Xi'an Jiaotong-Liverpool University
西交利物浦大学

DTS207TC - Database Development and Design - Introduction to Database Transaction

Course Leader: Zhang D, Co-Teacher: Affan Y

November 24, 2025

Contents

1 Lab Learning Outcomes and Revision

- ***Learning Outcomes*:** By the end of this lab, students should be able to:
  - Understand the purpose of transactions and their importance in maintaining data integrity and consistency.
  - Use transaction control commands in PostgreSQL, including `BEGIN`, `COMMIT`, and `ROLLBACK`.
  - Apply transactions to hypothetical scenarios to evaluate data consistency and manage error handling.
  - Learn the concept of transaction isolation and how it ensures data consistency in a multi-user environment.
  - Identify and explain the four isolation levels: `Read Uncommitted`, `Read Committed`, `Repeatable Read`, and `Serializable`.
  - Understand the performance for each isolation level, including potential risks like dirty reads, non-repeatable reads, and phantom reads.
  - Analyze the performance impact of different isolation levels, including their effect on concurrency, transaction speed, and system resource usage.
- **What is a Transaction in a Database?** A transaction in a database is a sequence of operations that are treated as a single unit. These operations must either all succeed or all fail together. If one part of the transaction fails, the whole transaction is rolled back, meaning no changes are made to the database. This ensures that the database remains in a consistent state.
- ***What is ACID Properties?* :** Transactions must follow **ACID** properties:
  - ***Atomicity***: Ensures that all steps in a transaction are completed successfully or none at all.
  - **Consistency**: Ensures that data moves from one valid state to another after a transaction.
  - **Isolation**: Ensures transactions do not interfere with each other.
  - **Durability**: Once a transaction is committed, changes are permanent even in case of system failure.
- **Real-Life Analogy: Shopping at a Store:** Imagine you are shopping at a store. You pick up a few items, go to the cashier, and proceed to pay. In this scenario, the shopping process is like a transaction. There are multiple steps involved: ***i)*** selecting the items, ***ii)*** paying for them, and ***iii)*** receiving a receipt as proof of purchase.

  Now, if something goes wrong during the payment process—say the cashier's system crashes before you get the receipt—the store will cancel the entire transaction. Even though you picked up the items and stood in line to pay, you will not be charged, and the items won't be yours. The store *rolls back* the entire process, ensuring that you're not accidentally charged for something you didn't complete.

  This is similar to how a database transaction works. If any part of the transaction fails, like an update or insertion operation, the database will undo all the changes, maintaining consistency and accuracy. The process of booking a flight, adding items to a shopping cart online, or transferring money between bank accounts also follows this principle. If one part fails, nothing is changed, ensuring no partial or incorrect transactions happen.

2 Lab Topic I: Transactions in SQL (PostgreSQL)

Objective:

Practice using transactions to manage data consistency and integrity within a relational database.

2.1 Introduction to Transactions

PostgreSQL Transaction Control Commands: In PostgreSQL, the main commands used in transaction control are:

- **BEGIN**: Starts a new transaction.
- **COMMIT**: Ends a transaction and makes all changes permanent.

ROLLBACK and SAVEPOINT in a database is used to undo changes made after a specific savepoint within a transaction, without affecting the entire transaction. A **savepoint** acts as a checkpoint within a transaction, allowing partial rollback to a certain point. This helps manage complex transactions by enabling selective rollback instead of canceling the entire transaction.

Example:

```
SAVEPOINT my_savepoint;
-- some database operations
ROLLBACK TO SAVEPOINT my_savepoint;  -- undoes changes made after 'my_savepoint'
```

This allows more control over transaction execution, especially in scenarios where errors occur partway through.

2.2 Lab Exercise

Problem Statement

You are working on a trading platform database that tracks currency pairs and their latest bid/ask prices. Data is organized in two tables:

- **Info table**: Contains information about trading pairs.
  - Columns: `base`, `quote_`, `symbol`, `old_symbol`
- **Market table**: Contains real-time trading data.
  - Columns: `timestamp`, `symbol`, `a` (ask price), `aq` (ask quantity), `b` (bid price), `bq` (bid quantity)

The goal is to:

1. Start a transaction that inserts new trading data into the `market` table.
2. Attempt to update the corresponding `info` table entry.
3. Roll back the transaction if the update fails (e.g., due to a unique constraint conflict or other database constraint).

If `EUR/USD` has outdated information in the `info` table, update it. If the update fails due to a constraint, roll back the transaction.

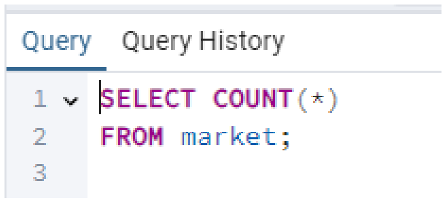
2.3 Lab Task Outline

1. **Start a transaction** using `BEGIN`.
2. **Insert** the new trade data into the `market` table.
3. **Attempt to update** the `info` table with new details.
4. **Roll back** the transaction if the update fails.
5. **Commit** the transaction if all operations are successful.

2.4 SQL Code Implementation

Step 1: Set Up Tables in PostgreSQL - *If not created earlier*

- Please note that the two lines of the code, which contain the `SELECT count(*)` (Figure 1), are used to check the number of rows in the table before executing the query.



```
SELECT COUNT(*)
FROM market;
```

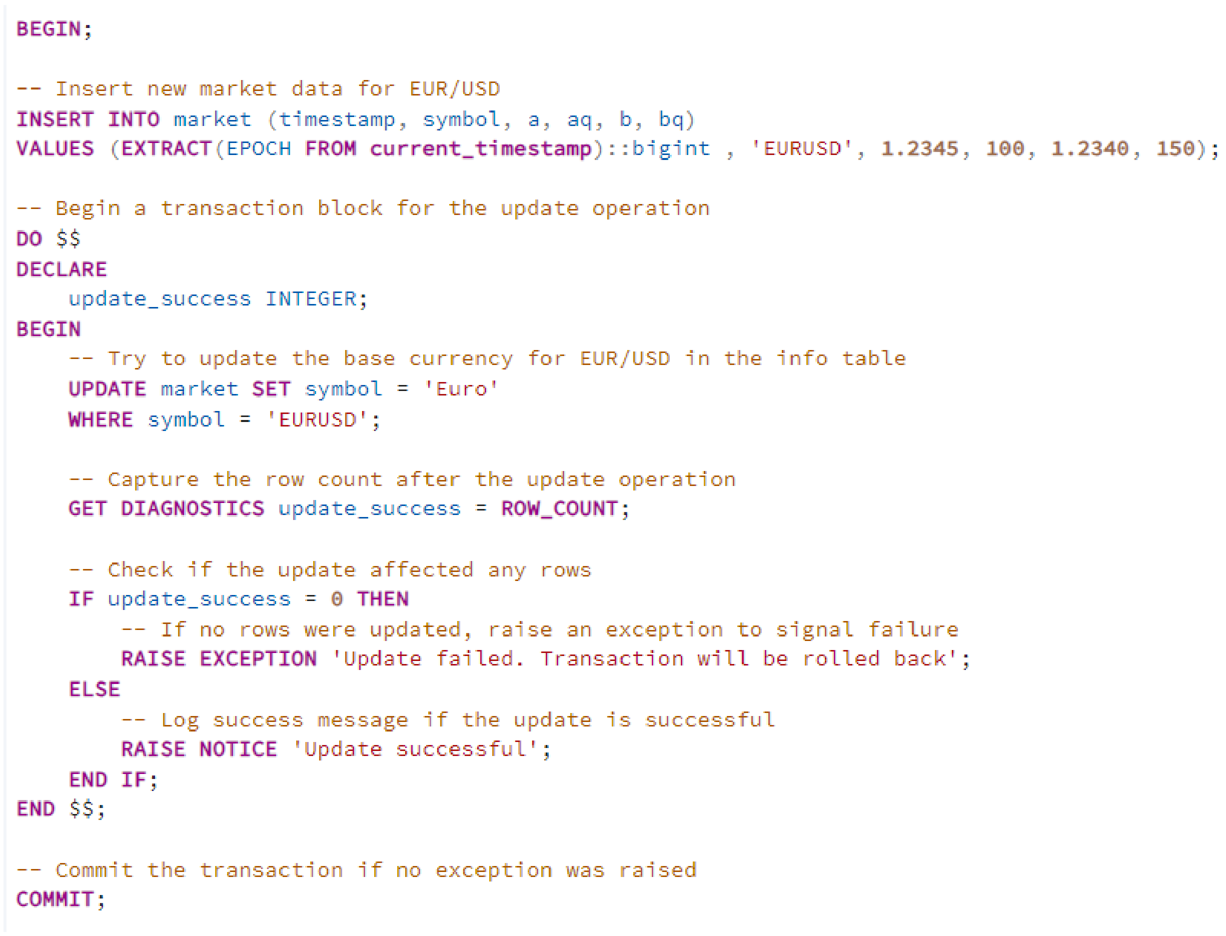
Figure 1: Check Number of Rows and note on paper

- **Running the Code in PostgreSQL:** To test this, write the code into PostgreSQL and run it. Ensure the `info` and `market` tables have been created and populated as specified. To check the number of rows in the market table use the SQL code given in Figure 1.
- **Execute the Transaction Logic:** The following code block demonstrates how to use PostgreSQL's PL/pgSQL procedural language to handle the transaction with conditional logic for rollback and commit. Figure 3 shows the failure of transaction for which we have to execute `ROLLBACK;` command explicitly. Figure 2 shows the code which is successful due to change in the Table `market` instead of `info`.

PLEASE NOTE THAT THE CODE FILES ARE WITH US, AND CAN BE SHARED UPON RECEIVING APPROVAL FROM THE COURSE INSTRUCTOR.

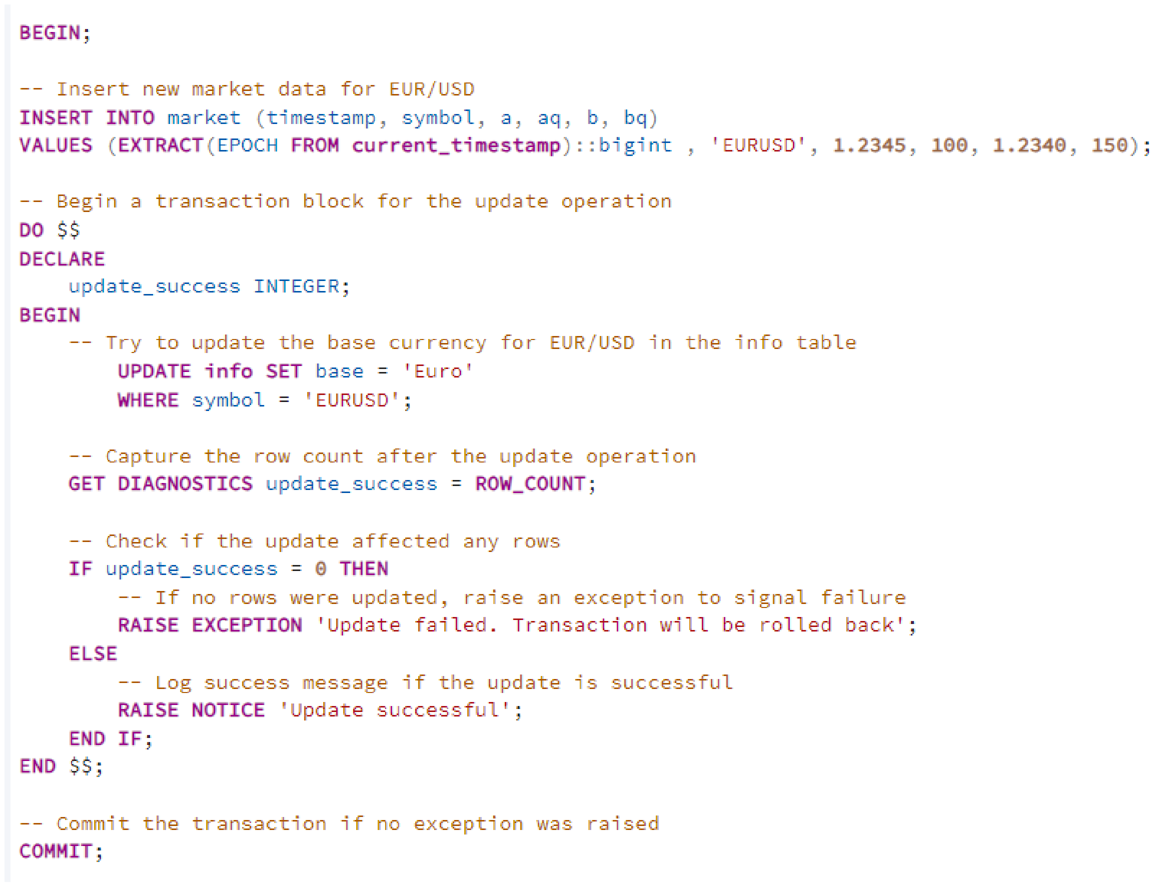
2.5 Explanation of the Code

- **BEGIN**: Starts the transaction, ensuring that all subsequent statements are executed together.
- **Insert Statement**: Inserts new trading data into the `market` table. (Optional - in case you don't have data in market Table)
- **DO Block**: This PL/pgSQL block attempts to update the `info` table and checks if the update was successful.
- **GET DIAGNOSTICS**: Captures the result of the update statement to determine if it affected any rows.
- **Rollback or Commit**:
  - If the update fails (`ROW_COUNT = 0`), the transaction rolls back.
  - If the update is successful, the transaction commits, saving both the insert and update changes.



```sql
BEGIN;

-- Insert new market data for EUR/USD
INSERT INTO market (timestamp, symbol, a, aq, b, bq)
VALUES (EXTRACT(EPOCH FROM current_timestamp)::bigint , 'EURUSD', 1.2345, 100, 1.2340, 150);

-- Begin a transaction block for the update operation
DO $$
DECLARE
    update_success INTEGER;
BEGIN
    -- Try to update the base currency for EUR/USD in the info table
    UPDATE market SET symbol = 'Euro'
    WHERE symbol = 'EURUSD';

    -- Capture the row count after the update operation
    GET DIAGNOSTICS update_success = ROW_COUNT;

    -- Check if the update affected any rows
    IF update_success = 0 THEN
        -- If no rows were updated, raise an exception to signal failure
        RAISE EXCEPTION 'Update failed. Transaction will be rolled back';
    ELSE
        -- Log success message if the update is successful
        RAISE NOTICE 'Update successful';
    END IF;
END $$;

-- Commit the transaction if no exception was raised
COMMIT;
```

Figure 2: Code for Transaction successful



```
BEGIN;

-- Insert new market data for EUR/USD
INSERT INTO market (timestamp, symbol, a, aq, b, bq)
VALUES (EXTRACT(EPOCH FROM current_timestamp)::bigint , 'EURUSD', 1.2345, 100, 1.2340, 150);

-- Begin a transaction block for the update operation
DO $$
DECLARE
    update_success INTEGER;
BEGIN
    -- Try to update the base currency for EUR/USD in the info table
        UPDATE info SET base = 'Euro'
        WHERE symbol = 'EURUSD';

    -- Capture the row count after the update operation
    GET DIAGNOSTICS update_success = ROW_COUNT;

    -- Check if the update affected any rows
    IF update_success = 0 THEN
        -- If no rows were updated, raise an exception to signal failure
        RAISE EXCEPTION 'Update failed. Transaction will be rolled back';
    ELSE
        -- Log success message if the update is successful
        RAISE NOTICE 'Update successful';
    END IF;
END $$;

-- Commit the transaction if no exception was raised
COMMIT;
```

Figure 3: Code for Transaction Failure

2.6 Summary

This lab exercise demonstrates the importance of transactions in maintaining data integrity. By using `BEGIN`, `COMMIT`, and `ROLLBACK`, students can manage multi-step operations in SQL, ensuring consistency even in case of errors.

Transactions are a powerful tool for controlling database operations, especially in cases requiring precise data integrity, such as trading platforms or inventory systems.

The provided code / shown above may encounter errors in various PostgreSQL environments due to certain common factors. If you face any issues, don't hesitate to ask for assistance during the lab session. The instructor or T.A. will gladly help you resolve the problem directly on your PC. Some of the reasons might be:

- **Mismatch in Data Types:** The column's data type in the table may differ from what the code assumes. For example, a column defined as BIGINT may cause issues if the code tries to insert a TIMESTAMP or vice versa.
- **Precision Differences:** When working with values like epoch time in milliseconds, differences in how environments handle numeric precision could lead to errors or unexpected results.
- **Regional or Local Settings:** Variations in local settings, such as date and time formats, may cause issues when interpreting `TO_TIMESTAMP` or other time related functions.
- **Trailing or Extra Spaces:** Data being inserted or copied might contain trailing spaces or formatting inconsistencies, causing parsing errors during operations like COPY.
- **Version Incompatibility:** PostgreSQL features and functions can vary slightly between versions, which might cause unexpected behavior.

3 Lab Topic II : PostgreSQL - Isolation Levels

PostgreSQL supports different **transaction isolation levels**, determining how transaction changes are visible to other transactions:

- **READ UNCOMMITTED**: Allows transactions to read uncommitted changes from other transactions. ***PostgreSQL does not support the "READ UNCOMMITTED" isolation level.***
- **READ COMMITTED**: Only committed changes from other transactions are visible.
- **REPEATABLE READ**: Ensures that all reads within a transaction are consistent.
- **SERIALIZABLE**: Provides the highest level of isolation, making it as though transactions are executed one after another.

3.1 Theoretical Knowledge

Real-Life Example: Booking an Airplane Seat on an E-commerce Website: Imagine you're on an e-commerce website looking to book a flight. You select a seat on the plane, but before you finish the booking, the price or availability changes. This scenario is a good analogy to explain how different database isolation levels affect transactions.

Scenario:

- **You (T1)** start the booking process for a seat on an airplane (e.g., Seat 10A).
- **Another customer (T2)** is also trying to book the same seat at the same time.

Now, let's see what happens in different isolation levels.

3.1.1 Read Committed

What happens? When you start looking at the seat, the system shows you the availability and price. However, while you are still selecting and filling out your details, the other customer (T2) might have already completed their booking and the seat could be taken. When you refresh the page or proceed with your booking, the system will show you the updated seat availability or price.

Why? In **Read Committed**, the system allows reading the most up-to-date data committed by other transactions. So, if **T2** finishes their transaction (book the seat), **T1** will see the new state (e.g., "Seat 10A is no longer available" or the price has changed) once they attempt to refresh or complete their booking.

Summary: You're selecting a seat, and while you're still in the process, someone else grabs the last available seat. When you refresh the page, you'll see the seat is no longer available or the price has increased.

3.1.2 Repeatable Read

What happens? When you start selecting your seat, the system ensures that you will continue to see the same price and availability for the seat throughout the entire transaction, even if the other customer (T2) tries to book the same seat. No matter how many times you refresh the page, you will see the same seat availability and price until you either finish or cancel the booking or session expires. If another customer (T2) tries to book the same seat during your transaction, they will be blocked from doing so until you finish.

Why? In *Repeatable Read*, once you start a transaction and read the data, the system guarantees that the data won't change for the duration of your transaction. This prevents "non-repeatable reads", meaning you won't see the seat availability change midway through your booking process. The other customer's (T2's) actions won't affect what you see until you commit.

Real-Life Analogy: Imagine you're in a store looking at a product and you ask the salesperson to hold it for you. Even if someone else tries to buy it while you're deciding, the store won't let them buy it until you're done. Your view of the product (price and availability) stays the same throughout the decision-making process.

3.1.3 Serializable

What happens? When you start selecting your seat, the system locks the entire set of seats so no one else can book any seats until your transaction completes. Even if another customer (T2) tries to book the same seat, or any other seat, they won't be able to do so until you're done. This provides the highest level of consistency and ensures that no other transactions can interfere with your transaction.

Why? In **Serializable**, the system behaves as if all transactions are occurring sequentially, one after another, even though they may be happening concurrently. This prevents ”phantom reads” and guarantees full isolation of the transactions, ensuring that no other transactions can modify data that might affect the outcome of the current transaction.

Real-Life Analogy: Imagine you're in a store, and the salesperson locks all the products in the aisle for you to look at and decide. No one else can purchase any of the products until you're finished making your decision and have completed your purchase. This ensures that no one else can affect your selection process, and the products stay exactly as you saw them.

Key Differences in Simple Terms:

Isolation Level	What You See	Real-World Example	Risk
Read Committed	You see the most recent committed data. If T2 updates the seat, you'll see the change when you refresh or commit your booking.	The seat availability or price may change while you're booking, especially if another customer completes the purchase.	**Non-repeatable reads**: The seat may no longer be available or the price may change when you refresh.
Repeatable Read	You see the same data throughout your transaction. If T2 tries to book the same seat, they can't until you finish.	The seat will stay the same for you throughout the booking process. Even if another customer tries to book it, they'll be blocked until you're done.	**Phantom reads**: New seats might appear or disappear, but the seat you're looking at won't change.
Serializable	You see the data as if no other transactions are happening. No one else can modify or book any seats until your transaction completes.	All products are locked for you to decide. No one else can book a seat or modify the price until you're finished.	**Full isolation**: Other transactions are completely blocked from affecting the data you're working with.

Conclusion:

- **Read Committed** allows for real-time updates, but you might see the availability or price change while you're still in the process of booking.
- **Repeatable Read** ensures that the seat availability and price you saw at the start of the transaction will stay the same until you're finished.
- **Serializable** locks the entire set of seats, ensuring no one else can book or modify any seats while you're deciding. It provides the highest level of isolation and guarantees no interference from other transactions.

3.2 Practical Task

Problem Statement

Imagine you're managing a financial database that stores exchange rate information for various currency pairs. You have two tables:

- **info** (contains basic information about currency pairs)
- **market** (stores market data like bid and ask prices)

The goal of this exercise is to ensure data consistency using transactions. We will:

- Update the `info` table with a new base currency.
- If the update fails, we will rollback the transaction to maintain the integrity of the database.

Setup the Environment

- Ensure a PostgreSQL instance is available.
- Import the given data into PostgreSQL. You will have two tables (`info` and `market`) with the following columns:
  - `info` table: `base`, `quote_`, `symbol`, `old_symbol`
  - `market` table: `timestamp`, `symbol`, `a`, `aq`, `b`, `bq`

3.2.1 Code Example - Read Committed

- Here's how to implement the "read committed" logic in PostgreSQL (Figure 6). The SELECT (*) query at both the beginning and the end is used to verify the changes that occurred before and after the update. Run individual scripts to see the effects (before and after). Look at the code in the Figure and try to run it in your own environment. If you face any issues, feel free to reach out to the Teacher or the Lab Teaching Assistant for help.

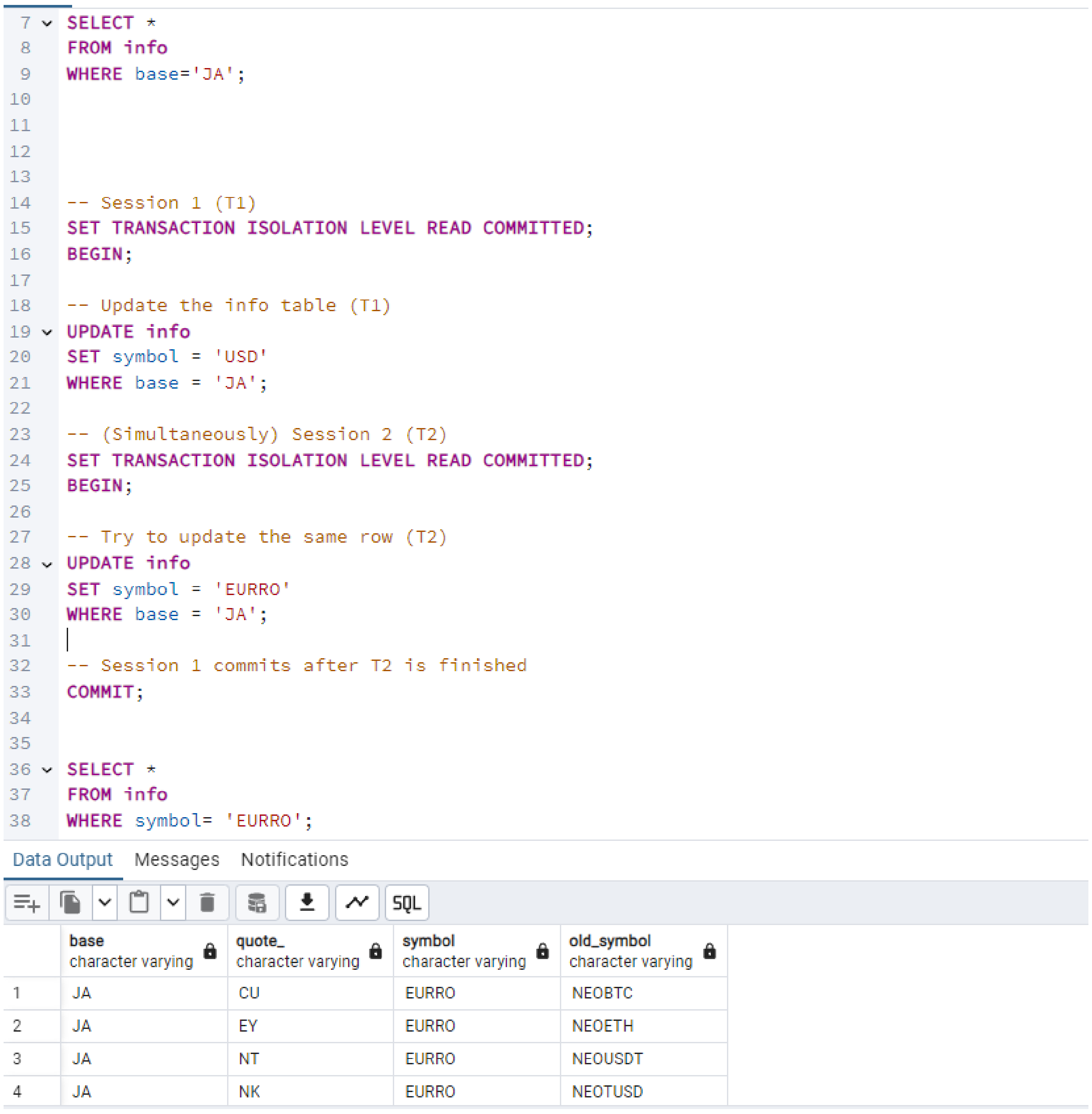
Explanation of Code - READ COMMITTED Isolation Level: In the `READ COMMITTED` isolation level, a transaction can only read data that has been committed by other transactions. Here's what happens in the given scenario:

- **Session 1 (T1)** begins and updates a row where `base = 'JA'` to `base = 'Euro'`. However, this update is not committed yet.
- **Session 2 (T2)** starts and tries to update the same row where `base = 'JA'`. However, because T1 has not committed yet, T2 cannot see T1's uncommitted changes.
- Since the row is locked by T1, T2 is blocked from updating it until T1 commits or rolls back.
- Once **T1** commits its changes (i.e., `base = 'USD'`), **T2** can proceed and see the updated value.
- The `SELECT` query in **T2** will return the updated value `base = 'EURRO'` once T1 has committed.

Thus, under `READ COMMITTED`, T2 cannot see uncommitted changes made by T1, and T2 will be blocked until T1 commits.

3.2.2 Code Example - Repeatable Read

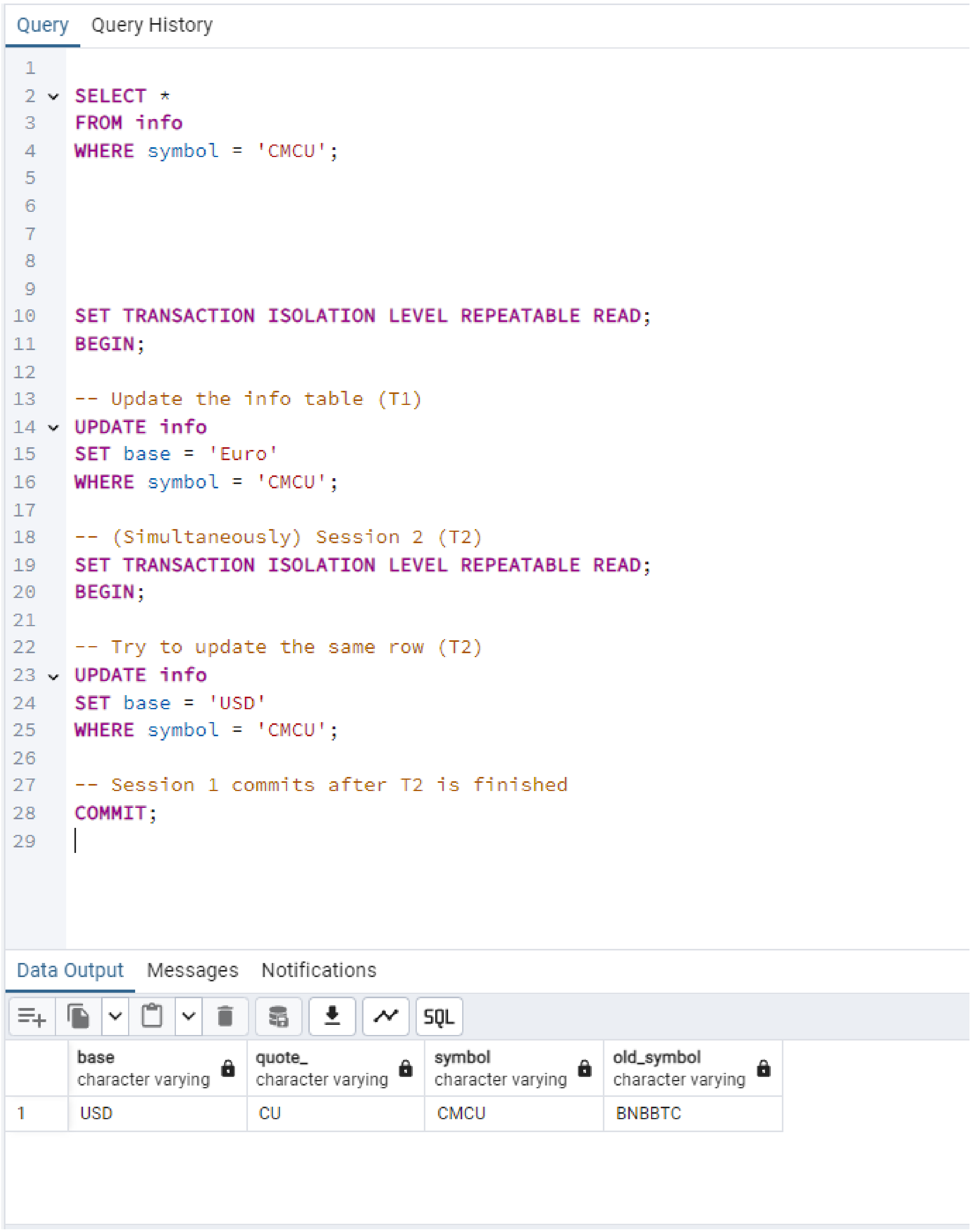
- Code for Repeatable read is shown in Figure 5. Look at the code in the Figure and try to run it in your own environment. If you face any issues, feel free to reach out to the Teacher or the Lab Teaching Assistant for help.



```
SELECT *
FROM info
WHERE base='JA';




-- Session 1 (T1)
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN;

-- Update the info table (T1)
UPDATE info
SET symbol = 'USD'
WHERE base = 'JA';

-- (Simultaneously) Session 2 (T2)
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN;

-- Try to update the same row (T2)
UPDATE info
SET symbol = 'EURRO'
WHERE base = 'JA';

-- Session 1 commits after T2 is finished
COMMIT;


SELECT *
FROM info
WHERE symbol= 'EURRO';
```

	base character varying	quote_ character varying	symbol character varying	old_symbol character varying
1	JA	CU	EURRO	NEOBTC
2	JA	EY	EURRO	NEOETH
3	JA	NT	EURRO	NEOUSDT
4	JA	NK	EURRO	NEOTUSD

Figure 4: Code fo Read Committed


Query | Query History

```sql

SELECT *
FROM info
WHERE symbol = 'CMCU';





SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;

-- Update the info table (T1)
UPDATE info
SET base = 'Euro'
WHERE symbol = 'CMCU';

-- (Simultaneously) Session 2 (T2)
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;

-- Try to update the same row (T2)
UPDATE info
SET base = 'USD'
WHERE symbol = 'CMCU';

-- Session 1 commits after T2 is finished
COMMIT;

```

Data Output | Messages | Notifications

	base character varying	quote_ character varying	symbol character varying	old_symbol character varying
1	USD	CU	CMCU	BNBBTC

Figure 5: Code for Repeatable Read

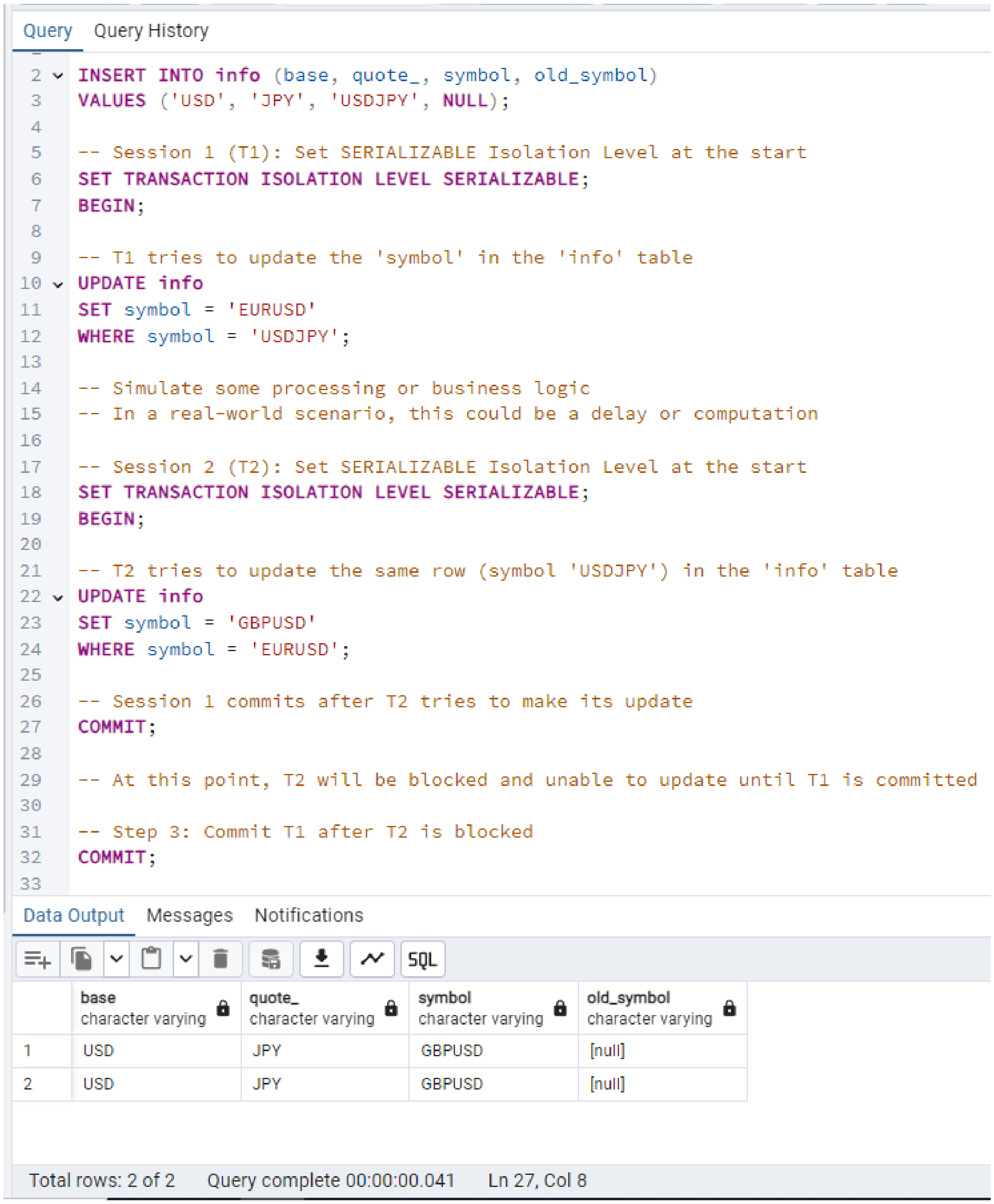
3.2.3 Code Example - Serializable

- Code for Serializable is shown in Figure 6. Look at the code in the Figure and try to run it in your own environment. If you face any issues, feel free to reach out to the Teacher or the Lab Teaching Assistant for help.

In this example, we demonstrate how the `SERIALIZABLE` isolation level works when two transactions (`T1` and `T2`) are running concurrently and attempting to modify the same row in the `info` table.

The `SERIALIZABLE` isolation level ensures that transactions behave as if they were executed sequentially, one after the other, even if they are actually executed concurrently. This means that:

- Transactions are fully isolated from each other.
- If two transactions try to modify the same data, the second transaction will be blocked until the first transaction commits, preventing any conflicts or inconsistencies.
- In our example, `T2` was blocked from committing until `T1` committed, ensuring that the updates to the `symbol` field were handled sequentially.


Query Query History

```sql
INSERT INTO info (base, quote_, symbol, old_symbol)
VALUES ('USD', 'JPY', 'USDJPY', NULL);

-- Session 1 (T1): Set SERIALIZABLE Isolation Level at the start
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;

-- T1 tries to update the 'symbol' in the 'info' table
UPDATE info
SET symbol = 'EURUSD'
WHERE symbol = 'USDJPY';

-- Simulate some processing or business logic
-- In a real-world scenario, this could be a delay or computation

-- Session 2 (T2): Set SERIALIZABLE Isolation Level at the start
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;

-- T2 tries to update the same row (symbol 'USDJPY') in the 'info' table
UPDATE info
SET symbol = 'GBPUSD'
WHERE symbol = 'EURUSD';

-- Session 1 commits after T2 tries to make its update
COMMIT;

-- At this point, T2 will be blocked and unable to update until T1 is committed

-- Step 3: Commit T1 after T2 is blocked
COMMIT;

```

Data Output Messages Notifications

	base character varying	quote_ character varying	symbol character varying	old_symbol character varying
1	USD	JPY	GBPUSD	[null]
2	USD	JPY	GBPUSD	[null]

Total rows: 2 of 2 Query complete 00:00:00.041 Ln 27, Col 8

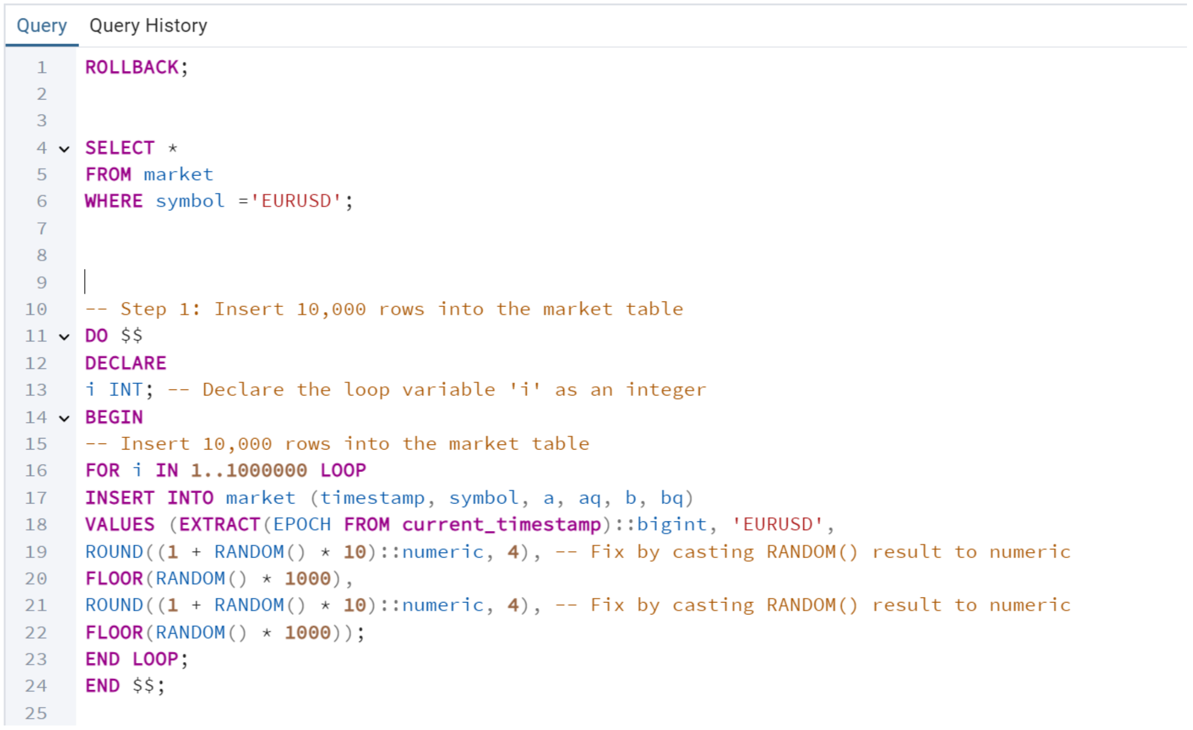
Figure 6: Code for SERIALIZABLE

4 Lab Topic III: Performance Comparison of Different Isolation Level

4.1 Step-by-Step Instructions for Running the Code and Analyzing Performance

1. **Generating the Data:**

   - Begin by executing the code provided in `Figure` 7. This will insert **1,000,000 random rows** into the `market` table with the symbol "EURUSD".



```sql
ROLLBACK;


SELECT *
FROM market
WHERE symbol ='EURUSD';



-- Step 1: Insert 10,000 rows into the market table
DO $$
DECLARE
i INT; -- Declare the loop variable 'i' as an integer
BEGIN
-- Insert 10,000 rows into the market table
FOR i IN 1..1000000 LOOP
INSERT INTO market (timestamp, symbol, a, aq, b, bq)
VALUES (EXTRACT(EPOCH FROM current_timestamp)::bigint, 'EURUSD',
ROUND((1 + RANDOM() * 10)::numeric, 4), -- Fix by casting RANDOM() result to numeric
FLOOR(RANDOM() * 1000),
ROUND((1 + RANDOM() * 10)::numeric, 4), -- Fix by casting RANDOM() result to numeric
FLOOR(RANDOM() * 1000));
END LOOP;
END $$;

```

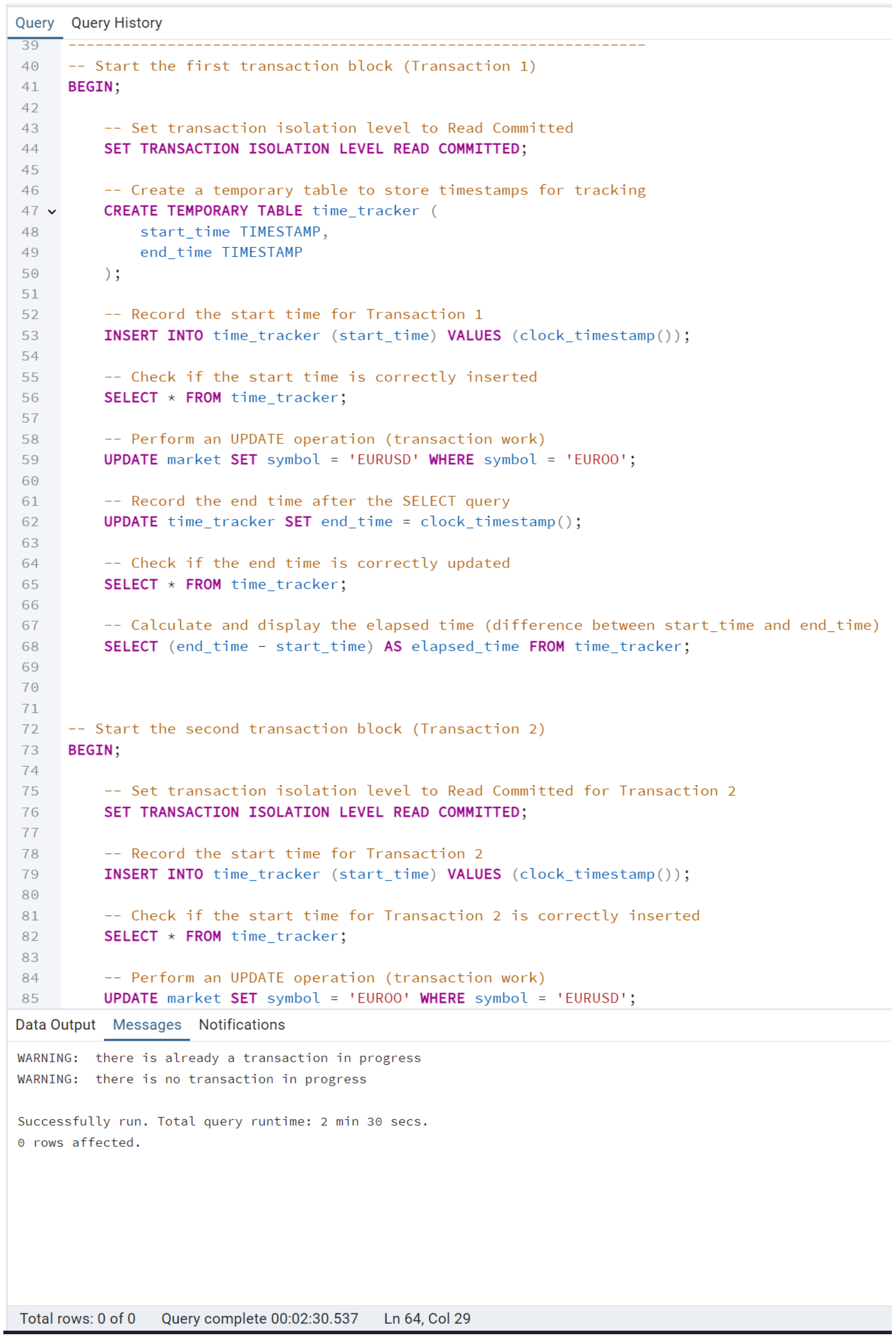
Figure 7: The generic code that helps generate the 1000000 rows in market table

2. **Verifying Data Insertion:**

   - The code in `Figure` 7 starts with a `SELECT` query, which is used to verify the successful insertion of the rows into the database. After running the insertion code, you can run the verification query to ensure the data is in place.

3. **Running the Performance Analysis:**

   - Next, refer to `Figures` 8 for the code you need to enter into the `pgAdmin 4` console.
   - Once the code is written, **RUN** the query. This will measure the **time taken** to execute the same query under different **isolation levels** (e.g., READ COMMITTED, REPEATABLE READ, SERIALIZABLE).
   - You will notice that the performance impact may seem minimal, as the query and dataset are not very large. However, it is important to imagine this scenario in a real-world application where millions of users might be accessing the system simultaneously—such as booking tickets on a train website, using services like **Alipay** or **Meituan**, or any application where a large number of users are making requests for data changes in a very short period of time.



```sql
-----------------------------------------------------------------
-- Start the first transaction block (Transaction 1)
BEGIN;

    -- Set transaction isolation level to Read Committed
    SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

    -- Create a temporary table to store timestamps for tracking
    CREATE TEMPORARY TABLE time_tracker (
        start_time TIMESTAMP,
        end_time TIMESTAMP
    );

    -- Record the start time for Transaction 1
    INSERT INTO time_tracker (start_time) VALUES (clock_timestamp());

    -- Check if the start time is correctly inserted
    SELECT * FROM time_tracker;

    -- Perform an UPDATE operation (transaction work)
    UPDATE market SET symbol = 'EURUSD' WHERE symbol = 'EUROO';

    -- Record the end time after the SELECT query
    UPDATE time_tracker SET end_time = clock_timestamp();

    -- Check if the end time is correctly updated
    SELECT * FROM time_tracker;

    -- Calculate and display the elapsed time (difference between start_time and end_time)
    SELECT (end_time - start_time) AS elapsed_time FROM time_tracker;



-- Start the second transaction block (Transaction 2)
BEGIN;

    -- Set transaction isolation level to Read Committed for Transaction 2
    SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

    -- Record the start time for Transaction 2
    INSERT INTO time_tracker (start_time) VALUES (clock_timestamp());

    -- Check if the start time for Transaction 2 is correctly inserted
    SELECT * FROM time_tracker;

    -- Perform an UPDATE operation (transaction work)
    UPDATE market SET symbol = 'EUROO' WHERE symbol = 'EURUSD';
```

```
WARNING:  there is already a transaction in progress
WARNING:  there is no transaction in progress

Successfully run. Total query runtime: 2 min 30 secs.
0 rows affected.
```

```
Total rows: 0 of 0     Query complete 00:02:30.537     Ln 64, Col 29
```

Figure 8: The code demonstrates the "Read Committed" isolation level. To use a different isolation level, simply replace the keyword with another, such as SERIALIZABLE or REPEATABLE READ. (Code for this is uploaded).

Use the same code to test different isolation levels by changing the isolation type to SET TRANSACTION ISOLATION LEVEL <SERIALIZABLE OR REPEATABLE READ>.

5 Summary of Transaction and Isolation Level Experiments

- **Experiment Objective**: The goal of these experiments is to observe how different isolation levels in PostgreSQL impact transaction behavior. We focused on **Read Committed**, **Repeatable Read**, and **Serializable** isolation levels, comparing how data consistency is maintained during concurrent transactions.
- ***Setup***:

  - **Data Population**: The market table was populated with random data for simulation.
  - **Isolation Levels Tested**: We tested three isolation levels:
  - **Read Committed**: Allows reading the most recent committed data. Other transactions may update data in between.
  - **Repeatable Read**: Ensures that the same data is seen throughout the transaction, but new rows can be inserted.
  - **Serializable**: Provides the highest level of isolation, treating all transactions as if they are executed sequentially.
- ***Key Observations***:
  - **Read Committed**: Transaction behavior was more prone to non-repeatable reads. If one transaction updates data (e.g., a seat booking), another transaction may see the updated value upon refresh.
  - **Repeatable Read**: Transactions were consistent throughout, preventing other transactions from changing the data being read. However, new data inserts (phantoms) could appear.
  - **Serializable**: The highest level of consistency, where transactions are isolated to the point that no other transactions can interfere, ensuring complete data integrity but at the cost of performance.
- ***Performance***: While the differences in performance between the isolation levels were not significant for small datasets, **Serializable** showed the highest overhead due to complete isolation of transactions. Larger datasets or high concurrency scenarios (e.g., millions of requests) would show more noticeable performance differences.
- ***Real-World Example***:
  - **Read Committed**: Similar to an online store where prices may change between browsing and purchasing.
  - **Repeatable Read**: Like a booking system where the same seat will remain available for a user until they complete the booking.
  - **Serializable**: Similar to a bank where multiple users cannot simultaneously withdraw money from the same account.