MultiVersion Concurrency Control in Postgres

Handling Concurrent access to a database by different users will always be an important factor in showing how efficient a database is. Postgres is well known for its promise in this regard - “Read never Blocks Write,Write never Blocks Read!”. It achieves this by a mechanism called MVCC or MultiVersion Concurrency Control. This method is not unique to Postgres. Many advanced databases have implemented their own versions of MVCC which suits their requirements.

We give a brief idea how this works internally.

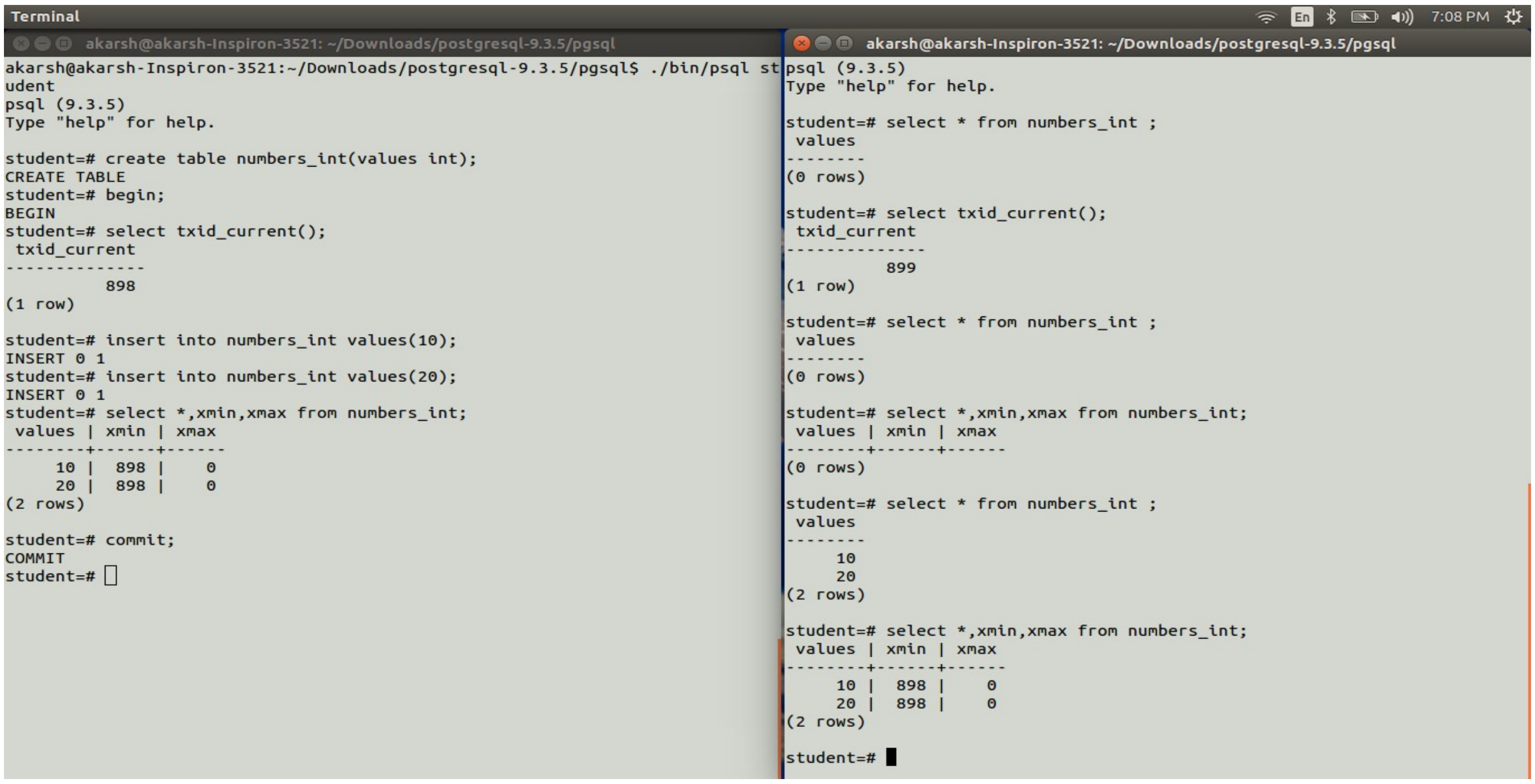
Every transaction in postgres gets a transaction ID called XID. This includes single one statement transactions such as an insert, update or delete, as well as explicitely wrapping a group of statements together via BEGIN - COMMIT. When a transaction starts, Postgres increments an XID and assigns it to the current transaction. This xid is not visible to the user, However it can be accessed using 'Select txid_current();' statement in the psql shell. Postgres also stores transaction information on every row in the system, which is used to determine whether a row is visible to the transaction or not.

Concurrency Control During Insertion:

When we insert a row, postgres will store the XID in the row and call it xmin. Every row that has been committed and has an xmin that is less than the current transaction's XID is visible to the transaction. This means that you can start a transaction and insert a row, and until that transaction COMMITs that row will not be visible to other transactions. Once it commits and other transaction BEGINS, they will be able to view the new row because they satisfy the xmin < XID condition – and the transaction that created the row has completed.

This is shown in the below example which was recorded on a Ubuntu 14.04 LTS machine running PostgreSQL 9.3.5

A Database 'Student' is already created. A table 'numbers_int' is created with a column 'values' of integer datatype.



```
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/psql st
udent
psql (9.3.5)
Type "help" for help.

student=# create table numbers_int(values int);
CREATE TABLE
student=# begin;
BEGIN
student=# select txid_current();
 txid_current
--------------
          898
(1 row)

student=# insert into numbers_int values(10);
INSERT 0 1
student=# insert into numbers_int values(20);
INSERT 0 1
student=# select *,xmin,xmax from numbers_int;
 values | xmin | xmax
--------+------+------
     10 |  898 |    0
     20 |  898 |    0
(2 rows)

student=# commit;
COMMIT
student=#
```

```
psql (9.3.5)
Type "help" for help.

student=# select * from numbers_int ;
 values
--------
(0 rows)

student=# select txid_current();
 txid_current
--------------
          899
(1 row)

student=# select * from numbers_int ;
 values
--------
(0 rows)

student=# select *,xmin,xmax from numbers_int;
 values | xmin | xmax
--------+------+------
(0 rows)

student=# select * from numbers_int ;
 values
--------
     10
     20
(2 rows)

student=# select *,xmin,xmax from numbers_int;
 values | xmin | xmax
--------+------+------
     10 |  898 |    0
     20 |  898 |    0
(2 rows)

student=#
```

A new transaction(A) is created by using BEGIN.Transaction ID is checked using the ' Select txid_current(); ' Command.(xid=898)

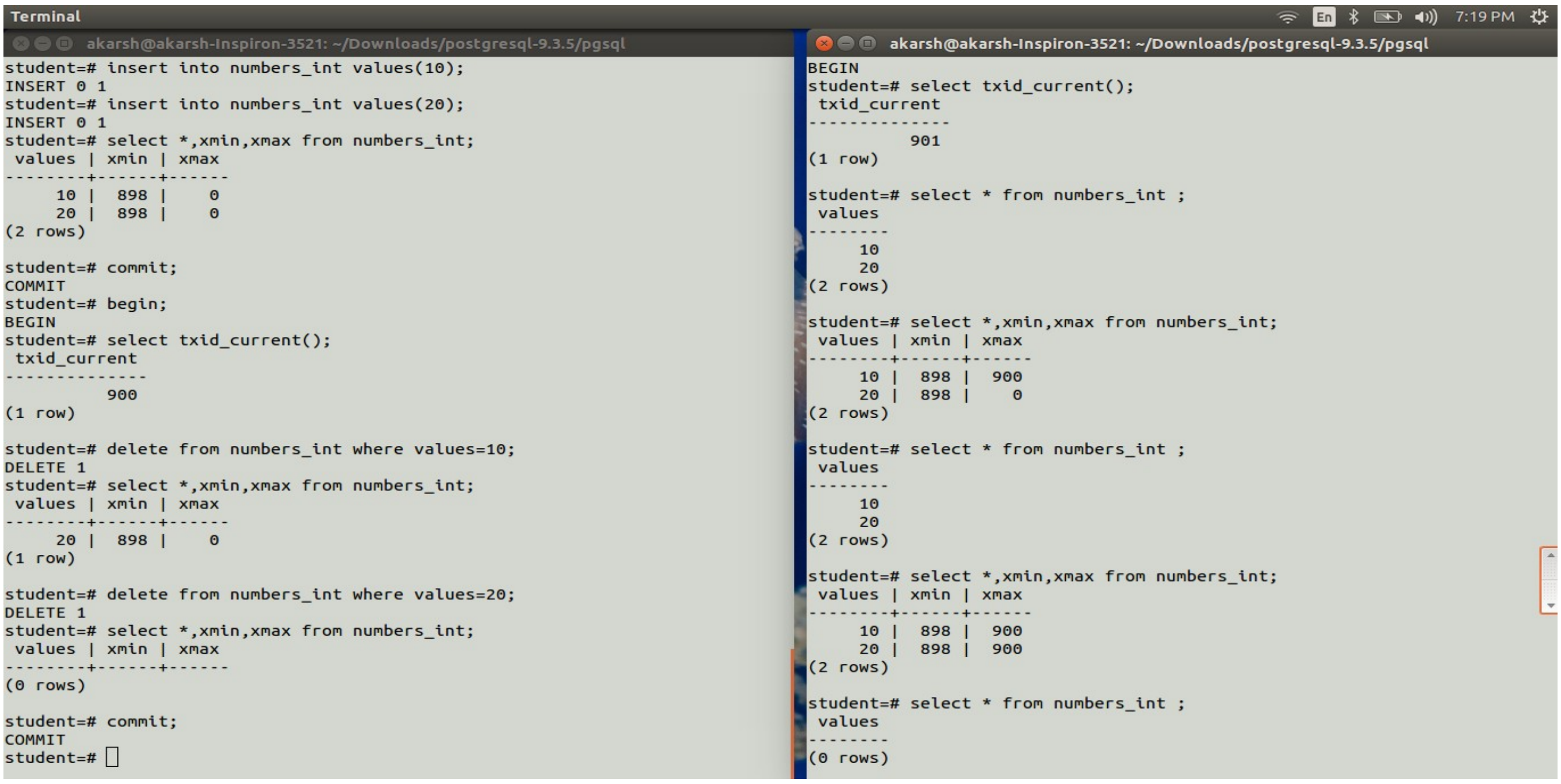
Concurrently, another terminal is accessing the same database and checks the contents of the table numbers_int. Lets call this Transaction as B with xid=899

Now,in T(A) we Insert values 10 and 20 into the column 'values' in the table. However, we see that on querying all the values of the table in T(B) it does not show the newly inserted values 10 & 20.

Since T(A) does insertion, the Xid of the transaction becomes the xmin value of the newly added rows. However, this does not appear in T(B) either.

Now we COMMIT the changes of T(A). The rows get updated along with their respective xmin values.Now T(B) can access these rows for further transactions.

Concurrency Control During Deletion:


Terminal
En
7:19 PM
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql

```
student=# insert into numbers_int values(10);
INSERT 0 1
student=# insert into numbers_int values(20);
INSERT 0 1
student=# select *,xmin,xmax from numbers_int;
 values | xmin | xmax
--------+------+------
     10 |  898 |    0
     20 |  898 |    0
(2 rows)

student=# commit;
COMMIT
student=# begin;
BEGIN
student=# select txid_current();
 txid_current
--------------
          900
(1 row)

student=# delete from numbers_int where values=10;
DELETE 1
student=# select *,xmin,xmax from numbers_int;
 values | xmin | xmax
--------+------+------
     20 |  898 |    0
(1 row)

student=# delete from numbers_int where values=20;
DELETE 1
student=# select *,xmin,xmax from numbers_int;
 values | xmin | xmax
--------+------+------
(0 rows)

student=# commit;
COMMIT
student=#
```

```
BEGIN
student=# select txid_current();
 txid_current
--------------
          901
(1 row)

student=# select * from numbers_int ;
 values
--------
     10
     20
(2 rows)

student=# select *,xmin,xmax from numbers_int;
 values | xmin | xmax
--------+------+------
     10 |  898 |  900
     20 |  898 |    0
(2 rows)

student=# select * from numbers_int ;
 values
--------
     10
     20
(2 rows)

student=# select *,xmin,xmax from numbers_int;
 values | xmin | xmax
--------+------+------
     10 |  898 |  900
     20 |  898 |  900
(2 rows)

student=# select * from numbers_int ;
 values
--------
(0 rows)
```

In this case, two transactions T(A) and T(B) with xid 900 and 901 respectively are shown working concurrently. T(A) first deletes the value=10 from the table, but it is not shown to T(B) . T(B) still has that row in its snapshot. But on checking the xmin and xmax values we see that the Transaction ID of T(A) which just deleted that row becomes the xmax value for that row. Same happens to value=20. On COMMIT, both the rows become invisible to Both the transactions as the xmax value is less than their current Transaction Ids.

Using Insertion and Deletion, we have seen how MVCC controls concurrency in Postgres. This type of handling the isolation of various snapshots of the database from different transactions is called 'Read Committed' which is the default Transaction isolation level in Postgres. Basically, this says that each transaction gets the most recently committed snapshot of the database. Thus untill one transaction COMMITs, another cannot access the modified tuples.

Postgres also has 2 other transaction Isolation levels:

-->Repeated Read Isolation level

By setting the isolation level to this, we allow postgres to provide the last committed snapshot before the transaction begins. And this snapshot does not change whether or not another concurrent transaction has commited some changes into the database. In case the isolated transaction wants to use the values that were committed after the transaction began, then the transaction rolls back (ROLLBACK) to the beginning of that transaction and re-executes the transaction taking the newly committed values as well.

The Repeatable Read mode guarantees that each transaction sees a completely stable view of the database. However, this view will not necessarily always be consistent with some serial (one at a time) execution of concurrent transactions of the same level.

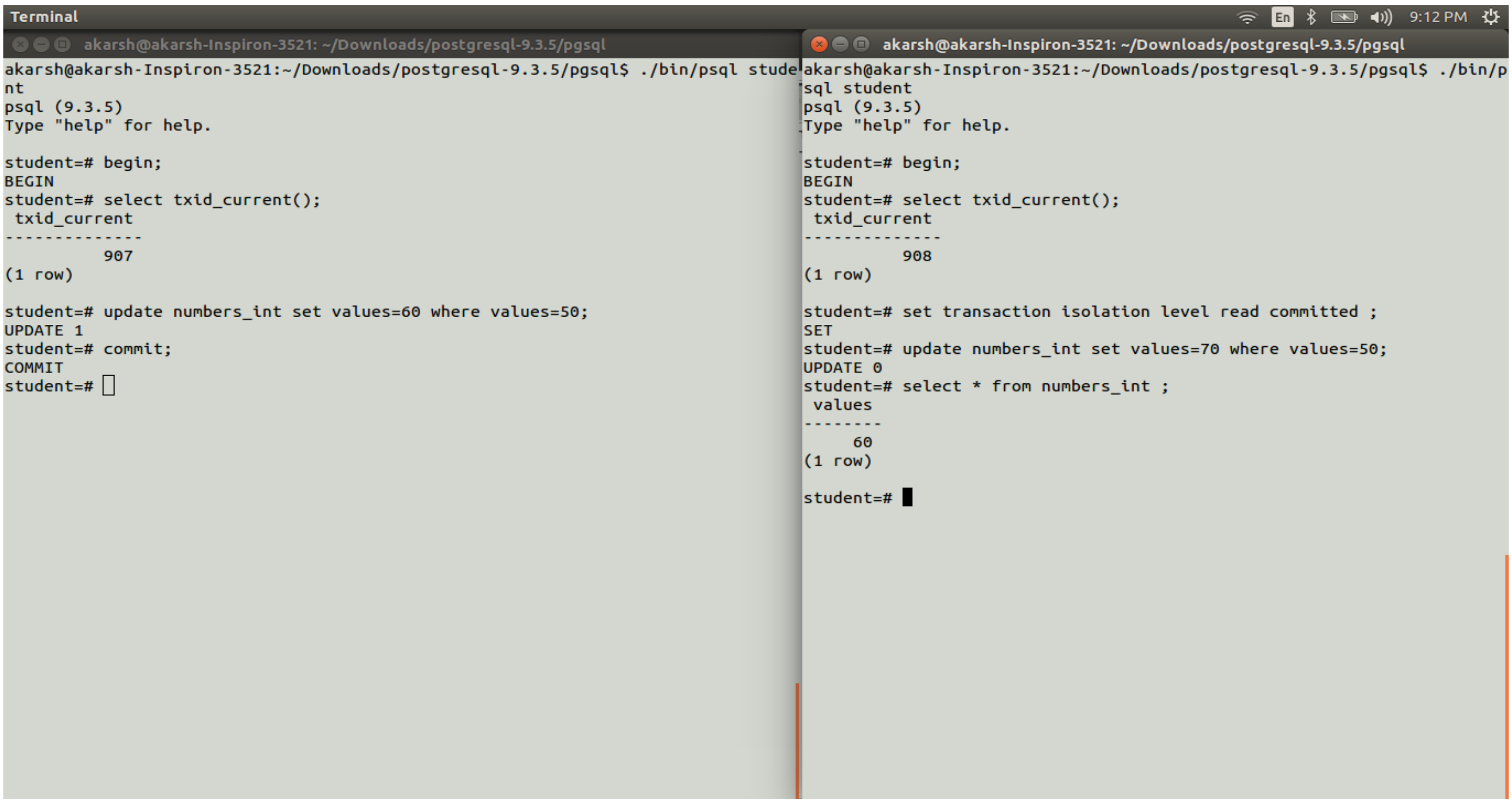
-->Serializable Isolation level:

This is the most strictest level of isolation postgres provides. This ensures that all the concurrent transactions will only be acceptable if they can be run serially, and produce the same outcome. In fact, this isolation level works exactly the same as Repeatable Read except that it monitors for conditions which could make execution of a concurrent set of serializable transactions behave in a manner inconsistent with all possible serial (one at a time) executions of those transactions. This monitoring does not introduce any blocking beyond that present in repeatable read, but there is some overhead to the monitoring, and detection of the conditions which could cause a *serialization anomaly* will trigger a *serialization failure*.

Concurrency control during Update:

->Read Committed Isolation level:

To show how this works, we have two concurrent transactions A and B with xid 907 and 908 respectively. In A we update the value from 50 to 60 but we do not commit it yet. In B we set the transaction isolation level to Read committed and then update the value of 50 to 70. This did not return the control back to the user . This is because B was waiting for A to commit the value since it has modified the value but yet to commit.Once A commits,B tries to update the value 50,but since value=50 does not exist anymore (It has been modified to 60 by A) ,the update fails.

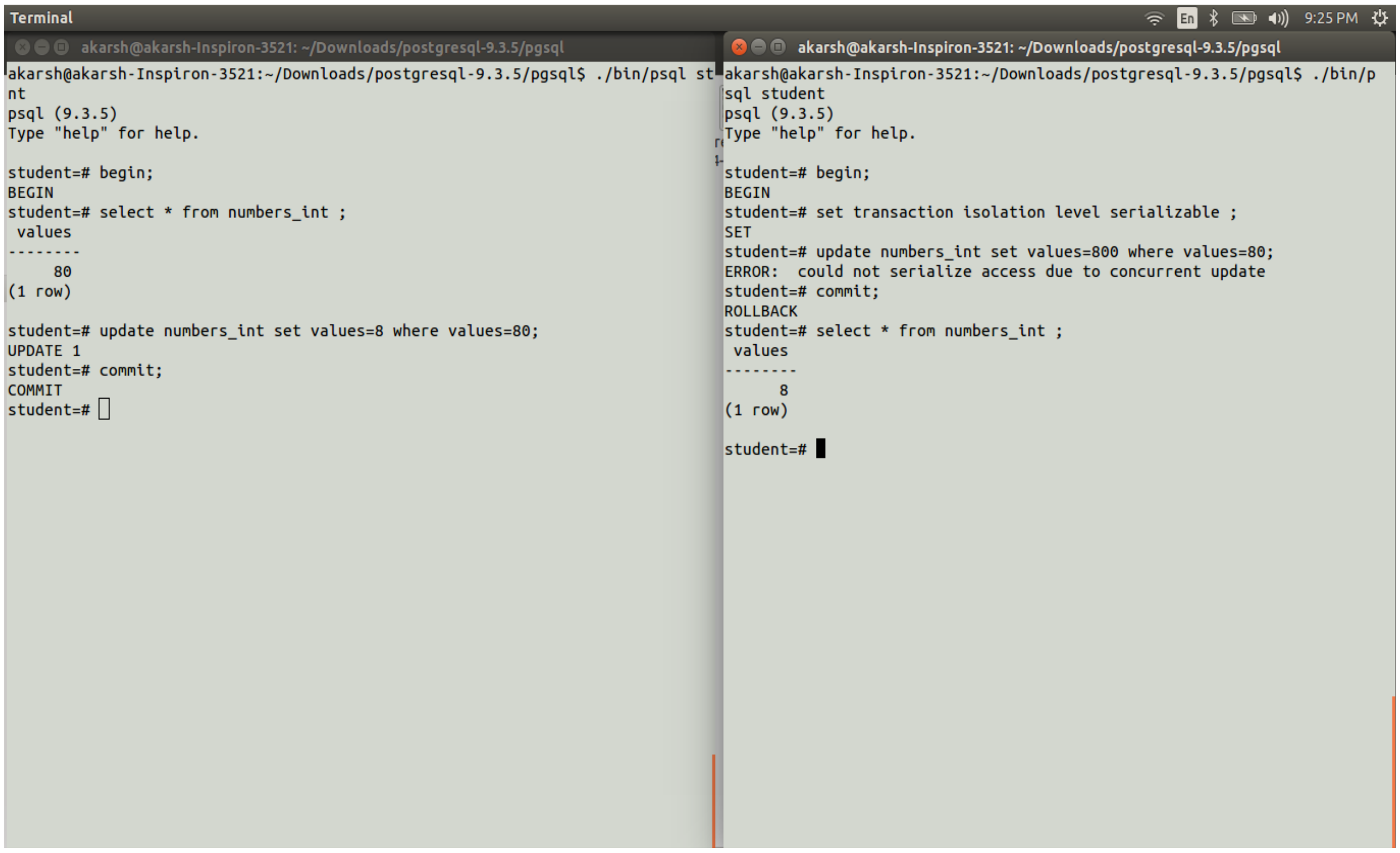

Terminal
En
9:12 PM
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql

```
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/psql stude
nt
psql (9.3.5)
Type "help" for help.

student=# begin;
BEGIN
student=# select txid_current();
 txid_current
--------------
          907
(1 row)

student=# update numbers_int set values=60 where values=50;
UPDATE 1
student=# commit;
COMMIT
student=# 
```

```
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/p
sql student
psql (9.3.5)
Type "help" for help.

student=# begin;
BEGIN
student=# select txid_current();
 txid_current
--------------
          908
(1 row)

student=# set transaction isolation level read committed ;
SET
student=# update numbers_int set values=70 where values=50;
UPDATE 0
student=# select * from numbers_int ;
 values
--------
     60
(1 row)

student=# 
```

->Serialization isolation level :

Just like in above mode, In A we update the value from 80 to 8 but we do not commit it yet. In B we set the transaction isolation level to serializable and then update the value of 80 to 800. This throws an error indicating serialization failure and ROLLBACK occurs on commit. We see that the updated value is the same done by A.


Terminal
9:25 PM
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql

```
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/psql st
nt
psql (9.3.5)
Type "help" for help.

student=# begin;
BEGIN
student=# select * from numbers_int ;
 values
--------
     80
(1 row)

student=# update numbers_int set values=8 where values=80;
UPDATE 1
student=# commit;
COMMIT
student=#
```

```
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/p
sql student
psql (9.3.5)
Type "help" for help.

student=# begin;
BEGIN
student=# set transaction isolation level serializable ;
SET
student=# update numbers_int set values=800 where values=80;
ERROR:  could not serialize access due to concurrent update
student=# commit;
ROLLBACK
student=# select * from numbers_int ;
 values
--------
      8
(1 row)

student=#
```

This is how MVCC works in Postgres. However there are other algorithms available which deals with Concurrency control less efficiently than MVCC. They can be implemented or rather, modified/optimized and then compared with the currently existing MVCC. This will be taken up in future implementations.