

**Batch:- B-2      Roll No:- 16010122151**

**Experiment / assignment / tutorial No. 8**  
**Grade: AA / AB / BB / BC / CC / CD / DD**  
**Signature of the Staff In-charge with date**

**Title: Implementing TCL/DCL**

**Objective:** To be able to Implement TCL and DCL.

**Expected Outcome of Experiment:**

CO 2: Convert entity-relationship diagrams into relational tables, populate a relational database and formulate SQL queries on the data Use SQL for creation and query the database.

CO 4: Demonstrate the concept of transaction, concurrency control and recovery techniques.

**Books/ Journals/ Websites referred:**

1. Dr. P.S. Deshpande, SQL and PL/SQL for Oracle 10g.Black book, Dreamtech Press
2. www.db-book.com
3. Korth, Silberchatz, Sudarshan : “Database Systems Concept”, 5<sup>th</sup> Edition , McGraw Hill
4. Elmasri and Navathe,”Fundamentals of database Systems”, 4<sup>th</sup> Edition,PEARSON Education.

**Resources used:** PostgreSQL

**Theory**

DCL stands for Data Control Language.

DCL is used to control user access in a database.

This command is related to the security issues.

Using DCL command, it allows or restricts the user from accessing data in database schema.

DCL commands are as follows,

GRANT

REVOKE

It is used to grant or revoke access permissions from any database user.

**GRANT command** gives user's access privileges to the database.

This command allows specified users to perform specific tasks.

**Syntax:**

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE |
REFERENCES | TRIGGER }
[, ...] | ALL [ PRIVILEGES ] }
ON { [ TABLE ] table_name [, ...]
| ALL TABLES IN SCHEMA schema_name [, ...] }
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT
OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name
[, ...] )
[, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
ON [ TABLE ] table_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT
OPTION ]
```

**Example**

```
GRANT INSERT ON films TO PUBLIC;
GRANT ALL PRIVILEGES ON kinds TO ram;
GRANT admins TO krishna;
```

**REVOKE command** is used to cancel previously granted or denied permissions.

This command withdraw access privileges given with the GRANT command.

It takes back permissions from user.

**Syntax:**

```
REVOKE [ GRANT OPTION FOR ]
{ { SELECT | INSERT | UPDATE | DELETE | TRUNCATE |
REFERENCES | TRIGGER }
[, ...] | ALL [ PRIVILEGES ] }
ON { [ TABLE ] table_name [, ...]
| ALL TABLES IN SCHEMA schema_name [, ...] }
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [,
...])
[, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
ON [ TABLE ] table_name [, ...]
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
        | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

### Example

```
REVOKE INSERT ON films FROM PUBLIC;
REVOKE ALL PRIVILEGES ON kinds FROM Madhav;
REVOKE admins FROM Keshav;
```

TCL stands for **Transaction Control Language**.

This command is used to manage the changes made by DML statements.

TCL allows the statements to be grouped together into logical transactions.

**TCL commands are as follows:**

1. COMMIT
2. SAVEPOINT
3. ROLLBACK
4. SET TRANSACTION

**COMMIT command** saves all the work done. It ends the current transaction and makes permanent changes during the transaction

#### Syntax:

```
commit;
```

**SAVEPOINT command** is used for saving all the current point in the processing of a transaction. It marks and saves the current point in the processing of a transaction. It is used to temporarily save a transaction, so that you can rollback to that point whenever necessary.

#### Syntax

```
SAVEPOINT savepoint_name
```

**ROLLBACK command** restores database to original since the last COMMIT. It is used to restores the database to last committed state.

#### Syntax:

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ]  
savepoint_name
```

#### Example

```
BEGIN;  
    INSERT INTO table1 VALUES (1);  
    SAVEPOINT my_savepoint;  
    INSERT INTO table1 VALUES (2);  
    ROLLBACK TO SAVEPOINT my_savepoint;  
    INSERT INTO table1 VALUES (3);  
COMMIT;
```

The above transaction will insert the values 1 and 3, but not 2.

**SET TRANSACTION** is used for placing a name on a transaction. You can specify a transaction to be read only or read write. This command is used to initiate a database transaction.

#### Syntax:

SET TRANSACTION [Read Write | Read Only];

The SET TRANSACTION command sets the characteristics of the current transaction. It has no effect on any subsequent transactions. SET SESSION CHARACTERISTICS sets the default transaction characteristics for subsequent transactions of a session. These defaults can be overridden by SET TRANSACTION for an individual transaction.

The available transaction characteristics are the transaction isolation level, the transaction access mode (read/write or read-only), and the deferrable mode. In addition, a snapshot can be selected, though only for the current transaction, not as a session default.

The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently:

#### **READ COMMITTED**

A statement can only see rows committed before it began. This is the default.

#### **REPEATABLE READ**

All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

#### **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.

### Examples

With the default read committed isolation level.

```
process A: BEGIN; -- the default is READ COMMITTED

process A: SELECT sum(value) FROM purchases;
--- process A sees that the sum is 1600

process B: INSERT INTO purchases (value) VALUES (400)
--- process B inserts a new row into the table while
--- process A's transaction is in progress

process A: SELECT sum(value) FROM purchases;
--- process A sees that the sum is 2000

process A: COMMIT;
```

If we want to avoid the changing sum value in process A during the lifespan of the transaction, we can use the repeatable read transaction mode.

```
process A: BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
process A: SELECT sum(value) FROM purchases;
--- process A sees that the sum is 1600

process B: INSERT INTO purchases (value) VALUES (400)
--- process B inserts a new row into the table while
--- process A's transaction is in progress

process A: SELECT sum(value) FROM purchases;
--- process A still sees that the sum is 1600

process A: COMMIT;
```

The transaction in process A will freeze its snapshot of the data and offer consistent values during the life of the transaction.

Repeatable reads are not more expensive than the default read commit transaction. There is no need to worry about performance penalties. However, applications must be prepared to retry transactions due to serialization failures.

Let's observe an issue that can occur while using the repeatable read isolation level — the `could not serialize access due to concurrent update` error.

```
process A: BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
process B: BEGIN;
process B: UPDATE purchases SET value = 500 WHERE id = 1;
process A: UPDATE purchases SET value = 600 WHERE id = 1;
```

```
-- process A wants to update the value while process B is changing it
-- process A is blocked until process B commits

process B: COMMIT;
process A: ERROR: could not serialize access due to concurrent update
-- process A immediately errors out when process B commits
```

If process B would roll back, then its changes are negated and repeatable read can proceed without issues. However, if process B commits the changes then the repeatable read transaction will be rolled back with the error message because it can not modify or lock the rows changed by other processes after the repeatable read transaction has began.

demonstrate the differences between the two isolation modes.

```
process A: BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
process A: SELECT sum(value) FROM purchases;
process A: INSERT INTO purchases (value) VALUES (100);
process B: BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
process B: SELECT sum(value) FROM purchases;
process B: INSERT INTO purchases (id, value);
process B: COMMIT;
process A: COMMIT;
```

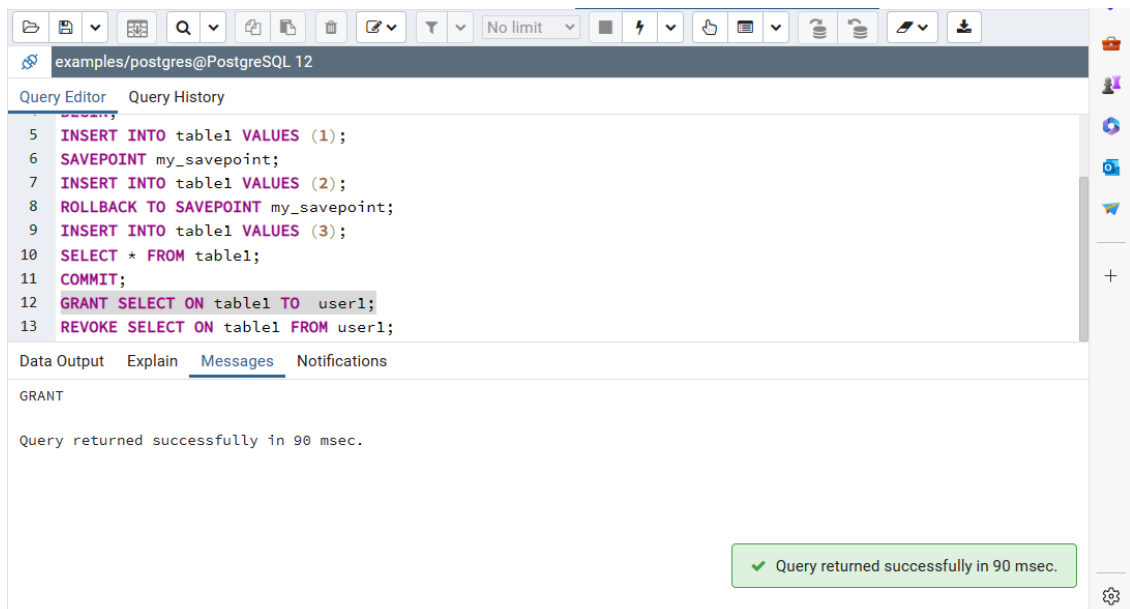
With Repeatable Reads everything works, but if we run the same thing with a Serializable isolation mode, process A will error out.

```
process A: BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
process A: SELECT sum(value) FROM purchases;
process A: INSERT INTO purchases (value) VALUES (100);
process B: BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
process B: SELECT sum(value) FROM purchases;
process B: INSERT INTO purchases (id, value);
process B: COMMIT;
process A: 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.
```

Both transactions have modified what the other transaction would have read in the select statements. If both would allow to commit this would violate the Serializable behaviour, because if they were run one at a time, one of the transactions would have seen the new record inserted by the other transaction.

## Implementation Screenshots (Problem Statement, Query and Screenshots of Results):

Demonstrate DCL and TCL language commands on your database.



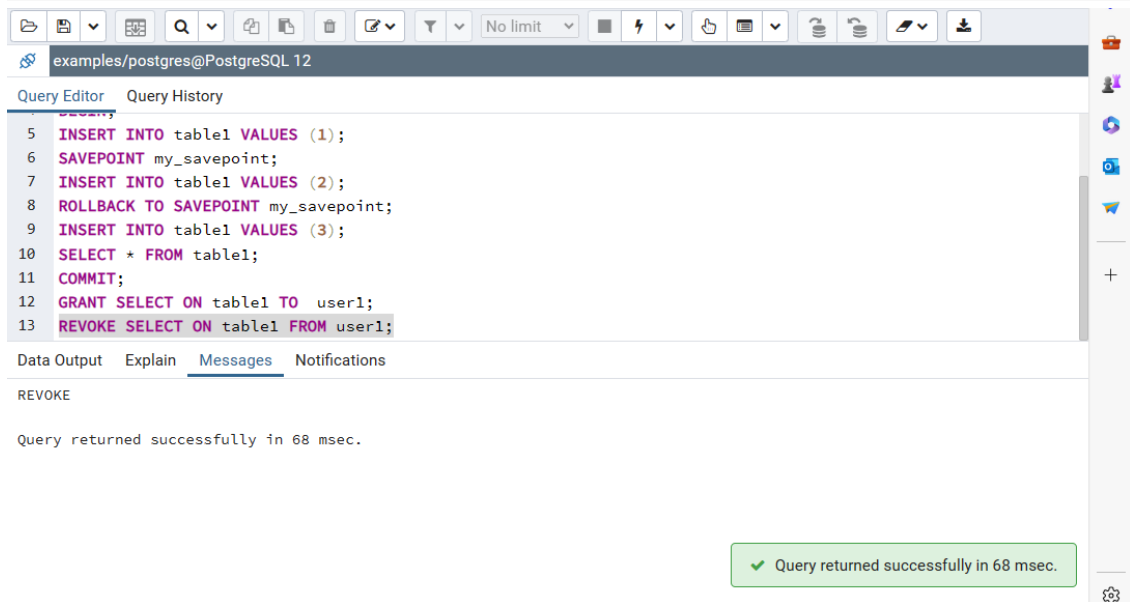
The screenshot shows the pgAdmin Query Editor interface. The query editor contains the following SQL commands:

```
1 BEGIN;  
5 INSERT INTO table1 VALUES (1);  
6 SAVEPOINT my_savepoint;  
7 INSERT INTO table1 VALUES (2);  
8 ROLLBACK TO SAVEPOINT my_savepoint;  
9 INSERT INTO table1 VALUES (3);  
10 SELECT * FROM table1;  
11 COMMIT;  
12 GRANT SELECT ON table1 TO user1;  
13 REVOKE SELECT ON table1 FROM user1;
```

The Messages tab is selected, showing the following output:

```
GRANT  
  
Query returned successfully in 90 msec.
```

A green status bar at the bottom right indicates: "Query returned successfully in 90 msec."



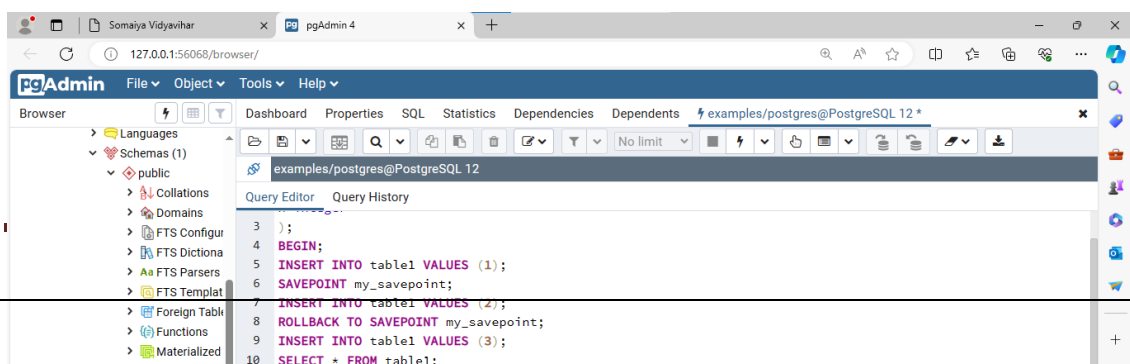
The screenshot shows the pgAdmin Query Editor interface. The query editor contains the following SQL commands:

```
1 BEGIN;  
5 INSERT INTO table1 VALUES (1);  
6 SAVEPOINT my_savepoint;  
7 INSERT INTO table1 VALUES (2);  
8 ROLLBACK TO SAVEPOINT my_savepoint;  
9 INSERT INTO table1 VALUES (3);  
10 SELECT * FROM table1;  
11 COMMIT;  
12 GRANT SELECT ON table1 TO user1;  
13 REVOKE SELECT ON table1 FROM user1;
```

The Messages tab is selected, showing the following output:

```
REVOKE  
  
Query returned successfully in 68 msec.
```

A green status bar at the bottom right indicates: "Query returned successfully in 68 msec."



The screenshot displays a PostgreSQL query editor interface with two panels. The top panel shows a query being executed, and the bottom panel shows the results and messages.

**Top Panel Query:**

```
21 );
22 BEGIN;
23     UPDATE employees SET salary = 60000 WHERE id = 1;
24     -- Simulate an error condition (e.g., invalid value or constraint violation)
25     -- For example, trying to set a negative salary which violates a constraint
26     UPDATE employees SET salary = -100 WHERE id = 2;
27 ROLLBACK;
28 SELECT * FROM employees;
29
```

**Top Panel Messages:**

```
ROLLBACK
Query returned successfully in 69 msec.
```

**Bottom Panel Query:**

```
34 );
35 BEGIN TRANSACTION;
36 SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
37
38 INSERT INTO transactions (transaction_id, transaction_date, amount, description)
39 VALUES (1, '2024-04-01', 100.00, 'Payment for goods');
40
41 INSERT INTO transactions (transaction_id, transaction_date, amount, description)
42 VALUES (2, '2024-04-02', 150.00, 'Refund for returned item');
```

**Bottom Panel Messages:**

```
WARNING: SET TRANSACTION can only be used in transaction blocks
SET
Query returned successfully in 69 msec.
```

A green status bar at the bottom right of the bottom panel indicates: **✓ Query returned successfully in 69 msec.**



Query Editor

```
1 CREATE TABLE employees(  
2     employee_id SERIAL PRIMARY KEY,  
3     first_name VARCHAR(50),  
4     last_name VARCHAR(50),  
5     department VARCHAR(50)  
6 );  
7 INSERT INTO employees (first_name, last_name, department) VALUES ('John','Doe','HR');  
8 INSERT INTO employees (first_name, last_name, department) VALUES ('Jane','Smith','Finance');  
9  
10 GRANT SELECT, INSERT, UPDATE, DELETE ON employees TO my_user;  
11  
12 REVOKE SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public FROM my_user;  
13
```

Messages

REVOKE

Query returned successfully in 68 msec.

### Post Lab question:

#### 1. Discuss ACID properties of transaction with suitable example

Ans- ACID (Atomicity, Consistency, Isolation, and Durability) properties are fundamental principles

In database management systems (DBMS) that ensure reliable and robust transaction processing. Here's a discussion of each property along with a suitable example:

##### 1. Atomicity:

Atomicity ensures that a transaction is treated as a single unit of operation, which means either all of its operations are successfully completed or none are. If any part of the transaction fails, the entire transaction is rolled back to its original state.

Example:

Consider a banking transaction where a customer transfers money from one account to another. The transaction involves two steps: deducting the amount from the sender's account

and crediting the same amount to the recipient's account. If the system deducts the amount

from the sender's account but fails to credit it to the recipient's account due to a system crash

or error, atomicity ensures that the entire transaction is rolled back, so the sender's account

remains unaffected.

## 2. Consistency:

Consistency ensures that the database remains in a consistent state before and after the execution of a transaction. This means that all integrity constraints, such as foreign key constraints, unique constraints, and business rules, must be maintained to ensure the validity of the data.

### Example:

In an online shopping application, when a customer places an order, the system checks if the ordered items are available in the inventory. If the items are available, the order is placed successfully, and the inventory is updated accordingly. Consistency ensures that the inventory is correctly updated after the order is placed, maintaining the integrity of the inventory data.

## 3. Isolation:

Isolation ensures that the concurrent execution of transactions does not interfere with each other. Each transaction should appear to execute in isolation, regardless of other concurrent transactions. This property prevents issues such as dirty reads, non-repeatable reads, and phantom reads.

### Example:

Consider two users simultaneously transferring money between their accounts in a banking system. Isolation ensures that even if the transactions occur simultaneously, each transaction is executed independently without interference from the other. Therefore, one user's transaction should not affect the outcome of the other user's transaction.

#### 4. Durability:

Durability ensures that once a transaction is committed, its effects persist even in the event of system failures. This means that once data is successfully written to the database, it remains there even if the system crashes or restarts.

Example:

After a successful funds transfer in a banking system, the changes to the account balances must be permanently recorded in the database. Even if the system crashes immediately after the transaction is committed, durability ensures that the transferred funds remain reflected in the account balances when the system is recovered.

In summary, the ACID properties are essential for ensuring the reliability, consistency, and integrity of transactions in a database system, which is crucial for maintaining data accuracy and reliability in various applications.

## **2. What is the purpose of the SAVEPOINT command in SQL?**

Ans- In SQL, the SAVEPOINT command serves the purpose of creating a named point within a transaction to which you can later roll back if needed. It allows for more granular control over transactions, especially in scenarios where you might want to undo only a part of the transaction without rolling back the entire transaction.

Here's a breakdown of the purpose and usage of the SAVEPOINT command:

#### 1. Granular Rollback:

- The SAVEPOINT command allows you to set a marker within a transaction, essentially creating a point to which you can rollback later.
- This is particularly useful when you have a complex transaction with multiple steps, and you want to undo only a portion of it in case of an error or unexpected condition.

#### 2. Nested Transactions:

- SAVEPOINTS can be nested within transactions, providing a way to create multiple points within the same transaction for potential rollbacks.

- This nested approach offers flexibility in handling transactions, allowing you to manage different parts of a transaction independently.

### 3. Improved Error Handling:

- By using SAVEPOINTS, you can implement more robust error handling mechanisms in your SQL code.
- If an error occurs during the execution of a transaction, you can rollback to a specific SAVEPOINT rather than rolling back the entire transaction, thus minimizing data loss and maintaining database consistency.