**Part-4**

We will be going through two codes

1. Transaction\_management.py
2. Distributed\_transactions.py

**Transaction\_management.py (for concurrency control)**

1. **Database Connection:**
   * The script uses the **psycopg2** library to connect to a PostgreSQL database running on localhost with the specified credentials (**dbname**, **user**, **password**, **host**, **port**).
2. **SQL Execution Functions:**
   * **execute\_sql**: Executes a SQL statement with optional parameters on the given database connection and commits the changes.
   * **execute\_sql\_and\_fetch**: Executes a SQL statement with optional parameters, fetches the result, and returns it.
3. **Read Team ID Functions:**
   * **read\_team\_id**: Reads the team ID for a specified player without transaction management. It opens a connection, executes a SELECT statement, sleeps for 2 seconds to simulate processing, and then reads the team ID again.
   * **read\_team\_id\_trans**: Reads the team ID for a specified player with transaction management. It begins a transaction with **SET TRANSACTION ISOLATION LEVEL SERIALIZABLE**, performs a similar read-sleep-read sequence, and commits the transaction.
4. **Update Team ID Function:**
   * **update\_team\_id** : Updates the team ID for a specified player using transaction management. It begins a transaction, executes an UPDATE statement, commits the transaction. If an error occurs, it rolls back the transaction.
5. **Concurrency Simulation:**
   * The script uses the **ThreadPoolExecutor** from the **concurrent.futures** module to simulate concurrent execution of tasks.
   * In the first concurrent block, it executes the **read\_team\_id** function and the **update\_team\_id** function concurrently for the same player ID.
   * After a 2-second sleep, it prints a message indicating the transition to the next scenario.
6. **Transaction Management Simulation:**
   * In the second concurrent block, it executes the **read\_team\_id\_trans** function (with transaction management) and the **update\_team\_id** function concurrently for the same player ID.
   * The **read\_team\_id\_trans** function uses **SET TRANSACTION ISOLATION LEVEL SERIALIZABLE** and commits the transaction after reading.
7. **Print Statements:**
   * The script includes print statements to provide detailed information about each step in the concurrent scenarios, including the thread ID, player ID, and the values read or updated.
8. **Sleep Statements:**
   * There are **time.sleep(2)** statements to simulate some processing time between reads and updates.
9. **Testing Player ID:**
   * A player ID (**player\_id = 1**) is specified for testing purposes. The script can be modified to test with different player IDs.
10. **Execution:**
    * The script is designed to be run as a standalone Python script. It demonstrates the behavior of concurrent read and update operations with and without transaction management in a multi-threaded environment.
    * Ensure database is populated with tables and sample data in Part1 before running the script.

This script is useful for understanding how transactions and concurrent database operations work in a multi-threaded context and how transaction management can affect the consistency of data in a database. Let’s see the output to see what happens with and without transaction management when thread is trying to read the data when some other thread is concurrently updating the data with and without transaction management.

A screen shot of a computer screen

Description automatically generated

**Distributed\_transactions.py (Distributed ACID compiant)**

1. **Database Connection:**
   * The script uses the **psycopg2** library to connect to two PostgreSQL databases running on localhost with the specified credentials (**dbname**, **user**, **password**, **host**, **port**).
2. **SQL Execution Functions:**
   * **execute\_sql**: Executes a SQL statement with optional parameters on the given database connection and commits the changes.
3. **Transaction Management Functions:**
   * **prepare\_transaction**: Prepares a transaction with a specified name.
   * **commit\_prepared\_transaction**: Commits a prepared transaction with the specified name after closing the initial transaction.
   * **rollback\_prepared\_transaction**: Rolls back a prepared transaction with the specified name after ensuring that the script is not inside a transaction block.
4. **Read Team ID Function:**
   * **read\_team\_id**: Reads the team ID for a specified player from the database.
5. **Distributed Transaction Function:**
   * **distributed\_transaction**: Performs a distributed transaction across two databases.
   * Generates a unique transaction name using a timestamp.
   * Connects to two PostgreSQL databases (**conn1** and **conn2**).
   * Begins a distributed transaction by preparing the transaction on both databases.
   * Performs updates (simulated as updating the team ID for Player 1 to 1) on both databases within the same distributed transaction.
   * Commits the distributed transaction if successful or rolls it back in case of an error.
   * Reads the team ID for Player 1 from both databases after committing the transaction.
6. **Testing and Execution:**
   * The script is designed to be run as a standalone Python script.
   * It demonstrates a simple scenario of updating the team ID for Player 1 in both databases within a distributed transaction.
7. **Print Statements:**
   * The script includes print statements to provide detailed information about each step in the distributed transaction, including the initial update, simulated processing time, and the final read of the team IDs.
8. **Sleep Statements:**
   * There is a **time.sleep(5)** statement to simulate processing time between updates.
9. **Database Cleanup:**
   * The script closes the database connections (**conn1** and **conn2**) in a **finally** block to ensure proper cleanup even in case of errors.
10. **Execution:**
    * Start the Docker containers: **docker-compose up -d**
    * Ensure both databases in both servers are populated using Part1/db\_tables.py. For one server the port is 5432 for other it is 5433.
    * Run following query for both servers in pgAdmin:- ALTER SYSTEM SET max\_prepared\_transactions = 10; to ensure multiple distributed transactions can take place.
11. **Assumptions:**
    * The script assumes the existence of a 'players' table in both databases with 'player\_id' and 'team\_id' columns. If not

Output:-

Case when transaction is successful in both servers

A black background with white text

Description automatically generated

Case when there is an error in one of the servers. As you can see transaction is rolled back achieving ACID compliance.

