**Batch: B1 Roll No.: 16010122104**

**Experiment / assignment / tutorial No. 8**

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

**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, Slberchatz, Sudarshan : “Database Systems Concept”, 5th Edition , McGraw

Hill

1. 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.

| could not serialize access due to concurrent update |
| --- |

Let’s observe an issue that can occur while using the repeatable read isolation level — the 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. |

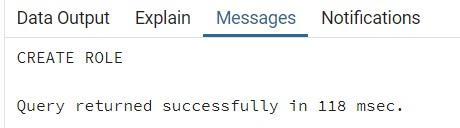
Both transactions have modified what the other transaction would have read in the select statements. If both would allow to commit this would violate the Serializable behaviour, because if they were run one at a time, one of the transactions would have seen the new record inserted by the other transaction.

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

Demonstrate DCL and TCL language commands on your database.

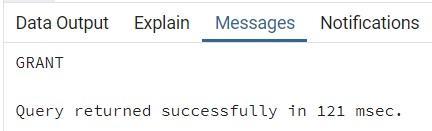
1. Creation of roles *create role Joe login*

*password 'Abcd1234';*



1. GRANT Admin:

*GRANT ALL PRIVILEGES ON Login TO Joe;*



Joe:

SELECT \* FROM Login;

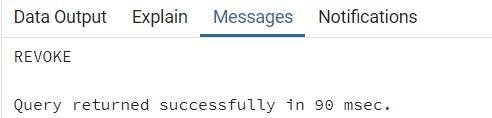


1. REVOKE Admin:

REVOKE SELECT

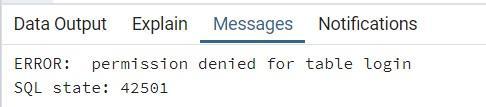
ON Login

FROM Joe;



Joe:

Select \* from Login;



1. COMMIT

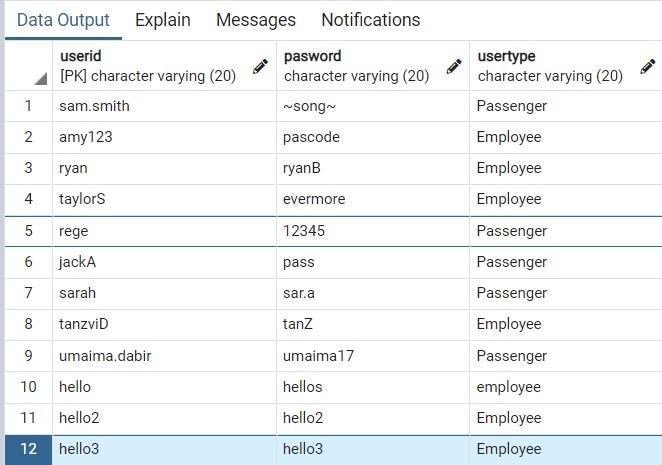
BEGIN;

INSERT INTO Login Values('hello3', 'hello3', 'Employee');

COMMIT;



SELECT \* from Login;



1. SAVEPOINT and ROLLBACK

BEGIN;

INSERT INTO Login Values('hello4', 'hello4', 'Employee');

SAVEPOINT point1;

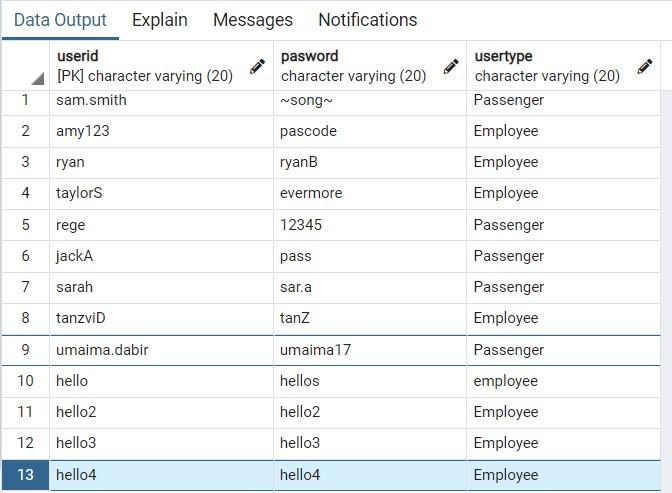
INSERT INTO Login Values('hello5', 'hello5', 'Employee');

ROLLBACK TO SAVEPOINT point1;

COMMIT;



SELECT \* FROM Login;



**Conclusion:**

Through this experiment we learnt about DCL and TCL commands like GRANT, REVOKE and applied the same in out problem statement database.

**Postlab question:**

1. Discuss ACID properties of transaction with suitable example  
   ans :   
   **Atomicity**: Imagine a banking transaction where you transfer money from one account to another. Atomicity ensures that if the transfer fails midway (due to system failure or any other reason), neither the money is deducted from the sender's account nor credited to the receiver's account.

**Consistency:** Consider a scenario where you have a constraint that each student's grade must be between A and F. If a transaction attempts to update a student's grade to "Z," which violates this constraint, consistency ensures that the transaction is aborted, and the database remains in a consistent state.

**Isolation**: Suppose two transactions are updating the same bank account balance concurrently. Isolation ensures that one transaction doesn't interfere with the other. For instance, even if one transaction updates the balance, the other transaction shouldn't read the intermediate, inconsistent value until the first transaction commits.

Durability: After a successful transaction, durability ensures that the changes made are permanent and survive system failures. For instance, if a user updates their profile information, the changes should persist in the database, even if the database server crashes immediately after the transaction commits.  
  
2.**What is the purpose of the SAVEPOINT command in SQL?**   
  
Ans :   
  
The SAVEPOINT command in SQL allows you to create a point within a transaction to which you can later roll back if needed. It provides a way to partially roll back a transaction, rather than rolling back the entire transaction. This command is useful when you want finer control over transaction management, especially in scenarios where you need to handle errors gracefully within a transaction or implement complex logic involving conditional rollback.