SE Comp A	Roll number : 9913
Experiment no. : 10	Date of Implementation :

Aim: Simple Transaction implementation

Tool Used : PostgreSQL/Mysql

Related Course outcome : At the end of the course, Students will be able to Use and Apply the concept of transaction, concurrency and recovery

Rubrics for assessment of Experiment:

Indicator	Poor	Average	Good
Timeliness ● **Maintains assignment deadline (3)**	**Assignment not done (0)**	**One or More than One week late (1-2)**	**Maintains deadline (3)**
Implementation of concepts (3)	**N/A**	**< 80% complete (1-2)**	**100% complete (3)**
Originality ● **Extent of plagiarism(2)**	**Copied it from someone else(0)**	**At least few parts of it have been done without copying(1)**	**Experiment has been solved completely without copying (2)**
Knowledge ● **In depth knowledge of the assignment(2)**	**Unable to answer 2 questions(0)**	**Unable to answer 1 question (1)**	**Able to answer 2 questions (2)**

Assessment Marks :

Timeliness	
Completeness and neatness	
Originality	
Knowledge	
Total	

Total : (Out of 10)

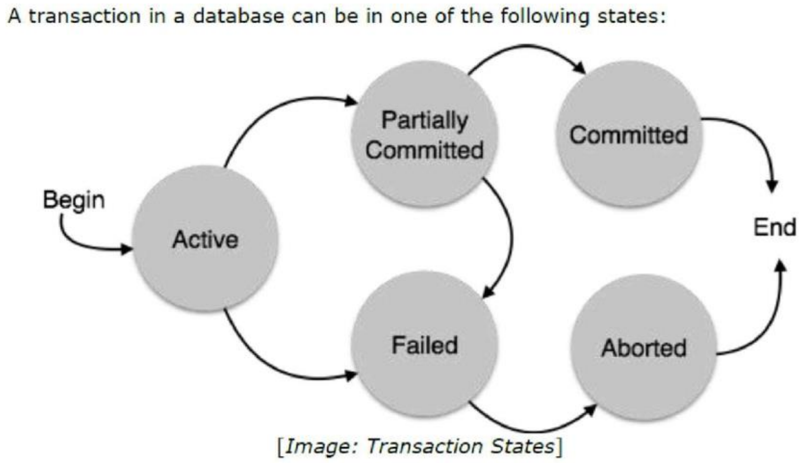
Teacher's Sign :

EXPERIMENT 10	***Transaction concept***
Aim	To implement Simple Transaction concept
Tools	PostgreSQL/Mysql

Theory	A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further. *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. **Properties of Transactions** Transactions have the following four standard properties, usually referred to by the acronym ACID − ● **Atomicity** − Ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure and previous operations are rolled back to their former state. ● **Consistency** − Ensures that the database properly changes states upon a successfully committed transaction. ● **Isolation** − Enables transactions to operate independently of and transparent to each other. ● **Durability** − Ensures that the result or effect of a committed transaction persists in case of a system failure. 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'; -- etc etc COMMIT; End;

<table>
<tr>
<td>Theory</td>
<td>

State Diagram :



A transaction in a database can be in one of the following states:

[*Image: Transaction States*]

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.

```
BEGIN;
--sql
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance - 100.00
   WHERE name = 'Alice';
UPDATE accounts SET balance = balance + 100.00
   WHERE name = 'Bob';
ROLLBACK TO my_savepoint; or commit;
--UPDATE accounts SET balance = balance + 100.00
   WHERE name = 'Wally';
COMMIT;
```

</td>
</tr>
</table>

Theory

DOCXTransaction Control

The following commands are used to control transactions −

- **BEGIN TRANSACTION** − To start a transaction.
- **COMMIT** − To save the changes, alternatively you can use **END TRANSACTION** command.
- **ROLLBACK** − To rollback the changes.

Transactional control commands are only used with the DML commands INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

The BEGIN TRANSACTION Command

Transactions can be started using BEGIN TRANSACTION or simply BEGIN command. Such transactions usually persist until the next COMMIT or ROLLBACK command is encountered. But a transaction will also ROLLBACK if the database is closed or if an error occurs.

The following is the simple syntax to start a transaction −

```
BEGIN;
```

or

```
BEGIN TRANSACTION;
```

The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for COMMIT command is as follows −

```
COMMIT;
```

or

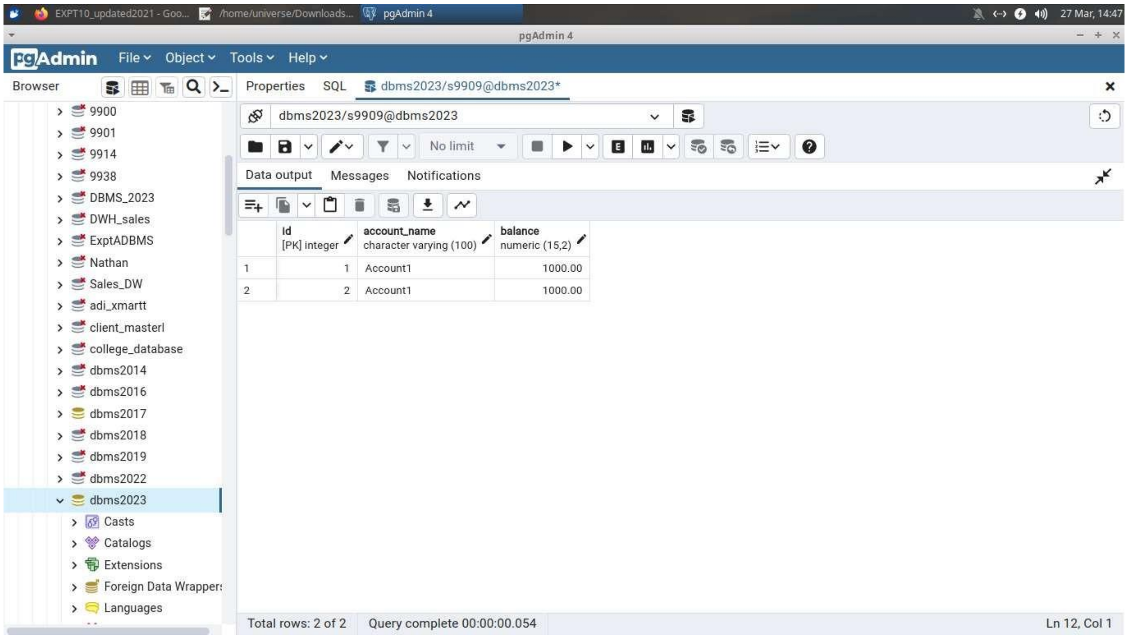
```
END TRANSACTION;
```

Theory	**The ROLLBACK Command** The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued. The syntax for ROLLBACK command is as follows – ROLLBACK; PostgreSQL Transaction (postgresqltutorial.com) PostgreSQL: Documentation: 15: 13.2. Transaction Isolation https://www.postgresql.org/docs/current/transaction-iso.html
Task	1. Create any sample table for transaction testing 2. Observe the effect of auto commit on /off on insert stmt(SET AUTOCOMMIT { = | TO } { ON | OFF }) 3. How to find out the current transaction mode(POSTGRESQL-MENU-QUERY-AUTOCOMMMIT) 4. Start transaction, insert records in table and use roll back 5. Start new transaction and commit the current transaction 6. Use save point and use rollback to and commit 7. What happens to the current transaction if a start transaction is executed? 8. What happens to the current transaction if the session is ended? 9. How to view and change the current transaction isolation level? 10. Demonstrate datalocks in transaction and how long a transaction will wait for a data lock? (OPTIONAL) 11. Wa java/php program using jdbc/odbc to implement transaction java-mysql or php-mysql (jdbc/odbc)[https://www.php.net/manual/en/mysqli.begin-transaction.php] (OPTIONAL) 12. Transfer 5k from A's account to B's account using transaction concept 13. demonstrate different locks(MySQL :: MySQL 8.0 Reference Manual :: 15.7.2.4 Locking Reads) (OPTIONAL)

Links	**https://www.postgresql.org/docs/9.1/sql-start-transaction.html** **https://www.postgresql.org/docs/8.3/tutorial-transactions.html** **https://pgdash.io/blog/postgres-transactions.html** **https://tapoueh.org/blog/2018/07/postgresql-concurrency-isolation-and-locking/** **http://dba.fyicenter.com/faq/mysql/mysql_sql_transaction_management.html** **locks** MySQL :: MySQL 8.0 Reference Manual :: 15.7.2.4 Locking Reads Processlist in postgresql - postgres > select `* from pg_stat_activity;` Terminate (kill) specific session in PostgreSQL database - PostgreSQL Data Dictionary Queries (dataedo.com) PostgreSQL: Documentation: 14: SET TRANSACTION
Post Lab Questions:	**1. How Does MySQL/postgresql Handle Read Consistency?** **2. What Are Transaction Isolation Levels IN MYSQL/ Postgresql** **3. What Are Impacts on Applications from Locks, Timeouts, and DeadLocks?**

1.
```
DROP TABLE IF EXISTS accounts;
BEGIN;
CREATE TABLE accounts (
        id SERIAL PRIMARY KEY,
        account_name VARCHAR(100),
        balance DECIMAL(15, 2)
);
COMMIT;
ROLLBACK;
```

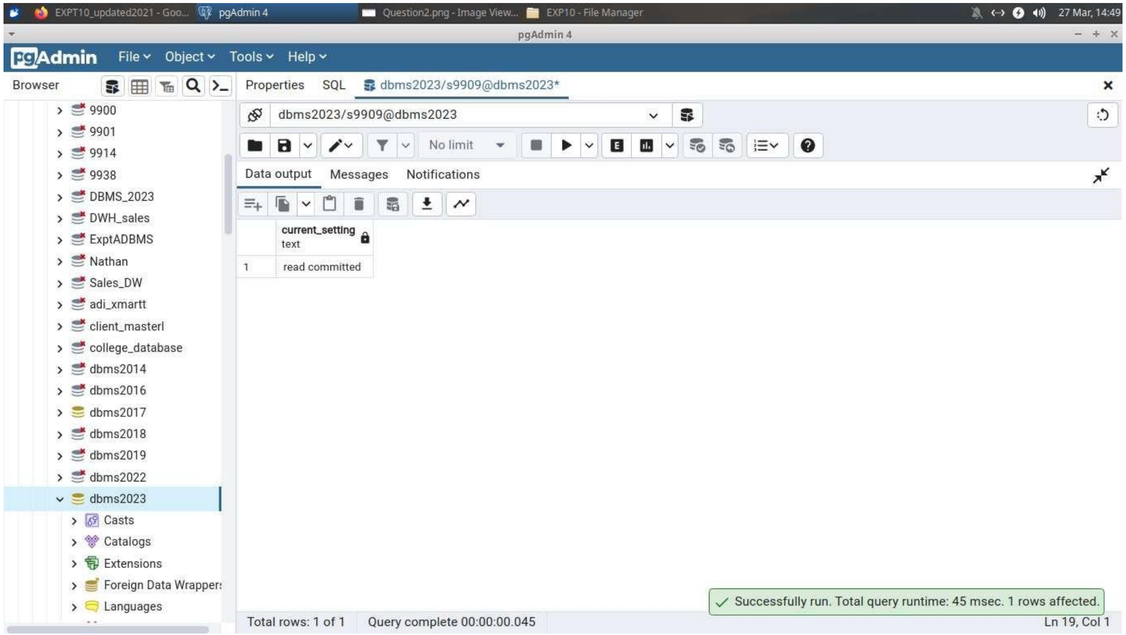
2.
```
BEGIN;
INSERT INTO accounts (account_name, balance) VALUES ('Account1', 1000);
COMMIT;
SELECT * FROM accounts;
```



	id [PK] integer	account_name character varying (100)	balance numeric (15,2)
1	1	Account1	1000.00
2	2	Account1	1000.00

Total rows: 2 of 2 Query complete 00:00:00.054 Ln 12, Col 1

3. SELECT current_setting('transaction_isolation');



	current_setting text
1	read committed

✓ Successfully run. Total query runtime: 45 msec. 1 rows affected.

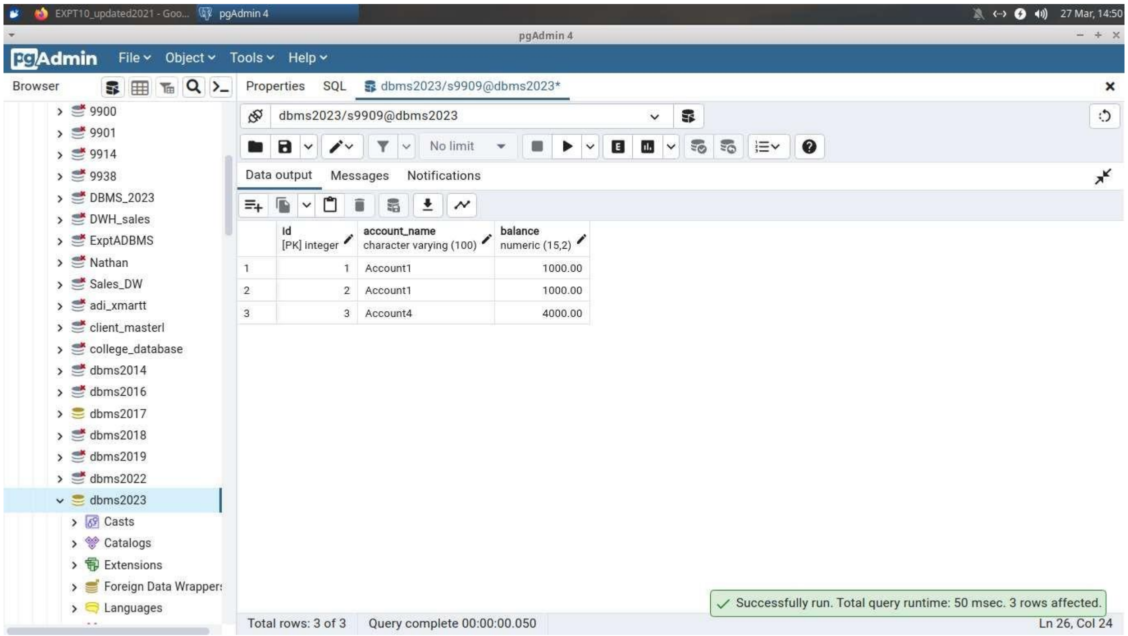
Total rows: 1 of 1 Query complete 00:00:00.045 Ln 19, Col 1

4. BEGIN;

   INSERT INTO accounts (account_name, balance) VALUES ('Account4', 4000);

5. COMMIT;

   SELECT * FROM accounts;



	id [PK] integer	account_name character varying (100)	balance numeric (15,2)
1	1	Account1	1000.00
2	2	Account1	1000.00
3	3	Account4	4000.00

```
✓ Successfully run. Total query runtime: 50 msec. 3 rows affected.
Total rows: 3 of 3    Query complete 00:00:00.050    Ln 26, Col 24
```

6. BEGIN;

   SAVEPOINT savepoint1;

   INSERT INTO accounts (account_name, balance) VALUES ('Account5', 5000);

   INSERT INTO accounts (account_name, balance) VALUES ('Account6', 6000);

   ROLLBACK TO SAVEPOINT savepoint1;

   SELECT * FROM accounts;

   COMMIT;

   SELECT * FROM accounts;

7. If a "start transaction" command is executed during a current transaction, it typically initiates a new transaction, effectively nesting the new transaction within the current one. This means that any changes made after the "start transaction" command will be part of the new transaction, and they can be committed or rolled back independently of the outer transaction.

8. If the session is ended during a current transaction without committing or rolling back the transaction explicitly, the transaction will typically be automatically rolled back by the database management system. This ensures data integrity and prevents any unintended changes from being permanently committed to the database.

9.

```
SHOW TRANSACTION ISOLATION LEVEL;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN;
UPDATE accounts SET balance = balance - 5000 WHERE account_name = 'A';
UPDATE accounts SET balance = balance + 5000 WHERE account_name = 'B';
COMMIT;
```

POSTLAB :

Q1 How Does MySQL/postgresql Handle Read Consistency?

MySQL and PostgreSQL both use Multi-Version Concurrency Control (MVCC) to ensure read consistency. In MySQL, different transaction isolation levels, like REPEATABLE READ, control consistency, while PostgreSQL defaults to READ COMMITTED isolation. Both systems provide mechanisms to ensure that transactions see a consistent view of the database despite concurrent modifications.

Q2 What Are Transaction Isolation Levels IN MYSQL/ Postgresql?

- MySQL:
  - READ UNCOMMITTED: Allows dirty reads.
  - READ COMMITTED: Prevents dirty reads, allows non-repeatable reads and phantom reads.
  - REPEATABLE READ: Prevents non-repeatable reads, allows phantom reads.
  - SERIALIZABLE: Prevents all anomalies: dirty reads, non-repeatable reads, and phantom reads.

- PostgreSQL:
  - READ UNCOMMITTED: Not directly supported.
  - READ COMMITTED: Prevents dirty reads.
  - REPEATABLE READ: Prevents non-repeatable reads.
  - SERIALIZABLE: Prevents all anomalies.

Q3 What Are Impacts on Applications from Locks, Timeouts, and DeadLocks?

- Locks: Can cause contention and blocking, leading to performance issues.
- Timeouts: Can result in incomplete transactions or failed operations.
- Deadlocks: Cause transactions to wait indefinitely for each other, potentially leading to application hangs or crashes.