Distributed HDMR (Team name)
Dr. Bharatesh Chakravarthi
CSE512
27 November 2023

Part 1: Design and Implementation of a Distributed Database System

1. Introduction

This project "EcoSphere: Distributed Commerce Gateway" is a collaborative effort to design, implement, and optimize a distributed database system tailored for the E-commerce domain. It addresses inherent challenges of ensuring data consistency, availability, and performance across distributed nodes. The plan of action is divided into six parts, ranging from the design and implementation of a comprehensive database system to exploring fragmentation and replication techniques, query processing and optimization, distributed transaction management, and the implementation of a NoSQL database system using MongoDB. The ultimate goal is to create a robust and efficient distributed database solution that caters to the complex requirements of an online retail environment. This document presents a report on Part 1 of the project.

2. Implementation Tools and Languages

Tools used: VS code, pgAdmin, Git

Database systems used: PostgreSQL

Languages: Python, SQL

3. Design and Implementation of a Distributed Database System

3.1. Distributed Database Schema:

In the proposed E-commerce distributed database, the key tables include Users, Products, Categories, Orders, OrderDetails, Transactions, Reviews, Inventory, and Shipping, each serving a distinct purpose. While creating the schema, the normalization techniques (1NF, 2NF, 3NF) were taken into consideration. The Users table stores registered user information, while Products and Categories manage product details and classifications of those products ("Electronics", "Clothing", "Books", "Home & Garden", "Sports & Outdoors") respectively. The Orders and OrderDetails tables track user orders and their contents. Transactions record payment details for each order. The Reviews table captures user feedback on products, storing their ratings as well as review text. Inventory table manages stock levels and locations, with information for the warehouse location. Finally the shipping monitors the delivery process for orders, the delivery dates and the status ("Shipped", "Processing", "Delivered"). The details of the column names, Primary and Foreign keys of the tables can be viewed from the ER diagram in Fig 1, as well as from the Queries screenshots in the next section. Relationships among these tables, such as the one-to-many connection between Users and Orders or between Products and Reviews, facilitate the representation of complex interactions and dependencies inherent to the E-commerce domain, providing a comprehensive view of the online retail environment. Following is the Entity-Relationship diagram corresponding to the above schema -

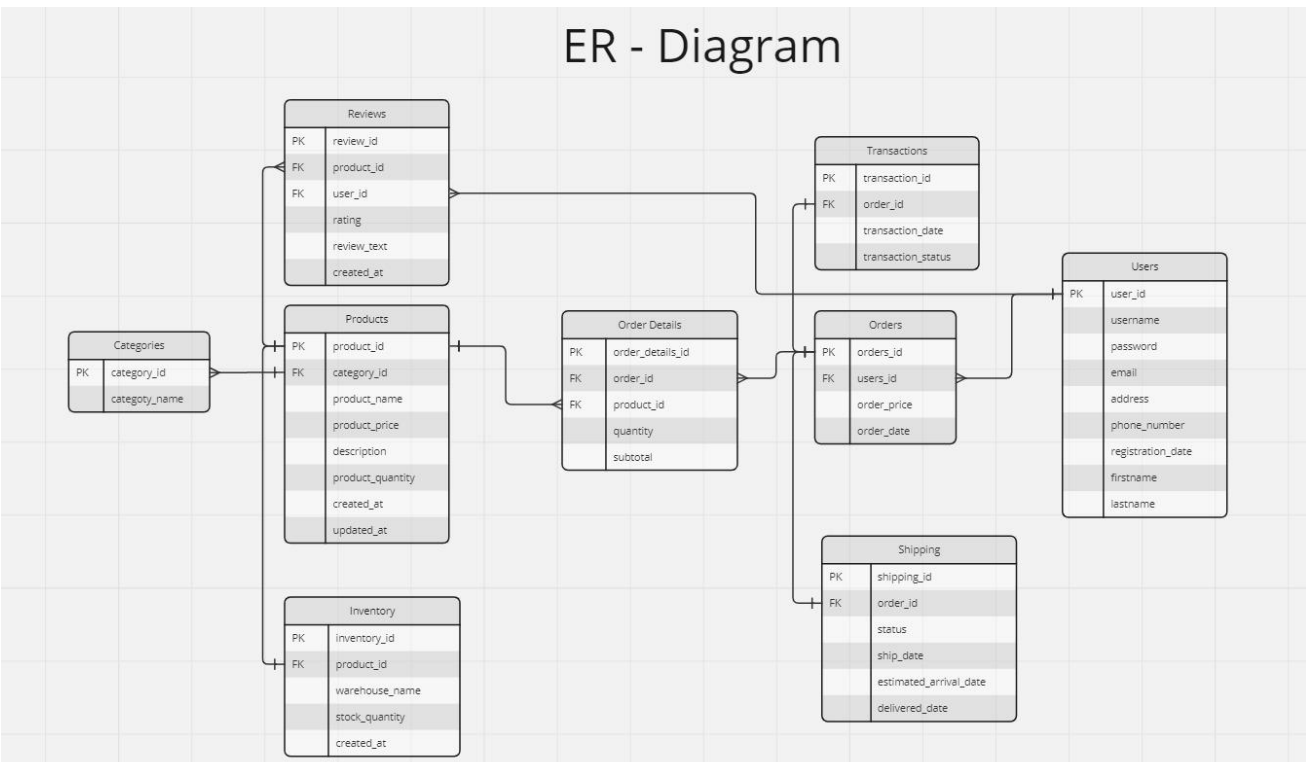


Fig 1: Entity-Relationship diagram for EcoSphere: Distributed Commerce Gateway

3.2. Database Tables:

We have used the postgres server to create our database named “e-commerce”, as shown in the code and the screenshot of the database from pgAdmin software as shown below -

```python
def connect_postgres(dbname):
    """Connect to the PostgreSQL using psycopg2 with default database
       Return the connection"""
    return psycopg2.connect(host=POSTGRES_CONFIG['HOST_NAME'], dbname=dbname, user=POSTGRES_CONFIG['USER_NAME'], password=POSTGRES_CONFIG['PASSWORD'], port=POSTGRES_CONFIG['PORT'])


def create_database(dbname):
    """Connect to the PostgreSQL by calling connect_postgres() function
       Create a database named 'dbname' passed in argument
       Close the connection"""
    conn = None
    cur = None
    try:
        conn = connect_postgres('postgres')
        conn.autocommit = True
        cur = conn.cursor()

        # Check if the database already exists
        cur.execute("SELECT 1 FROM pg_database WHERE datname = %s", (dbname,))
        if cur.fetchone():
            print(f"Database '{dbname}' already exists.")
        else:
            # Create the new database
            cur.execute(f"CREATE DATABASE {dbname}")
            print(f"Database '{dbname}' created successfully.")

    except Exception as error:
        print(error)

    finally:
        if cur is not None:
            cur.close()
        if conn is not None:
            conn.close()
```

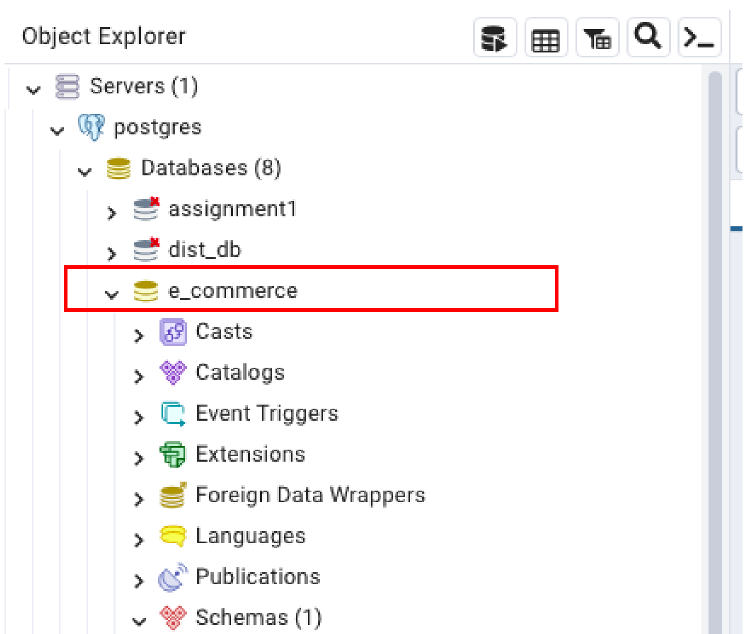


Fig 2: Code showing database creation (above) and the screenshot of database e-commerce created in pgAdmin (below)

There are a total of 9 tables created for the given distributed database schema. The details of the columns, their data types, primary and foreign keys can be gathered from the screenshots of the queries as shown below -

final_project.py 3, M   constants.py

Part 1 > constants.py > ...

```
# Tables Names
TABLE_NAMES = ["Users", "Categories", "Products", "Orders", "OrderDetails", "Transactions", "Reviews", "Inventory",
"Shipping"]

USERS_QUERY = f"""
            CREATE TABLE IF NOT EXISTS {TABLE_NAMES[0]} (
    user_id SERIAL,
    username TEXT NOT NULL,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    email TEXT NOT NULL,
    password TEXT NOT NULL,
    address TEXT,
    phone_number TEXT,
    registration_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, registration_date)
)
PARTITION BY RANGE (registration_date);
"""

CATEGORIES_QUERY = f"""
            CREATE TABLE IF NOT EXISTS {TABLE_NAMES[1]} (
    category_id SERIAL PRIMARY KEY,
    category_name TEXT NOT NULL
);
"""

PRODUCTS_QUERY = f"""
            CREATE TABLE IF NOT EXISTS {TABLE_NAMES[2]} (
    product_id SERIAL PRIMARY KEY,
    product_name TEXT NOT NULL,
    description TEXT,
    product_price DECIMAL NOT NULL,
    category_id INT REFERENCES Categories(category_id),
    product_quantity INT DEFAULT 0,
    created_at date DEFAULT CURRENT_DATE,
    updated_at date DEFAULT CURRENT_DATE
);
"""
```

```
ORDERS_QUERY =  f"""
            CREATE TABLE IF NOT EXISTS {TABLE_NAMES[3]} (
    order_id SERIAL PRIMARY KEY,
    user_id INT,
    user_registration_date TIMESTAMP,
    order_price DECIMAL NOT NULL,
    order_date date DEFAULT CURRENT_DATE,
    FOREIGN KEY (user_id, user_registration_date) REFERENCES Users(user_id, registration_date)
);
"""

ORDER_DETAILS_QUERY = f"""
            CREATE TABLE IF NOT EXISTS {TABLE_NAMES[4]} (
    order_detail_id SERIAL PRIMARY KEY,
    order_id INT REFERENCES Orders(order_id),
    product_id INT REFERENCES Products(product_id),
    quantity INT,
    subtotal DECIMAL
);
"""

TRANSACTIONS_QUERY = f"""
            CREATE TABLE IF NOT EXISTS {TABLE_NAMES[5]} (
    transaction_id SERIAL PRIMARY KEY,
    order_id INT REFERENCES Orders(order_id),
    transaction_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    transaction_status TEXT
);
"""
```

```
REVIEWS_QUERY = f"""
            CREATE TABLE IF NOT EXISTS {TABLE_NAMES[6]} (
    review_id SERIAL PRIMARY KEY,
    product_id INT REFERENCES Products(product_id),
    user_id INT,
    user_registration_date TIMESTAMP,
    rating INT CHECK (rating >= 1 AND rating <= 5),
    review_text TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id, user_registration_date) REFERENCES Users(user_id, registration_date)
);
"""

INVENTORY_QUERY = f"""
            CREATE TABLE IF NOT EXISTS {TABLE_NAMES[7]} (
    inventory_id SERIAL PRIMARY KEY,
    product_id INT REFERENCES Products(product_id),
    warehouse_name TEXT,
    stock_quantity INT DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
"""

SHIPPING_QUERY = f"""
            CREATE TABLE IF NOT EXISTS {TABLE_NAMES[8]} (
    shipping_id SERIAL PRIMARY KEY,
    order_id INT REFERENCES Orders(order_id),
    status TEXT,
    ship_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    estimated_arrival_date TIMESTAMP,
    delivered_date TIMESTAMP
);
"""
```

Fig 3: Screenshots for the SQL queries for 9 tables creation final_project.py 3, M × constants.py

Part 1 > final_project.py > ...

```python

def execute_query(conn, query):
    """
    Executes a given SQL query using the provided database connection.

    :param conn: Database connection object
    :param query: SQL query string to be executed
    """
    try:
        # Create a new cursor
        with conn.cursor() as cur:
            # Execute the query
            cur.execute(query)
            # Print success message
            print("Query executed successfully")
            # Commit the transaction
            conn.commit()
    except (Exception, psycopg2.Error) as error:
        # Rollback in case of error
        conn.rollback()
        print("Error executing query:", error)
        raise

def create_tables(conn):
    try:
        execute_query(conn, CREATE_QUERIES[0])
        printStatements("Succesfully created User Table")
        execute_query(conn, CREATE_QUERIES[1])
        printStatements("Succesfully created Categories Table")
        execute_query(conn, CREATE_QUERIES[2])
        printStatements("Succesfully created Products Table")
        execute_query(conn, CREATE_QUERIES[3])
        printStatements("Succesfully created Orders Table")
        execute_query(conn, CREATE_QUERIES[4])
        printStatements("Succesfully created Order Details Table")
        execute_query(conn, CREATE_QUERIES[5])
        printStatements("Succesfully created Transactions Table")
        execute_query(conn, CREATE_QUERIES[6])
        printStatements("Succesfully created Reviews Table")
        execute_query(conn, CREATE_QUERIES[7])
        printStatements("Succesfully created Inventory Table")
        execute_query(conn, CREATE_QUERIES[8])
        printStatements("Succesfully created Shipping Table")
        create_partition_table_users(conn)
    except (Exception, psycopg2.Error) as error:
        print("Error creating tables:", error)
```

Fig 4: Screenshot for Python code for table creation

Following is a screenshot of the tables that are successfully created in the database. As an example, a query is run in pgAdmin, fetching the details for User table, and we can see that the table is empty for now, with the schema as per the queries described above.

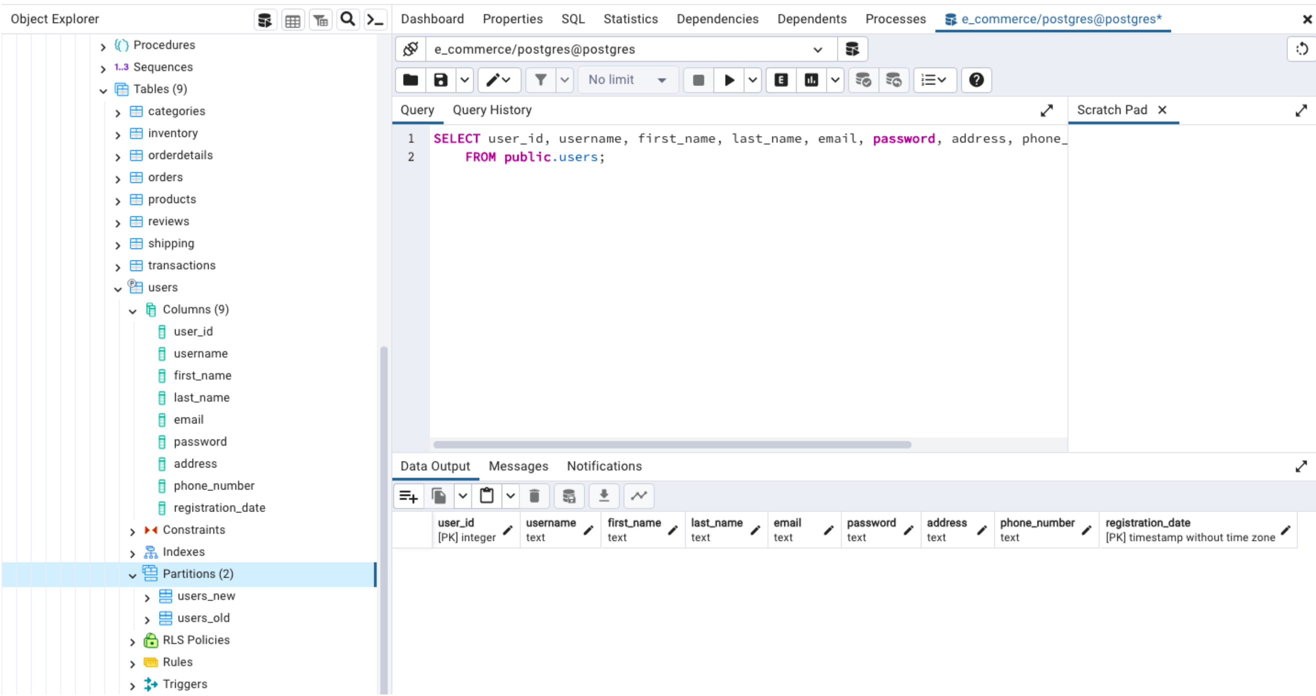


Fig 5: Screenshot of pgAdmin showing the tables created with the partitions and empty data

3.3. Data Distribution Plan:

After thorough analysis and consideration, the chosen data distribution strategy for this project centers around a partitioned and replicated database model. This approach involves fragmenting data into distinct partitions, or shards, each managed independently. These partitions can be distributed across multiple servers or nodes, allowing for load balancing and reducing the risk of single points of failure. The partitioning logic takes into account transaction volume, data access patterns, and geographical distribution of users. The details of partitioning will be discussed in the Part 2 report. To further enhance data availability and fault tolerance, a replication mechanism is also integrated. The data is replicated across another node and it is scalable so as to allow replication across multiple nodes, ensuring that in the event of a server failure, a backup is available to maintain uninterrupted service. For large amounts of data, replication also can aid in load balancing, particularly for read-heavy operations, by allowing read requests to be served by multiple nodes. The combined use of partitioning and replication in our data distribution plan aims to provide a robust, scalable solution, catering to the demands of high-volume, distributed applications while maintaining data integrity and consistency.

3.4. Data Insertion Mechanism:

For testing purposes, the project uses *Faker* library in python to generate and insert mock data into the database. This approach ensures a streamlined and efficient mechanism for populating the database with realistic data, taking into account the respective data types, for various testing scenarios that we have tested in later parts of the project. The insertion process is automated, with scripts designed to batch-insert this mock data, ensuring both time efficiency and consistency in test data creation. The following are the screenshots for the batch data insertion code -

Part 1 > final_project.py > insert_mock_data

```python
def insert_mock_data(conn):
    fake = Faker()
    try:
        cursor = conn.cursor()
        if conn and cursor:
            for _ in range(50):
                username = fake.user_name()
                first_name = fake.first_name()
                last_name = fake.last_name()
                email = fake.email()
                password = fake.password()
                address = fake.address()
                phone_number = fake.phone_number()
                registration_date = fake.date_time_this_decade()

                cursor.execute(f"""
                    INSERT INTO {TABLE_NAMES[0]} (username, first_name, last_name, email, password, address, phone_number, registration_date)
                    VALUES (%s, %s, %s, %s, %s, %s, %s, %s)
                """, (username, first_name, last_name, email, password, address, phone_number, registration_date))


            categories = ["Electronics", "Clothing", "Books", "Home & Garden", "Sports & Outdoors"]
            for category_name in categories:
                cursor.execute(f"""
                INSERT INTO {TABLE_NAMES[1]} (category_name) VALUES (%s)
                """, (category_name,))
```

```python
            cat_ids = []
            cursor.execute("SELECT category_id FROM Categories")
            cat_ids = [row[0] for row in cursor.fetchall()]
            for _ in range(50):
                product_name = fake.word().capitalize()
                description = fake.text(max_nb_chars=100)
                product_price = round(random.uniform(5, 200), 2)
                category_id = random.choice(cat_ids)
                product_quantity = random.randint(0, 100)
                cursor.execute(f"""
                    INSERT INTO {TABLE_NAMES[2]} (product_name, description, product_price, category_id, product_quantity)
                    VALUES (%s, %s, %s, %s, %s)
                """, (product_name, description, product_price, category_id, product_quantity))


            usrs = []
            cursor.execute("SELECT user_id, registration_date FROM Users")
            usrs = cursor.fetchall()

            for _ in range(50):
                user_id, registration_date = random.choice(usrs)
                order_price = round(random.uniform(10, 1000), 2)
                order_date = fake.date_between(start_date=registration_date, end_date="today")
                cursor.execute(f"""
                    INSERT INTO {TABLE_NAMES[3]} (user_id, user_registration_date, order_price, order_date)
                    VALUES (%s, %s, %s, %s)
                """, (user_id, registration_date, order_price, order_date))
```

```
        ord_ids = []
        cursor.execute("SELECT order_id FROM Orders")
        ord_ids = [row[0] for row in cursor.fetchall()]
        prod_ids = []
        cursor.execute("SELECT product_id FROM Products")
        prod_ids = [row[0] for row in cursor.fetchall()]
        for _ in range(50):
            order_id = random.choice(ord_ids)
            product_id = random.choice(prod_ids)
            quantity = random.randint(1, 10)
            subtotal = round(random.uniform(10, 100), 2)
            cursor.execute(f"""
                INSERT INTO {TABLE_NAMES[4]} (order_id, product_id, quantity, subtotal)
                VALUES (%s, %s, %s, %s)
            """, (order_id, product_id, quantity, subtotal))


        for _ in range(50):
            order_id = random.choice(ord_ids)
            transaction_date = fake.date_time_between(start_date="-1y", end_date="now")
            transaction_status = random.choice(["Success", "Pending", "Failed"])
            cursor.execute(f"""
                INSERT INTO {TABLE_NAMES[5]} (order_id, transaction_date, transaction_status)
                VALUES (%s, %s, %s)
            """, (order_id, transaction_date, transaction_status))
```

```
        for _ in range(50):
            product_id = random.choice(prod_ids)
            user_id, user_registration_date = random.choice(usrs)
            rating = random.randint(1, 5)
            review_text = fake.paragraph()
            created_at = fake.date_time_between(start_date="-365d", end_date="now")

            cursor.execute(f"""
                INSERT INTO {TABLE_NAMES[6]} (product_id, user_id, user_registration_date, rating, review_text,
                created_at)
                VALUES (%s, %s, %s, %s, %s, %s)
            """, (product_id, user_id, user_registration_date, rating, review_text, created_at))



        for _ in range(50):
            product_id = random.choice(prod_ids)
            warehouse_name = fake.company()
            stock_quantity = random.randint(0, 1000)
            created_at = fake.date_time_between(start_date="-30d", end_date="now")
            cursor.execute(f"""
                INSERT INTO {TABLE_NAMES[7]} (product_id, warehouse_name, stock_quantity, created_at)
                VALUES (%s, %s, %s, %s)
            """, (product_id, warehouse_name, stock_quantity, created_at))
```

```python
            for _ in range(1, 51):
                order_id = random.choice(ord_ids)
                status = random.choice(["Shipped", "Processing", "Delivered"])
                ship_date = fake.date_time_between(start_date="-30d", end_date="now")
                estimated_arrival_date = ship_date + timedelta(days=random.randint(1, 10))
                delivered_date = estimated_arrival_date + timedelta(days=random.randint(1, 5)) if status == "Delivered" else
                None

                cursor.execute(f"""
                    INSERT INTO {TABLE_NAMES[8]} (order_id, status, ship_date, estimated_arrival_date, delivered_date)
                    VALUES (%s, %s, %s, %s, %s)
                """, (order_id, status, ship_date, estimated_arrival_date, delivered_date))


            conn.commit()

            print(f"Inserted data!")

    except (Exception, psycopg2.Error) as error:
        print("Error inserting data:", error)
```

Fig 6: Screenshots of the code for data insertion

3.5. Data Retrieval Proof:

The above that has been inserted can be retrieved from the database using select SQL queries as shown below -

final_project.py 3, M × | Git Graph | constants.py | task_5.ipynb

Part 1 > final_project.py > insert_mock_data

```python
def select_data(conn):
    try:
        cursor = conn.cursor()
        if conn and cursor:
            for table_name in TABLE_NAMES:
                cursor.execute(f"SELECT * FROM {table_name};")
                rows = cursor.fetchall()
                headers = [desc[0] for desc in cursor.description]
                print(f"Data from Table: {table_name}")
                print(tabulate(rows, headers=headers, tablefmt="grid"))
                print("\n" + "-" * 50 + "\n")  # Separator between tables

    except (Exception, psycopg2.Error) as error:
        print("Error selecting data:", error)
```

PROBLEMS 3 | OUTPUT | TERMINAL | PORTS | JUPYTER | COMMENTS | DEBUG CONSOLE | zsh

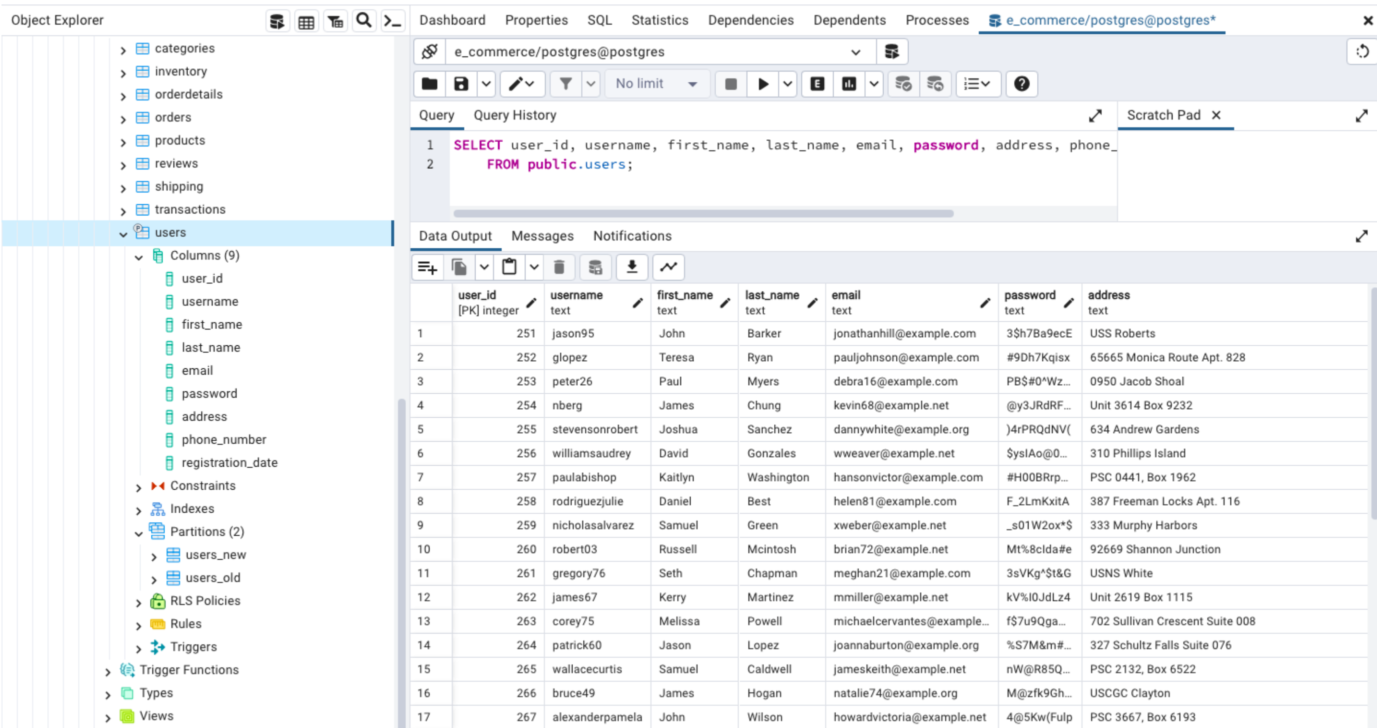
----

Data from Table: Shipping

shipping_id	order_id	status	ship_date	estimated_arrival_date	delivered_date
1	198	Processing	2023-11-23 19:51:45.177006	2023-11-25 19:51:45.177006	
2	169	Shipped	2023-11-13 14:02:23.395444	2023-11-21 14:02:23.395444	
3	174	Processing	2023-11-15 18:01:54.345766	2023-11-22 18:01:54.345766	
4	187	Delivered	2023-11-16 14:09:17.212365	2023-11-26 14:09:17.212365	2023-12-01 14:09:17.212365
5	185	Processing	2023-10-30 17:22:39.278301	2023-11-08 17:22:39.278301	
6	172	Shipped	2023-11-21 12:19:27.782103	2023-12-01 12:19:27.782103	
7	177	Shipped	2023-11-16 19:07:55.598897	2023-11-22 19:07:55.598897	
8	173	Delivered	2023-11-18 10:21:05.610372	2023-11-19 10:21:05.610372	2023-11-21 10:21:05.610372
9	179	Shipped	2023-11-10 22:12:11.412046	2023-11-17 22:12:11.412046	
10	190	Processing	2023-11-17 08:11:12.878829	2023-11-26 08:11:12.878829	
11	181	Delivered	2023-11-23 14:35:52.205136	2023-11-27 14:35:52.205136	2023-12-02 14:35:52.205136
12	168	Processing	2023-11-21 10:18:50.421653	2023-11-29 10:18:50.421653	
13	155	Processing	2023-10-31 17:04:45.705347	2023-11-01 17:04:45.705347	
14	167	Processing	2023-11-11 03:03:09.022673	2023-11-16 03:03:09.022673	

3 0 Git Graph | Ln 151, Col 53 Spaces: 4 UTF-8 LF Python 3.9.6 64-

Fig 7: Screenshot of data retrieval code and result for Shipping table on the console



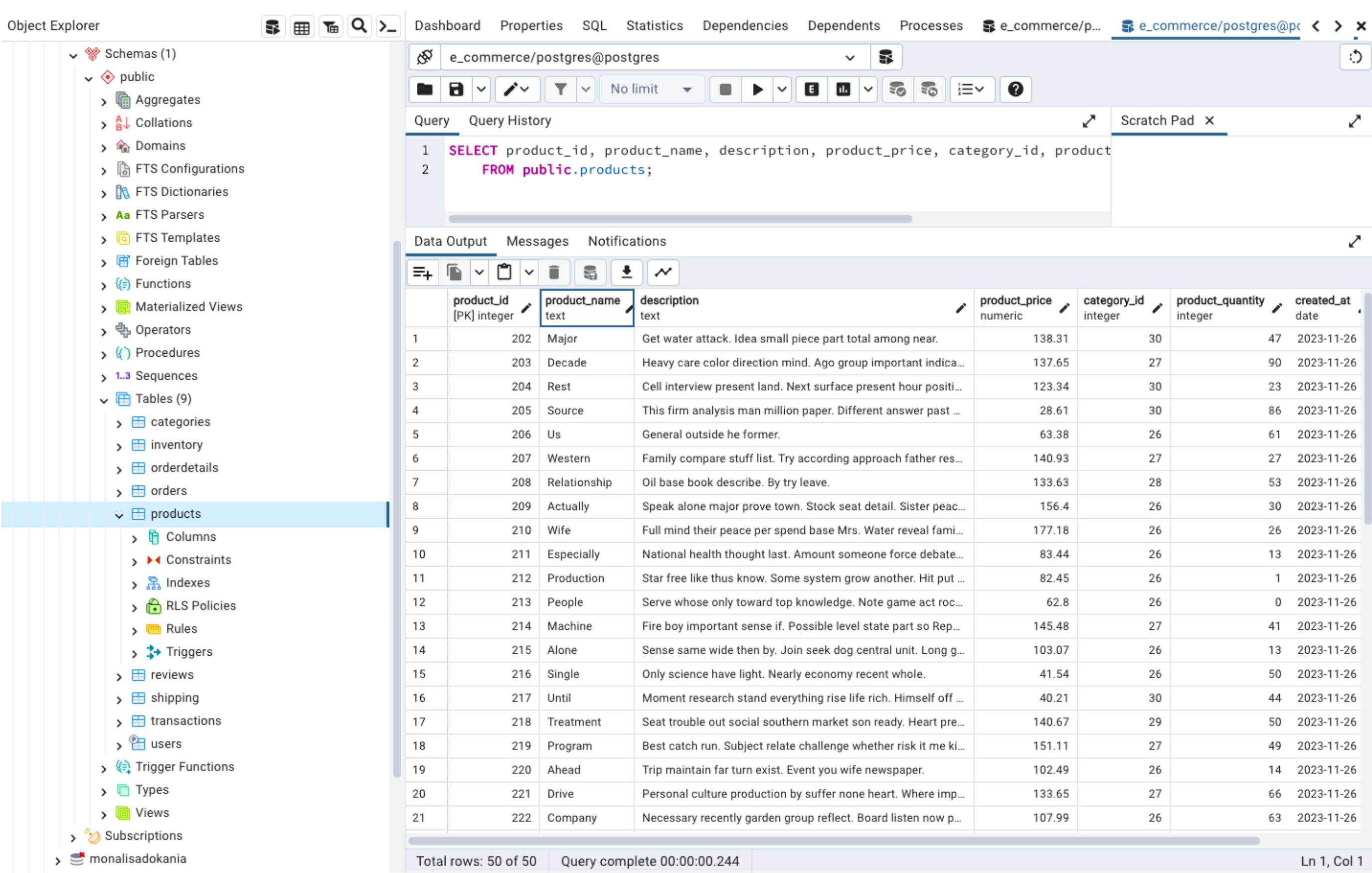


Fig 8: Screenshot for data retrieval proof for Users table (above) and Products table (below)

4. Conclusion

In Part 1 of the project, we successfully designed and implemented a distributed database system tailored to handle real-time data of an e-commerce system. Key deliverables include a distributed database schema with entity-relationship diagram, creation scripts for database tables with appropriate attributes, keys, and constraints, and a robust data distribution plan, an efficient data insertion mechanism, as well as evidence of successful data retrieval.