Project – DataBase – Louis ROY

Part 1: Essay

In the field of database management systems, transactions play a crucial role in ensuring the reliability, consistency, and security of data. A transaction can be defined as a logical unit of work that groups multiple database operations into a single execution. The fundamental principle is that either all operations within the transaction are executed successfully, or none of them are applied at all. This property is essential in real-world applications such as online banking, e-commerce, airline booking systems, and any environment where data integrity must be preserved despite errors, failures, or concurrent access.

A simple but illustrative example is a bank transfer. When a customer transfers money from one account to another, two operations are required: debiting the sender's account and crediting the receiver's account. If the first operation succeeds but the second fails due to a system crash, the financial institution would face serious inconsistencies. To prevent such issues, databases implement transactions to guarantee that either both operations are committed or both are rolled back.

The concept of transactions is governed by four fundamental properties, commonly referred to as the ACID principles: Atomicity, Consistency, Isolation, and Durability. Atomicity ensures that a transaction is indivisible: it either completes entirely or has no effect. Consistency means that the database moves from one valid state to another, maintaining all defined rules such as constraints and triggers. Isolation guarantees that concurrent transactions do not interfere with each other in a way that produces invalid results. Durability ensures that once a transaction has been committed, its results are permanently recorded, even in the event of a power outage or system crash [1].

In practice, transaction control is managed through specific commands. Most relational database systems, including PostgreSQL, MySQL, and

Oracle, use the keywords BEGIN or START TRANSACTION to mark the start of a transaction. At the end, the user can decide to either apply all changes using COMMIT, or undo them using ROLLBACK. For instance, the bank transfer mentioned earlier could be written in SQL as follows:

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

If any of these statements fails, the database ensures that the changes can be reverted, thereby protecting data integrity.

While transactions are essential, they also introduce several challenges, especially when multiple users or applications access the database simultaneously. This is where the notion of concurrency anomalies becomes important. A lost update occurs when two transactions attempt to update the same data at the same time, with one overwriting the other. A dirty read happens when one transaction reads data that has been modified by another transaction that has not yet been committed. Non-repeatable reads occur when a transaction reads the same record twice but gets different results because another transaction has modified it in between. Finally, phantom reads happen when repeated execution of a query returns different sets of rows because new records were inserted by another transaction [2].

To manage these anomalies, database systems implement isolation levels, which define how strictly concurrent transactions are separated. The common levels, as defined by the SQL standard, are Read Uncommitted, Read Committed, Repeatable Read, and Serializable. At the lowest level, transactions may experience anomalies but benefit from higher performance. At the Serializable level, transactions are executed as

if they occurred sequentially, which eliminates anomalies but may reduce performance. Choosing the right isolation level depends on the application's requirements: for example, financial systems typically favor stronger isolation, whereas web applications with heavy traffic may prioritize speed [3].

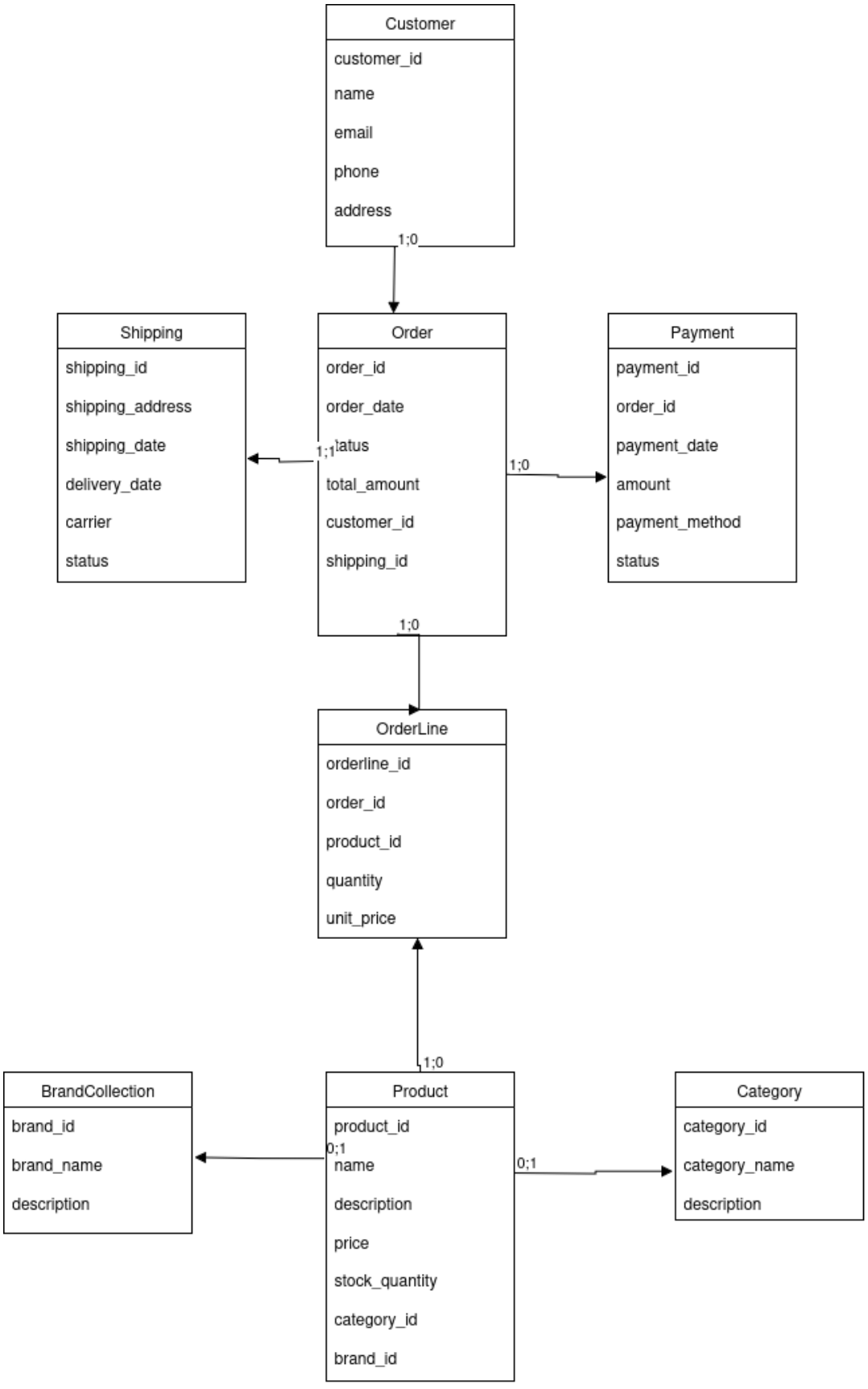
The importance of transactions is evident in many practical contexts. In financial systems, they ensure that money transfers, withdrawals, or payments cannot result in inconsistencies. In e-commerce, transactions are crucial for handling orders, payments, and inventory management to prevent overselling products. In transportation and ticket booking systems, they avoid the double reservation of seats or tickets. Even in online gaming, where players trade items or currencies, transactions guarantee that exchanges are executed fairly and reliably.

In conclusion, transactions are a cornerstone of database management systems. They provide a robust framework for maintaining the correctness and reliability of data despite failures, errors, or concurrent execution. The ACID properties and the mechanisms of isolation levels enable organizations to adapt transaction processing to their needs, striking a balance between accuracy and performance. In a world increasingly dependent on digital data, mastering the concept of transactions remains indispensable for both database professionals and application developers.

Références :

[1] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco: Morgan Kaufmann, 1993.
[2] H. Garcia-Molina, J. Ullman, and J. Widom, *Database Systems: The Complete Book*. Pearson, 2nd ed., 2008.
[3] A. Silberschatz, H. Korth, and S. Sudarshan, *Database System Concepts*. McGraw-Hill, 7th ed., 2019.

Part 2: Database Model Diagram :



This diagram represents an online store management system. Customers can place orders (Orders), and each order can include multiple products through the OrderLine table. Orders may be linked to a shipment (Shipping) and a payment (Payment). Products (Product) belong to a category (Category) and may also be associated with a brand or collection (BrandCollection). This model efficiently manages the relationships between customers, orders, products, shipping, and payments, while ensuring data integrity through well-defined foreign keys.

Part 3: Database Implementation :

Link Github : https://github.com/louisroyesiea/project-database

Database documentation:

Database Structure :

The database consists of **8 main tables** with logical relationships to manage an online store.

Customer : Store customers --> Primary Keys: customer_id

Category : Product categories --> Primary Keys: category_id

BrandCollection : Product brands or collections --> Primary Keys: brandcollection_id

Product : Products sold --> Primary Keys: product_id --> Foreingn Keys : category_id → Category, brandcollection_id → BrandCollection

Shipping : Shipping information --> Primary Keys: shipping_id

Orders : Customer orders --> Primary Keys: order_id --> Foreingn Keys : customer_id → Customer, shipping_id → Shipping

OrderLine: Order lines --> Primary Keys: (order_id, product_id) --> Foreingn Keys : order_id → Orders, product_id → Product

Payment: Payments --> Primary Keys: payment_id --> Foreingn Keys : order_id → Orders

Relationship Notes:

- Orders.customer_id → links an order to the customer who placed it.
- Orders.shipping_id → links an order to its shipment.
- OrderLine uses a composite primary key (order_id, product_id) to allow multiple products per order.
- Product.category_id and Product.brandcollection_id classify products by category and brand.
- Payment.order_id links a payment to its corresponding order.

I have posted the SQL scripts for the tables, queries and data on GitHub.

Implementation notes :

- Auto-increment primary keys are created using SERIAL.
- Foreign keys use ON DELETE CASCADE or ON DELETE SET NULL to maintain referential integrity.
- Table and column names use case-sensitive quotes ("Orders") to avoid conflicts with reserved words.
- The composite primary key in OrderLine allows multiple products per order.
- I'm not sure if you can open the .odb file because of the localhost server or the password: louis.

I used ChatGPT to generate the data for me to save time, as I was having problems with relationships that weren't working in my database. And for the “Customer orders with totals” query, I modified it slightly to better suit my project, particularly for displaying customers, which previously displayed names during calls, but I prefer to call using IDs.

customer orders, searching for products, etc.

Documentation: Provide documentation explaining the structure of your database, the SQL code used, and any additional notes on the implementation. // voici mon projet donne moi les data apres avoir vu mes tables et mon diagram

Parfait 😎, on va générer **les données d'exemple complètes pour ton projet** "OnlineStore", basées sur tes tables et ton diagramme ERD que tu m'as fourni indirectement. Je vais te préparer un script SQL `sample_data.sql` prêt à insérer dans PostgreSQL ou LibreOffice Base.

Je vais m'assurer que **toutes les relations FK** sont respectées.

`sample_data.sql` (données d'exemple)

sql — Copier le code

```sql
-- ========================
-- CUSTOMERS
-- ========================
INSERT INTO "Customer" (name, email, phone, address) VALUES
('Alice Dup', 'alice@example.com', '0123456789', '123 rue A'),
('Bob Martin', 'bob@example.com', '0987654321', '456 rue B'),
('Charlie Noel', 'charlie@example.com', '0112233445', '789 rue C');

-- ========================
-- CATEGORIES
-- ========================
INSERT INTO "Category" (category_name, description) VALUES
('Chaussures', 'Chaussures de sport et loisirs'),
('Accessoires', 'Sacs, ceintures, chapeaux'),
('Vêtements', 'T-shirts, pantalons, vestes');

-- ========================
-- BRAND COLLECTIONS
-- ========================
INSERT INTO "BrandCollection" (brand_name, description) VALUES
('Nike', 'Marque sportive américaine'),
('Adidas', 'Marque allemande célèbre'),
('Puma', 'Marque lifestyle et sport');

-- ========================
-- PRODUCTS
-- ========================
INSERT INTO "Product" (name, description, price, stock_quantity, category_id, brandcollect
('Air Max 90', 'Sneakers confortables', 120.00, 50, 1, 1),
('Adidas Superstar', 'Chaussures classiques', 90.00, 30, 1, 2),
('Nike Backpack', 'Sac à dos sport', 60.00, 20, 2, 1),
('Puma T-Shirt', 'T-shirt sport léger', 25.00, 40, 3, 3),
('Adidas Cap', 'Casquette Adidas', 20.00, 35, 2, 2);

-- ========================
-- SHIPPING
-- ========================
INSERT INTO "Shipping" (shipping_address, shipping_date, delivery_date, carrier, status) \
('123 rue A', '2025-09-16', '2025-09-18', 'DHL', 'Pending'),
('456 rue B', '2025-09-16', '2025-09-19', 'UPS', 'Shipped'),
('789 rue C', '2025-09-17', '2025-09-20', 'FedEx', 'Pending');

-- ========================
-- ORDERS
-- ========================
INSERT INTO "Orders" (order_date, status, total_amount, customer_id, shipping_id) VALUES
('2025-09-16', 'Paid', 240.00, 1, 1),
('2025-09-16', 'Pending', 90.00, 2, 2),
('2025-09-17', 'Paid', 85.00, 3, 3);

-- ========================
-- ORDERLINE
-- ========================
INSERT INTO "OrderLine" (order_id, product_id, quantity, unit_price) VALUES
(1, 1, 2, 120.00),   -- Alice commande 2 Air Max
(2, 2, 1, 90.00),    -- Bob commande 1 Adidas Superstar
(3, 4, 3, 25.00),    -- Charlie commande 3 Puma T-Shirt
```