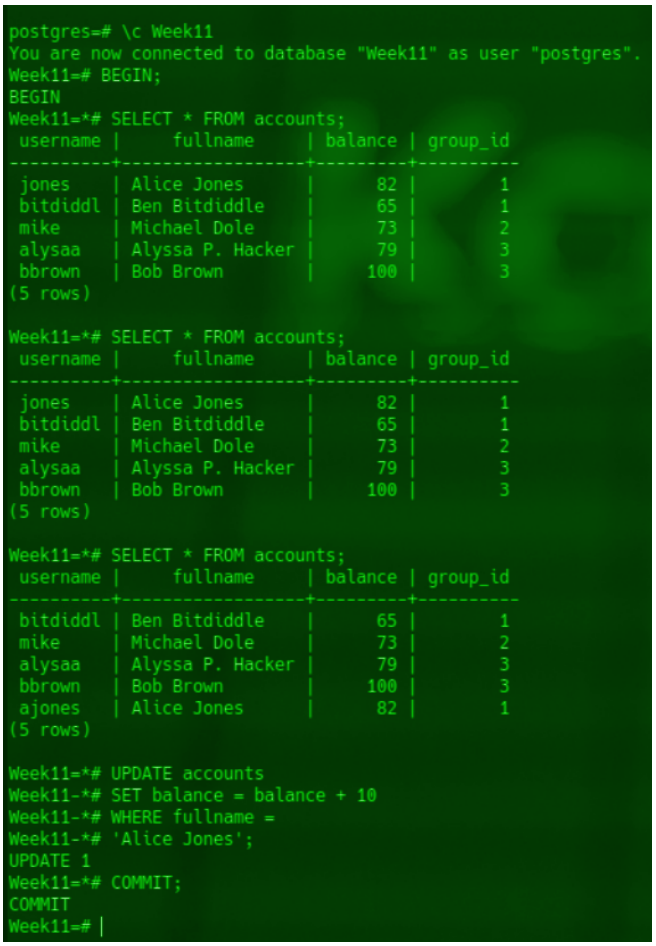
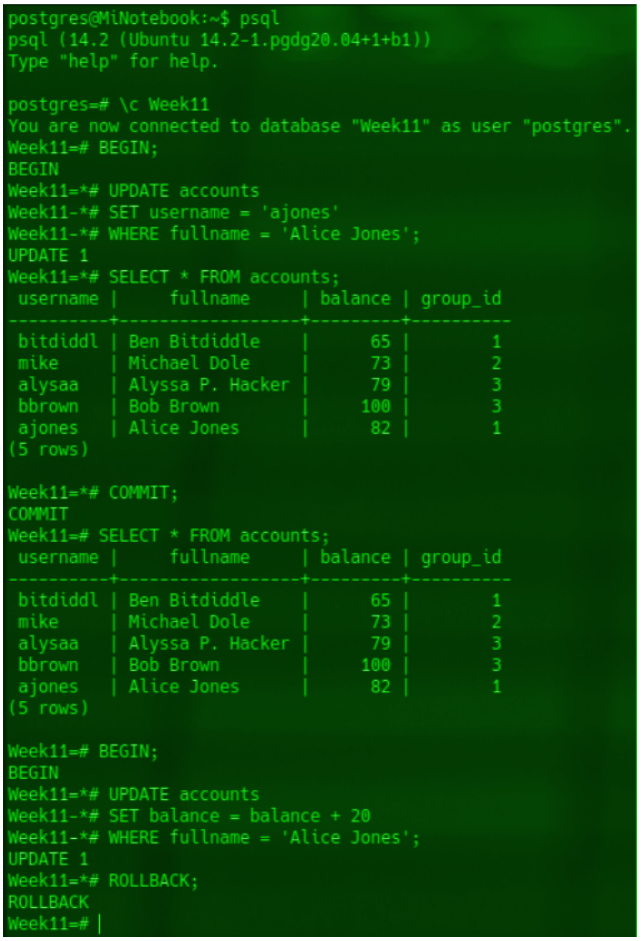
DB Lab 10 Report
Konstantin Smirnov

Exercise 2
To generate required dataset, I used the same approach as in the exercise 1 — python script. You can explore in the file ex2.py . The table I created and worked with is called accounts.

Read committed. First, let's evaluate the result of using READ COMMITTED isolation level transactions. To do it, we do not need to specify any isolation level information, it is chosen by default.

BEGIN;
SELECT * FROM accounts;
COMMIT;





Now, when we see the result, it's possible to answer questions:

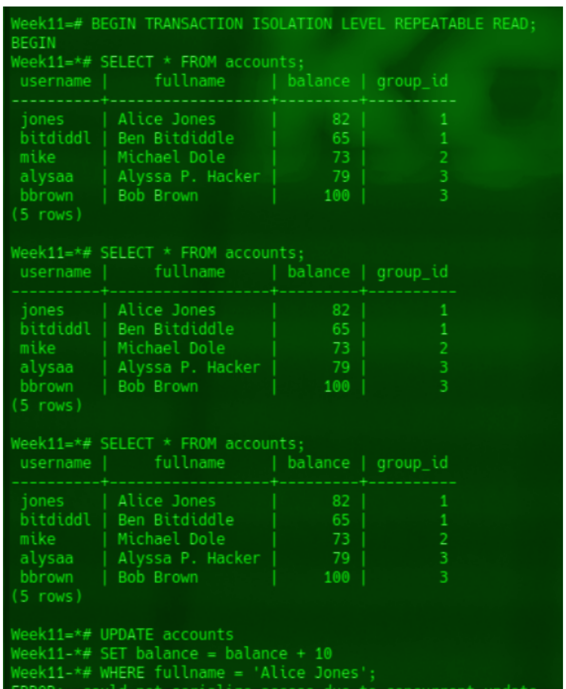
1. Do both terminals show the same information? Explain the reason. We can see that terminals see the different information: terminal 1 shows the old information before update, meanwhile terminal 2 shows information after an update. The reason is that the changes wasn't committed yet in the second terminal.
2. Not answer on the question, just mentioning. After committing the update in the second terminal, SELECT from terminal 1 shows new changes. As we would see below, this is a difference between READ COMMITTED and REPEATABLE READ isolation levels.
3. Explain the output form the second terminal. When I enter the update in the second terminal, it freezes for a while.

However, after committing changes in the
terminal 1, it automatically accepts and terminal 2 becomes available again.
As I understand, the Postgres freezes all the
read/write transactions, except of one (the firstone) and then allows them to execute after the end of current active transaction.

Repeatable read. Now, let's evaluate the result of using REPEATED READ isolation level transactions. Now we have to specify explicitly the isolation level we use:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM accounts;
COMMIT;
```

Below you can see screenshots from Terminal 1 and Terminal 2 with the result of completed steps:

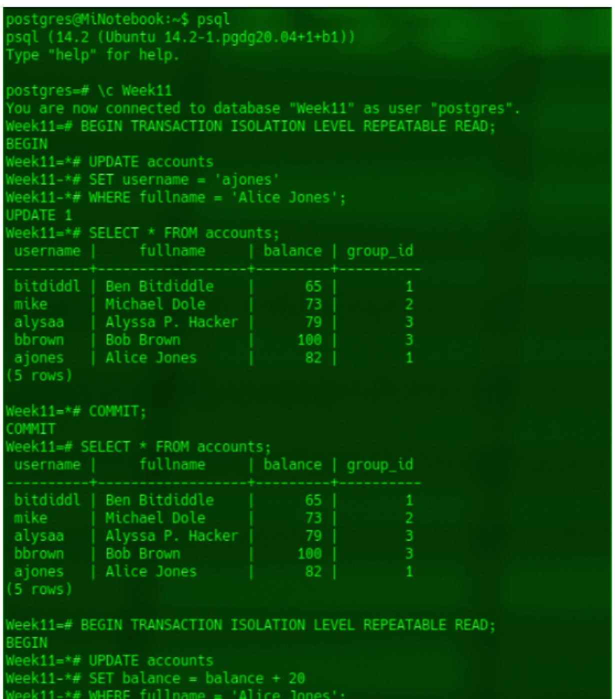


```
Week11=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
Week11=*# SELECT * FROM accounts;
 username |      fullname     | balance | group_id
----------+-------------------+---------+----------
 jones    | Alice Jones       |      82 |        1
 bitdiddl | Ben Bitdiddle     |      65 |        1
 mike     | Michael Dole      |      73 |        2
 alysaa   | Alyssa P. Hacker  |      79 |        3
 bbrown   | Bob Brown         |     100 |        3
(5 rows)

Week11=*# SELECT * FROM accounts;
 username |      fullname     | balance | group_id
----------+-------------------+---------+----------
 jones    | Alice Jones       |      82 |        1
 bitdiddl | Ben Bitdiddle     |      65 |        1
 mike     | Michael Dole      |      73 |        2
 alysaa   | Alyssa P. Hacker  |      79 |        3
 bbrown   | Bob Brown         |     100 |        3
(5 rows)

Week11=*# SELECT * FROM accounts;
 username |      fullname     | balance | group_id
----------+-------------------+---------+----------
 jones    | Alice Jones       |      82 |        1
 bitdiddl | Ben Bitdiddle     |      65 |        1
 mike     | Michael Dole      |      73 |        2
 alysaa   | Alyssa P. Hacker  |      79 |        3
 bbrown   | Bob Brown         |     100 |        3
(5 rows)

Week11=*# UPDATE accounts
Week11-*# SET balance = balance + 10
Week11-*# WHERE fullname = 'Alice Jones';
```



```
postgres@MiNotebook:~$ psql
psql (14.2 (Ubuntu 14.2-1.pgdg20.04+1+b1))
Type "help" for help.

postgres=# \c Week11
You are now connected to database "Week11" as user "postgres".
Week11=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
Week11=*# UPDATE accounts
Week11-*# SET username = 'ajones'
Week11-*# WHERE fullname = 'Alice Jones';
UPDATE 1
Week11=*# SELECT * FROM accounts;
 username |      fullname     | balance | group_id
----------+-------------------+---------+----------
 bitdiddl | Ben Bitdiddle     |      65 |        1
 mike     | Michael Dole      |      73 |        2
 alysaa   | Alyssa P. Hacker  |      79 |        3
 bbrown   | Bob Brown         |     100 |        3
 ajones   | Alice Jones       |      82 |        1
(5 rows)

Week11=*# COMMIT;
COMMIT
Week11=# SELECT * FROM accounts;
 username |      fullname     | balance | group_id
----------+-------------------+---------+----------
 bitdiddl | Ben Bitdiddle     |      65 |        1
 mike     | Michael Dole      |      73 |        2
 alysaa   | Alyssa P. Hacker  |      79 |        3
 bbrown   | Bob Brown         |     100 |        3
 ajones   | Alice Jones       |      82 |        1
(5 rows)

Week11=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
Week11=*# UPDATE accounts
Week11-*# SET balance = balance + 20
Week11-*# WHERE fullname = 'Alice Jones';
```

Now, when we see the result, it’s possible to answer questions:

1. Do both terminals show the same information? Explain the reason. Nothing changed since the previous case: same result and explanation.
2. Not answer on the question, just mentioning. After committing the update in the second terminal, we can see the difference between READ COMMITTED and REPEATABLE READ: terminal 1 still gets the old database snapshot, despite new changes was already committed.
3. Explain the output form the second terminal. The second terminal successfully updates database, since it works with the latest database snapshot. Meanwhile, the second terminal works with old snapshot, since it got the serialization error when it tries to update the database.

Read committed

It’s not hard to see that the result of both transactions is the same.
The explanation is that in both cases terminals work with initial data snapshot and
then the changes merge together as separate once. This is because both cases
suffer from serialization anomaly. This problem could be fixed if we use SERIALIZABLE isolation level.