Faculty of Sciences of the University of Lisbon

Database Technologies - Report 2

Concurrency Anomalies in PostgreSQL

Group 5 - António Rebelo · Duarte Balata · Filipa Serrano · Marco Mercier

taught by
Professor Cátia Pesquita

November 16, 2020

1 Introduction

For this part of the project, the goal was to study possible concurrency anomalies and how to solve them using multiversion concurrency control (MVCC) and explicit locks.

The SQL standard defines three consistency anomalies [3]: Dirty Reads, Non-repeatable Reads and Phantom Reads. A **dirty read** [1] manifests when a transaction is able to read uncommitted changes of some other concurrent transaction; a **non-repeatable read** happens when consecutive reads retrieve different results due to a concurring transaction that has changed the read values; and a **phantom read** [2] occurs when two queries, ran successively, within the same transaction, display different sets of results due to the insertion or deletion of one or more rows of data between read statements.

A lesser-known phenomenon is the **lost updates** anomaly: if two transactions want to change the same objects, the second transaction will overwrite the first one, therefore losing the first transaction update.

PostgreSQL does not allow the occurrence of dirty reads [3], therefore this type of error was not studied in this project. All the other types were successfully replicated and solved.

The first step of this part of the project was to "translate" the queries from the previous part of the project to PL/pgSQL, which is a procedural language that allows the creation of complex functions in PostgreSQL. This is the language used to facilitate concurrency testing.

2 Query 1

The first query was already split, since a VIEW was used. It was slightly altered, so instead of creating a view or a temporary table, the result from the first part (the band id with the maximum average album length) was saved into a new variable, which was then used for the second part of the query, which selects the artists in that band, as in appendix 5. The function created was also generalized, receiving an argument with the genre intended, instead of assuming "Indie rock", like the procedure in SQL. For the purpose of this part of the project, the genre "Indie rock" was still the one used, so it was easier to evaluate the results.

2.1 Phantom Read

To test for this issue using the first query, a transaction was ran where a dataset was queried twice to display the members of the Indie Rock band with the maximum average album length. As there is a time window between the execution of each query, due to the presence of a PG_SLEEP statement, a new Indie Rock band was inserted between reads, by adding this genre to a band previously present in the dataset as follows:

```
T1: BEGIN ;
    SELECT * FROM QUERY1_genre('Indie Rock');
    SELECT PG_SLEEP(3)
    SELECT * FROM QUERY1_genre('Indie Rock');
    COMMIT;
```

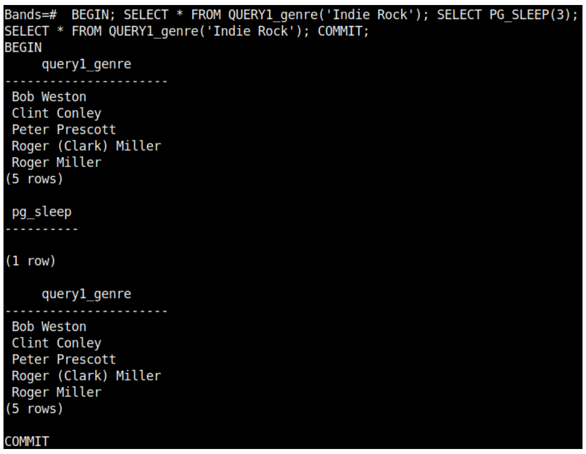
```
T2: BEGIN;
    SELECT insert_band_genre(7705,45);
    COMMIT;
```

Since this band presents a higher average album length than any of the Indie Rock genre bands previously present on the dataset,the list of its members becomes the expected result of the query, causing the second query of the transaction to display a different set of results than the first one.

Listing 1: Transaction schema for Phantom Read experiment. Time goes from left to right.

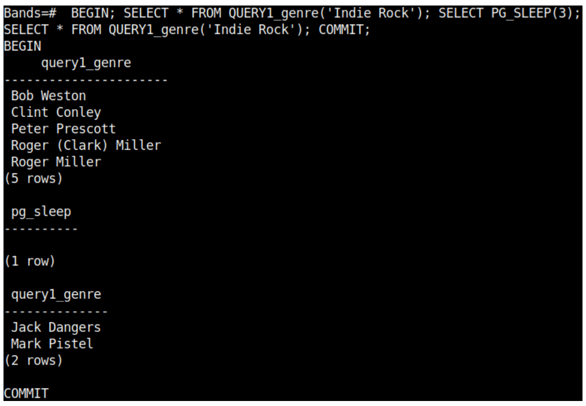
```
T1: QUERY                                QUERY C
T2: READ     INSERT band_genre C
```

This concurrency problem is illustrated Figures 1 and 2.



```
Bands=#  BEGIN; SELECT * FROM QUERY1_genre('Indie Rock'); SELECT PG_SLEEP(3);
SELECT * FROM QUERY1_genre('Indie Rock'); COMMIT;
BEGIN
     query1_genre
---------------------
 Bob Weston
 Clint Conley
 Peter Prescott
 Roger (Clark) Miller
 Roger Miller
(5 rows)

 pg_sleep
----------

(1 row)

     query1_genre
---------------------
 Bob Weston
 Clint Conley
 Peter Prescott
 Roger (Clark) Miller
 Roger Miller
(5 rows)

COMMIT
```

Figure 1: Execution of T1 without concurrency problems.



```
Bands=#  BEGIN; SELECT * FROM QUERY1_genre('Indie Rock'); SELECT PG_SLEEP(3);
SELECT * FROM QUERY1_genre('Indie Rock'); COMMIT;
BEGIN
     query1_genre
---------------------
 Bob Weston
 Clint Conley
 Peter Prescott
 Roger (Clark) Miller
 Roger Miller
(5 rows)

 pg_sleep
----------

(1 row)

 query1_genre
--------------
 Jack Dangers
 Mark Pistel
(2 rows)

COMMIT
```

Figure 2: Execution of T1 displaying a Phantom Read problem caused by the insertion of a new band between queries.

Solving Phantom Read Phenomena

The main approach typically used to solve the Phantom Read problem is to serialize the execution of the transactions. A serializable execution is defined to be an execution of the operations of concurrently executing SQL transactions that produces the same effect as one of the possible serial executions of the transactions, meaning that each SQL transaction executes to completion before the next SQL-transaction begins [4].

The serialization of the transactions can be implemented by writing "TRANSACTION ISOLATION LEVEL REPEATABLE READ" next to the BEGIN statement since this isolation mode has the same behaviour as SERIALIZABLE in PostgreSQL towards phantom reads. The code for the solved transactions for this and all concurrency problems are shown in appendix 8.

Another way to prevent the Phantom Read phenomenon is to use explicit locks. Since the conflicting transaction was caused by the association of a new band with the genre Indie Rock on the "Bands_Genres" table, the problem can be solved by applying a lock on this table that prevents a change in the relations, by inserting or deleting table rows. Multiple locking modes were tested in order to find the most granular solution that could solve the Phantom Read problem, without compromising performance too much. Using this trial and error approach, the SHARE MODE lock was found to be the least conflicting lock solution capable of solving this issue. This lock can be implemented into the transactions, by writing "LOCK TABLE Bands_Genres IN SHARE MODE" after the BEGIN statement, before the execution of the first query.

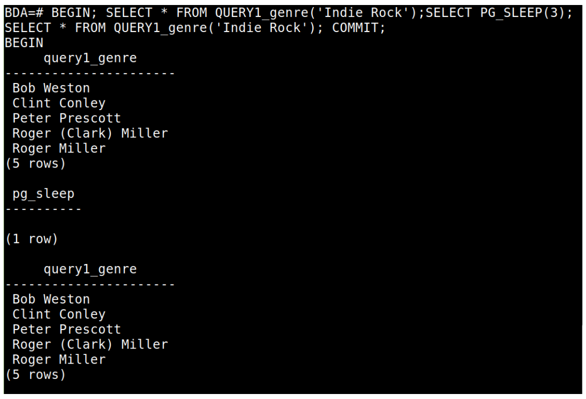
By adding this statement to T1, the row insertion in T2 is withheld until T1 is complete, thus preventing the insertion of a new row halfway through the first transaction, avoiding the concurrency issue. Further testing revealed that 'FOR UPDATE', a row-level lock, is also a viable solution for this problem, when inserted in the query function, at the end of the SELECT statement that retrieves the IDs of the Bands belonging to the genre Indie Rock. Even though, this more specific type of approach can be beneficial in some cases, it also presents more risks when compared to full table locks, since it implies that it is known in advance what rows are subject to be modified.

2.2 Non-repeatable Read

To portray a Non-repeatable Read error, two queries that are exactly the same are run sequentially, but in between an update is made so that the results differ. In this case an album belonging to an "Indie Rock" band has 5000 minutes added to its length:

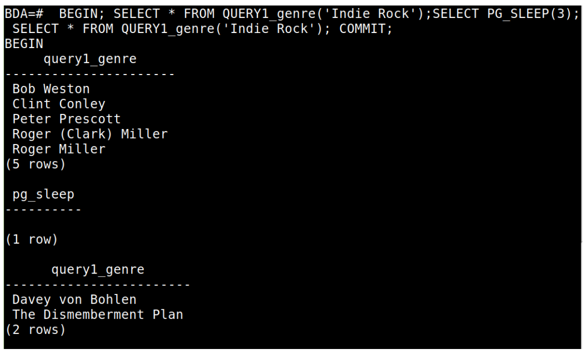
```
T1: BEGIN;
    SELECT * FROM QUERY1_genre('Indie Rock');
    SELECT PG_SLEEP(3);
    SELECT * FROM QUERY1_genre('Indie Rock');
    COMMIT;
```

```
T2: BEGIN;
    SELECT add_album_length(1,5000);
    COMMIT;
```



```
BDA=# BEGIN; SELECT * FROM QUERY1_genre('Indie Rock');SELECT PG_SLEEP(3);
SELECT * FROM QUERY1_genre('Indie Rock'); COMMIT;
BEGIN
     query1_genre
---------------------
 Bob Weston
 Clint Conley
 Peter Prescott
 Roger (Clark) Miller
 Roger Miller
(5 rows)

 pg_sleep
----------

(1 row)

     query1_genre
---------------------
 Bob Weston
 Clint Conley
 Peter Prescott
 Roger (Clark) Miller
 Roger Miller
(5 rows)
```

Figure 3: Execution of T1 without concurrency problems.



```
BDA=#  BEGIN; SELECT * FROM QUERY1_genre('Indie Rock');SELECT PG_SLEEP(3);
 SELECT * FROM QUERY1_genre('Indie Rock'); COMMIT;
BEGIN
     query1_genre
---------------------
 Bob Weston
 Clint Conley
 Peter Prescott
 Roger (Clark) Miller
 Roger Miller
(5 rows)

 pg_sleep
----------

(1 row)

      query1_genre
-----------------------
 Davey von Bohlen
 The Dismemberment Plan
(2 rows)
```

Figure 4: Execution of T1 displaying a Non-repeatable problem caused by the update of an album length.

Listing 2: Transaction schema for Non-repeatable Read experiment. Time goes from left to right.

```
T1: QUERY                                 QUERY C
T2: READ    UPDATE album_length C
```

Solving Non-repeatable Read Phenomena

In order to solve the Non-repeatable Read problem there were two approaches that worked. On one hand, we could set the isolation level to repeatable read (or serializable) by writing "TRANSACTION ISOLATION LEVEL REPEATABLE READ/SERIALIZABLE" next to the BEGIN statement.

It is also possible to lock the "Albums" table in order to prevent the update from happening before the conclusion of the first transaction by writing "LOCK TABLE Albums IN SHARE MODE" after the BEGIN statement, before the execution of the first query. The lock works for SHARE or any higher lock level, but not for any lower.

2.3 Lost Update

To support the Lost Update phenomena experiment, two updates were done concurrently concerning the same row as follows:

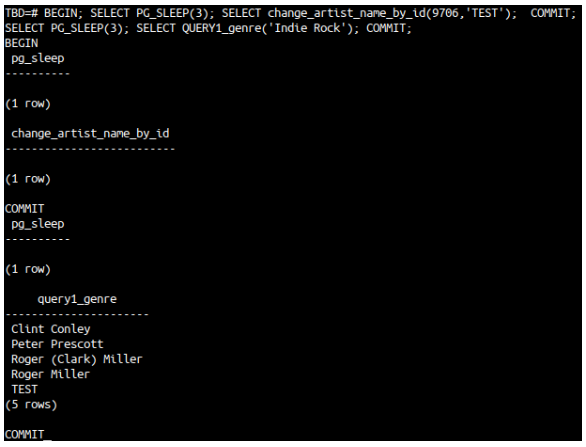
```
T1: BEGIN; SELECT PG_SLEEP(3);
    SELECT change_artist_name_by_id
    (9706,'TEST');
    COMMIT; SELECT PG_SLEEP(3);
    SELECT * FROM QUERY1_genre('Indie Rock');
    COMMIT;
```

```
T2: BEGIN; SELECT PG_SLEEP(3);
    SELECT change_artist_name_by_id
    (9706,'BOB THE DRAG QUEEN');
    COMMIT; SELECT PG_SLEEP(3);
    SELECT * FROM QUERY1_genre('Indie Rock');
    COMMIT;
```

The first transaction reads the id and updates the artist name in the "Belongs" table while the second transaction reads the id and updates the same artist name. T1 commits and after that T2 commits, overwriting T1 as a Lost Update, as depicted in Figures 5 and 6. The result of the query in both transactions show the artist name as ’BOB THE DRAG QUEEN’. The artist name being updated belongs to the band retrieved by the query and the pg_sleep functions provide time to begin both transactions slightly out of phase.

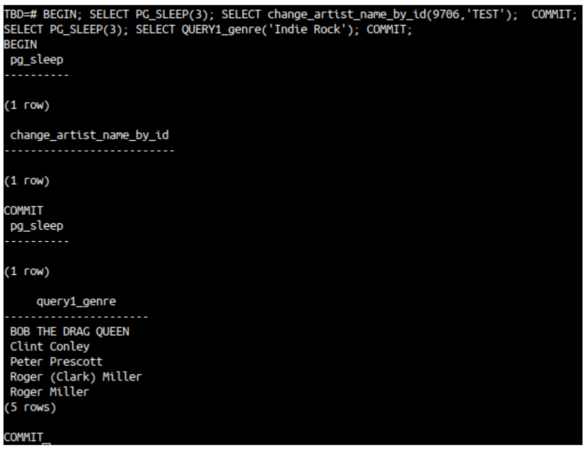
Listing 3: Transaction schema for Lost Update experiment. Time goes from left to right.

```
T1: READ    UPDATE artist_name  C          QUERY C
T2:     READ    UPDATE artist_name C       QUERY C
```



```
TBD=# BEGIN; SELECT PG_SLEEP(3); SELECT change_artist_name_by_id(9706,'TEST');  COMMIT;
SELECT PG_SLEEP(3); SELECT QUERY1_genre('Indie Rock'); COMMIT;
BEGIN
 pg_sleep
----------

(1 row)

 change_artist_name_by_id
--------------------------

(1 row)

COMMIT
 pg_sleep
----------

(1 row)

     query1_genre
---------------------
 Clint Conley
 Peter Prescott
 Roger (Clark) Miller
 Roger Miller
 TEST
(5 rows)

COMMIT_
```

Figure 5: Execution of T1 without concurrency problems.



```
TBD=# BEGIN; SELECT PG_SLEEP(3); SELECT change_artist_name_by_id(9706,'TEST');  COMMIT;
SELECT PG_SLEEP(3); SELECT QUERY1_genre('Indie Rock'); COMMIT;
BEGIN
 pg_sleep
----------

(1 row)

 change_artist_name_by_id
--------------------------

(1 row)

COMMIT
 pg_sleep
----------

(1 row)

     query1_genre
---------------------
 BOB THE DRAG QUEEN
 Clint Conley
 Peter Prescott
 Roger (Clark) Miller
 Roger Miller
(5 rows)

COMMIT_
```

Figure 6: Execution of T1 displaying the Lost Update phenomena caused by concurrent update of the same artist name.

Solving Lost Update Phenomena

The first approach to solve the Lost Update problem was to change the isolation level to REPEATABLE READ, preventing the transaction to see rows committed before the first query or data-modification statement was executed in this transaction. Setting the transaction isolation level in the BEGIN statement, as showed in appendix 8, provided an output error stating it could not serialize access due to concurrent update.

The next step was to use explicit locks to prevent the update of the table "Belongs" between concurrent transactions. Using the SHARE mode row lock stating "LOCK TABLE Belongs IN SHARE MODE" after the BEGIN statement in both transactions resulted in an error shown in the output stating that a deadlock was detected since both processes want to access resources that would be mutually locked by each other.

In both approaches, the result of T1 is showed in the query retrieval, eliminating the Lost Update phenomena.

3 Query 2

The second query was also split into two parts. It was also generalized, receiving an argument with the rank intended, instead of just retrieving the results with rank=1. Firstly, the different sets of decade, genre and respective ranking and number of albums are saved into a cursor. Secondly, the cursor is looped trough and returns the sets which correspond to the ranking requested, as demonstrated in appendix 6.

3.1 Phantom Read

Using Query 2, it was possible to recreate the Phantom Read problem by running the read query twice within the same transaction block (T1), while simultaneously performing a transaction (T2) that inserts a new album into the second most popular genre of the 50's decade.

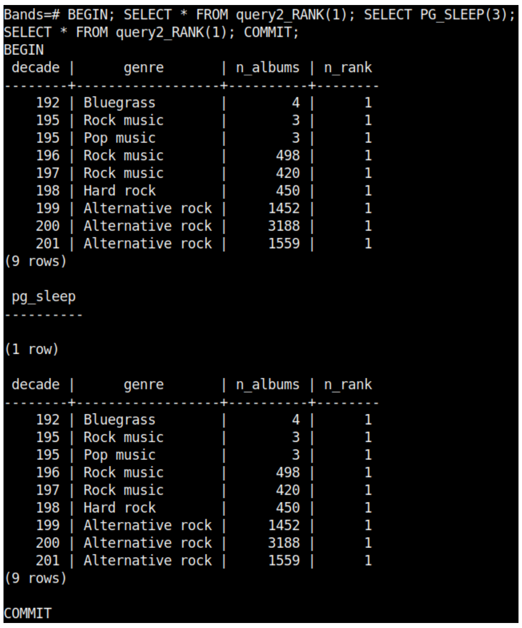
```
T1: BEGIN ;
    SELECT * FROM query2_RANK(1);
    PG_SLEEP(3)
    SELECT * FROM query2_RANK(1);
    COMMIT;
```

```
T2: BEGIN;
    SELECT insert_album( 408 , 'TESTE',
    '1950-10-10', 0, 100.0, 'Test album');
    COMMIT;
```

This second transaction, makes the previously second most popular genre of 50's tie for first place, altering the expected results of the query. For that reason, the popular genres displayed for each run of the query, inside the same transaction, will differ.

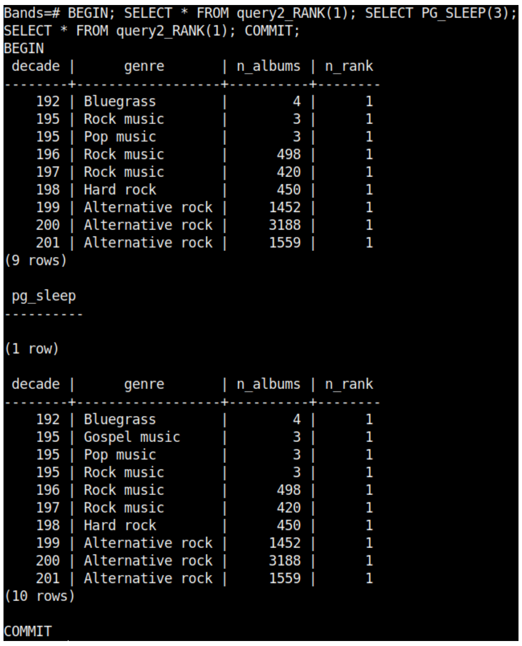
Listing 4: Transaction schema for Phantom Read experiment. Time goes from left to right.

```
T1: QUERY                                  QUERY C
T2: READ     INSERT album C
```



```
Bands=# BEGIN; SELECT * FROM query2_RANK(1); SELECT PG_SLEEP(3);
SELECT * FROM query2_RANK(1); COMMIT;
BEGIN
 decade |       genre       | n_albums | n_rank
--------+-------------------+----------+--------
    192 | Bluegrass         |        4 |      1
    195 | Rock music        |        3 |      1
    195 | Pop music         |        3 |      1
    196 | Rock music        |      498 |      1
    197 | Rock music        |      420 |      1
    198 | Hard rock         |      450 |      1
    199 | Alternative rock  |     1452 |      1
    200 | Alternative rock  |     3188 |      1
    201 | Alternative rock  |     1559 |      1
(9 rows)

 pg_sleep
----------

(1 row)

 decade |       genre       | n_albums | n_rank
--------+-------------------+----------+--------
    192 | Bluegrass         |        4 |      1
    195 | Rock music        |        3 |      1
    195 | Pop music         |        3 |      1
    196 | Rock music        |      498 |      1
    197 | Rock music        |      420 |      1
    198 | Hard rock         |      450 |      1
    199 | Alternative rock  |     1452 |      1
    200 | Alternative rock  |     3188 |      1
    201 | Alternative rock  |     1559 |      1
(9 rows)

COMMIT
```

Figure 7: Execution of T1 without concurrency problems.



```
Bands=# BEGIN; SELECT * FROM query2_RANK(1); SELECT PG_SLEEP(3);
SELECT * FROM query2_RANK(1); COMMIT;
BEGIN
 decade |       genre       | n_albums | n_rank
--------+-------------------+----------+--------
    192 | Bluegrass         |        4 |      1
    195 | Rock music        |        3 |      1
    195 | Pop music         |        3 |      1
    196 | Rock music        |      498 |      1
    197 | Rock music        |      420 |      1
    198 | Hard rock         |      450 |      1
    199 | Alternative rock  |     1452 |      1
    200 | Alternative rock  |     3188 |      1
    201 | Alternative rock  |     1559 |      1
(9 rows)

 pg_sleep
----------

(1 row)

 decade |       genre       | n_albums | n_rank
--------+-------------------+----------+--------
    192 | Bluegrass         |        4 |      1
    195 | Gospel music      |        3 |      1
    195 | Pop music         |        3 |      1
    195 | Rock music        |        3 |      1
    196 | Rock music        |      498 |      1
    197 | Rock music        |      420 |      1
    198 | Hard rock         |      450 |      1
    199 | Alternative rock  |     1452 |      1
    200 | Alternative rock  |     3188 |      1
    201 | Alternative rock  |     1559 |      1
(10 rows)

COMMIT
```

Figure 8: Execution of T1 displaying a Phantom Read problem caused by the insertion of a new album in the 50s decade between queries.

Figures 7 and 8 display the results for runs of the first transaction under normal circumstances and with a Phantom Read problem, respectively.

As can be seen in the outputs, the insertion of a new album into the 'Gospel music' genre halfway through the first transaction, makes this genre the most popular of the 50's decade (tied with Rock music and Pop music) with 3 albums released, causing the results of the second query to differ from the first one.

Solving Phantom Read Phenomena

In order to solve this concurrency problem, the same solutions as presented on Query 1 were used. These solutions consisted in serializing the execution of the transactions, or explicitly locking the Albums table during the querying transaction in order to avoid the insertion or deletion of new data that would alter the query's results. The only difference was that, due to the way the query is written, the second query didn't allow for the use of 'FOR UPDATE' to perform a more specific locking, since it is based on aggregate functions that make use every row in the accessed tables.

3.2 Non-repeatable Read

The "Bluegrass" genre is part of the results yielded in the query before any changes are made. An unrepeatable read is caused by updating that genre name to TEST in between the two queries as follows:

```
T1: BEGIN;                                  T2: BEGIN;
    SELECT * FROM query2_RANK(1);               SELECT add_album_length(1,5000);
    SELECT PG_SLEEP(3);                         COMMIT;
    SELECT * FROM query2_RANK(1);
    COMMIT;
```

Listing 5: Transaction schema for Non-repeatable Read experiment. Time goes from left to right.

```
T1: QUERY                            QUERY C
T2: READ    UPDATE genre_name C
```

Solving Non-repeatable Read Phenomena

The issue here is solved exactly in the same way it was solved by the first query: either by setting the transaction level to unrepeatable read/serializable or by using a share lock or higher in the Genres table, as presented in appendix 8.

3.3 Lost Update

The Lost Update phenomena experiment was created using the model for the first query, having both transactions updating the same information and querying the result as follows:

```
T1: BEGIN; SELECT PG_SLEEP(3);
    SELECT change_genre_name_by_id(357,'BLUE');
    COMMIT; SELECT PG_SLEEP(3);
    SELECT * FROM query2_RANK(1); COMMIT;
```

```
T2: BEGIN; SELECT PG_SLEEP(3);
    SELECT change_genre_name_by_id(357,'GRASS');
    COMMIT; SELECT PG_SLEEP(3);
    SELECT * FROM query2_RANK(1); COMMIT;
```

The first transaction reads the id and updates the genre name in the "Genres" table while the second transaction reads the same id and updates the genre with another name. T1 commits and after that T2 commits, overwriting T1 as a Lost Update. The result of the query in both transactions show the genre name as 'GRASS'.

Listing 6: Transaction schema for Lost Update experiment. Time goes from left to right.

```
T1: READ    UPDATE genre_name  C           QUERY C
T2:     READ    UPDATE genre_name C        QUERY C
```

Solving Lost Update Phenomena

The Lost Update problem was solved using the same techniques used for the first query. Setting the transaction to REPEATABLE READ or higher in the BEGIN statement or locking explicitly the table "Genres" with LOCK TABLE Genres IN SHARE MODE after the begin statement provided that the update made in T1 was retrieved by both transactions, making the update of T2 ROLLBACK.

4 Conclusions

Firstly it's important to note that the locking strategies were successfully implemented and they allowed a solution for the given problems. One of the main takeaways from this report, is that setting the transaction level to SERIALIZABLE solves all of the presented concurrency phenomena. However, it's important to note that locking always takes a toll on performance (higher locking levels yield worse performance). READ COMMITED is the standard isolation level for PostgreSQL and in most cases provides a very reasonable compromise between isolation guarantees and performance. For that reason, there might be situations where non-default locking is unnecessary. As an example let's consider the case for the non-repeatable read for the second query, where the name of the genre "Bluegrass" was changed to "TEST", while there was another ongoing transaction. In a real bands database a change in genre name wouldn't really be a problem, so locking it would only decrease the performance of the database.

It's also important to notice that Multiversion Concurrency Control (MVCC) and explicit locks work differently. In the case of the MVCC model, it allows for write commits to be done even when there are concurrency problems and solves them under the hood, only presenting results that are correct. The fact that writing transactions don't block the ones which read and vice versa is one of the main advantages of MVCC. This is possible due to the Serializable Snapshot Isolation (SSI) system. Also MVCC generally provides better performance at the cost of an higher memory usage, due to the storage of snapshots. However, some situations might not require full transaction isolation control.

In these cases, it is possible to use table or row level explicit locking. Despite generally presenting poorer performances, since one must wait until one can have the desired lock in order to write and commit, it may be useful to explicitly manage particular points of conflict in the transactions. Another drawback of using explicit locks is that it can lead to deadlocks more frequently.

Overall, it is necessary to weigh the trade-offs between data consistency and concurrency to decide which is more important, considering how the database will be used.

References

[1] Vlad Mihalcea. *A beginner's guide to ACID and database transactions.* URL: `https://vladmihalcea.com/a-beginners-guide-to-acid-and-database-transactions/`.

[2] Vlad Mihalcea. *A beginner's guide to Phantom Read anomaly.* URL: `https://vladmihalcea.com/phantom-read/`.

[3] PostgreSQL. *Documentation.* URL: `https://www.postgresql.org/docs/`.

[4] Raghu Ramakrishnan and Johannes Gehrke - 3^{rd}*edition. Database Management Systems.* McGraw-Hill, 2003. ISBN: 0072465638.

5 Appendix: Function for first query in PL/pgSQL language.

```
CREATE OR REPLACE FUNCTION query1_genre(text)   -- text is the intended genre
RETURNS SETOF VARCHAR(150) AS $$
DECLARE
var_r record;
bandid int;
BEGIN
SELECT B_avg.band_id into bandid FROM   -- bandid gets the band_id with maximum average album length
(SELECT A.band_id AS band_id, AVG(A.time) AS avg_length
         FROM Albums A
         GROUP BY A.band_id
         HAVING A.band_id IN (SELECT BA.band_id
            FROM Bands BA, Bands_Genres BG, Genres G
            WHERE BA.band_id = BG.band_id
              AND BG.genre_id = G.genre_id
              AND G.genre ILIKE $1)) AS B_avg
         ORDER BY B_avg.avg_length DESC -- ordering the results
         LIMIT 1;        -- Limiting to the one output, which is the maximum average length
RETURN QUERY             -- Retrieve the artists names from that band
        SELECT DISTINCT BE.artist_name FROM Belongs BE WHERE BE.band_id = bandid;
END;
$$ LANGUAGE plpgsql;
```

6 Appendix: Function for second query in PL/pgSQL language.

```
CREATE OR REPLACE FUNCTION query2_rank(int) -- int is the intended rank
RETURNS TABLE (decade double precision, genre varchar(100), n_albums bigint, n_rank int) AS $$
DECLARE
var_r record;
Ranking CURSOR FOR      -- Creates a cursor with Decades and Genres organized by ranking
        SELECT Decade_Genre.decade AS decade, Decade_Genre.genre AS genre,
        RANK() OVER (PARTITION BY Decade_Genre.decade
        ORDER BY Decade_Genre.album_nr DESC) album_rank, Decade_Genre.album_nr as album_nr
        FROM (SELECT (EXTRACT (DECADE FROM A.release)) AS decade, G.genre AS genre,
        COUNT(A.album_id) AS album_nr
                FROM Albums A, Bands BA, Bands_Genres BG, Genres G
                WHERE A.band_id = BA.band_id
                AND BA.band_id = BG.band_id
                AND BG.genre_id = G.genre_id
                GROUP BY (EXTRACT (DECADE FROM A.release)), G.genre
                ORDER BY (EXTRACT (DECADE FROM A.release))) AS Decade_Genre;
BEGIN
OPEN Ranking;
LOOP                     -- Loop inside the Ranking cursor
FETCH Ranking into var_r;        -- Fecth the value of the cursor onto a record variable
IF NOT found THEN
        exit ;           -- Exit if the cursor ends
END IF;
IF var_r.album_rank = $1 THEN    -- If the rank is the intended as input
        n_rank := var_r.album_rank;
        decade := var_r.decade;
        genre := var_r.genre;
        n_albums := var_r.album_nr;
        RETURN NEXT;             -- Returns the table line
END IF;
END LOOP;
CLOSE Ranking;
RETURN;
END;
$$ LANGUAGE plpgsql;
```

7 Appendix: Functions for concurrency experiments in PL/pgSQL language.

```
-- Insert a new band-genre relation in Bands_Genres table
CREATE OR REPLACE FUNCTION insert_band_genre( band_id Bands.band_id%TYPE, genre_id Genres.genre_id%TYPE)
        RETURNS void AS $$
        BEGIN
        INSERT INTO Bands_Genres VALUES (band_id, genre_id);
        END;
$$ LANGUAGE plpgsql;


-- Update album length by adding a value
CREATE OR REPLACE FUNCTION add_album_length(id_ Albums.album_id%TYPE, add_ Albums.time%TYPE)
        RETURNS void AS $$
        BEGIN
        UPDATE Albums SET time = time + add_ WHERE album_id=id_;
        END;
$$ LANGUAGE plpgsql;


-- Update an artist name by getting its id first in Belongs table
CREATE OR REPLACE FUNCTION change_artist_name_by_id( id_ Belongs.artist_id%TYPE, new Belongs.artist_name%T
        RETURNS void AS $$
        BEGIN
        UPDATE Belongs SET artist_name=new WHERE artist_id=id_;
        END;
$$ LANGUAGE plpgsql;


-- Insert a new album in the Albums table
CREATE OR REPLACE FUNCTION insert_album( band_id_ albums.band_id%TYPE,
    album_name_ Albums.album_name%TYPE, release_ Albums.release%TYPE,
    sales_ Albums.sales%TYPE, time_ Albums.time%TYPE, abstract_ Albums.abstract%TYPE)
        RETURNS void AS $$
        BEGIN
        INSERT INTO Albums (band_id, album_name, release, sales, time, abstract)
        VALUES (band_id_, album_name_, release_, sales_, time_, abstract_);
        END;
$$ LANGUAGE plpgsql;


-- Update a genre name by getting its old name first in Genres table
CREATE OR REPLACE FUNCTION change_genre_name( old Genres.genre%TYPE, new Genres.genre%TYPE)
        RETURNS void AS $$
        BEGIN
        UPDATE Genres SET genre=new WHERE genre=old;
        END;
$$ LANGUAGE plpgsql;


-- Update a genre name by getting its id first in Genres table
CREATE OR REPLACE FUNCTION change_genre_name_by_id( id_ Genres.genre_id%TYPE, new Genres.genre%TYPE)
        RETURNS void AS $$
        BEGIN
        UPDATE Genres SET genre=new WHERE genre_id=id_;
        END;
$$ LANGUAGE plpgsql;
```

8 Appendix: Transactions to solve concurrency problems.

(a) Concurrent transactions to solve Phantom Read in the first query using transaction isolation:

```
T1: BEGIN TRANSACTION ISOLATION LEVEL          T2: BEGIN;
    REPEATABLE READ;                               SELECT insert_band_genre(7705,45);
    SELECT * FROM QUERY1_genre('Indie Rock');      COMMIT;
    SELECT PG_SLEEP(3);
    SELECT * FROM QUERY1_genre('Indie Rock');
    COMMIT;
```

(b) Concurrent transactions to solve Phantom Read in the first query using explicit lock of table "Bands_Genres":

```
T1: BEGIN;                                     T2: BEGIN;
    LOCK TABLE Bands_Genres IN SHARE MODE;         SELECT insert_band_genre(7705,45);
    SELECT * FROM QUERY1_genre('Indie Rock');      COMMIT;
    SELECT PG_SLEEP(3);
    SELECT * FROM QUERY1_genre('Indie Rock');
    COMMIT;
```

(c) Query1 function with FOR UPDATE lock to prevent Phantom Read on specific table rows:

```
CREATE OR REPLACE FUNCTION query1_genre_FORUPDATE(text) -- text is the intended genre
RETURNS SETOF VARCHAR(150) AS $$
DECLARE
var_r record;
bandid int;
BEGIN
SELECT B_avg.band_id into bandid FROM   -- bandid gets the band_id with maximum average album length
(SELECT A.band_id AS band_id, AVG(A.time) AS avg_length
         FROM Albums A
         GROUP BY A.band_id
         HAVING A.band_id IN (SELECT BA.band_id
            FROM Bands BA, Bands_Genres BG, Genres G
            WHERE BA.band_id = BG.band_id
              AND BG.genre_id = G.genre_id
              AND G.genre ILIKE $1 FOR UPDATE)) AS B_avg
              -- FOR UPDATE will lock the rows used by the query
              -- on "Bands", "Genres" and "Bands_Genres" tables where genre is the input
         ORDER BY B_avg.avg_length DESC -- ordering the results
         LIMIT 1;        -- Limiting to one output, which is the maximum average length
RETURN QUERY             -- Retrieve the artists names from that band
        SELECT DISTINCT BE.artist_name FROM Belongs BE WHERE BE.band_id = bandid;
END;
$$ LANGUAGE plpgsql;
```

(d) Concurrent transactions to solve Non-Repeatable Read in the first query using transaction isolation:

```
T1: BEGIN TRANSACTION ISOLATION LEVEL          T2: BEGIN;
    REPEATABLE READ;                               SELECT add_album_length(1,5000);
    SELECT * FROM QUERY1_genre('Indie Rock');      COMMIT;
    SELECT PG_SLEEP(3);
    SELECT * FROM QUERY1_genre('Indie Rock');
    COMMIT;
```

(e) Concurrent transactions to solve Non-Repeatable Read in the first query by locking explicitly the "Albums" table:

```
T1: BEGIN;                                     T2: BEGIN;
    LOCK TABLE Albums IN SHARE MODE;               SELECT add_album_length(1,5000);
    SELECT * FROM QUERY1_genre('Indie Rock');      COMMIT;
    SELECT PG_SLEEP(3);
    SELECT * FROM QUERY1_genre('Indie Rock');
    COMMIT;
```

(f) Concurrent transactions to solve Lost Update in the first query using transaction isolation:

```
T1: BEGIN TRANSACTION ISOLATION LEVEL
    REPEATABLE READ;
    SELECT PG_SLEEP(3);
    SELECT change_artist_name_by_id
    (9706,'TEST'); COMMIT;
    SELECT PG_SLEEP(3);
    SELECT * FROM QUERY1_genre('Indie Rock');
    COMMIT;
```

```
T2: BEGIN TRANSACTION ISOLATION LEVEL
    REPEATABLE READ;
    SELECT PG_SLEEP(3);
    SELECT change_artist_name_by_id
    (9706,'BOB THE DRAG QUEEN'); COMMIT;
    SELECT PG_SLEEP(3);
    SELECT * FROM QUERY1_genre('Indie Rock');
    COMMIT;
```

(g) Concurrent transactions to solve Lost Update in the first query by locking explicitly the "Belongs" table:

```
T1: BEGIN; LOCK TABLE Belongs IN SHARE MODE;
    SELECT PG_SLEEP(3);
    SELECT change_artist_name_by_id
    (9706,'TEST'); COMMIT;
    SELECT PG_SLEEP(3);
    SELECT * FROM QUERY1_genre('Indie Rock');
    COMMIT;
```

```
T2: BEGIN; LOCK TABLE Belongs IN SHARE MODE;
    SELECT PG_SLEEP(3);
    SELECT change_artist_name_by_id
    (9706,'BOB THE DRAG QUEEN'); COMMIT;
    SELECT PG_SLEEP(3);
    SELECT * FROM QUERY1_genre('Indie Rock');
    COMMIT;
```

(h) Concurrent transactions to solve Phantom Read in the second query using transaction isolation:

```
T1: BEGIN TRANSACTION ISOLATION LEVEL
    REPEATABLE READ;
    SELECT * FROM query2_RANK(1);
    SELECT PG_SLEEP(3);
    SELECT * FROM query2_RANK(1);
    COMMIT;
```

```
T2: BEGIN;
    SELECT insert_album( 408 , 'TESTE', '1950-10-10',
    0, 100.0, 'Test album');
    COMMIT;
```

(i) Concurrent transactions to solve Phantom Read in the second query using explicit lock on "Albums" table:

```
T1: BEGIN;
    LOCK TABLE Albums IN SHARE MODE;
    SELECT * FROM query2_RANK(1);
    SELECT PG_SLEEP(3);
    SELECT * FROM query2_RANK(1);
    COMMIT;
```

```
T2: BEGIN;
    SELECT insert_album( 408 , 'TESTE', '1950-10-10',
    0, 100.0, 'Test album');
    COMMIT;
```

(j) Concurrent transactions to solve Non-Repeatable Read in the second query using transaction isolation:

```
T1: BEGIN TRANSACTION ISOLATION LEVEL
    REPEATABLE READ;
    SELECT * FROM query2_RANK(1);
    SELECT PG_SLEEP(3);
    SELECT * FROM query2_RANK(1);
    COMMIT;
```

```
T2: BEGIN;
    SELECT add_album_length(1,5000);
    COMMIT;
```

(k) Concurrent transactions to solve Non-Repeatable Read in the second query explicitly locking the table "Albums":

```
T1: BEGIN;
    LOCK TABLE Albums IN SHARE MODE;
    SELECT * FROM query2_RANK(1);
    SELECT PG_SLEEP(3);
    SELECT * FROM query2_RANK(1);
    COMMIT;
```

```
T2: BEGIN;
    SELECT add_album_length(1,5000);
    COMMIT;
```

(l) Concurrent transactions to solve Lost Update in the second query using transaction isolation:

```
T1: BEGIN TRANSACTION ISOLATION LEVEL
    REPEATABLE READ;
    SELECT PG_SLEEP(3);
    SELECT change_genre_name_by_id(357,'BLUE');
    COMMIT; SELECT PG_SLEEP(3);
    SELECT * FROM query2_RANK(1); COMMIT;
```

```
T2: BEGIN TRANSACTION ISOLATION LEVEL
    REPEATABLE READ;
    SELECT PG_SLEEP(3);
    SELECT change_genre_name_by_id(357,'GRASS');
    COMMIT; SELECT PG_SLEEP(3);
    SELECT * FROM query2_RANK(1); COMMIT;
```

(m) Concurrent transactions to solve Lost Update in the second query using an explicit lock on "Genres" table:

```
T1: BEGIN;
    LOCK TABLE Genres IN SHARE MODE;
    SELECT PG_SLEEP(3);
    SELECT change_genre_name_by_id(357,'BLUE');
    COMMIT; SELECT PG_SLEEP(3);
    SELECT * FROM query2_RANK(1); COMMIT;
```

```
T2: BEGIN;
    LOCK TABLE Genres IN SHARE MODE;
    SELECT PG_SLEEP(3);
    SELECT change_genre_name_by_id(357,'GRASS');
    COMMIT; SELECT PG_SLEEP(3);
    SELECT * FROM query2_RANK(1); COMMIT;
```