Batch: A1 Roll No.: 16010123012
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 3: Utilize SQL for Relational Database Operations
CO 5: Apply Transaction Management, 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, Slberchatz, Sudarshan : "Database Systems Concept", 5th Edition , McGraw Hill
4. Elmasri and Navathe,"Fundamentals of database Systems", 4th 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 COMMITED

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 fill 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 immidiatly errors out when process B commits
```

If process B would rolls 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.

DCL:

```
C:\Users\STUDENT>cd ..

C:\Users>cd ..

C:\>cd Program*

C:\Program Files>cd postg*

C:\Program Files\PostgreSQL>cd 17

C:\Program Files\PostgreSQL\17>cd bin

C:\Program Files\PostgreSQL\17\bin>
```

```
C:\Program Files\PostgreSQL\17\bin>psql -d empview -U postgres -W
Password:
psql (17.2)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

empview=# CREATE USER aaryan WITH PASSWORD 'password';
CREATE ROLE
empview=# CREATE USER aditey WITH PASSWORD 'password';
CREATE ROLE
empview=# CREATE USER aditya WITH PASSWORD 'password';
CREATE ROLE
empview=# GRANT SELECT, INSERT ON department TO aaryan;
GRANT
empview=# GRANT ALL ON department TO aditey;
GRANT
empview=# GRANT SELECT ON department TO aditya;
GRANT
empview=#
```

Command Prompt - psql -d empview -U aaryan -W

```
Microsoft Windows [Version 10.0.19045.5247]
(c) Microsoft Corporation. All rights reserved.

C:\Users\STUDENT>cd ..

C:\Users>cd ..

C:\>cd "C:\Program Files\PostgreSQL\17\bin"

C:\Program Files\PostgreSQL\17\bin>psql -d empview -U aaryan -W
Password:
psql (17.2)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

empview=> SELECT * FROM department;
   dname    | dnumber |  mgr_ssn  | mgr_start_date
------------+---------+-----------+----------------
 HR         |       1 | 111111111 | 2020-01-10
 Finance    |       2 | 222222222 | 2019-03-15
 IT         |       3 | 333333333 | 2021-06-20
 Marketing  |       4 | 444444444 | 2018-07-25
 Operations |       5 | 555555555 | 2022-02-05
(5 rows)


empview=> DELETE FROM department WHERE mgr_ssn = '111111111';
ERROR:  permission denied for table department
empview=> INSERT INTO DEPARTMENT(dname, dnumber, mgr_ssn, mgr_start_date) VALUES ('Creative', 6, '123456789', '2025-03-28')
empview-> ;
INSERT 0 1
empview=> SELECT * FROM department;
   dname    | dnumber |  mgr_ssn  | mgr_start_date
------------+---------+-----------+----------------
 HR         |       1 | 111111111 | 2020-01-10
 Finance    |       2 | 222222222 | 2019-03-15
 IT         |       3 | 333333333 | 2021-06-20
 Marketing  |       4 | 444444444 | 2018-07-25
 Operations |       5 | 555555555 | 2022-02-05
 Creative   |       6 | 123456789 | 2025-03-28
(6 rows)


empview=>
```

Command Prompt - psql -d empview -U aditya -W

```
C:\Users>cd ..

C:\>cd "C:\Program Files\PostgreSQL\17\bin"

C:\Program Files\PostgreSQL\17\bin>psql -d empview -U aditya -W
Password:
psql (17.2)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

empview=> SELECT * FROM department;
   dname    | dnumber |  mgr_ssn  | mgr_start_date
------------+---------+-----------+----------------
 HR         |       1 | 111111111 | 2020-01-10
 Finance    |       2 | 222222222 | 2019-03-15
 IT         |       3 | 333333333 | 2021-06-20
 Marketing  |       4 | 444444444 | 2018-07-25
 Operations |       5 | 555555555 | 2022-02-05
 Creative   |       6 | 123456789 | 2025-03-28
(6 rows)


empview=>  INSERT INTO DEPARTMENT(dname, dnumber, mgr_ssn, mgr_start_date) VALUES ('Creative', 6, '123456789', '2025-03-28')
empview-> ;
ERROR:  permission denied for table department
empview=>
```

```
empview=# REVOKE SELECT ON department FROM aditya;
REVOKE
empview=#
```

Command Prompt - psql -d empview -U aditya -W

```
empview=> SELECT * FROM department;
ERROR:  permission denied for table department
empview=>
```

TCL:

Query | Query History

```
1  SELECT * FROM department;
2
```

Data Output | Messages | Notifications

	dname character varying (100)	dnumber [PK] integer	mgr_ssn character (9)	mgr_start_date date
1	HR	1	111111111	2020-01-10
2	Finance	2	222222222	2019-03-15
3	IT	3	333333333	2021-06-20
4	Marketing	4	444444444	2018-07-25
5	Operations	5	555555555	2022-02-05
6	Creative	6	123456789	2025-03-28

Query | Query History

```
1  begin transaction;
```

Data Output | Messages | Notifications

```
BEGIN

Query returned successfully in 97 msec.
```

Query | Query History

```
1  INSERT INTO department VALUES ('Tech', 7, 'lllllllll', '2025-04-04');
```

Data Output | Messages | Notifications

```
INSERT 0 1

Query returned successfully in 41 msec.
```

Query | Query History

```
1  SELECT * FROM department;
2  BEGIN transaction;
3  INSERT INTO department VALUES ('Tech', 7, '111111111', '2025-04-04');
4  SAVEPOINT A;
5  INSERT INTO department VALUES ('Social', 8, '222222222', '2025-04-04');
6  ROLLBACK TO A;
7  END transaction;
8  SELECT * FROM department;
```

Data Output | Messages | Notifications

```
SAVEPOINT

Query returned successfully in 47 msec.
```

Query | Query History

```
1  SELECT * FROM department;
2  BEGIN transaction;
3  INSERT INTO department VALUES ('Tech', 7, '111111111', '2025-04-04');
4  SAVEPOINT A;
5  INSERT INTO department VALUES ('Social', 8, '222222222', '2025-04-04');
6  ROLLBACK TO A;
7  END transaction;
8  SELECT * FROM department;
```

Data Output | Messages | Notifications

```
COMMIT

Query returned successfully in 45 msec.
```

Query | Query History

```
1  SELECT * FROM department;
2  BEGIN transaction;
3  INSERT INTO department VALUES ('Tech', 7, '111111111', '2025-04-04');
4  SAVEPOINT A;
5  INSERT INTO department VALUES ('Social', 8, '222222222', '2025-04-04');
6  ROLLBACK TO A;
7  END transaction;
8  SELECT * FROM department;
```

Data Output | Messages | Notifications

	dname character varying (100)	dnumber [PK] integer	mgr_ssn character (9)	mgr_start_date date
1	HR	1	111111111	2020-01-10
2	Finance	2	222222222	2019-03-15
3	IT	3	333333333	2021-06-20
4	Marketing	4	444444444	2018-07-25
5	Operations	5	555555555	2022-02-05
6	Creative	6	123456789	2025-03-28
7	Tech	7	111111111	2025-04-04

Query | Query History

```
1  SELECT * FROM department;
2  BEGIN transaction;
3  INSERT INTO department VALUES ('PM', 8, '111111111', '2025-04-04');
4  SAVEPOINT A;
5  INSERT INTO department VALUES ('CM', 9, '222222222', '2025-04-04');
6  ROLLBACK TO A;
7  END transaction;
8  SELECT * FROM department;
```

Data Output | Messages | Notifications

	dname character varying (100)	dnumber [PK] integer	mgr_ssn character (9)	mgr_start_date date
1	HR	1	111111111	2020-01-10
2	Finance	2	222222222	2019-03-15
3	IT	3	333333333	2021-06-20
4	Marketing	4	444444444	2018-07-25
5	Operations	5	555555555	2022-02-05
6	Creative	6	123456789	2025-03-28
7	Tech	7	111111111	2025-04-04

Query | Query History

```
1  SELECT * FROM department;
2  BEGIN transaction;
3  INSERT INTO department VALUES ('PM', 8, '111111111', '2025-04-04');
4  SAVEPOINT A;
5  INSERT INTO department VALUES ('CM', 9, '222222222', '2025-04-04');
6  ROLLBACK TO A;
7  END transaction;
8  SELECT * FROM department;
```

Data Output | Messages | Notifications

```
BEGIN

Query returned successfully in 112 msec.
```

Query | Query History

```
1  SELECT * FROM department;
2  BEGIN transaction;
3  INSERT INTO department VALUES ('PM', 8, '111111111', '2025-04-04');
4  SAVEPOINT A;
5  INSERT INTO department VALUES ('CM', 9, '222222222', '2025-04-04');
6  ROLLBACK TO A;
7  END transaction;
8  SELECT * FROM department;
```

Data Output | Messages | Notifications

```
SAVEPOINT

Query returned successfully in 50 msec.
```

Query | Query History

```
1  SELECT * FROM department;
2  BEGIN transaction;
3  INSERT INTO department VALUES ('PM', 8, '111111111', '2025-04-04');
4  SAVEPOINT A;
5  INSERT INTO department VALUES ('CM', 9, '222222222', '2025-04-04');
6  ROLLBACK TO A;
7  END transaction;
8  SELECT * FROM department;
```

Data Output | Messages | Notifications

```
INSERT 0 1

Query returned successfully in 45 msec.
```

Query | Query History

```
1  SELECT * FROM department;
2  BEGIN transaction;
3  INSERT INTO department VALUES ('PM', 8, '111111111', '2025-04-04');
4  SAVEPOINT A;
5  INSERT INTO department VALUES ('CM', 9, '222222222', '2025-04-04');
6  ROLLBACK TO A;
7  END transaction;
8  SELECT * FROM department;
```

Data Output | Messages | Notifications

```
COMMIT

Query returned successfully in 48 msec.
```

Query | Query History

```
1  SELECT * FROM department;
2  BEGIN transaction;
3  INSERT INTO department VALUES ('PM', 8, '111111111', '2025-04-04');
4  SAVEPOINT A;
5  INSERT INTO department VALUES ('CM', 9, '222222222', '2025-04-04');
6  ROLLBACK TO A;
7  END transaction;
8  SELECT * FROM department;
```

Data Output | Messages | Notifications

	dname character varying (100)	dnumber [PK] integer	mgr_ssn character (9)	mgr_start_date date
1	HR	1	111111111	2020-01-10
2	Finance	2	222222222	2019-03-15
3	IT	3	333333333	2021-06-20
4	Marketing	4	444444444	2018-07-25
5	Operations	5	555555555	2022-02-05
6	Creative	6	123456789	2025-03-28
7	Tech	7	111111111	2025-04-04
8	PM	8	111111111	2025-04-04
9	CM	9	222222222	2025-04-04

Conclusion:

I have successfully implemented and demonstrated the usage of TCL and DCL commands in PostgreSQL and cmd. Through this experiment, I explored how

transaction control commands like COMMIT, ROLLBACK, and SAVEPOINT help manage data consistency and integrity during database operations. Additionally, the use of DCL commands such as GRANT and REVOKE allowed me to control user permissions effectively. This hands-on experience deepened my understanding of transaction management and access control, which are crucial for maintaining secure and reliable database systems.

Post Lab question:

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

In database systems, ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability. These properties ensure that database transactions are processed reliably and maintain the integrity of data.

1. **Atomicity**
   This property ensures that a transaction is treated as a single, indivisible unit. It either completes fully or not at all. If any part of the transaction fails, the entire transaction is rolled back, and the database remains unchanged.
   Example: If you transfer ₹1000 from Account A to Account B, the deduction from A and addition to B must both occur. If the addition fails, the deduction must also be undone.

2. **Consistency**
   Consistency ensures that a transaction brings the database from one valid state to another, maintaining all predefined rules, constraints, and integrity.
   Example: If there's a rule that an account balance cannot be negative, a transaction that would violate this rule must be rejected to maintain consistency.

3. **Isolation**
   Isolation means that concurrent transactions occur independently without interference. The intermediate state of a transaction must be invisible to others.
   Example: If two users are transferring money at the same time, their transactions should not affect each other, and results should appear as if the transactions were run one after another.

4. **Durability**
   Once a transaction is committed, its changes are **permanently saved** in the database, even in the event of a system failure.
   Example: After you complete a fund transfer and receive a confirmation, the changes should persist even if the server crashes immediately afterward.

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

The SAVEPOINT command in SQL is used to create a specific point within a transaction to which you can later roll back if needed. It allows partial rollbacks of a transaction without affecting the entire sequence of operations. This is especially useful in complex transactions where certain operations might fail, but you still want to preserve the successful parts executed before the error. By setting savepoints, developers can handle errors more gracefully and maintain finer control over data consistency. For example, if a transaction includes multiple insert operations and one fails, using ROLLBACK TO SAVEPOINT can undo just that part while keeping the rest intact. This makes SAVEPOINT a valuable tool for enhancing flexibility, reliability, and error handling within SQL transactions.