations.
     - Adding proper indexing on join keys and filtering conditions can mitigate such performance issues.
   - **Network Overhead:**
     - Returning large result sets (e.g., with thousands of rows containing `username`, `order_id`, `product_name`, and other details) can lead to network latency and client-side processing overhead.
8) **Performance improvement**
   Proposed solutions can be used as queries to create indexes to improve performance. .

Query 2

1) **Group By Query for Sales Analysis**

Query | Query History

```
1  SELECT c.category_name, SUM(oi.quantity * oi.price) AS total_sales
2  FROM order_items oi
3  JOIN products p ON oi.product_id = p.product_id
4  JOIN categories c ON p.category_id = c.category_id
5  GROUP BY c.category_name
6  ORDER BY total_sales DESC;
```

2) **Performance Issue:**
   - **Aggregations (SUM):** The aggregation `SUM(oi.quantity * oi.price)` requires scanning all rows in `order_items` to compute total sales, which is slow without indexes.
   - **Join Processing:** The joins (`oi.product_id = p.product_id` and `p.category_id = c.category_id`) involve scanning `order_items` and `products`, adding computational cost.
   - **Lack of Indexes:** Without indexes on `order_items.product_id`, `products.product_id`, and `products.category_id`, PostgreSQL performs full table scans to find matching rows, slowing down query execution.
   - **Table Size Impact:** As the sizes of `order_items`, `products`, and `categories`

grow, the query performance degrades significantly due to increased scan times.

- **Optimization:** Adding indexes on `order_items.product_id`, `products.product_id`, and `products.category_id` can improve the efficiency of joins and aggregation.

3) **Execution Plan Analysis:** The execution plan generated using `EXPLAIN` reveals inefficiencies primarily due to the lack of indexes.

```
Query  Query History
1  EXPLAIN ANALYZE
2  SELECT c.category_name, SUM(oi.quantity * oi.price) AS total_sales
3  FROM order_items oi
4  JOIN products p ON oi.product_id = p.product_id
5  JOIN categories c ON p.category_id = c.category_id
6  GROUP BY c.category_name
7  ORDER BY total_sales DESC;
```

4) **Analysis of the Execution Plan (Before Indexing):**

	QUERY PLAN text
1	Sort (cost=114.63..115.13 rows=200 width=38) (actual time=3.626..3.643 rows=196 loops=1)
2	Sort Key: (sum(((oi.quantity)::numeric * oi.price))) DESC
3	Sort Method: quicksort Memory: 32kB
4	-> HashAggregate (cost=104.48..106.98 rows=200 width=38) (actual time=3.432..3.524 rows=196 loops=1)
5	Group Key: c.category_name
6	Batches: 1 Memory Usage: 160kB
7	-> Hash Join (cost=41.00..84.54 rows=1994 width=16) (actual time=0.519..2.063 rows=1994 loops=1)
8	Hash Cond: (p.category_id = c.category_id)
9	-> Hash Join (cost=34.50..72.70 rows=1994 width=14) (actual time=0.404..1.385 rows=1994 loops=1)
10	Hash Cond: (oi.product_id = p.product_id)
11	-> Seq Scan on order_items oi (cost=0.00..32.94 rows=1994 width=14) (actual time=0.013..0.227 rows=1994 loo...
12	-> Hash (cost=22.00..22.00 rows=1000 width=8) (actual time=0.379..0.379 rows=1000 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 48kB
14	-> Seq Scan on products p (cost=0.00..22.00 rows=1000 width=8) (actual time=0.013..0.222 rows=1000 loop...
15	-> Hash (cost=4.00..4.00 rows=200 width=10) (actual time=0.100..0.100 rows=200 loops=1)
16	Buckets: 1024 Batches: 1 Memory Usage: 17kB
17	-> Seq Scan on categories c (cost=0.00..4.00 rows=200 width=10) (actual time=0.020..0.054 rows=200 loops=1)
18	Planning Time: 4.856 ms
19	Execution Time: 3.775 ms

Fig. 28. Execution plan before indexing

Observations:

- **Sequential Scans:** Full table scans on `products`, `order_items`, and `categories`, with low costs (e.g., 0.00–22.00) and execution times (e.g., 0.013–0.227 ms). These scale poorly for larger datasets.
- **Hash Joins:** Efficient for small datasets but may degrade with larger ones due to memory limits or increased hash table creation time. Cost ranges from 34.50–84.54 with execution times of 0.404–2.063 ms.
- **Sorting Operation:** Sorting by `SUM(oi.quantity * oi.price)` incurs a cost of 114.63–115.13 and execution time of 3.626–3.643 ms, becoming a bottleneck for large aggregated datasets.
- **Memory Usage:** Minimal memory usage (e.g., 32 kB for quicksort and 160 kB for hash aggregates) indicates suitability for the current small dataset but poor scalability.
- **Key Issues:** Lack of indexing results in inefficiencies in scanning, joining, aggregating, and sorting.

5) **Proposed Solution:**
1. Indexing Join Columns: Creating indexes on the join columns improves join performance by enabling faster matching:

```
Query  Query History
1  CREATE INDEX idx_products_category_id ON products(category_id);
2  CREATE INDEX idx_order_items_product_id ON order_items(product_id);
3  CREATE INDEX idx_categories_category_id ON categories(category_id);
```

2. Indexing Aggregated Columns: Creating a composite index supports efficient aggregation for `SUM(oi.quantity * oi.price)` and join operations:

```
Query  Query History
1  CREATE INDEX idx_order_items_quantity_price ON order_items(product_id, quantity, price);
```

After creating these indexes, re-run `EXPLAIN` to evaluate performance improvements.

6) **Key Changes in the New Execution Plan(After Indexing**

Data Output Messages Notifications SQL

	QUERY PLAN text
1	Sort (cost=114.63..115.13 rows=200 width=38) (actual time=2.327..2.337 rows=196 loops=1)
2	Sort Key: (sum(((oi.quantity)::numeric * oi.price))) DESC
3	Sort Method: quicksort Memory: 32kB
4	-> HashAggregate (cost=104.48..106.98 rows=200 width=38) (actual time=2.222..2.271 rows=196 loops=1)
5	Group Key: c.category_name
6	Batches: 1 Memory Usage: 160kB
7	-> Hash Join (cost=41.00..84.54 rows=1994 width=16) (actual time=0.312..1.313 rows=1994 loops=1)
8	Hash Cond: (p.category_id = c.category_id)
9	-> Hash Join (cost=34.50..72.70 rows=1994 width=14) (actual time=0.248..0.892 rows=1994 loops=1)
10	Hash Cond: (oi.product_id = p.product_id)
11	-> Seq Scan on order_items oi (cost=0.00..32.94 rows=1994 width=14) (actual time=0.007..0.141 rows=1994 loo...
12	-> Hash (cost=22.00..22.00 rows=1000 width=8) (actual time=0.231..0.232 rows=1000 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 48kB
14	-> Seq Scan on products p (cost=0.00..22.00 rows=1000 width=8) (actual time=0.009..0.143 rows=1000 loop...
15	-> Hash (cost=4.00..4.00 rows=200 width=10) (actual time=0.059..0.059 rows=200 loops=1)
16	Buckets: 1024 Batches: 1 Memory Usage: 17kB
17	-> Seq Scan on categories c (cost=0.00..4.00 rows=200 width=10) (actual time=0.012..0.032 rows=200 loops=1)
18	Planning Time: 3.590 ms
19	Execution Time: 2.437 ms

Fig. 29. Execution plan after indexing

Since our dataset is sparse, we don't see any improvements in performance and thus this is not a problematic query for now, but as the dataset grows below are the common scenarios Where Queries Could Lead to Poor Performance .

7) **Common Scenarios Leading to Poor Performance:**

- **Larger Tables:** The order_items, products, and categories tables grow significantly (e.g., order_items has millions of rows)..
- **Frequent Aggregations:** Queries similar to the one we have will run often to calculate sales totals by category.

- **Joins Over Large Tables:** The joins between order_items, products, and categories will require scanning large portions of the tables.
- **Sorting and Grouping:** The GROUP BY and ORDER BY clauses will need significant memory and time, especially without indexes.
- **I/O Bottlenecks:** Sequential scans of large tables result in high I/O costs, degrading query performance.

8) **Performance Improvement:**
   - **Indexing Strategy:** The proposed indexes ensure efficient join operations and aggregations, significantly enhancing scalability.

Query 3

1) **Subquery to Find Products Not Reviewed**

Query | Query History

```
1  SELECT name
2  FROM products
3  WHERE product_id NOT IN (SELECT
4  DISTINCT product_id FROM reviews);
```

2) **Performance Issue**
   - **Subquery with NOT IN:** The `NOT IN` condition requires evaluating the subquery (`SELECT DISTINCT product_id FROM reviews`) for each row in `products`, which is slow with large datasets.
   - **Subquery Evaluation:** The subquery must scan the `reviews` table to identify distinct `product_id` values, making the query inefficient.
   - **Lack of Indexes:** Without an index on `reviews.product_id`, PostgreSQL must scan the entire `reviews` table to find distinct product IDs, slowing down the subquery.
   - **Table Size Impact:** As `products` and `reviews` grow in size, the performance of the subquery becomes worse, especially with the `NOT IN` operation.
   - **Optimization:** Adding an index on `reviews.product_id` can speed up the subquery by enabling faster lookups, reducing the overall query time.

3) **Execution Plan Analysis** The execution plan generated using `EXPLAIN` reveals inefficiencies primarily due to the lack of indexes

Query | Query History

```
1  EXPLAIN ANALYZE
2  SELECT name
3  FROM products
4  WHERE product_id NOT IN (SELECT
5  DISTINCT product_id FROM reviews);
```

4) **Analysis of the Execution Plan (Before Indexing)**

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on products (cost=46.93..71.43 rows=500 width=6) (actual time=0.661..0.854 rows=207 loops=1)
2	Filter: (NOT (hashed SubPlan 1))
3	Rows Removed by Filter: 793
4	SubPlan 1
5	-> HashAggregate (cost=37.01..44.94 rows=793 width=4) (actual time=0.472..0.542 rows=793 loops=1)
6	Group Key: reviews.product_id
7	Batches: 1 Memory Usage: 73kB
8	-> Seq Scan on reviews (cost=0.00..33.21 rows=1521 width=4) (actual time=0.010..0.147 rows=1521 loop...
9	Planning Time: 2.182 ms
10	Execution Time: 0.905 ms

Fig. 30. Execution plan before indexing

- **Sequential Scans:** Both `products` and `reviews` tables use sequential scans with costs of 46.93-71.43 and 0.00-33.21, making them inefficient for larger datasets.
- **Filter Operation:** The `NOT IN` filter leads to nested loop evaluations, increasing costs and execution time without indexing on `product_id`.
- **Subquery:** A HashAggregate fetches distinct `product_id` values from `reviews`, relying on sequential scans and lacking indexing, which slows performance as data grows.Costs range from 37.01–44.94, with execution times of 0.472–0.542 ms.
- **Rows Filtered:** 793 rows were removed unnecessarily due to inefficient filtering caused by the lack of indexes.
- **Execution Time:** Currently at 0.905 ms, but expected to degrade significantly with larger datasets.
- **Key Observations:** Sequential scans, lack of indexing, and the expensive `NOT IN` subquery create scalability and performance bottlenecks.

5) **Proposed Solution**

Query | Query History

```
1  CREATE INDEX idx_reviews_product_id
2  ON reviews(product_id);
```

6) **Key Changes in the New Execution Plan(After Indexing**

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on products (cost=52.88..77.38 rows=500 width=6) (actual time=1.334..1.657 rows=207 loops=1)
2	Filter: (NOT (hashed SubPlan 1))
3	Rows Removed by Filter: 793
4	SubPlan 1
5	-> Unique (cost=0.28..50.90 rows=793 width=4) (actual time=0.548..1.052 rows=793 loops=1)
6	-> Index Only Scan using idx_reviews_product_id on reviews (cost=0.28..47.09 rows=1521 width=4) (actual time=0.546..0.820 rows=1521 loop...
7	Heap Fetches: 0
8	Planning Time: 2.440 ms
9	Execution Time: 1.710 ms

Fig. 31. Execution plan after indexing

- **Sequential Scan on Products:**
  - The query performs a sequential scan on the `products` table with a cost range of `52.88..77.38` and an actual execution time of `1.334..1.657 ms`.
  - A total of 207 rows are processed in this scan, but 793 rows are removed by the filter condition, indicating inefficiency in data retrieval.
- **Filter Application:**
  - A filter using `NOT hashed SubPlan 1` is applied to exclude rows. This operation eliminated 793 rows, suggesting that a significant portion of the dataset was deemed irrelevant after the initial scan.
- **SubPlan 1 - Hashing and Uniqueness:**
  - The subplan identifies unique product IDs from the `reviews` table. It has a cost range of `0.28..50.90` with an actual execution time of `0.548..1.052 ms`.
  - A total of 793 unique rows are processed.
- **Index Only Scan:**
  - An index-only scan is performed on the `reviews` table using the `idx_reviews_product_id` index. This operation has a cost range of `0.28..47.09` and an actual execution time of `0.546..0.820 ms`.
  - A total of 1521 rows are retrieved efficiently without heap fetches, indicating effective use of indexing.
- **Heap Fetches:**
  - No heap fetches were performed, meaning that the required data was entirely available in the index, reducing I/O operations and improving query performance.
- **Planning and Execution Time:**
  - The query planning time is `2.440 ms`, and the execution time is `1.710 ms`. These timings are acceptable for the current dataset size but may not scale well for larger datasets.

XI. Tableau Visualization

This report analyzes the online shopping system's performance from December 31, 2023, to September 29, 2024. The analysis covers trends in order volumes, payment methods, revenue generation, and regional contributions. The findings offer insights into customer behavior and highlight opportunities for improvement.

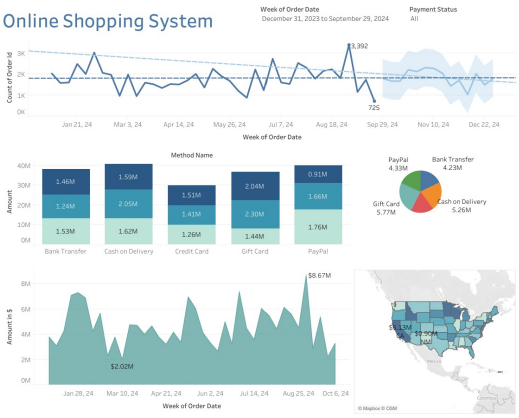


Fig. 32. Visualizations of Online Shopping System Metrics

A. Key Insights

1) Order Trends Over Time: The line chart at the top visualizes the weekly order counts across the analyzed period. Important insights include the following:

- Early 2024 experienced significant fluctuations, with a peak of 3027 orders in early February.
- The highest order count occurred in early-September 2024, reaching 3,392 orders, possibly reflecting seasonal or promotional activities.
- It comes back down to 725 orders, which may indicate lower demand or system problems.
- The declining trend line suggests a drop in customer engagement or operational issues.
- Also the light blue line shows the forecast of number of orders in future weeks which can be used for making decisions.

2) Payment Method Analysis: The payment method breakdown (Pie chart) reveals:

- Gift Cards contributed $5.77M, the highest among all methods.
- Cash on Delivery ($5.26M) and PayPal ($4.33M) follow as preferred methods.
- Bank transfers, generating $4.23M, are the least utilized option.

3) Revenue Trends Over Time: Weekly revenue trends (Area chart at bottom left) highlight:

- Revenue fluctuations between $2M and $6M, with peaks reaching $8.67M in early September which aligns with first line graph where orders were more in early September.

- Sharp declines in March and September align with drops in order counts.
- Spikes in revenue suggest periodic promotions or holidays driving sales.

4) Geographical Distribution: The geographical analysis (Map chart) shows:

- California leads with $5.13M in revenue, followed by Texas and New York.
- States like New Mexico ($0.90M) exhibit low engagement, offering potential for expansion.

B. Recommendations

- Address declining order trends by evaluating customer satisfaction and market dynamics.
- Promote high-performing payment methods and incentivize underutilized ones.
- Conduct targeted promotions during peak seasons to sustain revenue growth.
- Expand market presence in low-performing states through localized marketing strategies.

C. Interactivity

Advanced data filtering has been enabled on the dashboard for increased data exploration and interactivity. Dashboard has filters for selecting a specific range of weeks or statuses of payments pending, completed, and failed, which enables the users to focus on specific subsets of data. In addition, selecting one payment method from one chart dynamically updates all other visualizations that allow consistency, so further analysis of trends, revenues, and regional performances could be made based on the chosen filter.

Below is the dashboard when the data is filtered to show the week of the year with the most orders, together with only the completed payment transactions. This selection emphasizes the exact trends and metrics for that peak week and shows how well the completed orders performed during the most active period.

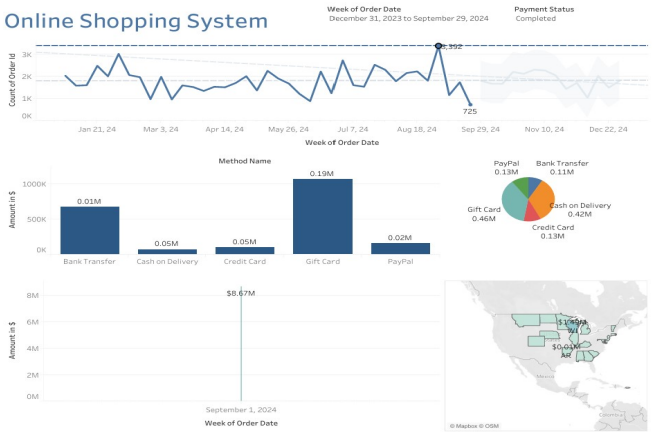


Fig. 33. Visualizations of Online Shopping System Metrics with filters

XII. Conclusion

This project successfully develops a robust and scalable database management system designed for e-commerce, addressing critical challenges like data integrity, scalability, and security. By using techniques like normalization, indexing, and triggers, the database performs well even as the data grows.The use of SQL queries and procedures makes tasks like tracking inventory, managing orders, and processing payments smooth and efficient. Additionally, Tableau visualizations provide valuable insights into customer behavior, revenue trends, and regional performance. Overall, the system improves efficiency, supports business growth, and enhances the shopping experience for customers.

Evaluation Criteria	Mounika Pasupuleti Team Member 1	Sahithya Arveti Nagaraju Team Member 2	Sushmitha Manjunatha Team Member 3
How effectively did your group mate work with you?	5	5	5
Contribution in writing the report	5	5	5
Demonstrates a cooperative and supportive attitude	5	5	5
Contributes significantly to the success of the project	5	5	5
Total	**20**	**20**	**20**

Fig. 34. Group Evaluation Table

References

[1] Faker Documentation https://faker.readthedocs.io/en/master/

[2] Online Shopping and Customer Relationship Management System By S.M. Mostafa Eakram, Sheikh Sharfuddin Mim And Md. Mynul Ahsan, January 2013.

[3] Background Study of online Shopping, May 18, 2018, https://www.scribd.com/doc/31988584/Background-to-the-Study

[4] Categories of e-commerce models, May 18, 2018, https://www.e- commercetutorialspointspdf.com