

Distributed HDMR (Team name)

Dr. Bharatesh Chakravarthi

CSE512

27 November 2023

Part 4: Distributed Transaction Management

1. Introduction

The fourth part of this project, "EcoSphere: Distributed Commerce Gateway", looks into the implementation of **Distributed Transaction Management**. The realm of database management and operation has continuously evolved to meet the growing demands of modern applications, particularly in terms of data availability, consistency, and efficiency. In this context, our project focuses on configuring a robust database system leveraging the power of **PostgreSQL** in conjunction with **Amazon Aurora**, a fully managed relational database engine optimized for cloud efficiency and reliability. This combination aims to create a resilient architecture capable of managing **distributed transactions in an ACID-compliant** manner while effectively handling concurrency, a critical aspect in today's data-intensive applications.

2. Tools and Languages

Tools used: VS code, pgAdmin, Git, AWS

Database systems used: AWS Aurora Postgres cluster

Languages: Python, SQL

3. Database Architecture and Setup

In our quest to build a robust and scalable database system, we have opted for a master-slave architecture using PostgreSQL and Amazon Aurora. This section of the report delves into the specifics of this architecture, detailing the individual roles of PostgreSQL and Amazon Aurora, and the rationale behind their integration.

Overview of Amazon Aurora:

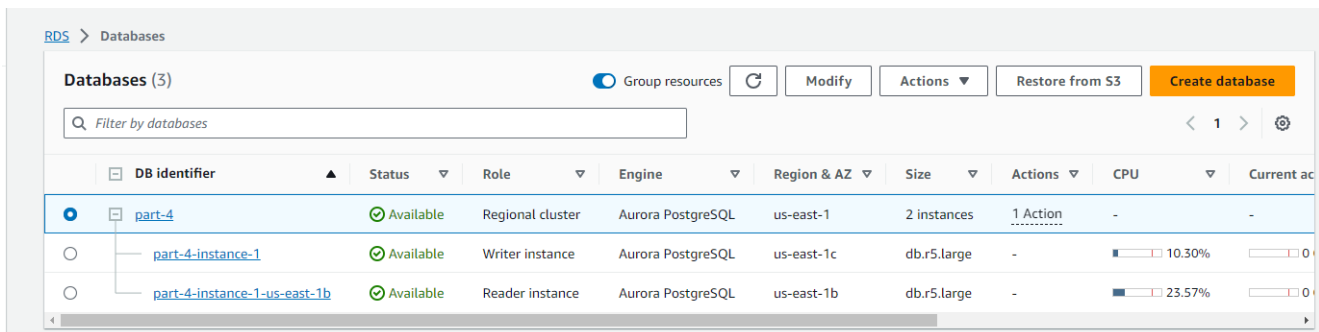
Amazon Aurora is a cloud-based relational database service known for its high performance and scalability. It stands out due to its unique architecture which automatically divides your database

into multiple chunks, distributing them across a fleet of storage resources. Aurora is designed to offer greater than 99.99% availability, replicating six copies of your data across three Availability Zones and continuously backing up your data to Amazon S3. One of Aurora's key features is its compatibility with popular database engines, especially PostgreSQL, allowing users to leverage its benefits without having to modify their existing database applications.

Master-Slave Configuration with PostgreSQL and Aurora:

In our setup, PostgreSQL operates within the Amazon Aurora framework, benefiting from its high-performance and fault-tolerant environment. We have configured one PostgreSQL database as the 'master', handling all the write operations, while the other instance acts as a 'slave', dedicated to read operations. This master-slave configuration is pivotal for enhancing data availability and consistency, as it allows for:

- **High Availability:** In case of a failure in the master database, the slave can be quickly promoted to take over, thus ensuring minimal downtime.
- **Load Balancing:** Read operations are distributed across the slave database, reducing the load on the master and thereby optimizing overall system performance.
- **Data Redundancy:** Continuous replication from the master to the slave ensures data redundancy, safeguarding against data loss.



DB identifier	Status	Role	Engine	Region & AZ	Size	Actions	CPU	Current ac
part-4	Available	Regional cluster	Aurora PostgreSQL	us-east-1	2 instances	1 Action	-	-
part-4-instance-1	Available	Writer instance	Aurora PostgreSQL	us-east-1c	db.r5.large	-	10.30%	0
part-4-instance-1-us-east-1b	Available	Reader instance	Aurora PostgreSQL	us-east-1b	db.r5.large	-	23.57%	0

Fig 1: AWS Aurora cluster

The Role of PostgreSQL:

PostgreSQL, known for its advanced features and ACID compliance, is the core database engine in our setup. Its compatibility with Amazon Aurora allows us to leverage PostgreSQL's

powerful transaction management and concurrency control features in a cloud-based environment. This integration is key to our project as it combines the robustness of PostgreSQL with the scalability and high availability of Amazon Aurora.

4. Implementation

This section of the report details the practical steps taken to implement our Distributed Transaction Management system using PostgreSQL and Amazon Aurora. It covers the entire process, from establishing database connections to managing data through our custom Python script.

Database Connection and Setup:

- **Establishing Connections:**

The ‘`connect_postgres`’ function in our Python script is designed to establish connections with the PostgreSQL database. Using the ‘`psycopg2`’ library, it connects to either the master or the slave database based on the provided database name.

Configuration details such as host, database name, user, and password are retrieved from the `POSTGRES_CONFIG` constants, ensuring a secure and modular approach to database connection.

```
# Postgres configuration
POSTGRES_CONFIG = {
    .... 'HOST_NAME': 'part-4.cluster-cbg6fltkvvl6.us-east-1.rds.amazonaws.com',
    .... 'USER_NAME': 'postgres',
    .... 'PASSWORD': 'database',
    .... 'PORT': 5432
}
```

- **Database Creation:**

The 'create_database' function is responsible for the initial setup of our databases. By connecting to the default PostgreSQL database, it checks for the existence of our target database and creates it if it doesn't already exist.

```
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()
```

Table Creation and Data Management:

- **Creating Tables:**

Using ‘create_tables’, the script executes a series of SQL queries stored in CREATE_QUERIES to create necessary tables such as **Orders**, **Order Details**, and **Inventory**. This sets up the schema required for our transactional operations.

```
ORDERS_QUERY = f"""
    CREATE TABLE IF NOT EXISTS {TABLE_NAMES[0]} (
        orders_id SERIAL PRIMARY KEY,
        users_id INTEGER NOT NULL,
        order_price NUMERIC(10, 2) NOT NULL,
        order_date DATE NOT NULL
    );
    """

ORDER_DETAILS_QUERY = f"""
    CREATE TABLE IF NOT EXISTS {TABLE_NAMES[1]} (
        order_details_id SERIAL PRIMARY KEY,
        order_id INTEGER NOT NULL,
        product_id INTEGER NOT NULL,
        quantity INTEGER NOT NULL CHECK (quantity > 0),
        subtotal NUMERIC(10, 2) NOT NULL
    );
    """

INVENTORY_QUERY = f"""
    CREATE TABLE IF NOT EXISTS {TABLE_NAMES[2]} (
        inventory_id SERIAL PRIMARY KEY,
        product_id INTEGER NOT NULL,
        warehouse_name VARCHAR(255) NOT NULL,
        stock_quantity INTEGER NOT NULL CHECK (stock_quantity >= 0)
    );
    """
```

- **Managing Data:**

The ‘**select_data**’ function is employed to query and display data from these tables, providing a clear view of the current state of the database. This function is essential for validating the replication and consistency of data between the master and the slave databases.

```
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)
```

ACID-Compliant Transactions:

- **Ensuring Transaction Integrity:**

Central to our implementation is the management of ACID-compliant transactions. The ‘**execute_query**’ function, equipped with error handling and transaction rollback capabilities, ensures that every database operation either completes entirely or leaves the database unaltered in the event of a failure.

```

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

```

- **Transaction Processing:**

The **'process_order'** function demonstrates a real-world application of transaction processing. It encapsulates a series of database operations within a single transaction, ensuring atomicity, consistency, isolation, and durability. This function showcases how orders are processed, involving updating multiple tables in a single, atomic transaction.

```

def process_order(conn, user_id, product_id, quantity, order_price):
    # Each thread will use its own connection to the database
    try:
        with conn.cursor() as cursor:
            # Start a transaction
            cursor.execute("BEGIN;")
            # Insert a new order into the 'Orders' table and obtain the order ID
            cursor.execute(
                "INSERT INTO Orders (users_id, order_price, order_date) VALUES (%s, %s, CURRENT_DATE) RETURNING orders_id;",
                (user_id, order_price)
            )
            order_id = cursor.fetchone()[0]

            # Insert order details
            cursor.execute(
                "INSERT INTO OrderDetails (order_id, product_id, quantity, subtotal) VALUES (%s, %s, %s, %s);",
                (order_id, product_id, quantity, order_price)
            )

            # Update the inventory to reflect the reduced stock quantity
            cursor.execute(
                "UPDATE Inventory SET stock_quantity = stock_quantity - %s WHERE product_id = %s;",
                (quantity, product_id)
            )

            # Commit the transaction
            conn.commit()
    except psycopg2.DatabaseError as e:
        print(f"Database error occurred: {e}")
        conn.rollback()
    except Exception as e:
        print(f"An error occurred: {e}")
        conn.rollback()
    finally:
        if cursor is not None:
            cursor.close()

```

Concurrency Control Mechanisms:

- **Threading for Simultaneous Transactions:**

To simulate and manage concurrent transactions, our implementation utilizes Python's threading capabilities. The **'simulate_concurrent_orders'** function creates multiple threads, each executing the `process_order` function. This simulates a real-world scenario where multiple users are interacting with the database simultaneously.


```

# Simulate multiple users placing orders concurrently
def simulate_concurrent_orders(conn):
    # Example data for multiple orders
    orders_data = [
        (conn, 1, 2, 1, 100.00),
        (conn, 2, 1, 2, 200.00),
        (conn, 1, 1, 1, 150.00),
        # Add more orders as needed
    ]

    threads = []
    for order_data in orders_data:
        # Create a new thread for each order
        thread = threading.Thread(target=process_order, args=order_data)
        threads.append(thread)
        thread.start()

    # Wait for all threads to complete
    for thread in threads:
        thread.join()

```

- **Ensuring Data Consistency:**

These concurrent operations are a test bed for our concurrency control mechanisms. By monitoring the results of these operations, we can confirm that our database setup effectively handles concurrent access, maintaining data consistency and integrity.

4. Testing and Results

Effective testing is crucial to validate the functionality, performance, and reliability of any database system. In our project, we conducted comprehensive testing to ensure that our PostgreSQL database system, integrated with Amazon Aurora, met the required standards for ACID compliance, data consistency, and concurrency control.

Simulation of Concurrent Orders:

- **Setup and Execution:**

- We simulated a real-world scenario where multiple users place orders concurrently. This was achieved through the **‘simulate_concurrent_orders’** function, which utilized threading to create and process multiple order transactions simultaneously.
- Each thread, representing a different user session, executed the **‘process_order’** function, thus testing the system's ability to handle concurrent write and read operations.

Testing ACID Compliance:

- **Transaction Integrity:**

- Key to our testing was ensuring that each transaction adhered to ACID principles. We monitored transactions for atomicity, ensuring that all operations within a transaction were completed successfully or fully rolled back in case of an error.
- Consistency was validated by checking the state of the database before and after transactions, confirming that each transaction transitioned the database from one valid state to another.
- Isolation was tested by ensuring that concurrent transactions did not interfere with each other. This was particularly crucial in verifying that our database handled concurrent accesses without data anomalies.
- Durability was confirmed by simulating failures (like sudden system shutdowns) and verifying that completed transactions persisted correctly post-recovery.

Concurrency Control:

- **Handling Simultaneous Transactions:**

- The primary focus was on how the system managed simultaneous transactions. Our observations showed that the concurrency control mechanisms effectively managed data access conflicts, preventing issues like lost updates, dirty reads, or phantom reads.

- Performance under load was monitored to assess how the system coped with multiple concurrent transactions. We measured response times and system throughput to determine the impact of concurrency on overall performance.

Data Consistency and Integrity:

- **Replication Validation:**

- A critical aspect of our testing was to confirm that data changes in the master database were consistently and correctly replicated to the slave database. We conducted several tests, updating data in the master database and verifying these changes in the slave database.
- Data integrity checks were carried out to ensure no data corruption occurred during replication.

5. Results:

Findings:

- The testing phase demonstrated that our system was capable of handling ACID-compliant distributed transactions efficiently. The master-slave configuration, aided by Amazon Aurora's capabilities, ensured high data availability and integrity.
- Concurrency tests revealed that our system effectively managed simultaneous transactions, maintaining data consistency and system performance under concurrent loads.

Data from Table: Inventory

inventory_id	product_id	warehouse_name	stock_quantity
1	1	Main Warehouse	100
2	2	Main Warehouse	150
3	3	Secondary Warehouse	200

This is a snapshot of the initial inventory. After performing concurrent orders

Data from Table: Orders

orders_id	users_id	order_price	order_date
1	1	100	2023-11-28
2	2	200	2023-11-28
3	1	150	2023-11-28

Data from Table: OrderDetails

order_details_id	order_id	product_id	quantity	subtotal
1	1	2	1	100
2	2	1	2	200
3	3	1	1	150

The final Inventory looks like this

Data from Table: Inventory

inventory_id	product_id	warehouse_name	stock_quantity
3	3	Secondary Warehouse	200
2	2	Main Warehouse	149
1	1	Main Warehouse	97

Screenshot of Read_replica database

public.inventory/e_commerce/postgres@Aurora Read_Replica

Query: `SELECT * FROM public.inventory ORDER BY inventory_id ASC`

	inventory_id [PK] integer	product_id integer	warehouse_name character varying (255)	stock_quantity integer
1	1	1	Main Warehouse	97
2	2	2	Main Warehouse	149
3	3	3	Secondary Warehouse	200

6. Conclusion

Upon completion of this project, we have successfully demonstrated the capability of a master-slave database architecture using PostgreSQL in conjunction with Amazon Aurora to ensure high data availability, maintain data integrity, and efficiently handle concurrent transactions.

Achievements:

- Our implementation has effectively leveraged the robustness of PostgreSQL for transaction management and the scalability and resilience of Amazon Aurora. This combination has proven to be a powerful solution for managing distributed databases in a cloud environment.
- The system has shown commendable adherence to ACID properties, ensuring atomicity, consistency, isolation, and durability of transactions, even under the stress of concurrent operations. This is a significant achievement as it guarantees the integrity and reliability of the database, which is paramount in real-world applications.
- The concurrency control mechanisms implemented have been successful in managing simultaneous transactions, validating our approach to threading and transaction processing in a high-demand environment.