Experiment No.: 07

Title: To implement the ACID properties for transaction and concurrency.

.

Batch:A2　　　　　　　　**Roll No.:16010421063**　　　　　　**Experiment No.: 07**

Aim: To implement the ACID properties for transaction and concurrency.

Resources needed: PostgreSQL 9.3

Theory:

Transactions are a fundamental concept of all database systems. The essential point of a transaction is that it bundles multiple steps into a single, all-or-nothing operation. The intermediate states between the steps are not visible to other concurrent transactions, and if some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all.

For example, consider a bank database that contains balances for various customer accounts, as well as total deposit balances for branches. Suppose that we want to record a payment of $100.00 from Alice's account to Bob's account. Simplifying outrageously, the SQL commands for this might look like:

UPDATE accounts SET balance = balance – 100.00 WHERE name = 'Alice';

UPDATE branches SET balance = balance – 100.00 WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');

UPDATE accounts SET balance = balance + 100.00 WHERE name = 'Bob';

UPDATE branches SET balance = balance + 100.00 WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');

The details of these commands are not important here; the important point is that there are several separate updates involved to accomplish this rather simple operation. Our bank's officers will want to be assured that either all these updates happen or none of them happen. It would certainly not do for a system failure to result in Bob receiving $100.00 that was not debited from Alice. Nor would Alice long remain a happy customer if she was debited without Bob being credited. We need a guarantee that if something goes wrong partway through the operation, none of the steps executed so far will take effect. Grouping the updates into a transaction gives us this guarantee. A transaction is said to be atomic: from the point of view of other transactions, it either happens completely or not at all. We also want a guarantee that once a transaction is completed and acknowledged by the database system, it has indeed been permanently recorded and won't be lost even if a crash ensues shortly thereafter. For example, if we are recording a cash withdrawal by Bob, we do not want any chance that the debit to his account will disappear in a crash just after he walks out the bank door. A transactional database guarantees that all the updates made by a transaction are logged in permanent storage (i.e., on disk) before the transaction is reported complete.

Another important property of transactional databases is closely related to the notion of atomic up-dates: when multiple transactions are running concurrently, each one should not be able to see the incomplete changes made by others. For example, if one transaction is busy totaling all the branch balances, it would not do for it to include the debit from Alice's branch but not the credit to Bob's branch, nor vice versa. So transactions must be all-or-nothing not only in terms of their permanent effect on the database, but also in terms of their visibility as they happen. The updates made so far by an open transaction are invisible to other transactions until the transaction completes, whereupon all the updates become visible simultaneously.

Procedure:
Transactions:

In PostgreSQL, a transaction is set up by surrounding the SQL commands of the transaction with BEGIN and COMMIT commands. So our banking transaction would actually look like:

```
BEGIN;
UPDATE accounts SET balance = balance – 100.00 WHERE name = 'Alice';
COMMIT;
```

If, partway through the transaction, we decide we do not want to commit (perhaps we just noticed thatAlice's balance went negative), we can issue the command ROLLBACK instead of COMMIT, and all our updates so far will be canceled.

PostgreSQL actually treats every SQL statement as being executed within a transaction. If you do not issue a BEGIN command, then each individual statement has an implicit BEGIN and (if successful) COMMIT wrapped around it. A group of statements surrounded by BEGIN and COMMIT is sometimes called a transaction block.

Note: Some client libraries issue BEGIN and COMMIT commands automatically, so that you might get the effect of transaction blocks without asking.

In this exercise, you will see how to rollback or commit transactions. By default PostgreSQL commits each SQL statement as soon as it is submitted. To prevent the transaction from committing immediately, you have to issue a command **begin;** to tell PostgreSQL to not commit immediately. You can issue any number of SQL statements after this, and then either **commit;** to commit the transaction, or **rollback;** to rollback the transaction. To see the effect, execute the following commands *one at a time for your table and attribute:*

```
begin ;
select * from student where name = 'Tanaka';
delete from student where name = 'Tanaka';
select * from student where name = 'Tanaka';
rollback;
select * from student where name = 'Tanaka';
```

In your results, explain what you observed and why it happened.

Concurrency:

PostgreSQL implements concurrency control using **read committed** isolation level as the default, but also supports a concurrent control mechanism called **serializable snapshot isolation** which can be turned on by executing the command.

Type below command for current isolation level**:**

show transaction isolation level;

Note: please do not set transaction isolation level as serializable right now, you will do it in later exercises.

In the **read committed** isolation level, each statement sees the effects of all preceding transactions that have committed, but does not see the effect of concurrently running transactions(i.e. updates that have not been committed yet). This low level of consistency can cause problems with transactions, and it is safer to use the **serializable** level if concurrent updates occur with multiple statement transactions.

In **snapshot isolation**, where a transaction gets a conceptual snapshot of data at the time it started, and all values it reads are as per this snapshot. In snapshot isolation, if two transactions concurrently update the same data item, one of them will be rolled back. Snapshot isolation does NOT guarantee serializability of transactions.

For example, it is possible that transaction T1 reads A and performs an update B =A, while transaction T2 reads B and performs an update A=B. In this case, there is no conflict on the update, since different tuples are updated by the two transactions, but the execution may not be serializable: in any serial schedule, A and B will become the same value, but with snapshot isolation, they may exchange values.

Oracle uses snapshot isolation for concurrency control when asked to set the isolation level to serializable, even though it does not really guarantee serializability. Microsoft SQL Server supports snapshot isolation, but uses two-phase locking for the serializable isolation level. PostgreSQL versions prior to 9.1 used snapshot isolation when the isolation level was set to serializable.

However, since version 9.1, PostgreSQL uses an improved version of snapshot isolation, called serializable snapshot isolation, when asked to set the isolation level to serializable. This mechanism in fact offers true serializability, unlike plain snapshot isolation.

Case 1: In this exercise you will run transactions concurrently from two different pgAdmin3 windows, to see how updates by one transaction affect another.
1. Open two pgAdmin3 connections to the same database. Execute your own command in sequence in the first window as given the following commands:

   begin;

   update student set tot_cred = 55 where name = 'Tanaka';
2. Now in the second window execute

   begin;

   select * from student where name = 'Tanaka';

In result write the value of tot_cred (your attribute). Can you figure out why you got the result that you saw? What does this tell you about concurrency control in PostgreSQL?

    3. Now in the second window execute

        commit;

        select * from student where name = 'Tanaka';

In result write the value of tot_cred (your attribute). Can you figure out why you got the result that you saw? The second transaction committed successfully before the first, it got the old value for tot_cred. What does this tell you about concurrency control in PostgreSQL?

    4. Now in the first window execute

        commit;

Observe the the value of tot_cred (your attribute) in the second window with select * from student where name = 'Tanaka'; .

Case 2: Now, let us try to update the same tuple concurrently from two windows. In one window execute
1. begin;
2. update student set tot_cred = 44 where name = 'Tanaka'
3. Then in the second window, execute one after another:
4. begin;
5. update student set tot_cred = 44 where name = 'Tanaka'

See what happens at this point. The query appears to be hanging: PostgreSQL is waiting for the other query that updates student to complete.
6. Now in the first window, execute
    commit;

Write in result what happens in the second window.
7. Then execute commit;

Observe in the second window and write in result.
8. Now in the second window execute
      select * from student where name = 'Tanaka';

This is the value after the above update was committed. Explain why you got the above value.

Case 3: Next, in both windows execute the command after the begin command set transaction isolation level serializable.

SERIALIZABLE
All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction. If a pattern of reads and writes among concurrent serializable transactions would create a situation which could not have occurred for any serial (one-at-a-time) execution of those transactions, one of them will be rolled back with a serialization_failure error.

    Open two connections (two new query windows) and type the following:
1. Create table instructor with attributes id and salary.
2. Insert values required values:
    insert into instructor values(3, 1000);
    insert into instructor values(4, 2000);
3. Run select * from instructor in both the window and note the result.

To begin a transaction and set isolation level to serializable:

4. Run the query in window 1:
   Begin transaction isolation level serializable;
   update instructor set salary=(select salary from instructor where id=4) where id= 3;
5. Run the query in window 2:
   Begin transaction isolation level serializable;
   update instructor set salary=(select salary from instructor where id=3) where id= 4;
6. commit window 1
7. commit window 2 (you will get error. Note down the error)
8. What happened above? Check the state of the system by running the query select * from instructor in both the windows.

Is this equivalent to any serializable schedule? In your results, explain what you observed and why it happened.

Results: (Query printout with output)

16010421063_Arya on postgres@PostgreSQL 9.6

```
1   Begin;
2   SELECT * FROM employee where password='test123';
3
4   DELETE from employee where name='arya1';
5
6   SELECT * FROM employee where password='test123';
7   rollback;
8   SELECT * FROM employee where password='test123';
```

Data Output | Explain | Messages | History

eloginid text	name text	password text	
112	arya	test123	
113	arya1	test123	
115	arya2	test123	
117	arya8	test123	
119	arya9	test123	

16010421063_Arya on postgres@PostgreSQL 9.6

```
1   Begin;
2   SELECT * FROM employee where password='test123';
3
4   DELETE from employee where name='arya1';
5
6   SELECT * FROM employee where password='test123';
7   rollback;
8   SELECT * FROM employee where password='test123';
```

Data Output | Explain | Messages | History

eloginid text	name text	password text	
112	arya	test123	
115	arya2	test123	
117	arya8	test123	
119	arya9	test123	

16010421063_Arya on postgres@PostgreSQL 9.6

```
5
6   SELECT * FROM employee where password='test123';
7   rollback;
8   SELECT * FROM employee where password='test123';
9
10
11
12
```

Data Output | Explain | Messages | History

eloginid text	name text	password text	
112	arya	test123	
113	arya1	test123	
115	arya2	test123	
117	arya8	test123	
119	arya9	test123	

We can see that when we run the rollback query the database goes back to the state consistent state before we ran the begin query. Thus transaction is all or nothing .

Concurrency

16010421063_Arya on postgres@PostgreSQL 9.6

```
1   show transaction isolation level;
2
```

Data Output   Explain   Messages   History

	transaction_isolation text	
	read committed	

16010421063_Arya on postgres@PostgreSQL 9.6

```
1   begin;
2   update employee set name='arya_changed' where name='aryal';
3   commit;
4   SELECT * FROM employee where password='test123';
```

Data Output   Explain   **Messages**   History

UPDATE 1

Query returned successfully in 421 msec.

16010421063_Arya on postgres@PostgreSQL 9.6

```
1  begin;
2  SELECT * FROM employee where password='test123'
3
```

Data Output | Explain | Messages | History

eloginid text	name text	password text	
112	arya	test123	
115	arya2	test123	
117	arya8	test123	
119	arya9	test123	
113	arya1	test123	

16010421063_Arya on postgres@PostgreSQL 9.6

```
1  begin;
2  update employee set name='arya_changed' where name='arya1';
3  commit;
4  SELECT * FROM employee where password='test123';
```

Data Output | Explain | Messages | History

eloginid text	name text	password text	
112	arya	test123	
115	arya2	test123	
117	arya8	test123	
119	arya9	test123	
113	arya_changed	test123	

16010421063_Arya on postgres@PostgreSQL 9.6

```
1  begin;
2  SELECT * FROM employee where password='test123';
3  commit;
4
```

Data Output | Explain | **Messages** | History

```
WARNING:  there is already a transaction in progress
COMMIT

Query returned successfully in 332 msec.
```

We see no change before we commit as there is isolation at both the windows and the change is not reflected as we dont change in the database until we actually commit.

Case2

```
16010421063_Arya on postgres@PostgreSQL 9.6

1   begin;
2   update employee set name='arya_1' where name='arya_changed';
3
```

Data Output   Explain   **Messages**   History

```
UPDATE 1

Query returned successfully in 318 msec.
```

```
16010421063_Arya on postgres@PostgreSQL 9.6

1   begin;
2   update employee set name='arya_1' where name='arya_changed';
3
4
```

Data Output   Explain   **Messages**   History

Waiting for the query execution to complete...

```
begin;
update employee set name='arya_1' where name='arya_changed';
```

UPDATE 0

Query returned successfully in 53 secs.

```
begin;
update employee set name='arya_1' where name='arya_changed';
commit;
```

COMMIT

Query returned successfully in 422 msec.

16010421063_Arya on postgres@PostgreSQL 9.6

```
select * from employee where password='test123'
```

eloginid text	name text	password text
112	arya	test123
115	arya2	test123
117	arya8	test123
119	arya9	test123
113	arya_1	test123

hence we see that at one point only one transaction can be committed on the same database. The transaction will happen one after the other.

Serialisable

16010421063_Arya on postgres@PostgreSQL 9.6

```
1   CREATE TABLE instructor(
2       id int PRIMARY KEY,
3       salary int
4   )
```

Data Output Explain **Messages** History

CREATE TABLE

Query returned successfully in 559 msec.

16010421063_Arya on postgres@PostgreSQL 9.6

```
1   insert into instructor values( 3, 1000);
2   insert into instructor values( 4, 2000);
```

Data Output Explain **Messages** History

INSERT 0 1

Query returned successfully in 302 msec.

16010421063_Arya on postgres@PostgreSQL

```
1   SELECT * from instructor
```

Data Output Explain Messages History

id integer	salary integer	
3	1000	
4	2000	

16010421063_Arya on postgres@PostgreSQL 9.6

```
1   Begin transaction isolation level serializable;
2
3   update instructor set salary=(select salary from instructor where id=4) where id= 3
4
```

Data Output   Explain   **Messages**   History

UPDATE 1

Query returned successfully in 309 msec.

16010421063_Arya on postgres@PostgreSQL 9.6

```
1   Begin transaction isolation level serializable;
2
3   update instructor set salary=(select salary from instructor where id=3) where id= 4;
4
```

Data Output   Explain   **Messages**   History

UPDATE 1

Query returned successfully in 319 msec.

```
16010421063_Arya on postgres@PostgreSQL 9.6

1   Begin transaction isolation level serializable;
2
3   update instructor set salary=(select salary from instructor where id=4) where id= 3;
4   commit;
5
```

Data Output   Explain   **Messages**   History

```
COMMIT

Query returned successfully in 460 msec.
```

```
16010421063_Arya on postgres@PostgreSQL 9.6

1   Begin transaction isolation level serializable;
2
3   update instructor set salary=(select salary from instructor where id=3) where id= 4;
4
5   commit;
6
```

Data Output   Explain   **Messages**   History

```
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.
********** Error **********

ERROR: could not serialize access due to read/write dependencies among transactions
SQL state: 40001
Detail: Reason code: Canceled on identification as a pivot, during commit attempt.
Hint: The transaction might succeed if retried.
```

Whenever you request SERIALIZABLE isolation the DB will attempt to make concurrent sets of queries appear to have executed serially in terms of the results they produce. This is not always possible, e.g. when two transactions have mutual depenencies. In this case, PostgreSQL will abort one of the transactions with a serialization failure error, telling you that you should retry it.

Code that uses SERIALIZABLE must always be prepared to re-try transactions. It must check the SQLSTATE and, for serialization failures, repeat the transaction.

Outcomes:

Apply the concept of transaction, concurrency control and recovery techniques.

Conclusion:

The concepts of transaction,concurrency control and recovery techniques was understood and implemented practically on our database. This deepened my understanding of the concept and taught me how to apply them in real life applications.

(Autonomous College Affiliated to University of Mumbai)

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of faculty in-charge with date

_____

References:
Books:

1 Elmasri and Navathe, "Fundamentals of Database Systems", 6th Edition, Pearson Education
2 Korth, Slberchatz,Sudarshan, :"Database System Concepts", 6th Edition, McGraw – Hill.