# LAB 11

## Exercise 1

Result of Step 1:

```
username |     fullname      | balance | group_id
----------+-------------------+---------+----------
 bitdiddl | Ben Bitdiddle     |      65 |        1
 mike     | Michael Dole      |      73 |        2
 alyssa   | Alyssa P. Hacker |      79 |        3
 bbrown   | Bob Brown         |     100 |        3
 jones    | Alice Jones       |      82 |        1
```

Result of step 3:

```
username |     fullname      | balance | group_id
----------+-------------------+---------+----------
 bitdiddl | Ben Bitdiddle     |      65 |        1
 mike     | Michael Dole      |      73 |        2
 alyssa   | Alyssa P. Hacker |      79 |        3
 bbrown   | Bob Brown         |     100 |        3
 jones    | Alice Jones       |      82 |        1
```

Result of step 4:

```
username |     fullname      | balance | group_id
----------+-------------------+---------+----------
 bitdiddl | Ben Bitdiddle     |      65 |        1
 mike     | Michael Dole      |      73 |        2
 alyssa   | Alyssa P. Hacker |      79 |        3
 bbrown   | Bob Brown         |     100 |        3
 ajones   | Alice Jones       |      82 |        1
(5 rows)
```

Result of step 5:

```
username |     fullname      | balance | group_id
----------+-------------------+---------+----------
 bitdiddl | Ben Bitdiddle     |      65 |        1
 mike     | Michael Dole      |      73 |        2
 alyssa   | Alyssa P. Hacker |      79 |        3
 bbrown   | Bob Brown         |     100 |        3
 ajones   | Alice Jones       |      92 |        1     - both
```

Result of step 8:

```
update account
set balance = balance +20
where fullname='Alice Jones’;
| <–cursor
```

*The idea behind read committed is that within a given transaction we can read only committed changes, as seen from steps 3–4, until changes are not committed, transactions see the account table differently. When T2 commits changes, T1 automatically see these changes, as seen from step 5.*

*From postgresql doc:*

“two successive SELECT commands can see different data, even though they are within a single transaction, if other transactions commit changes after the first SELECT starts and before the second SELECT starts.”

*As for update, T2 will wait for T1 to either commit or rollback and then it will automatically execute the query with message: UPDATE 1.*

## Repeatable read:

Same outputs at steps 1–4 for the repeatable read

Result at step 5:

T1:

```
username |     fullname     | balance | group_id
----------+------------------+---------+----------
 bitdiddl | Ben Bitdiddle    |      65 |        1
 mike     | Michael Dole     |      73 |        2
 alyssa   | Alyssa P. Hacker |      79 |        3
 bbrown   | Bob Brown        |     100 |        3
 ajones   | Alice Jones      |      92 |        1
```

T2:

```
username |     fullname     | balance | group_id
----------+------------------+---------+----------
 bitdiddl | Ben Bitdiddle    |      65 |        1
 mike     | Michael Dole     |      73 |        2
 alyssa   | Alyssa P. Hacker |      79 |        3
 bbrown   | Bob Brown        |     100 |        3
 jones    | Alice Jones      |      92 |        1
```

Result at step 8:

```
update account
set balance = balance +20
where fullname='Alice Jones';
| <–cursor
```

*As for repeatable read, we can see that steps 3–4 result in the same output. However, step 5 result in different output. T2 sees the changes, T1 doesn't. This is due to the fact that 'repeatable read'* transaction sees a snapshot as of the start of the first SELECT statement in the transaction. It guarantees that read data would stay unchanged within a transaction, unless altered within the transaction.

*As for update, T2 will wait for T1 to either commit or rollback (as shown in step 8). However, if T1 commits, the T2 will display a message*

*ERROR: could not serialize access due to concurrent update*

*(happened, when I accidentally committed T1 first). As I understood, this happens because repeatable read cannot change rows modified by other transactions.*

*If T1 rolls back then T2 commits easily.*

# Exercise 2

Result of step 2:

```
 username |   fullname   | balance | group_id
----------+--------------+---------+----------
 mike     | Michael Dole |      73 |        2
(1 row)
```

Result of step 4:

```
 username |   fullname   | balance | group_id
----------+--------------+---------+----------
 mike     | Michael Dole |      73 |        2
```

```
(1 row)
```

Result at step 6:

T1:

```
username |     fullname     | balance | group_id
----------+------------------+---------+----------
 bitdiddl | Ben Bitdiddle    |      65 |        1
 alyssa   | Alyssa P. Hacker |      79 |        3
 jones    | Alice Jones      |      92 |        1
 bbrown   | Bob Brown        |     100 |        2
 mike     | Michael Dole     |      88 |        2
(5 rows)
```

T2:

```
username |     fullname     | balance | group_id
----------+------------------+---------+----------
 bitdiddl | Ben Bitdiddle    |      65 |        1
 alyssa   | Alyssa P. Hacker |      79 |        3
 jones    | Alice Jones      |      92 |        1
 bbrown   | Bob Brown        |     100 |        2
 mike     | Michael Dole     |      88 |        2
(5 rows)
```

Again, for read committed we can see that does not matter what T2 does, T1 will not see uncommitted changes within a transaction. No update conflict then, since for T1 only Mike is in group 2, whilst T2 operates on Bob. When both changes are committed, both T1 and T2 gather changes from each other (as seen from the last step)

## Repeatable read:

Result at step 2:

```
username |   fullname   | balance | group_id
----------+--------------+---------+----------
 mike     | Michael Dole |      88 |        2
(1 row)
```

Result at step 4:

```
username |   fullname   | balance | group_id
----------+--------------+---------+----------
 mike     | Michael Dole |      88 |        2
```

```
(1 row)

Result at step 6:

 username |     fullname      | balance | group_id
----------+-------------------+---------+----------
 bitdiddl | Ben Bitdiddle     |      65 |        1
 alyssa   | Alyssa P. Hacker  |      79 |        3
 jones    | Alice Jones       |      92 |        1
 bbrown   | Bob Brown         |     100 |        2
 mike     | Michael Dole      |     103 |        2
(5 rows)
```

As for repeatable read we can see that changes within transactions do not affect other. Consecutive select operations keep the access tables consistent. No conflict on updates at steps 3 and 5, since each transaction updates unchanged row, which is allowed. On the last step commits do not conflict also. The output table accommodates changes from T1 and T2. On the other hand, if select in T1 was executed before transaction was ended – no changes would be seen.

## Exercise 3

```
Result at step 3:

 sum
-----
 103
(1 row)

Result at step 5:
 username |   fullname   | balance | group_id
----------+--------------+---------+----------
 mike     | Michael Dole |     103 |        2
(1 row)

Result at step 7:

T1:

username |   fullname   | balance | group_id
----------+--------------+---------+----------
 mike     | Michael Dole |     206 |        2
(1 row)
```

```
T2:

username |   fullname   | balance | group_id
----------+--------------+---------+----------
 mike     | Michael Dole |     103 |        2
 bbrown   | Bob Brown    |     100 |        2
(2 rows)

At last :

username |     fullname     | balance | group_id
----------+------------------+---------+----------
 bitdiddl | Ben Bitdiddle    |      65 |        1
 alyssa   | Alyssa P. Hacker |      79 |        3
 jones    | Alice Jones      |      92 |        1
 bbrown   | Bob Brown        |     100 |        2
 mike     | Michael Dole     |     206 |        2
(5 rows) - both
```

## Serializable:

```
Result at step 3:

sum
-----
 206
(1 row)

Result at step 5:


username |   fullname   | balance | group_id
----------+--------------+---------+----------
 mike     | Michael Dole |     206 |        2
(1 row)

Result at step 7:

T1:

username |   fullname   | balance | group_id
----------+--------------+---------+----------
 mike     | Michael Dole |     412 |        2
(1 row)
T2:
```

```
username |   fullname   | balance | group_id
----------+--------------+---------+----------
 mike     | Michael Dole |     206 |        2
 bbrown   | Bob Brown    |     100 |        2
(2 rows)
```

Result at step 9 (T2 commit):

```
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

I wanted to explain this first)) Everything is fine as we work in separate transactions until we want to commit. If T1 and T2 try to commit, T1 will be committed and T2 will get message:

```
ERROR:  could not serialize access due to read/write dependencies among transactions
```

This happens since serialisable considers whether the serial order of execution consistent with the result. Let us explore our case:
If T1 was executed before T2 we would have the output:

```
 username |     fullname      | balance | group_id
----------+-------------------+---------+----------
 bitdiddl | Ben Bitdiddle     |      65 |        1
 alyssa   | Alyssa P. Hacker  |      79 |        3
 jones    | Alice Jones       |      92 |        1
 mike     | Michael Dole      |     412 |        2
 bbrown   | Bob Brown         |     100 |        2 <—T2 changes after
(5 rows)
```

However, if T2 was executed first, the table would look like this:

```
 username |     fullname      | balance | group_id
----------+-------------------+---------+----------
 bitdiddl | Ben Bitdiddle     |      65 |        1
 alyssa   | Alyssa P. Hacker  |      79 |        3
 jones    | Alice Jones       |      92 |        1
 mike     | Michael Dole      |     512 |        2
 bbrown   | Bob Brown         |     406 |        2
(5 rows)
```

So the balance column is different, that’s why serialisable does not allow such manipulations.

As for repeatable read, such commits are ok with the given order of commands.

## Commands used:

```
–select ∗ from account;

–begin;

– set transaction isolation level read committed;

–update account
set username = 'ajones'
where fullname='Alice Jones’; –updates Alice’s username

–update account
set balance = balance +20
where fullname='Alice Jones';–updates Alice’s balance

–update account
set
    balance = balance +15
where
    group_id = 2; –updates balance for all people with id 2

–select sum(balance)
from account
where group_id=2; –gets the sum of balances with id 2

–set
    balance = balance + (select sum(balance) from account where
group_id=2)
where
    group_id = 2; –add sum from prev transaction to balances of id=2

–select ∗ from account where group_id=2; –get all accounts with id=2

–update account
set group_id = '2'
where fullname='Bob Brown’; –move Bob to group 2
```