Faculdade de Ciências da Universidade de Lisboa

Concurrency Anomalies in PostgreSQL

Advanced Databases - Report 2

Group 1

Daniela Vieira - 57634

João Raimundo - 57454

João Rato - 57434

Maria Vieira - 53346

Oriented by: Professor Cátia Pesquita

22 November 2021

Table of Contents

1 - Introduction

The main goal of the second part of the project is to study concurrency anomalies and how to solve them using *Multiversion Concurrency Control* (MVCC) and explicit locks. According to the *PostgreSQL* documentation, about transaction isolation, there are four main types of concurrency anomalies: ***Dirty Reads***, ***Nonrepeatable Reads***, ***Phantom Reads*** and ***Serialization*** anomaly.

Dirty Reads occur when a transaction is allowed to read a row that has been modified by another transaction which is not committed, mainly occurring when multiple transactions run at a time which are not yet committed. Similarly, the *Nonrepeatable read* occurs when a second transaction accesses the same row several times and reads different data each time. The main differences between them are that in a *Nonrepeatable read*, the data read by the second transaction is already committed, and also involves multiple reads of the same row.

A *Phantom Read* occurs when one transaction performs two queries successively, and it gets a different set of results due to the occurrence of a second transaction. Phantom reads and nonrepeatable reads are also similar, but the first above mentioned occurs when one or more rows are inserted and/or deleted between the read statements.

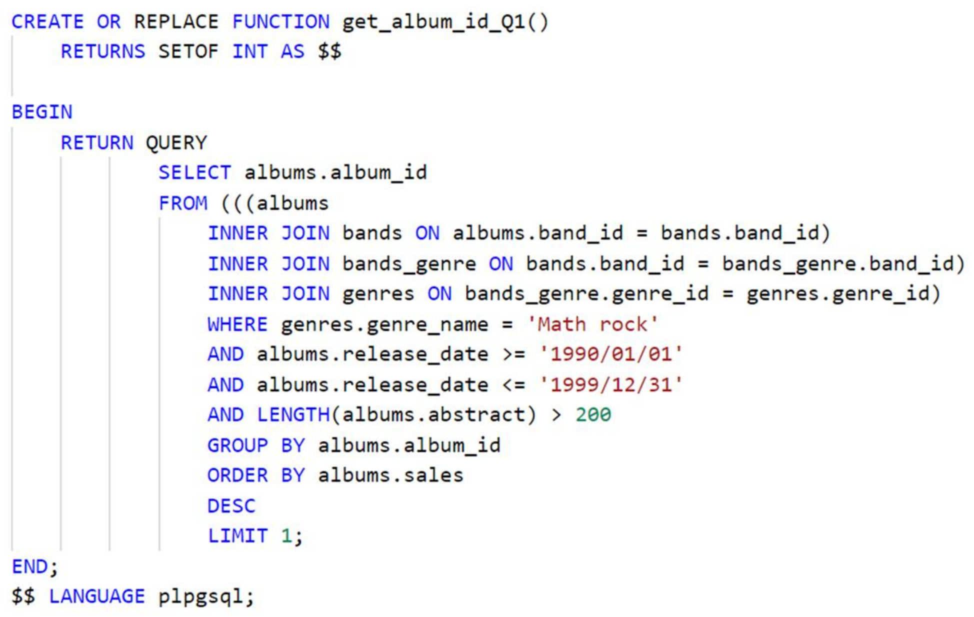
A *Serialization* anomaly occurs when the result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time. Lastly, there is one lesser-known anomaly which is the ***Lost updates*** anomaly which happens when two transactions change the same object - the second transaction will overwrite the first one, losing the result of the first transaction.

In the next topics we will cover how we transformed the queries from the previous part of the project to *PL/pgSQL*, and after that we will show some examples of anomalies that can occur and how to solve them with locks and isolation levels.

2 - Query 1

The first query is composed of two parts, a select and an UPDATE query. In order to “translate” these queries to the procedural language *PL/pgSQL* (to facilitate concurrency testing), we created a function for each part.

In the select function - “*get_album_id_Q1()” -* we used the same query but slightly modified. It only retrieves the ‘*album_id*’ value - return of the function as an integer.

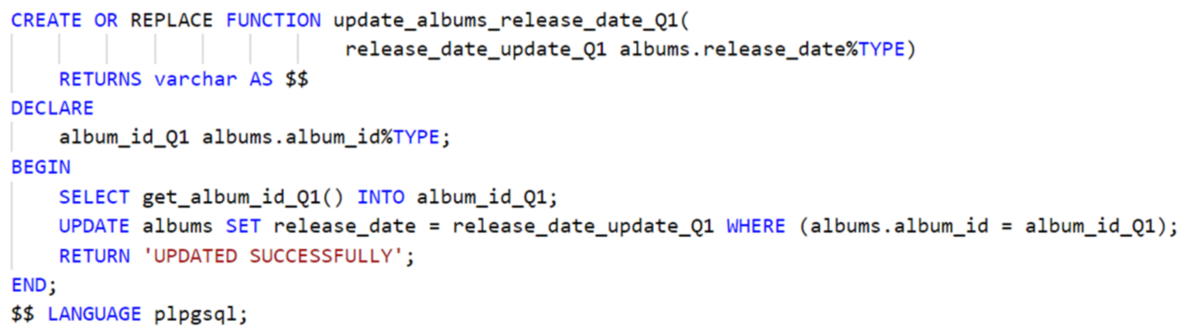


```
CREATE OR REPLACE FUNCTION get_album_id_Q1()
    RETURNS SETOF INT AS $$

BEGIN
    RETURN QUERY
            SELECT albums.album_id
            FROM (((albums
                INNER JOIN bands ON albums.band_id = bands.band_id)
                INNER JOIN bands_genre ON bands.band_id = bands_genre.band_id)
                INNER JOIN genres ON bands_genre.genre_id = genres.genre_id)
                WHERE genres.genre_name = 'Math rock'
                AND albums.release_date >= '1990/01/01'
                AND albums.release_date <= '1999/12/31'
                AND LENGTH(albums.abstract) > 200
                GROUP BY albums.album_id
                ORDER BY albums.sales
                DESC
                LIMIT 1;
END;
$$ LANGUAGE plpgsql;
```

Figure 1 - Select function (*query 1*) in *PL/pgSQL* - “*get_album_id_Q1()”*.

For the update function - *“update_albums_release_date_Q1(‘date’)”* - we started by creating an input variable to set the date that we wished to modify the existing one to - ‘*release_date_update_q1*’. Then we declared another variable that stores the output of the select function ‘*album_id_q1*’. The type of the created variables was defined as the same of the columns ‘*albums.release_date*’ and ‘*albums.album_id’*, respectively.

Finally, we defined the UPDATE query where we used the variable ‘*release_date_update_q1*’ as the value of the ‘*release_date’* that will be updated to, and the variable ‘*album_id_q1*’ as a constraint to find the specific album row.



```
CREATE OR REPLACE FUNCTION update_albums_release_date_Q1(
                        release_date_update_Q1 albums.release_date%TYPE)
    RETURNS varchar AS $$
DECLARE
    album_id_Q1 albums.album_id%TYPE;
BEGIN
    SELECT get_album_id_Q1() INTO album_id_Q1;
    UPDATE albums SET release_date = release_date_update_Q1 WHERE (albums.album_id = album_id_Q1);
    RETURN 'UPDATED SUCCESSFULLY';
END;
$$ LANGUAGE plpgsql;
```

Figure 2 - Update function (*query 1*) in *PL/pgSQL* - “*update_albums_release_date_Q1(‘date’)”*.

2.1 - Dirty Read Phenomena

The functions described before were built in a transaction process **(**Appendix 1**),** which was run in distinct *Linux* shells simultaneously (shell 1 - transaction 1 (T1), shell 2 - transaction 2 (T2). The transaction is composed, after the 'BEGIN' statement, with the update function; right after, runs two select queries after another, separated with the statement '*pg_sleep(3)*', that allows us to start a second transaction in another shell in time (shell 2). The first SELECT query retrieves the '*album_id*' and the '*release_date*', with the constraints of *query 1*, but this time we only wanted the set of values for the year '*1980-01-01*', in order to infer the rows that were modified by the update function. On the other hand, the second query is basically the *query 1,* but only outputs the 'album_id' - Figure 3 describes this procedure.

Given the fact that *PL/pgSQL* does not allow concurrency anomalies, we defined in each transaction "*\set AUTCOMMIT off*", before the 'BEGIN' statement. Thus, the control system will not commit the transactions automatically allowing it to test some anomalies, in order to understand the *PL/pgSQL* backend process. At the end, we returned the control system to the normal configuration with the statement "*\set AUTCOMMIT on* ", at the end of each transaction in this project.

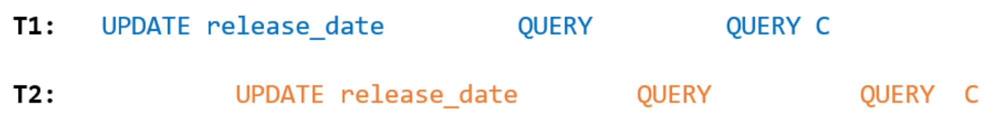
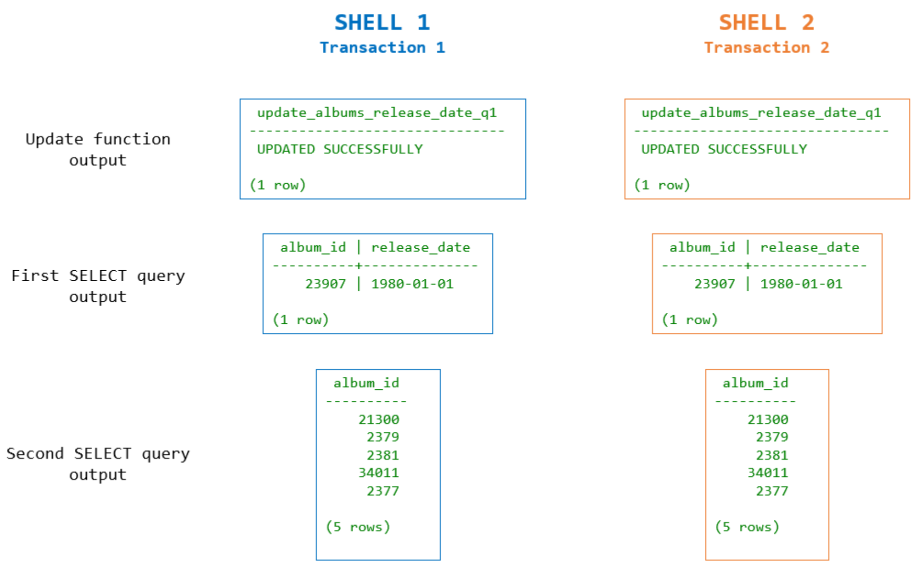


Figure 3 - Transaction schema for query *1* in *PL/pgSQL*.

Thus, we obtained an output for both transactions (T1 and T2) - **Figure 4**. For both transactions we received the same results, which indicates the occurrence of an consistency anomaly defined as *Dirty Read*, as both transactions are updating the same piece of data: the '*release_date'* of the "*album_id* = *23907*" to the year '*1980-01-01*'.

In order to solve this phenomena, we used two approaches, implementation of explicit locks and isolation levels. These procedures will be explained in the next chapters.



```
 update_albums_release_date_q1
-------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

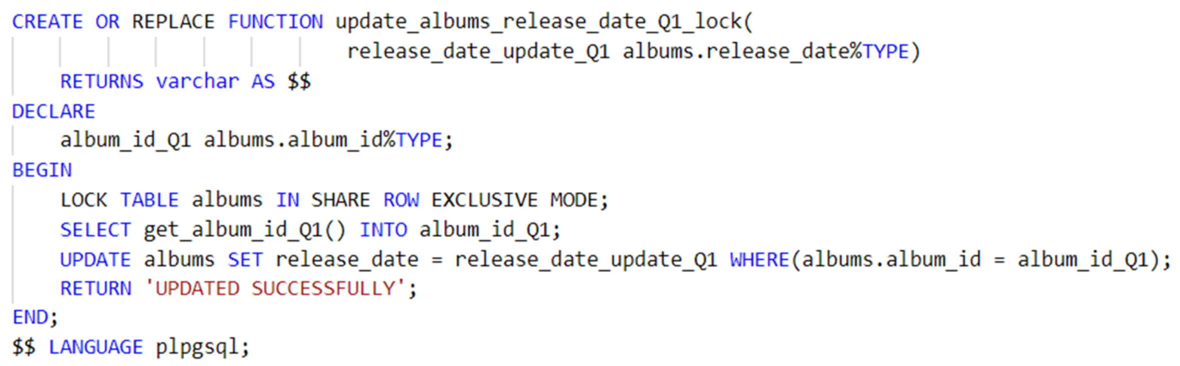
```
 album_id | release_date
----------+--------------
    23907 | 1980-01-01

(1 row)
```

```
 album_id
----------
    21300
     2379
     2381
    34011
     2377

(5 rows)
```

Figure 4 - Outputs of transaction 1 and 2 (*query 1*) in *PL/pgSQL*.

2.1.1 - Implementing Lock-Based Concurrency Control

The first approach to solve this concurrency problem was using explicit *locks*. Thus, we locked the rows of table '*albums',* stating in the update function "*LOCK TABLE albums IN SHARE ROW EXCLUSIVE MODE*", which created the new function "update*_albums_release_date_Q1_lock('data')*" (Figure 5). This chosen lock mode will protect the rows of the table '*albums*' against reading and writing concurrent data changes, until a transaction is committed - only one session can hold it at time.

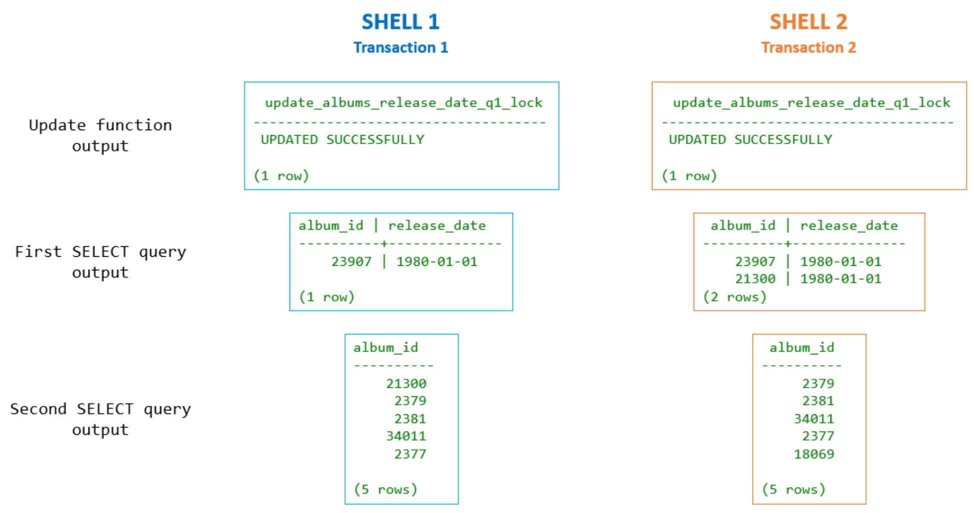


```
CREATE OR REPLACE FUNCTION update_albums_release_date_Q1_lock(
                                release_date_update_Q1 albums.release_date%TYPE)
    RETURNS varchar AS $$
DECLARE
    album_id_Q1 albums.album_id%TYPE;
BEGIN
    LOCK TABLE albums IN SHARE ROW EXCLUSIVE MODE;
    SELECT get_album_id_Q1() INTO album_id_Q1;
    UPDATE albums SET release_date = release_date_update_Q1 WHERE(albums.album_id = album_id_Q1);
    RETURN 'UPDATED SUCCESSFULLY';
END;
$$ LANGUAGE plpgsql;
```

Figure 5 - Query 1 update function with lock implementation in *PL/pgSQL*.

Therefore, we run the transaction (Appendix 2) in distinct shells simultaneously, which have the same schema presented in Figure 3, but with the update function switched to the newly created one (Figure 5).

This way, we obtain the set of results present in Figure 6. The transactions 1 and 2 outputs were different this time, due to lock implementation. Transaction 2 waited until transaction 1 was committed to proceed, and as we can infer, transaction 1 updated the *'release_date'* for the "*album_id* = *'23907'*", and the transaction 2 updated the *'release_date'* for the "*album_id* = *'21300*'". If a third transaction were to be run, it would update the *'release_date'* for the "*album_id* = *'2389*'". This happens because when an album is updated, it will no longer be a part of the set of results given by the second SELECT query. This would allow, for example, to know which '*album_id'* would be updated next (the *ids* are sorted in a descending order by the value of the '*sales'* attribute).



```
 update_albums_release_date_q1_lock
------------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

```
album_id | release_date
----------+--------------
    23907 | 1980-01-01

(1 row)
```

```
album_id
----------
    21300
     2379
     2381
    34011
     2377

(5 rows)
```

```
 update_albums_release_date_q1_lock
------------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

```
album_id | release_date
----------+--------------
    23907 | 1980-01-01
    21300 | 1980-01-01
(2 rows)
```

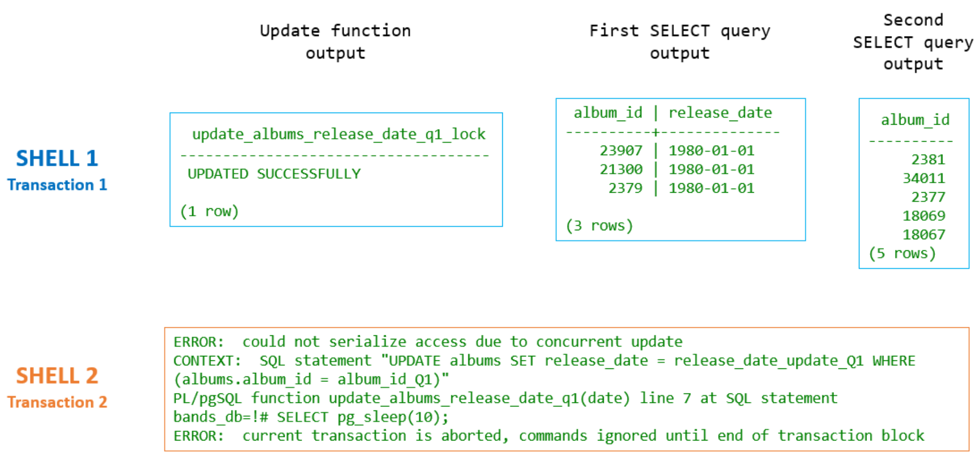
```
album_id
----------
     2379
     2381
    34011
     2377
    18069

(5 rows)
```

Figure 6 - Outputs of transaction 1 and 2 (*query 1*) in *PL/pgSQL*, with locks implemented.

2.1.2 - Implementing Isolation Levels

In this second approach to solve the *Dirty Read* phenomena we implemented an isolation level in our transaction. Even though *PostgreSQL* documentation states that less restricted levels, like *uncommitted* and *committed reads*, do not allow *Dirty Reads* phenomena to happen, in our case it still happened, so we set the isolation level to *repeatable read* - which is a more restricted level - by writing “*TRANSACTION ISOLATION LEVEL REPEATABLE READ*” after the ‘*BEGIN*’ statement in our transaction (Appendix 3).

After the transactions were runned, we got the outputs presented in Figure 7. The transaction 2 was aborted, due to the transaction block established by the isolation level - “Error: could not serialize access due to concurrent update”; “Error: current transaction is aborted, commands ignored until end of transaction block”. Conversely, transaction 1 updated the ‘*release_date*’ of the *“album_id = 2379*” to ‘*1980-01-01*’.



```
 update_albums_release_date_q1_lock
------------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

```
 album_id | release_date
----------+--------------
    23907 | 1980-01-01
    21300 | 1980-01-01
     2379 | 1980-01-01

(3 rows)
```

```
 album_id
----------
     2381
    34011
     2377
    18069
    18067
(5 rows)
```

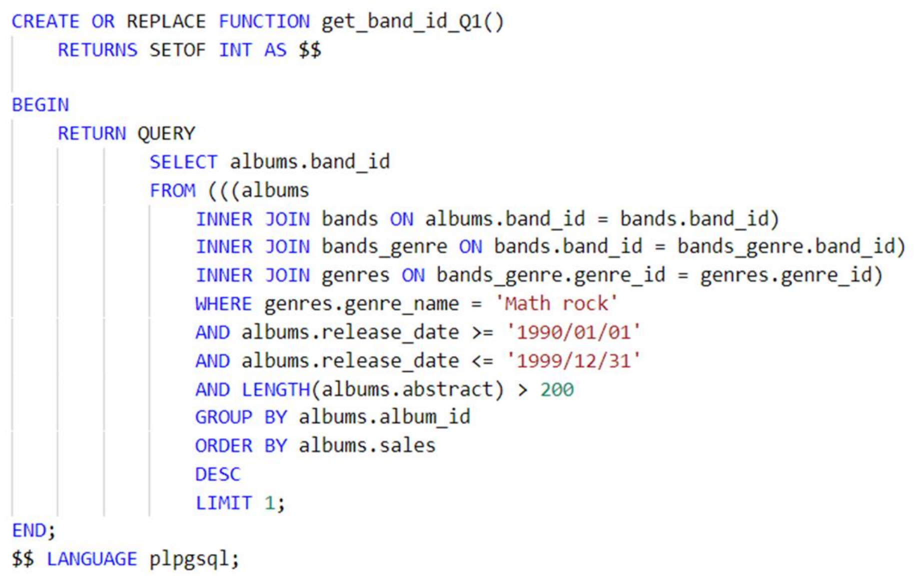
```
ERROR:  could not serialize access due to concurrent update
CONTEXT:  SQL statement "UPDATE albums SET release_date = release_date_update_Q1 WHERE
(albums.album_id = album_id_Q1)"
PL/pgSQL function update_albums_release_date_q1(date) line 7 at SQL statement
bands_db=!# SELECT pg_sleep(10);
ERROR:  current transaction is aborted, commands ignored until end of transaction block
```

Figure 7 - Outputs of transaction 1 and 2 (*query 1*) in *PL/pgSQL*, with isolation levels implemented.

2.2 - Inducing Phantom Read Phenomena

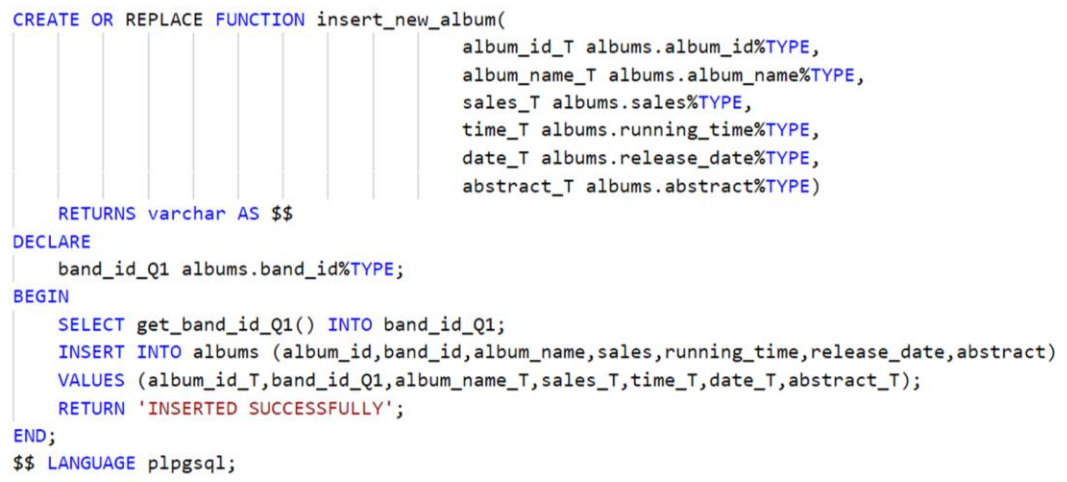
To induce a *Phantom Read* phenomena, taking in consideration the principle previously described in the introductory part, we started by creating a new set of functions in *PL/pgSQL* - a select and a insert function (Figure 8 and 9, respectively).

For the select function “*get_band_id()*”, we used query 1 which only outputs the ‘*band_id*’ (Figure 8). Thus, we proceeded with a similar approach implemented in the update function (Figure 2) for dirty reads, to create the insert function “*insert_new_album()*” - Figure 9. We started by defining the input variables that will be used as the values of the complex insert query. Then, the result of the select function was saved in the declared variable *‘band_id_Q1’ .* As a final step, we proceed to define the insert query using the values of the input variables and the ‘*band_id_Q1*’ to create a new album row in the database.



```
CREATE OR REPLACE FUNCTION get_band_id_Q1()
    RETURNS SETOF INT AS $$

BEGIN
    RETURN QUERY
            SELECT albums.band_id
            FROM (((albums
                INNER JOIN bands ON albums.band_id = bands.band_id)
                INNER JOIN bands_genre ON bands.band_id = bands_genre.band_id)
                INNER JOIN genres ON bands_genre.genre_id = genres.genre_id)
                WHERE genres.genre_name = 'Math rock'
                AND albums.release_date >= '1990/01/01'
                AND albums.release_date <= '1999/12/31'
                AND LENGTH(albums.abstract) > 200
                GROUP BY albums.album_id
                ORDER BY albums.sales
                DESC
                LIMIT 1;
END;
$$ LANGUAGE plpgsql;
```

Figure 9 - Select function for concurrency experiments with Phantom Read phenomena in *PL/pgSQL*.



```
CREATE OR REPLACE FUNCTION insert_new_album(
                                        album_id_T albums.album_id%TYPE,
                                        album_name_T albums.album_name%TYPE,
                                        sales_T albums.sales%TYPE,
                                        time_T albums.running_time%TYPE,
                                        date_T albums.release_date%TYPE,
                                        abstract_T albums.abstract%TYPE)
    RETURNS varchar AS $$
DECLARE
    band_id_Q1 albums.band_id%TYPE;
BEGIN
    SELECT get_band_id_Q1() INTO band_id_Q1;
    INSERT INTO albums (album_id,band_id,album_name,sales,running_time,release_date,abstract)
    VALUES (album_id_T,band_id_Q1,album_name_T,sales_T,time_T,date_T,abstract_T);
    RETURN 'INSERTED SUCCESSFULLY';
END;
$$ LANGUAGE plpgsql;
```

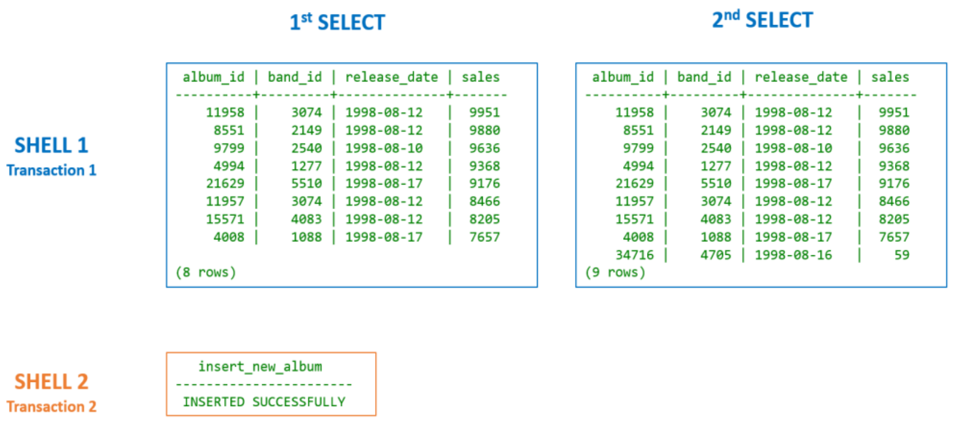
Figure 10 - Insert function for concurrency experiments with Phantom Read phenomena in *PL/pgSQL*.

Consequently, with the insert function created, we builded a new transaction process to test the *Phantom Read* phenomena. The transaction process consists of running two distinct transactions, simultaneously, in distinct *Linux* shells. Thus, the first transaction (Appendix 4-A) - after the 'BEGIN' statement - will be our *query* 1 which runned twice, and between both queries we used the statement '*pg_sleep(20)*'. On the other hand, transaction 2 (Appendix 4-B) will run the insert function "*insert_new_album()", after the 'BEGIN'* statement. This process is described in the schema of Figure 11.

```
T1:   QUERY                          QUERY C

T2:          insert_new_album() C
```

Figure 11 - Transaction schema for *query 1* in *PL/pgSQL*, for concurrency experiments with Phantom Read phenomena.

The set of results presented in Figure 12, were obtained after the end of the transaction process. It allowed us to infer that the set of results of transaction 2 was affected by the first transaction: transaction 2 added a new album row - “*album_id = 34716*” - which is embedded in the set of constraints of the queries from transaction 1.



```
 album_id | band_id | release_date | sales
----------+---------+--------------+-------
    11958 |    3074 | 1998-08-12   |  9951
     8551 |    2149 | 1998-08-12   |  9880
     9799 |    2540 | 1998-08-10   |  9636
     4994 |    1277 | 1998-08-12   |  9368
    21629 |    5510 | 1998-08-17   |  9176
    11957 |    3074 | 1998-08-12   |  8466
    15571 |    4083 | 1998-08-12   |  8205
     4008 |    1088 | 1998-08-17   |  7657

(8 rows)
```

```
 album_id | band_id | release_date | sales
----------+---------+--------------+-------
    11958 |    3074 | 1998-08-12   |  9951
     8551 |    2149 | 1998-08-12   |  9880
     9799 |    2540 | 1998-08-10   |  9636
     4994 |    1277 | 1998-08-12   |  9368
    21629 |    5510 | 1998-08-17   |  9176
    11957 |    3074 | 1998-08-12   |  8466
    15571 |    4083 | 1998-08-12   |  8205
     4008 |    1088 | 1998-08-17   |  7657
    34716 |    4705 | 1998-08-16   |    59
(9 rows)
```

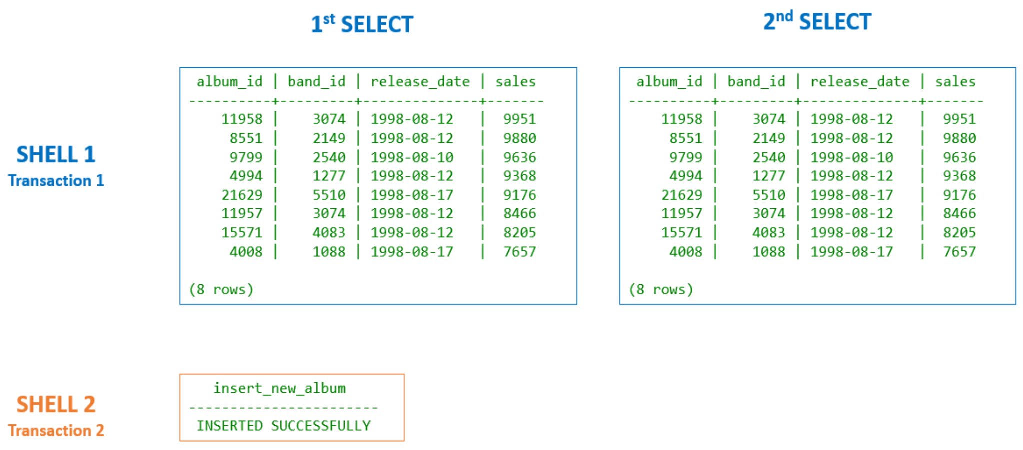
```
   insert_new_album
-----------------------
 INSERTED SUCCESSFULLY
```

Figure 12 - Outputs of transaction 1 and 2 in *PL/pgSQL*, for concurrency experiments with Phantom Read phenomena.

2.2.1 - Implementing Lock-Based Concurrency Control

To solve the phenomena above mentioned, we applied a lock-based concurrency control in the transaction 1 (Appendix 5), which prevents other transactions to write in table ‘*albums*’ while the first transaction is still running and not yet committed - it was added the “*LOCK TABLE albums IN EXCLUSIVE MODE*” after the 'BEGIN'’ statement.

Therefore, we performed both transactions simultaneously, getting the results presented in Figure 13. As concluded, transaction 2 waited until transaction 1’s process was committed to start, and transaction 1 showed the same set of results in both queries.



```
 album_id | band_id | release_date | sales
----------+---------+--------------+-------
    11958 |    3074 | 1998-08-12   |  9951
     8551 |    2149 | 1998-08-12   |  9880
     9799 |    2540 | 1998-08-10   |  9636
     4994 |    1277 | 1998-08-12   |  9368
    21629 |    5510 | 1998-08-17   |  9176
    11957 |    3074 | 1998-08-12   |  8466
    15571 |    4083 | 1998-08-12   |  8205
     4008 |    1088 | 1998-08-17   |  7657

(8 rows)
```

```
 album_id | band_id | release_date | sales
----------+---------+--------------+-------
    11958 |    3074 | 1998-08-12   |  9951
     8551 |    2149 | 1998-08-12   |  9880
     9799 |    2540 | 1998-08-10   |  9636
     4994 |    1277 | 1998-08-12   |  9368
    21629 |    5510 | 1998-08-17   |  9176
    11957 |    3074 | 1998-08-12   |  8466
    15571 |    4083 | 1998-08-12   |  8205
     4008 |    1088 | 1998-08-17   |  7657

(8 rows)
```

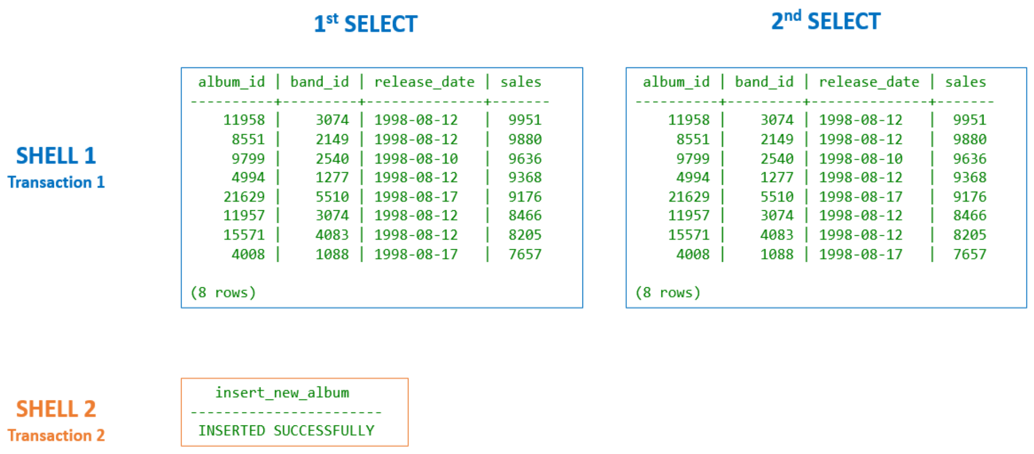
```
   insert_new_album
-----------------------
 INSERTED SUCCESSFULLY
```

Figure 13 - Outputs of transaction 1 and 2 with implemented locks in *PL/pgSQL*, for concurrency experiments with Phantom Read phenomena.

2.2.2 - Implementing Isolation Levels

Another solution implemented to solve the *Phantom Read* anomaly was the isolation level in transaction 1. Thus, we chose the '*REPEATABLE READ*' as an isolation level, since less restricted levels allow the occurrence of the phenomena. This way, we used the statement "*TRANSACTION ISOLATION LEVEL REPEATABLE READ*" after the '*BEGIN*' statement in transaction 1 (Appendix 6).

Therefore, we performed the transaction with the implemented isolation level which retrieves the output presented in Figure 14. Similar to the output obtained for transaction 1 with lock implementation, with this approach we achieved the same results as expected - transaction 1 had the same set of results for both queries. Transaction 2 was committed during the process of transaction 1 without affecting its results, and did not report any error.



```
 album_id | band_id | release_date | sales
----------+---------+--------------+-------
    11958 |    3074 | 1998-08-12   |  9951
     8551 |    2149 | 1998-08-12   |  9880
     9799 |    2540 | 1998-08-10   |  9636
     4994 |    1277 | 1998-08-12   |  9368
    21629 |    5510 | 1998-08-17   |  9176
    11957 |    3074 | 1998-08-12   |  8466
    15571 |    4083 | 1998-08-12   |  8205
     4008 |    1088 | 1998-08-17   |  7657

(8 rows)
```

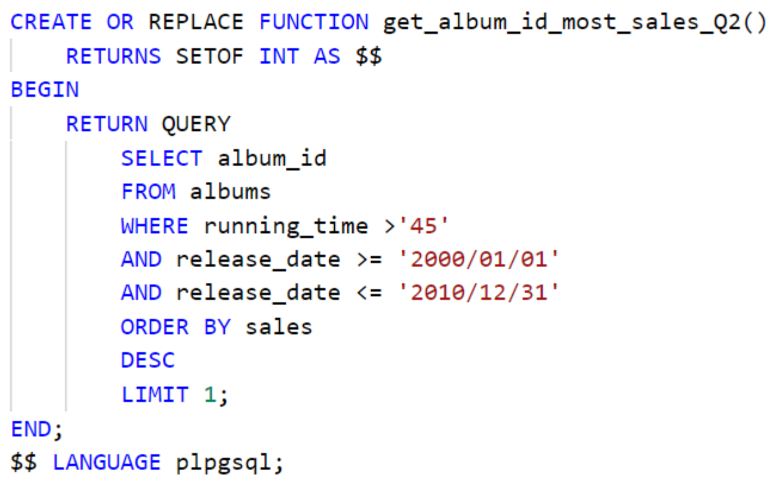
```
 album_id | band_id | release_date | sales
----------+---------+--------------+-------
    11958 |    3074 | 1998-08-12   |  9951
     8551 |    2149 | 1998-08-12   |  9880
     9799 |    2540 | 1998-08-10   |  9636
     4994 |    1277 | 1998-08-12   |  9368
    21629 |    5510 | 1998-08-17   |  9176
    11957 |    3074 | 1998-08-12   |  8466
    15571 |    4083 | 1998-08-12   |  8205
     4008 |    1088 | 1998-08-17   |  7657

(8 rows)
```

```
   insert_new_album
-----------------------
 INSERTED SUCCESSFULLY
```

Figure 14 - Outputs of transaction 1 and 2 with implemented isolation levels in *PL/pgSQL*, for concurrency experiments with Phantom Read phenomena.

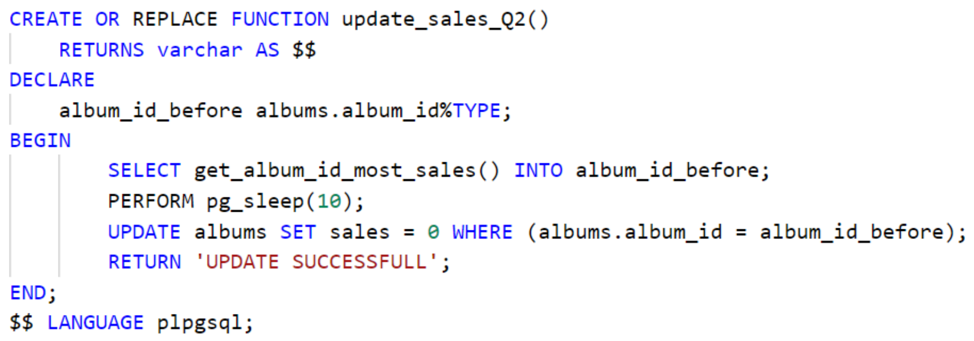
3 - Query 2

For *query 2,* we proceeded to do the same as we did for *query 1*: we created two separate functions, one for the *SELECT query* - “*get_album_id_most_sales_Q2()*” (Figure 15) - and the second one for the *update - “update_sales_Q2()”* (Figure 16). In the *update function* we declared the variable *‘album_id_before’,* which had the same value type as the column *‘album_id’* from the *‘albums’* table (*integer*), and proceeded to store the output of the *select function* in this variable.



```
CREATE OR REPLACE FUNCTION get_album_id_most_sales_Q2()
    RETURNS SETOF INT AS $$
BEGIN
    RETURN QUERY
        SELECT album_id
        FROM albums
        WHERE running_time >'45'
        AND release_date >= '2000/01/01'
        AND release_date <= '2010/12/31'
        ORDER BY sales
        DESC
        LIMIT 1;
END;
$$ LANGUAGE plpgsql;
```

Figure 15 - Select function (*query 2*) in *PL/pgSQL* - “*get_album_id_most_sales_Q2()”.*



```
CREATE OR REPLACE FUNCTION update_sales_Q2()
    RETURNS varchar AS $$
DECLARE
    album_id_before albums.album_id%TYPE;
BEGIN
        SELECT get_album_id_most_sales() INTO album_id_before;
        PERFORM pg_sleep(10);
        UPDATE albums SET sales = 0 WHERE (albums.album_id = album_id_before);
        RETURN 'UPDATE SUCCESSFULL';
END;
$$ LANGUAGE plpgsql;
```

Figure 16 - Update function (*query 2)* for concurrency experiments with Dirty Read phenomena in *PL/pgSQL* - “*update_sales_Q2()*”.

3.1 - Dirty Read Phenomena

The functions described for the *query 2* were also built in a transaction process **(**Appendix 7**),** which was run in distinct *Linux* shells simultaneously (shell 1 - transaction 1 (T1), shell 2 - transaction 2 (T2). Both transactions are built in a way where the first and second *SELECT query* retrieve the ‘*album_id*’ and are saved in the local variable ‘*album_id_before*’, then we put the transactions on *‘pg_sleep’* so that the *update function* gets executed at the same time, the Figure X describes this procedure.

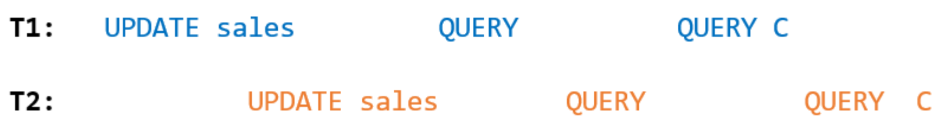
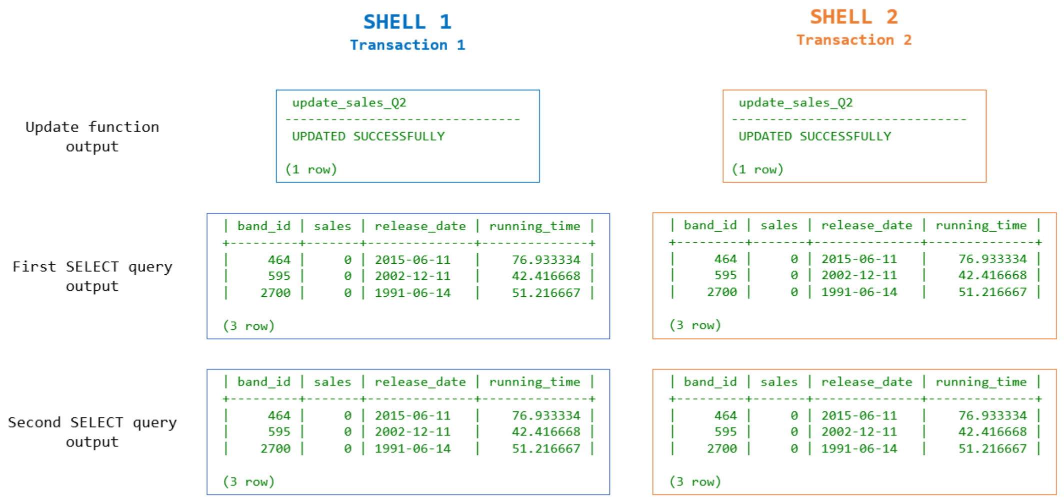

T1: UPDATE sales QUERY QUERY C
T2: UPDATE sales QUERY QUERY C

Figure 17 - Transaction schema for *query 2* in *PL/pgSQL*.

Therefore, after the execution of both transactions we obtained an output (T1 and T2) - Figure 18 - where the results were the same as if only one transaction was executed, which indicated the occurrence of an consistency anomaly defined as Dirty Read, since both transactions were updating the same piece of data, the *sales* of the "*album_id = 464*" to the '*sales*' number '*0'*.

In order to solve this phenomena we used the same procedures as the ones used for *query 1*, which will be explained in the next chapters.



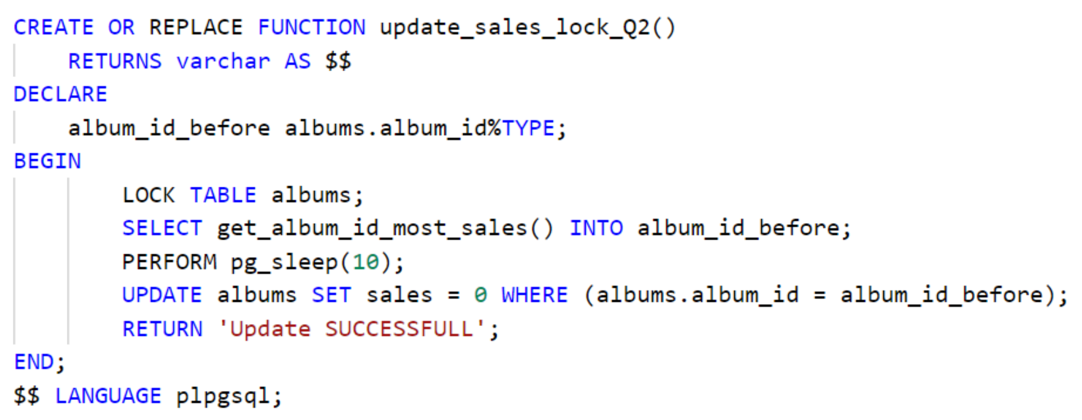
```
update_sales_Q2
--------------------------------
UPDATED SUCCESSFULLY

(1 row)
```

```
| band_id | sales | release_date | running_time |
+---------+-------+--------------+--------------+
|     464 |     0 | 2015-06-11   |    76.933334 |
|     595 |     0 | 2002-12-11   |    42.416668 |
|    2700 |     0 | 1991-06-14   |    51.216667 |

(3 row)
```

Figure 18 - Outputs of transaction 1 and 2 (*query 2*) in *PL/pgSQL*.

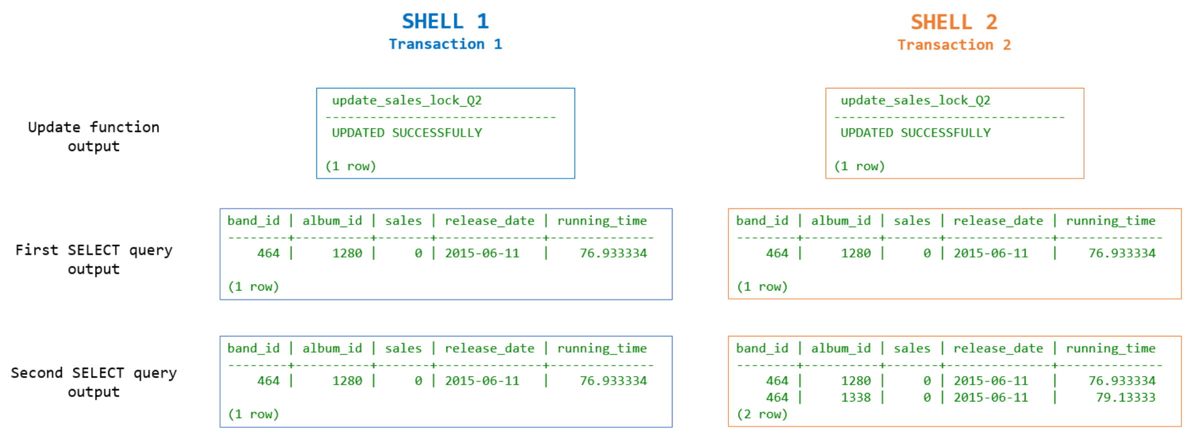
3.1.1 – Implementing Lock-Based Concurrency Control

To solve the *Dirty Read* phenomena for this query we started by adding an explicit lock which controls concurrent access to data in tables (Figure 19). Usually, the database management system takes care of these issues, however sometimes it is required to add explicit locks to obtain the desired result. In the following query we added a line where we inserted a lock table that locks the whole table, including all data and indexes and it lasts until the end of the current transaction, which means no other transactions can either read, write, delete or make updates in this table while the first transaction does not end - Appendix 8.



```
CREATE OR REPLACE FUNCTION update_sales_lock_Q2()
    RETURNS varchar AS $$
DECLARE
    album_id_before albums.album_id%TYPE;
BEGIN
        LOCK TABLE albums;
        SELECT get_album_id_most_sales() INTO album_id_before;
        PERFORM pg_sleep(10);
        UPDATE albums SET sales = 0 WHERE (albums.album_id = album_id_before);
        RETURN 'Update SUCCESSFULL';
END;
$$ LANGUAGE plpgsql;
```

Figure 19 - Update function (*query 2*) with lock implementation in *PL/pgSQL*.



```
 update_sales_lock_Q2
------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

```
 band_id | album_id | sales | release_date | running_time
---------+----------+-------+--------------+--------------
     464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

```
 band_id | album_id | sales | release_date | running_time
---------+----------+-------+--------------+--------------
     464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

```
 update_sales_lock_Q2
------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

```
 band_id | album_id | sales | release_date | running_time
---------+----------+-------+--------------+--------------
     464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

```
 band_id | album_id | sales | release_date | running_time
---------+----------+-------+--------------+--------------
     464 |     1280 |     0 | 2015-06-11   |    76.933334
     464 |     1338 |     0 | 2015-06-11   |     79.13333
(2 row)
```

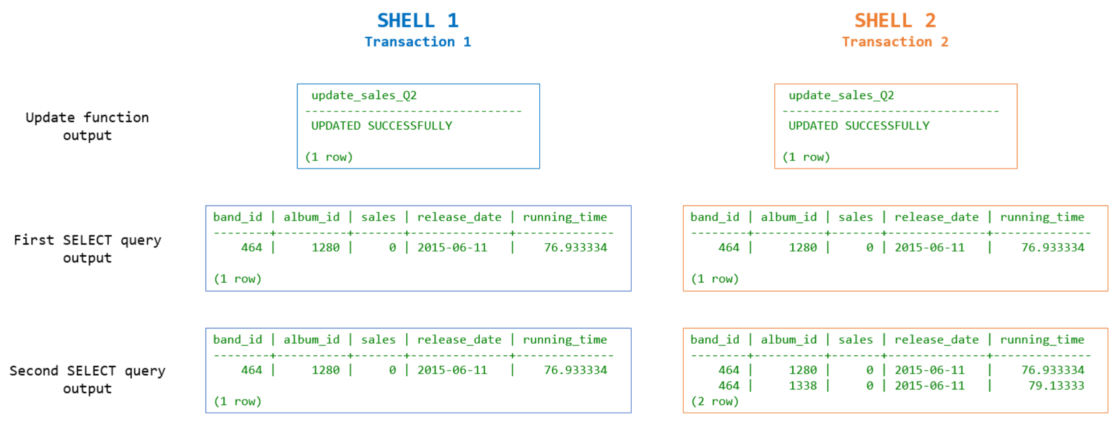
Figure 20 - Outputs of transaction 1 and 2 (*query 2*) in *PL/pgSQL*, with locks implemented.

As it can be seen in the figure above (Figure 20), this time the second shell waited for the first transaction to end and then proceeded to start the second transaction this way obtaining the desired result.

3.1.2 - Implementing Isolation Levels

Taking in consideration the phenomena occurred in *query 2* and the documentation about isolation levels, one can infer that *read uncommitted* is the best isolation level to use in our query since the other levels are more restrict and the main goal is to not impact other users and to maintain availability and consistency at the same time. However, *PostgreSQL read uncommitted* has the same behavior as *read committed*, so internally only *read committed* is implemented.

In the Appendix 9 it is possible to see how the isolation level was implemented in our code, as well as the results obtained. After the *"BEGIN"*, the "*BEGIN TRANSACTION ISOLATION LEVEL Read Committed;*" statement was added.



Shell 1 – Transaction 1

```
 update_sales_Q2
--------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

```
band_id | album_id | sales | release_date | running_time
--------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

```
band_id | album_id | sales | release_date | running_time
--------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

Shell 2 – Transaction 2

```
 update_sales_Q2
--------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

```
band_id | album_id | sales | release_date | running_time
--------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

```
band_id | album_id | sales | release_date | running_time
--------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334
    464 |     1338 |     0 | 2015-06-11   |     79.13333
(2 row)
```

Figure 21 - Outputs of transaction 1 and 2 (*query 2*) in *PL/pgSQL*, with isolation levels implemented.

As depicted, the first and second transaction ended with one changed row, meaning it was successful (Figure 21) - this happens because *read committed* checks if there are other transactions occurring at the same time for the same piece of data that is being updated. After the update of the first one, the second transaction will check if the result of the first transaction is still applied to the selection that we seek - if applied, then it would update the same piece of data. Nevertheless, in our case, the data got updated to '0' and no longer satisfied the condition of the highest sales, so the second transaction checked again which part of the data satisfied the condition and performed the update.

```
CREATE OR REPLACE FUNCTION update_sales_lost_update_Q2()
    RETURNS varchar AS $$
DECLARE
    album_id_before albums.album_id%TYPE;
BEGIN
        SELECT get_album_id_most_sales() INTO album_id_before;
        PERFORM pg_sleep(10);
        UPDATE albums SET sales = 1 WHERE (albums.album_id = album_id_before);
        RETURN 'UPDATE SUCCESSFUL';
END;
$$ LANGUAGE plpgsql;
```

Figure 22 - Update function (*query 2)* for concurrency experiments with Lost Update phenomena in *PL/pgSQL*

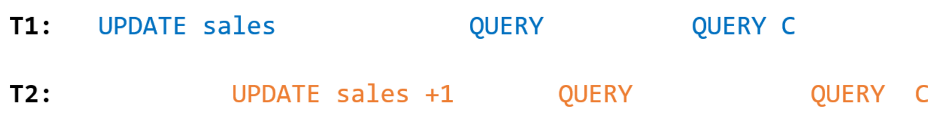
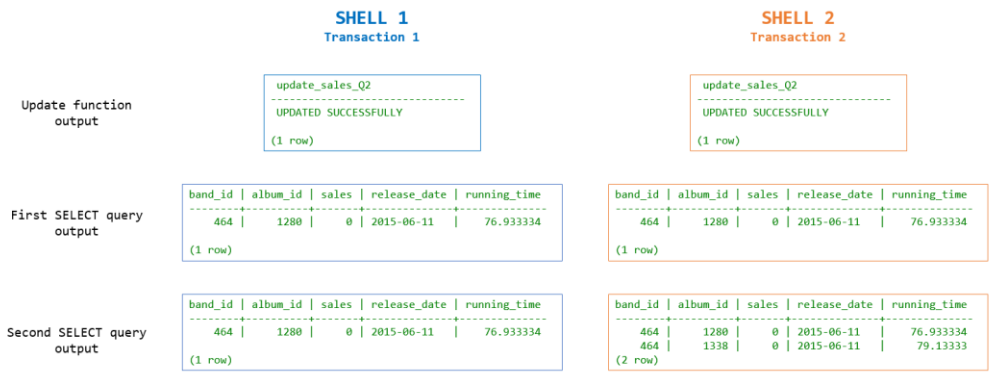


Figure 23 - Transaction schema for *query 2* in *PL/pgSQL*, for concurrency experiments with Lost Update phenomena in *PL/pgSQL*

3.2 - Inducing Lost Update Phenomena

As previously explained, *Lost Update* is another phenomena that can occur when the second update overwrites the first update. To achieve this result, we changed our update function creating a new one (Figure 22) - "*update_sales_lost_update_Q2()*" - to ensure that the second transaction would update to the value '1', instead of updating the sales number to '0', as depicted in the Figure 23. The transaction in the Appendix 10, was performed simultaneously in distinct *Linux* shells, and its result is shown in Figure 24.



Shell 1 – Update function output:

```
 update_sales_Q2
--------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

Shell 1 – First SELECT query output:

```
band_id | album_id | sales | release_date | running_time
--------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

Shell 1 – Second SELECT query output:

```
band_id | album_id | sales | release_date | running_time
--------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

Shell 2 – Update function output:

```
 update_sales_Q2
--------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

Shell 2 – First SELECT query output:

```
band_id | album_id | sales | release_date | running_time
--------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

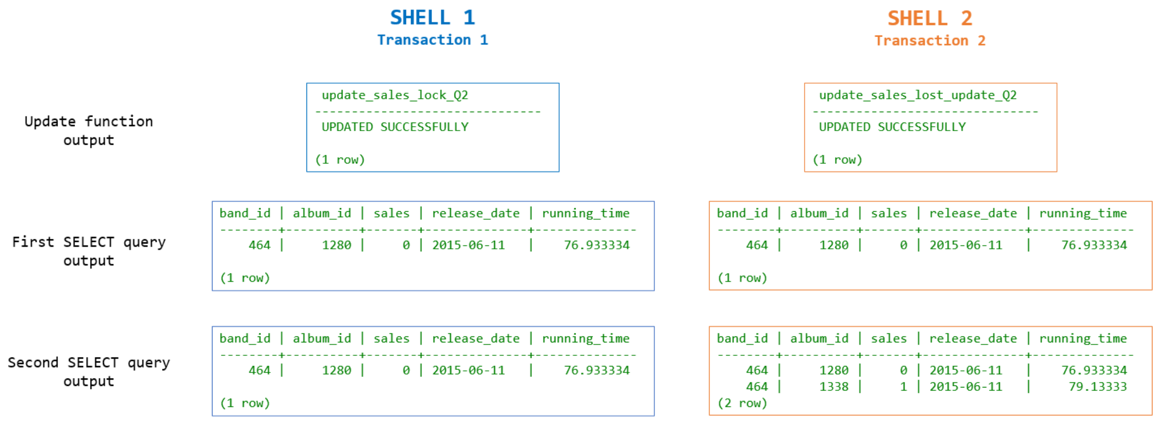
Shell 2 – Second SELECT query output:

```
band_id | album_id | sales | release_date | running_time
--------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334
    464 |     1338 |     0 | 2015-06-11   |     79.13333
(2 row)
```

Figure 24 - Outputs of transaction 1 and 2 with lost update (*query 2*) in *PL/pgSQL*.

3.2.1 – Implementing Lock-Based Concurrency Control

The resolution consisted in using the same method as explained for Figure 19, where we inserted a lock table statement - "*LOCK TABLE albums*" after the '*BEGIN*' - to prevent any changes to the table '*albums*', therefore obtaining the desired results (Figure 25**)** and solving the *Lost Update* phenomena. The code of the performed transaction is available in Appendix 11.



Shell 1 – Update function output:

```
 update_sales_lock_Q2
--------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

Shell 1 – First SELECT query output:

```
band_id | album_id | sales | release_date | running_time
--------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

Shell 1 – Second SELECT query output:

```
band_id | album_id | sales | release_date | running_time
--------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

Shell 2 – Update function output:

```
 update_sales_lost_update_Q2
--------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

Shell 2 – First SELECT query output:

```
band_id | album_id | sales | release_date | running_time
--------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

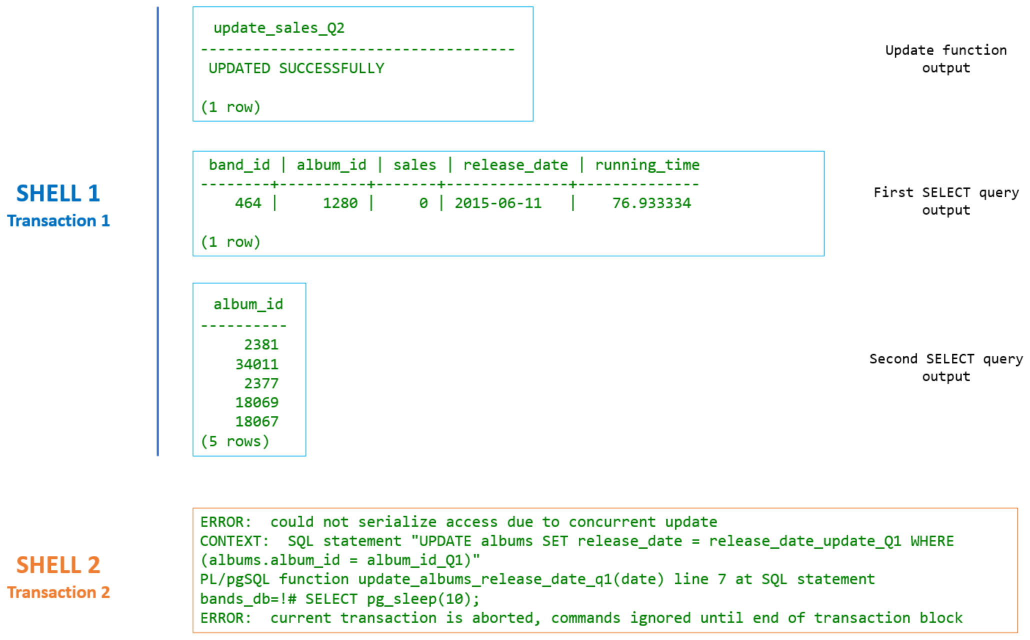
Shell 2 – Second SELECT query output:

```
band_id | album_id | sales | release_date | running_time
--------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334
    464 |     1338 |     1 | 2015-06-11   |     79.13333
(2 row)
```

Figure 25 - Outputs of transaction 1 and 2 with lock implement (*query 2*), for concurrency experiments with Lost Update phenomena in *PL/pgSQL*

3.2.2 - Implementing Isolation level

To solve the *Lost Update* phenomena, we can also use the *Repeatable Read Isolation Level* or higher, which will not allow another transaction to change the data while the first one is still running. However, the isolation levels like *Repeatable read* or higher will not put other transactions on standby - instead, an error stating that it is not possible to access that data because it is being used in other transactions will appear. The code for this transaction is provided in the Appendix 12, as well as the result of the transaction in Figure 26.



```
  update_sales_Q2
------------------------------------
 UPDATED SUCCESSFULLY

(1 row)
```

```
 band_id | album_id | sales | release_date | running_time
---------+----------+-------+--------------+-------------
    464 |     1280 |     0 | 2015-06-11   |    76.933334

(1 row)
```

```
  album_id
----------
     2381
    34011
     2377
    18069
    18067
(5 rows)
```

```
ERROR:  could not serialize access due to concurrent update
CONTEXT:  SQL statement "UPDATE albums SET release_date = release_date_update_Q1 WHERE
(albums.album_id = album_id_Q1)"
PL/pgSQL function update_albums_release_date_q1(date) line 7 at SQL statement
bands_db=!# SELECT pg_sleep(10);
ERROR:  current transaction is aborted, commands ignored until end of transaction block
```

Figure 26 - Outputs of transaction 1 and 2 with isolation levels implemented (*query 2*), for concurrency experiments with Lost Update phenomena in *PL/pgSQL*

Since an error occurred, the database management system automatically made a rollback on the data, this way getting the previous state before the start of the transaction. In this isolation level we lose the availability of the data to obtain consistency.

4 – Conclusion

With this project, we deduced how to avoid and solve concurrency anomalies with different approaches, always considering the database consistency and availability.

It is necessary to keep in mind that either *Lock-Based Concurrency Control or Isolation Levels* implementations can sometimes have a negative impact on the database performance.

Explicit locking modes could be beneficial in situations where the MVCC does not give the necessary application-control. It is important to consider that locking strategies were successfully implemented and always provided a reasonable solution to sort out the tested concurrency phenomenons. Nevertheless, it's necessary to weigh in carefully when choosing the appropriate lock mode - for example, if we are carrying a transaction that is only embedded in a set of a few rows from a table and we lock the full table, the availability and performance of the database will decrease.

On the other hand, it is also relevant to notice that transaction isolation levels could negatively impact the database performance, if the chosen level is less or higher restricted. By default, PostgreSQL uses the *READ COMMITTED* level which could be insufficient to handle some concurrent anomalies, like the *Phantom Read* and *Nonrepeatable Read* phenomenons. A *SERIALIZABLE* level does not allow the occurrence of any anomaly, due to being the most strict - requiring the highest level of system resources. However, its implementation could be an ‘overkill’ to a database, when the workload does not require such strict guarantees. Therefore, it is important to choose the right isolation level, considering each database implementation.

For the reasons above mentioned, the most coherent approach to use in our case is the explicit locks, as it does not demonstrate any implementation concerns in contrast to the isolation levels, and proved to be the best option to ensure a better availability of the database.

In conclusion, it is necessary to evaluate the trade-offs between data consistency and concurrency, considering what use the database will be given.

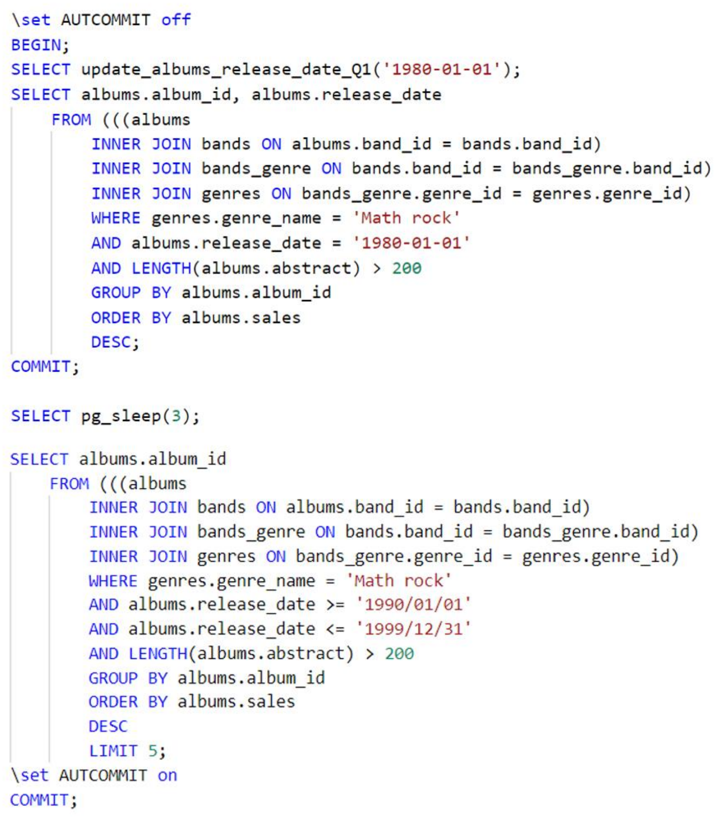
5 - References

[1] *Raghu Ramakrishnan e Johannes Gehrke, Database Management Systems, McGraw Hill, 3ª edição, 2003, ISBN 0072465638.*

[2] *PostgreSQL. PostgreSQL Documentation - Explicit Locking. https://www.postgresql.org/docs/9.1/explicit-locking.html*

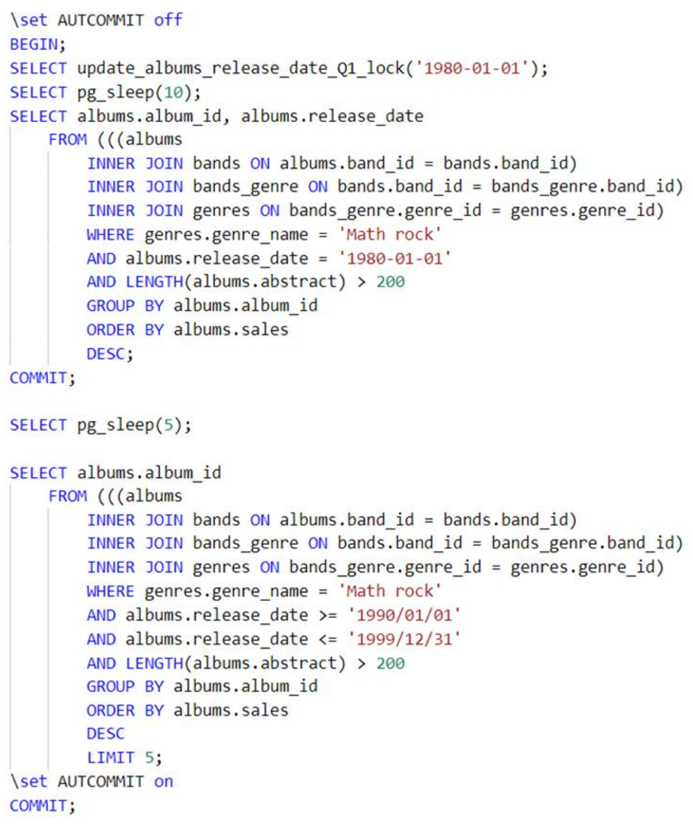
[3] *PostgreSQL. PostgreSQL Documentation - Transaction Isolation. https://www.postgresql.org/docs/9.5/transaction-iso.html*

6- Appendices



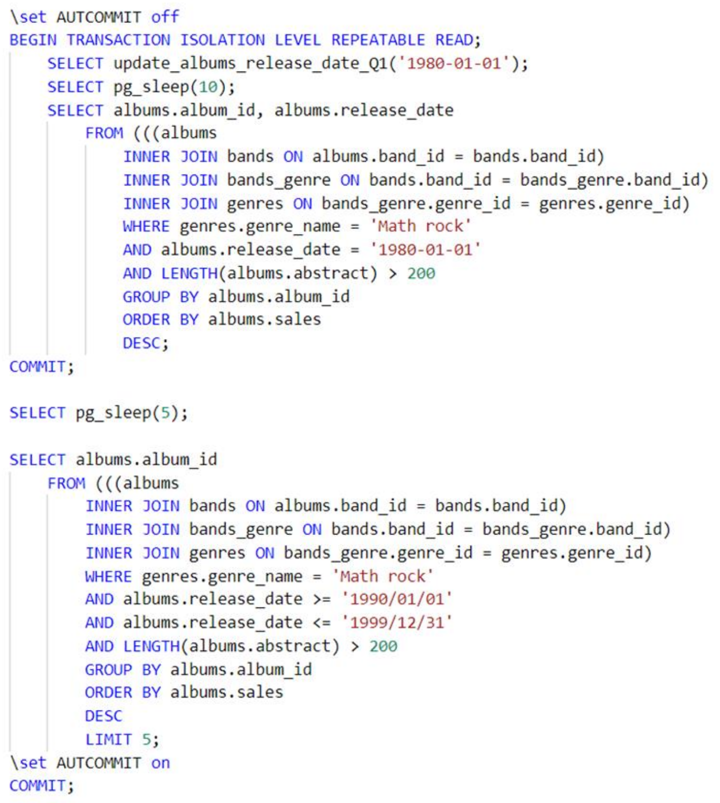
```
\set AUTOCOMMIT off
BEGIN;
SELECT update_albums_release_date_Q1('1980-01-01');
SELECT albums.album_id, albums.release_date
    FROM (((albums
        INNER JOIN bands ON albums.band_id = bands.band_id)
        INNER JOIN bands_genre ON bands.band_id = bands_genre.band_id)
        INNER JOIN genres ON bands_genre.genre_id = genres.genre_id)
        WHERE genres.genre_name = 'Math rock'
        AND albums.release_date = '1980-01-01'
        AND LENGTH(albums.abstract) > 200
        GROUP BY albums.album_id
        ORDER BY albums.sales
        DESC;
COMMIT;

SELECT pg_sleep(3);

SELECT albums.album_id
    FROM (((albums
        INNER JOIN bands ON albums.band_id = bands.band_id)
        INNER JOIN bands_genre ON bands.band_id = bands_genre.band_id)
        INNER JOIN genres ON bands_genre.genre_id = genres.genre_id)
        WHERE genres.genre_name = 'Math rock'
        AND albums.release_date >= '1990/01/01'
        AND albums.release_date <= '1999/12/31'
        AND LENGTH(albums.abstract) > 200
        GROUP BY albums.album_id
        ORDER BY albums.sales
        DESC
        LIMIT 5;
\set AUTOCOMMIT on
COMMIT;
```

Appendix 1 - Query 1 transaction in *PL/pgSQL*.



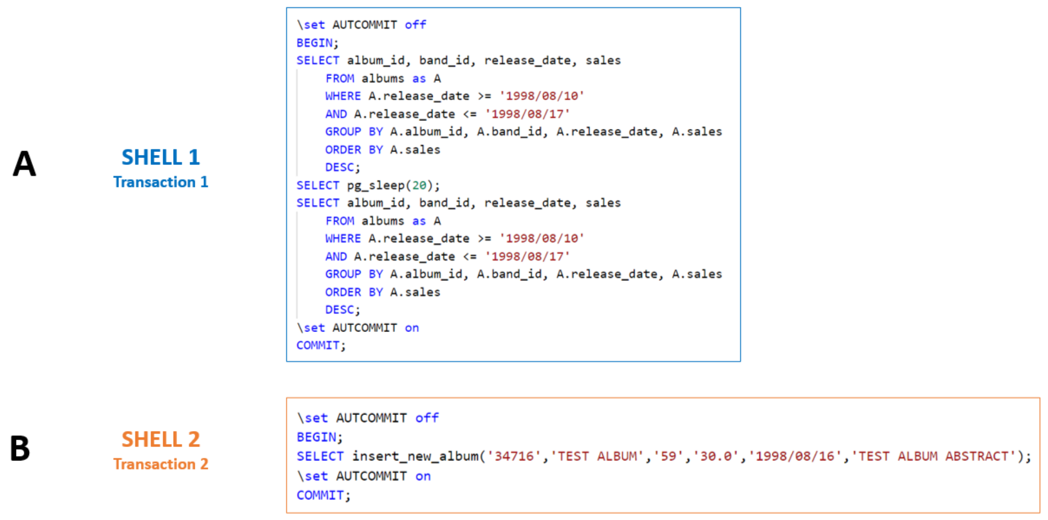
```
\set AUTOCOMMIT off
BEGIN;
SELECT update_albums_release_date_Q1_lock('1980-01-01');
SELECT pg_sleep(10);
SELECT albums.album_id, albums.release_date
    FROM (((albums
        INNER JOIN bands ON albums.band_id = bands.band_id)
        INNER JOIN bands_genre ON bands.band_id = bands_genre.band_id)
        INNER JOIN genres ON bands_genre.genre_id = genres.genre_id)
        WHERE genres.genre_name = 'Math rock'
        AND albums.release_date = '1980-01-01'
        AND LENGTH(albums.abstract) > 200
        GROUP BY albums.album_id
        ORDER BY albums.sales
        DESC;
COMMIT;

SELECT pg_sleep(5);

SELECT albums.album_id
    FROM (((albums
        INNER JOIN bands ON albums.band_id = bands.band_id)
        INNER JOIN bands_genre ON bands.band_id = bands_genre.band_id)
        INNER JOIN genres ON bands_genre.genre_id = genres.genre_id)
        WHERE genres.genre_name = 'Math rock'
        AND albums.release_date >= '1990/01/01'
        AND albums.release_date <= '1999/12/31'
        AND LENGTH(albums.abstract) > 200
        GROUP BY albums.album_id
        ORDER BY albums.sales
        DESC
        LIMIT 5;
\set AUTOCOMMIT on
COMMIT;
```

Appendix 2 - Query 1 transaction in *PL/pgSQL*, with explicit locks implemented.



```
\set AUTCOMMIT off
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
    SELECT update_albums_release_date_Q1('1980-01-01');
    SELECT pg_sleep(10);
    SELECT albums.album_id, albums.release_date
        FROM (((albums
            INNER JOIN bands ON albums.band_id = bands.band_id)
            INNER JOIN bands_genre ON bands.band_id = bands_genre.band_id)
            INNER JOIN genres ON bands_genre.genre_id = genres.genre_id)
            WHERE genres.genre_name = 'Math rock'
            AND albums.release_date = '1980-01-01'
            AND LENGTH(albums.abstract) > 200
            GROUP BY albums.album_id
            ORDER BY albums.sales
            DESC;
COMMIT;

SELECT pg_sleep(5);

SELECT albums.album_id
    FROM (((albums
        INNER JOIN bands ON albums.band_id = bands.band_id)
        INNER JOIN bands_genre ON bands.band_id = bands_genre.band_id)
        INNER JOIN genres ON bands_genre.genre_id = genres.genre_id)
        WHERE genres.genre_name = 'Math rock'
        AND albums.release_date >= '1990/01/01'
        AND albums.release_date <= '1999/12/31'
        AND LENGTH(albums.abstract) > 200
        GROUP BY albums.album_id
        ORDER BY albums.sales
        DESC
        LIMIT 5;
\set AUTCOMMIT on
COMMIT;
```

Appendix 3 - Query 1 transaction in *PL/pgSQL*, with isolation level implemented.



```
\set AUTCOMMIT off
BEGIN;
SELECT album_id, band_id, release_date, sales
    FROM albums as A
    WHERE A.release_date >= '1998/08/10'
    AND A.release_date <= '1998/08/17'
    GROUP BY A.album_id, A.band_id, A.release_date, A.sales
    ORDER BY A.sales
    DESC;
SELECT pg_sleep(20);
SELECT album_id, band_id, release_date, sales
    FROM albums as A
    WHERE A.release_date >= '1998/08/10'
    AND A.release_date <= '1998/08/17'
    GROUP BY A.album_id, A.band_id, A.release_date, A.sales
    ORDER BY A.sales
    DESC;
\set AUTCOMMIT on
COMMIT;
```

```
\set AUTCOMMIT off
BEGIN;
SELECT insert_new_album('34716','TEST ALBUM','59','30.0','1998/08/16','TEST ALBUM ABSTRACT');
\set AUTCOMMIT on
COMMIT;
```

Appendix 4 - Transactions for concurrency experiments with Phantom Read phenomena in *PL/pgSQL*.

```
\set AUTCOMMIT off
BEGIN;
LOCK TABLE albums IN EXCLUSIVE MODE;
SELECT album_id, band_id, release_date, sales
    FROM albums as A
    WHERE A.release_date >= '1998/08/10'
    AND A.release_date <= '1998/08/17'
    GROUP BY A.album_id, A.band_id, A.release_date, A.sales
    ORDER BY A.sales
    DESC;
SELECT pg_sleep(20);
SELECT album_id, band_id, release_date, sales
    FROM albums as A
    WHERE A.release_date >= '1998/08/10'
    AND A.release_date <= '1998/08/17'
    GROUP BY A.album_id, A.band_id, A.release_date, A.sales
    ORDER BY A.sales
    DESC;
\set AUTCOMMIT on
COMMIT;
```

Appendix 5 - Transactions 1 for concurrency experiments with Phantom Read phenomena in *PL/pgSQL*, with implemented locks.

```
\set AUTCOMMIT off
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT album_id, band_id, release_date, sales
    FROM albums as A
    WHERE A.release_date >= '1998/08/10'
    AND A.release_date <= '1998/08/17'
    GROUP BY A.album_id, A.band_id, A.release_date, A.sales
    ORDER BY A.sales
    DESC;
SELECT pg_sleep(20);
SELECT album_id, band_id, release_date, sales
    FROM albums as A
    WHERE A.release_date >= '1998/08/10'
    AND A.release_date <= '1998/08/17'
    GROUP BY A.album_id, A.band_id, A.release_date, A.sales
    ORDER BY A.sales
    DESC;
\set AUTCOMMIT on
COMMIT;
```

Appendix 6 - Transactions 1 for concurrency experiments with Phantom Read phenomena in *PL/pgSQL*, with implemented isolation levels.

```
\set AUTCOMMIT off
BEGIN;
    SELECT update_sales_Q2();
    SELECT band_id, sales, release_date, running_time FROM albums WHERE sales = 0;
COMMIT;

SELECT pg_sleep(3);

SELECT band_id, sales, release_date, running_time FROM albums WHERE sales = 0;
\set AUTCOMMIT on
COMMIT;
```

Appendix 7 - Query 2 transaction in *PL/pgSQL*.

```
\set AUTOCOMMIT off
BEGIN;
    SELECT update_sales_lock_Q2();
    SELECT band_id,album_id, sales, release_date, running_time FROM albums WHERE sales = 0;
COMMIT;

SELECT pg_sleep(3);

SELECT band_id,album_id, sales, release_date, running_time FROM albums WHERE sales = 0;
\set AUTOCOMMIT on
COMMIT;
```

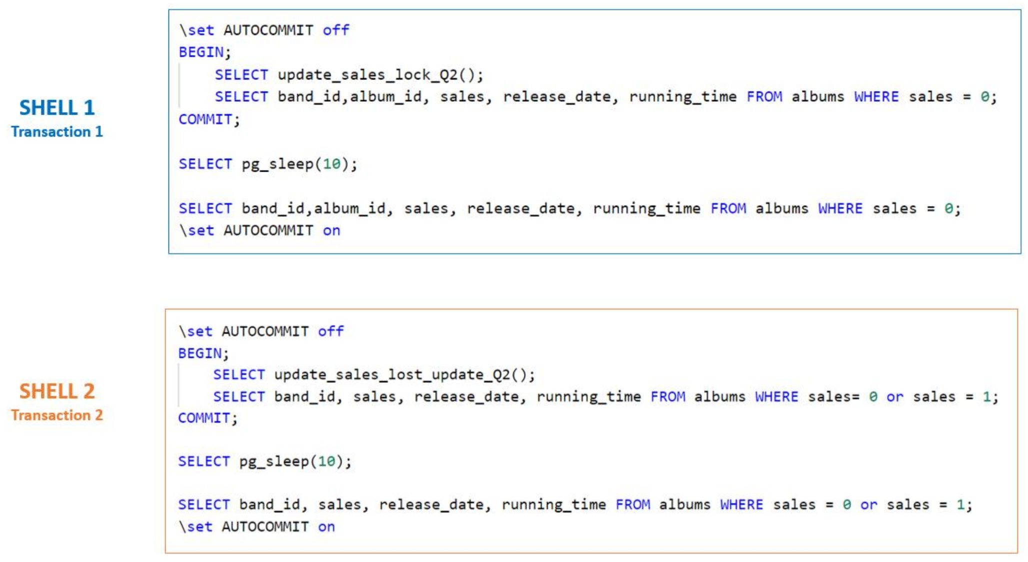
Appendix 8 - Query 2 transaction in *PL/pgSQL*, with explicit locks implemented.

```
\set AUTOCOMMIT off
BEGIN TRANSACTION ISOLATION LEVEL Read Committed;
    SELECT update_sales_Q2();
    SELECT band_id, sales, release_date, running_time FROM albums WHERE sales = 0;
COMMIT;

SELECT pg_sleep(10);

SELECT band_id, sales, release_date, running_time FROM albums WHERE sales = 0;
\set AUTOCOMMIT on
COMMIT;
```

Appendix 9 - Query 2 transaction in *PL/pgSQL*, with isolation level implemented.

```
\set AUTCOMMIT off
BEGIN;
    SELECT update_sales_lost_update_Q2();
    SELECT band_id, sales, release_date, running_time FROM albums WHERE sales = 1;
COMMIT;

SELECT pg_sleep(10);

SELECT band_id, sales, release_date, running_time FROM albums WHERE sales = 1;
\set AUTCOMMIT on
```

Appendix 10 - Transaction (*query 2*) for concurrency experiments with *Lost Update* Phenomena in *PL/pgSQL*.

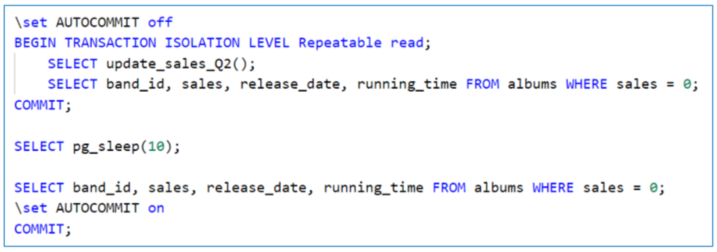


```
\set AUTOCOMMIT off
BEGIN;
    SELECT update_sales_lock_Q2();
    SELECT band_id,album_id, sales, release_date, running_time FROM albums WHERE sales = 0;
COMMIT;

SELECT pg_sleep(10);

SELECT band_id,album_id, sales, release_date, running_time FROM albums WHERE sales = 0;
\set AUTOCOMMIT on
```

```
\set AUTOCOMMIT off
BEGIN;
    SELECT update_sales_lost_update_Q2();
    SELECT band_id, sales, release_date, running_time FROM albums WHERE sales= 0 or sales = 1;
COMMIT;

SELECT pg_sleep(10);

SELECT band_id, sales, release_date, running_time FROM albums WHERE sales = 0 or sales = 1;
\set AUTOCOMMIT on
```

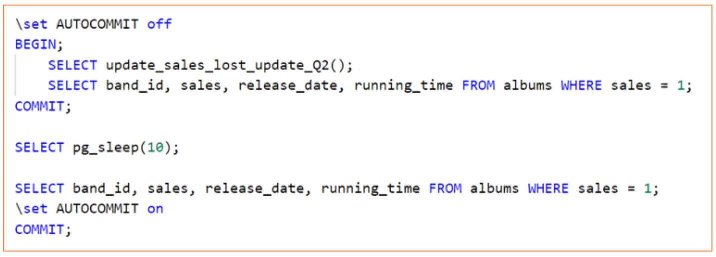
Appendix 11 - Transaction (*query 2*) with implemented lock in transaction 1, for concurrency experiments with *Lost Update* Phenomena in *PL/pgSQL*.

SHELL 1
Transaction 1



```
\set AUTOCOMMIT off
BEGIN TRANSACTION ISOLATION LEVEL Repeatable read;
    SELECT update_sales_Q2();
    SELECT band_id, sales, release_date, running_time FROM albums WHERE sales = 0;
COMMIT;

SELECT pg_sleep(10);

SELECT band_id, sales, release_date, running_time FROM albums WHERE sales = 0;
\set AUTOCOMMIT on
COMMIT;
```

SHELL 2
Transaction 2



```
\set AUTOCOMMIT off
BEGIN;
    SELECT update_sales_lost_update_Q2();
    SELECT band_id, sales, release_date, running_time FROM albums WHERE sales = 1;
COMMIT;

SELECT pg_sleep(10);

SELECT band_id, sales, release_date, running_time FROM albums WHERE sales = 1;
\set AUTOCOMMIT on
COMMIT;
```

Appendix 12 - Transaction (*query 2*) with implemented isolation level in transaction 1, for concurrency experiments with *Lost Update* Phenomena in *PL/pgSQL*.