Project Part 2

Done by: Group 18

- Pedro Santos 53129
- Rafael Oliveira 52838
- Rodrigo Serra 57313
- Rui Roque 57588

1. Introduction

For this part of the project, the goals were to learn how to code complex operations in a procedural language and then use them in concurrency anomaly experiments and learn how to identify, demonstrate and solve concurrency anomalies.

In our project we access the presence of two possible concurrency anomalies depending on the order in which the queries are executed. To do this a procedural language is used (PL/PGSQL), which allows us to create functions for various queries and call them when needed. These concurrence anomalies were then solved by the use of locks.

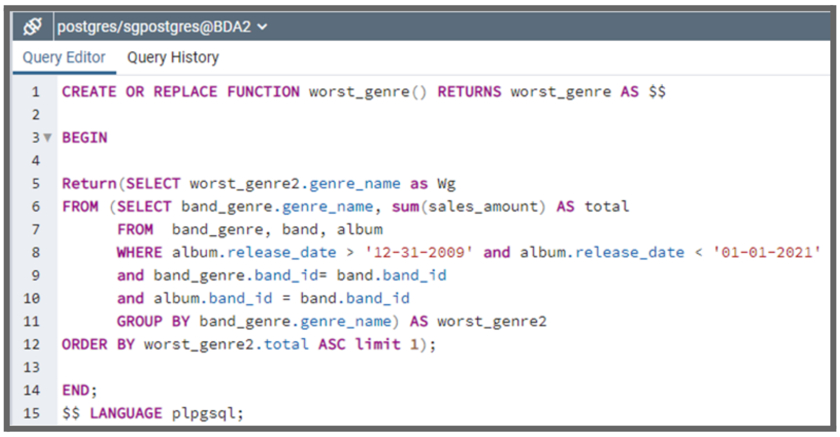
2. Implementation of our complex operations in PL/pgSQL

To facilitate concurrency testing, four functions were created based on our queries created in part 1 using a procedural language that allows the creation of complex functions in PostgreSQL.

Query 1

The first query was splitted in two functions:

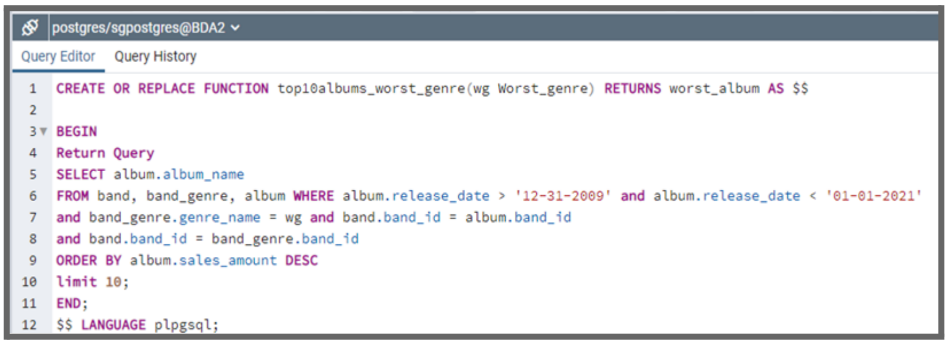
- The first function (worst_genre()) doesn't take arguments and was created to return the worst genre in the discussed decade which is later used in the second function;



```sql
CREATE OR REPLACE FUNCTION worst_genre() RETURNS worst_genre AS $$

BEGIN

Return(SELECT worst_genre2.genre_name as Wg
FROM (SELECT band_genre.genre_name, sum(sales_amount) AS total
      FROM  band_genre, band, album
      WHERE album.release_date > '12-31-2009' and album.release_date < '01-01-2021'
      and band_genre.band_id= band.band_id
      and album.band_id = band.band_id
      GROUP BY band_genre.genre_name) AS worst_genre2
ORDER BY worst_genre2.total ASC limit 1);

END;
$$ LANGUAGE plpgsql;
```

Figure 1: Worst Genre Function

- The second function (top10albums_worst_genre(text)) uses the text returned by the previous function and returns a table with the name of the albums that belong to the worst genre and the sales amount of each of those albums.

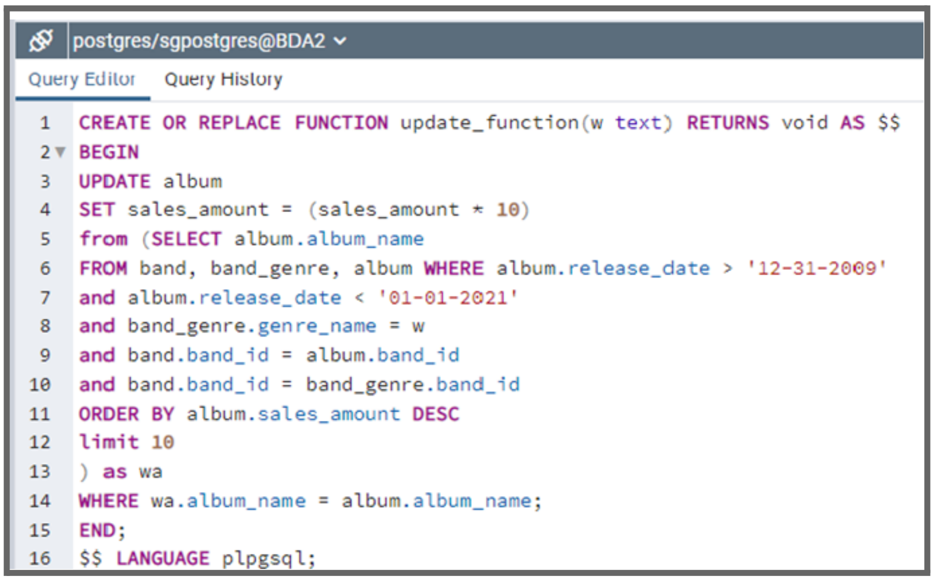


```sql
CREATE OR REPLACE FUNCTION top10albums_worst_genre(wg Worst_genre) RETURNS worst_album AS $$

BEGIN
Return Query
SELECT album.album_name
FROM band, band_genre, album WHERE album.release_date > '12-31-2009' and album.release_date < '01-01-2021'
and band_genre.genre_name = wg and band.band_id = album.band_id
and band.band_id = band_genre.band_id
ORDER BY album.sales_amount DESC
limit 10;
END;
$$ LANGUAGE plpgsql;
```

Figure 2: Function of Top 10 Albums of the Worst Genre

Query 2

For the second query only one function was created (update_function(text)), this function uses the text returned by the first function of query one and updates the table albums by multiplying the sales amount by 10.

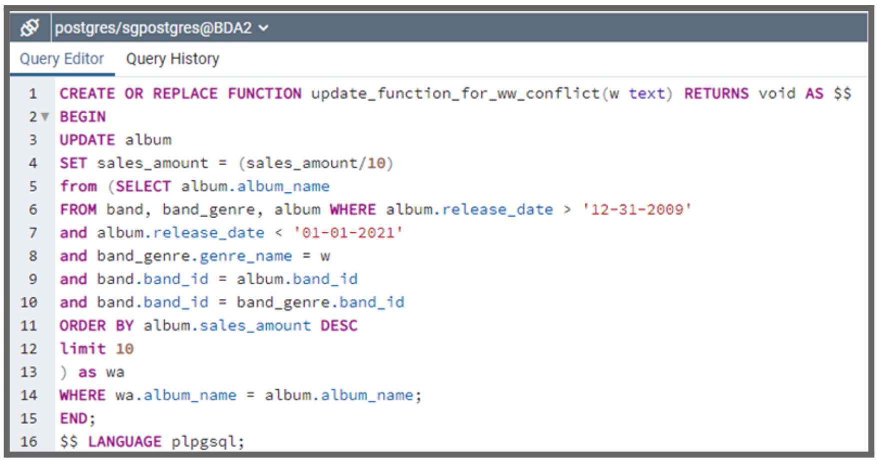

postgres/sgpostgres@BDA2

Query Editor Query History

```sql
CREATE OR REPLACE FUNCTION update_function(w text) RETURNS void AS $$
BEGIN
UPDATE album
SET sales_amount = (sales_amount * 10)
from (SELECT album.album_name
FROM band, band_genre, album WHERE album.release_date > '12-31-2009'
and album.release_date < '01-01-2021'
and band_genre.genre_name = w
and band.band_id = album.band_id
and band.band_id = band_genre.band_id
ORDER BY album.sales_amount DESC
limit 10
) as wa
WHERE wa.album_name = album.album_name;
END;
$$ LANGUAGE plpgsql;
```

Figure 3: Update Function

Based on the second query, another function was created so that some anomalies could be tested (lost update). This function also uses the text from the first function of query one and updates the table albums but this time instead of multiplying by 10, it divides by 10.


postgres/sgpostgres@BDA2

Query Editor Query History

```sql
CREATE OR REPLACE FUNCTION update_function_for_ww_conflict(w text) RETURNS void AS $$
BEGIN
UPDATE album
SET sales_amount = (sales_amount/10)
from (SELECT album.album_name
FROM band, band_genre, album WHERE album.release_date > '12-31-2009'
and album.release_date < '01-01-2021'
and band_genre.genre_name = w
and band.band_id = album.band_id
and band.band_id = band_genre.band_id
ORDER BY album.sales_amount DESC
limit 10
) as wa
WHERE wa.album_name = album.album_name;
END;
$$ LANGUAGE plpgsql;
```

Figure 4: Update Function Created to Test Anomalies

3. Concurrency anomalies experiments

3.1 Types of concurrency anomalies that occur with the concurrent execution of our complex operations

After the creation of the complex functions, two types of concurrency anomalies occurred during the concurrent execution of our functions, which were:

- Unrepeatable read;
- Lost update.

A dirty read problem occurs when a transaction reads data written by a concurrent uncommitted transaction. Since the postgreSQL's Read Uncommitted mode behaves like Read Committed, only three distinct isolation levels are implemented in postgreSQL, which makes the read uncommitted used in dirty read not allowed in postgreSQL so this anomaly hasn't been tested in this project.

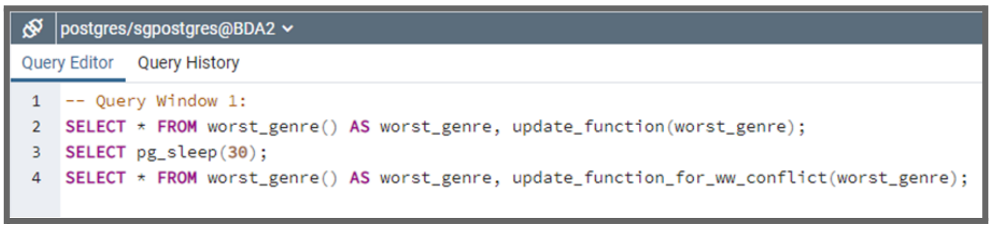
The difference between unrepeatable reads and phantom reads is that instead of reading incorrect data the data has been deleted, creating an inconsistency very similar to the process of dirty reads.

3.2 Concurrency issues

All of the issues that were created and explained below were with auto commit off and isolation level Read Committed.

Lost update

In this specific case and in order to test this problem, it was executed the first window with the following queries:



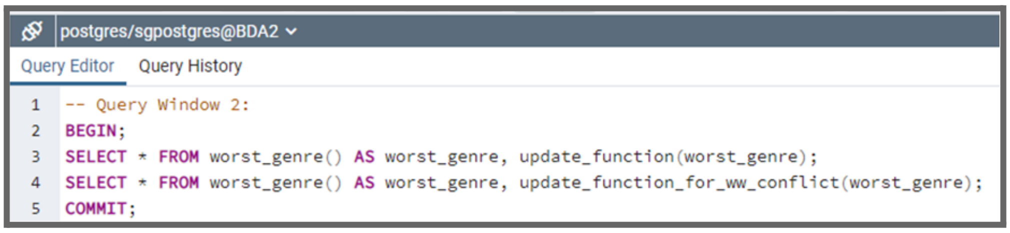
postgres/sgpostgres@BDA2

Query Editor   Query History

```
-- Query Window 1:
SELECT * FROM worst_genre() AS worst_genre, update_function(worst_genre);
SELECT pg_sleep(30);
SELECT * FROM worst_genre() AS worst_genre, update_function_for_ww_conflict(worst_genre);
```

Figure 5: Query Window 1 Lost Update

And while the first window was running, because of the **pg_sleep()** statement, a second window was executed with the following queries:



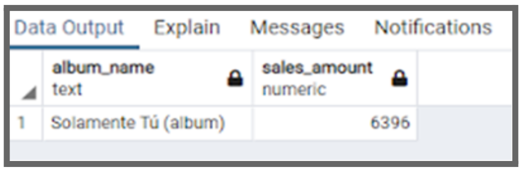
postgres/sgpostgres@BDA2

Query Editor   Query History

```
-- Query Window 2:
BEGIN;
SELECT * FROM worst_genre() AS worst_genre, update_function(worst_genre);
SELECT * FROM worst_genre() AS worst_genre, update_function_for_ww_conflict(worst_genre);
COMMIT;
```

Figure 6: Query Window 2 Lost Update

Before those queries were run, the value of the top 10 albums of the worst genre at the time, Norteño Music were the following:

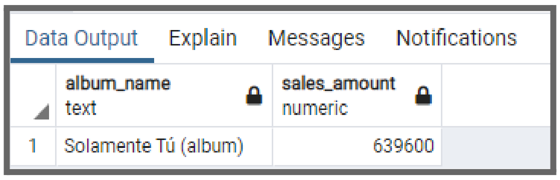


Data Output   Explain   Messages   Notifications

	album_name text	sales_amount numeric
1	Solamente Tú (album)	6396

Figure 7: Inicial Result Before Testing

If run in serial schedule, both queries would cancel themselves out, because one query multiplied by 10 and the other update query divided by 10, but in this case we lost both divisions losing both updates and therefore increasing the value of that album to 100x the original value.

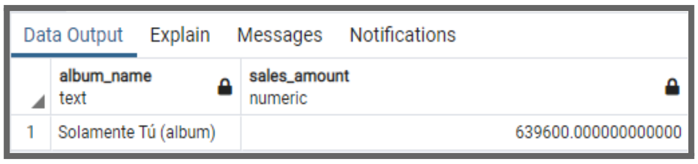


Data Output   Explain   Messages   Notifications

	album_name text	sales_amount numeric
1	Solamente Tú (album)	639600

Figure 8: Lost Update Result

After the lost update we ran both queries in serial schedule, having the expected results,as shown below:



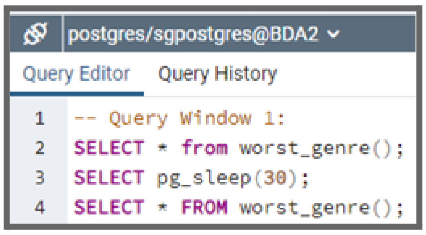
Data Output   Explain   Messages   Notifications

	album_name text	sales_amount numeric
1	Solamente Tú (album)	639600.000000000000

Figure 9 : Serial Exec After the First Update

Unrepeatable read

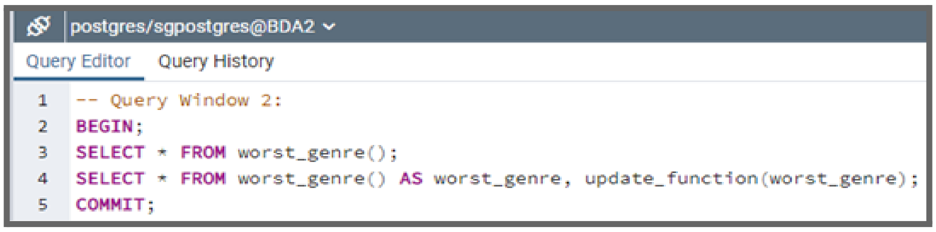
In this specific case and in order to test this problem, the first window was executed with the following queries:



```
-- Query Window 1:
SELECT * from worst_genre();
SELECT pg_sleep(30);
SELECT * FROM worst_genre();
```

Figure 10: Query Window 1 Unrepeatable Read

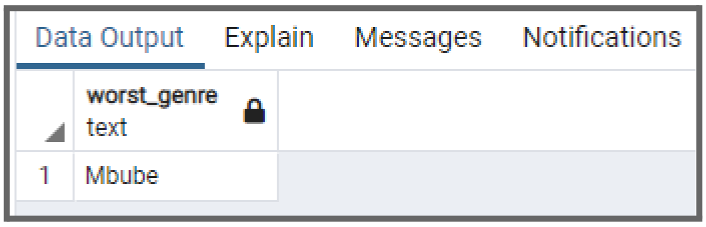
And while the first window was running, because of the **pg_sleep()** statement, a second window was executed with the following queries:



```
-- Query Window 2:
BEGIN;
SELECT * FROM worst_genre();
SELECT * FROM worst_genre() AS worst_genre, update_function(worst_genre);
COMMIT;
```

Figure 11: Query Window 2 Unrepeatable Read

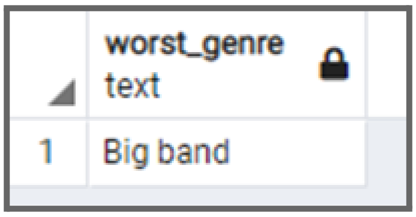
With this execution, it was possible to observe the concurrency issue of this anomaly. In this case the second read of the worst genre in the first window is different from the first read due to the fact that there is another transaction running between these two readings that modifies the value of the sales amount, making the second read of the first transaction give a different worst genre than what it was supposed to.
Before those queries were run, the value of the worst genre at the time, was the following:



	worst_genre text
1	Mbube

Figure 12: Inicial Result Before Testing

If run in serial schedule,the second read of the first transaction would read the same worst genre as the first read, but in this case the second transaction updates the value of the worst genre, changing the reading of the second read in transaction 1.



	worst_genre text
1	Big band

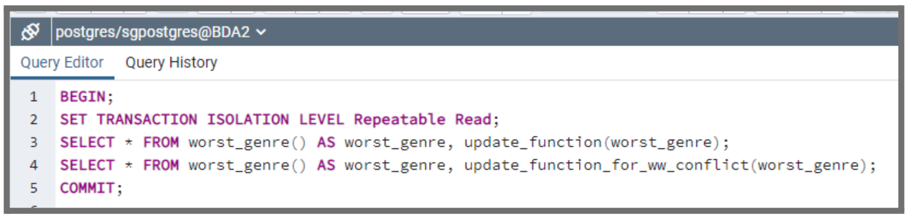
Figure 13: Unrepeatable Read Result

3.3 Solving the anomalies

Lost update

Isolation levels

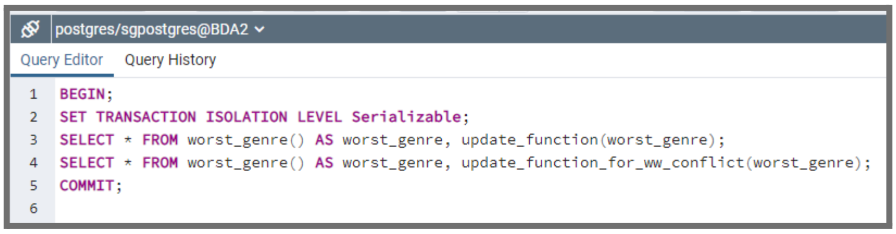
For this anomaly, after the isolation level was on read committed we tried to solve it with isolation levels. For this particular anomaly we tested setting the isolation level to the Repeatable Read level, but with the same outcome as before.



```
BEGIN;
SET TRANSACTION ISOLATION LEVEL Repeatable Read;
SELECT * FROM worst_genre() AS worst_genre, update_function(worst_genre);
SELECT * FROM worst_genre() AS worst_genre, update_function_for_ww_conflict(worst_genre);
COMMIT;
```

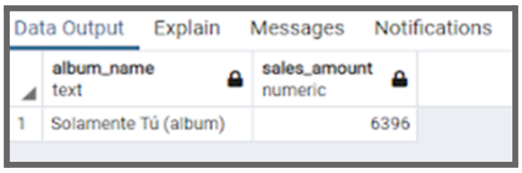
Figure 14: Lost Update with Repeatable Read Isolation Level

After that we decided to set the isolation to Serializable that gave us the result that we wanted. Although the Serializable isolation level will hurt performance in this case, no other isolation level below it was enough to make it work, and so we have to use this level.



```sql
BEGIN;
SET TRANSACTION ISOLATION LEVEL Serializable;
SELECT * FROM worst_genre() AS worst_genre, update_function(worst_genre);
SELECT * FROM worst_genre() AS worst_genre, update_function_for_ww_conflict(worst_genre);
COMMIT;
```

Figure 15: Lost Update with Serializable Isolation Level

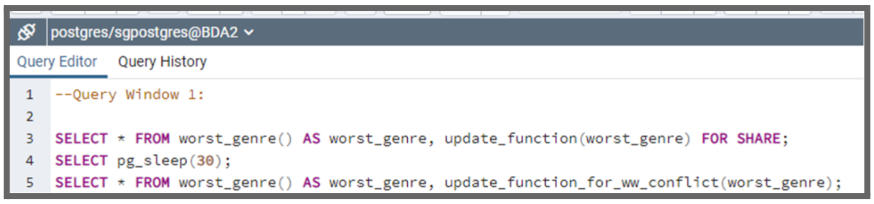


	album_name text	sales_amount numeric
1	Solamente Tú (album)	6396

Figure 16: Result of Using Serializable Isolation Level With Lost Update

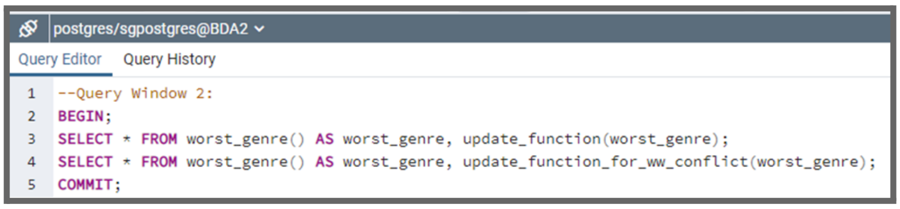
After that we tried to also solve the problem with locks.

Locks



```sql
--Query Window 1:

SELECT * FROM worst_genre() AS worst_genre, update_function(worst_genre) FOR SHARE;
SELECT pg_sleep(30);
SELECT * FROM worst_genre() AS worst_genre, update_function_for_ww_conflict(worst_genre);
```

Figure 17: Queries of Transaction 1 With Lock to Solve the Lost Update Anomaly



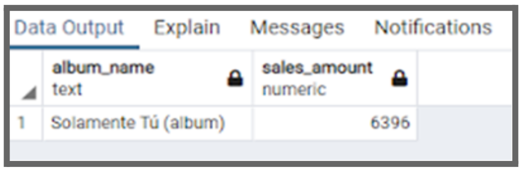
```sql
--Query Window 2:
BEGIN;
SELECT * FROM worst_genre() AS worst_genre, update_function(worst_genre);
SELECT * FROM worst_genre() AS worst_genre, update_function_for_ww_conflict(worst_genre);
COMMIT;
```

Figure 18: Queries of Transaction 2 With Lock

As can be seen in the code above, the simple addition of a shared lock (**FOR SHARE**) to the first **SELECT** in the first query window solves the concurrence anomaly. This is used because it acquires a shared lock that makes it so that transaction 2 can't perform an **UPDATE**, **DELETE**, **SELECT FOR UPDATE** or **SELECT FOR NO KEY UPDATE**; This lock is stronger than what would happen with the command **FOR KEY SHARE**, which would not solve the anomaly because it doesn't block updates that don't change the key value. It is also better than an exclusive lock as it allows for some kinds of concurrency, which can help with the efficiency of our queries.

Results

After solving the anomaly with locks, the queries above were executed and it was possible to observe that the sales amount value returned was the expected.



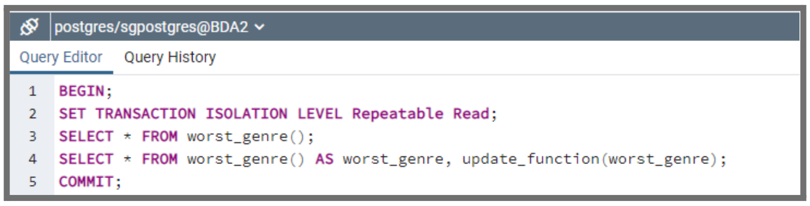
	album_name text	sales_amount numeric
1	Solamente Tú (album)	6396

Figure 19: Result of Using Locks With Lost Update

Unrepeatable read

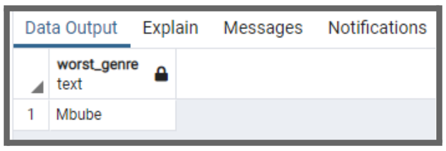
Isolation levels

For this anomaly, after the isolation level was on read committed we tried to solve it with isolation levels. For this particular anomaly we tested setting the isolation level to the Repeatable Read level, which resolved our issue as expected. The Serializable level would have also worked but that would make every transaction run in a non parallel execution which would hurt performance. Since the Unrepeatable Read level was enough, we decided that was the best isolation level to deal with this issue.



```sql
BEGIN;
SET TRANSACTION ISOLATION LEVEL Repeatable Read;
SELECT * FROM worst_genre();
SELECT * FROM worst_genre() AS worst_genre, update_function(worst_genre);
COMMIT;
```

Figure 20: Unrepeatable Read with Repeatable ReadIsolation Level

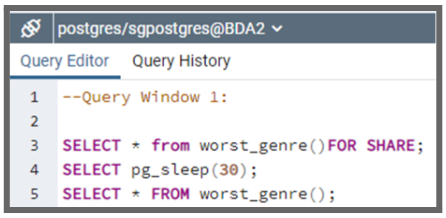


```
  worst_genre
  text
1 Mbube
```

Figure 21: Result of Using Repeatable Read Isolation Level With Unrepeatable Read

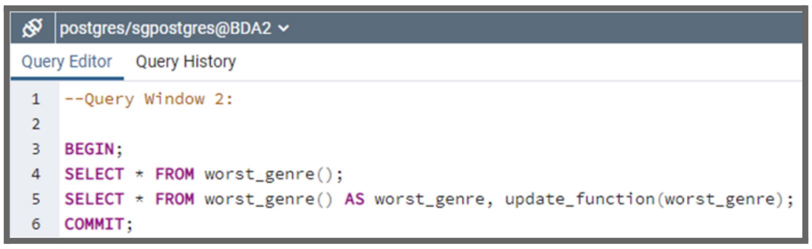
After that we tried to also solve the problem with locks.

Locks



```sql
--Query Window 1:

SELECT * from worst_genre()FOR SHARE;
SELECT pg_sleep(30);
SELECT * FROM worst_genre();
```

Figure 22: Queries of Transaction 1 With Lock to Solve the Unrepeatable Read Anomaly



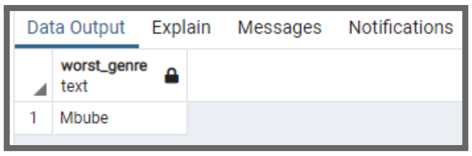
```sql
--Query Window 2:

BEGIN;
SELECT * FROM worst_genre();
SELECT * FROM worst_genre() AS worst_genre, update_function(worst_genre);
COMMIT;
```

Figure 23: Queries of Transaction 2

Just as in the previous anomaly a shared lock sufficed to solve the anomaly. To do this a **FOR SHARE** was added to the first **SELECT** of the first query window which initiated the function **worst_genre()**. This lock blocks the function **update_function()** from trying to perform an **UPDATE** at the same time as the **SELECT** is being run, which gets rid of the unrepeatable read problem.

Results

After solving the anomaly with locks, the queries above were executed and it was possible to observe that the worst genre value returned was the expected.



```
  worst_genre
  text
1 Mbube
```

Figure 24: Result of Using Locks With Unrepeatable Read