Batch: E-2 **Roll No.: 16010123325**

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 [, ...]
      ROM { [ 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.
```

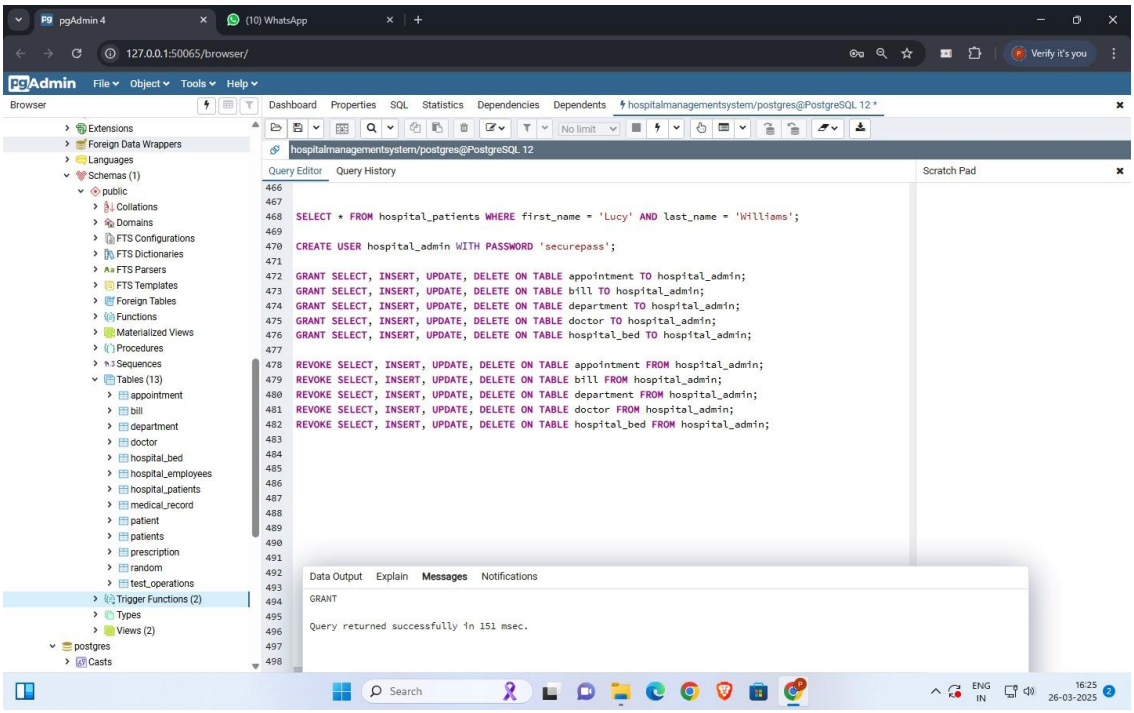
Both transactions have modified what the other transaction would have read in the select statements. If both would allow to commit this would violate the Serializable behaviour, because if they were run one at a time, one of the transactions would have seen the new record inserted by the other transaction.

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

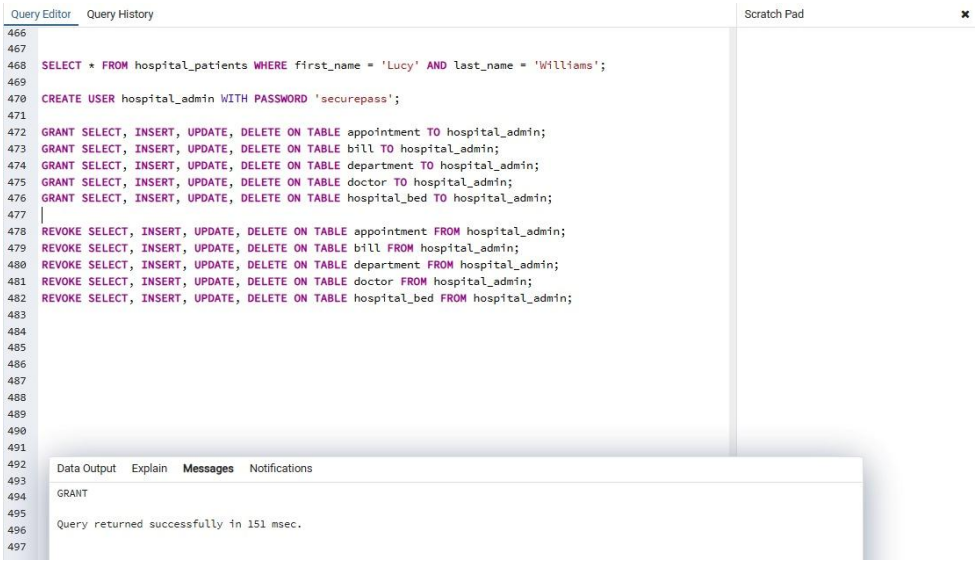
Demonstrate DCL and TCL language commands on your database.

DCL

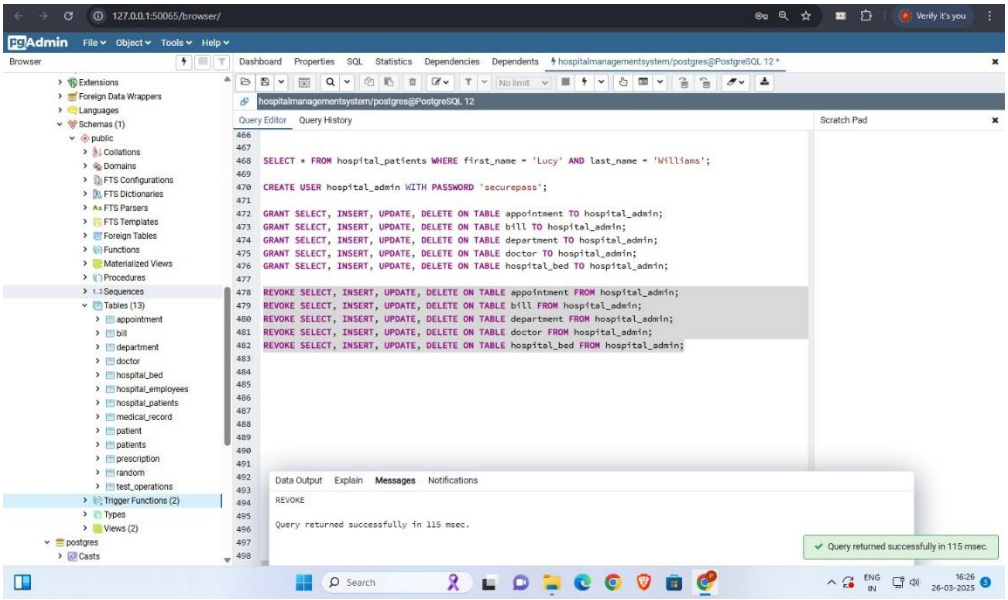
Create hospital_admin



Grant permission



Permission Revoke



Grant insert permission on appointment table (insert at appointment_id=11)

```
hospitalmanagementdb=> INSERT INTO appointment (appointment_id, patient_id, doctor_id, date, status, fees)
VALUES (11, 9, 9, '2025-04-01 10:00:00', 'Scheduled', 1800.00);
INSERT 0 1
hospitalmanagementdb=> SELECT * FROM appointment WHERE appointment_id = 11;
 appointment_id | patient_id | doctor_id |        date         |  status   |  fees
----------------+------------+-----------+---------------------+-----------+---------
             11 |          9 |         9 | 2025-04-01 10:00:00 | Scheduled | 1800.00
(1 row)
```

Grant delete permission on appointment table (delete appointment_id=11)

```
hospitalmanagementdb=> DELETE FROM appointment WHERE appointment_id = 11;
DELETE 1
hospitalmanagementdb=> SELECT * FROM appointment WHERE appointment_id = 11;
 appointment_id | patient_id | doctor_id | date | status | fees
----------------+------------+-----------+------+--------+------
(0 rows)
```

Grant update permission on appointment table

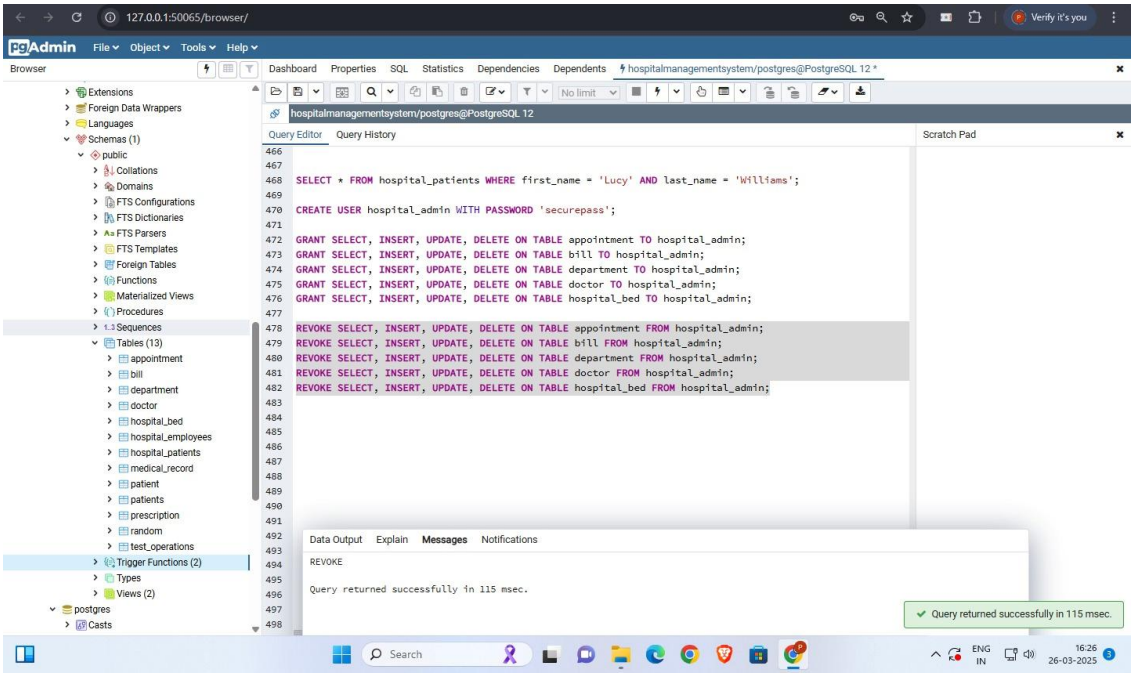
```
hospitalmanagementdb=> UPDATE appointment
SET status = 'Completed', fees = 2000.00
WHERE appointment_id = 11;
UPDATE 1
hospitalmanagementdb=> SELECT * FROM appointment WHERE appointment_id = 11;
 appointment_id | patient_id | doctor_id |        date         |  status   |  fees
----------------+------------+-----------+---------------------+-----------+---------
             11 |          9 |         9 | 2025-04-01 10:00:00 | Completed | 2000.00
(1 row)
```

Grant truncate permission on appointment table

```
hospitalmanagementdb=# SELECT * FROM dummy_appointment;
 appointment_id | patient_name | doctor_name |  appointment_date   |  status   |  fees
----------------+--------------+-------------+---------------------+-----------+---------
              1 | John Doe     | Dr. Smith   | 2025-04-01 10:00:00 | Scheduled | 1800.00
              2 | Alice Brown  | Dr. Johnson | 2025-04-02 11:30:00 | Completed | 2000.00
              3 | Bob White    | Dr. Lee     | 2025-04-03 14:00:00 | Scheduled | 2500.00
(3 rows)

hospitalmanagementdb=# TRUNCATE TABLE dummy_appointment;
TRUNCATE TABLE
hospitalmanagementdb=# SELECT * FROM dummy_appointment;
 appointment_id | patient_name | doctor_name | appointment_date | status | fees
----------------+--------------+-------------+------------------+--------+------
(0 rows)
```

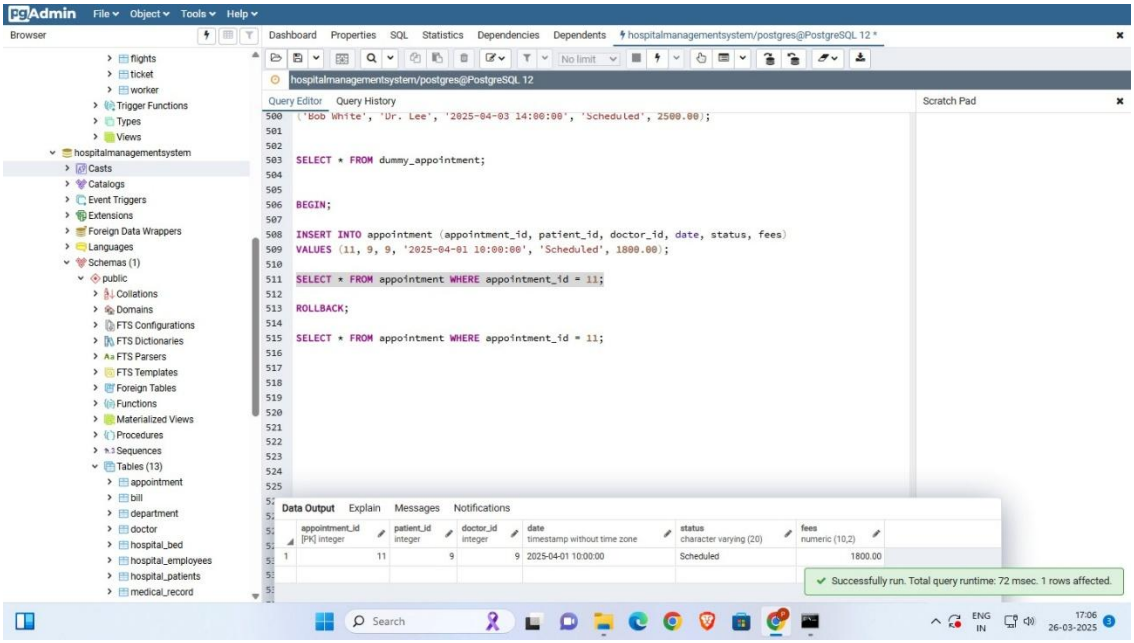
All Queries to grant, revoke and finally drop user



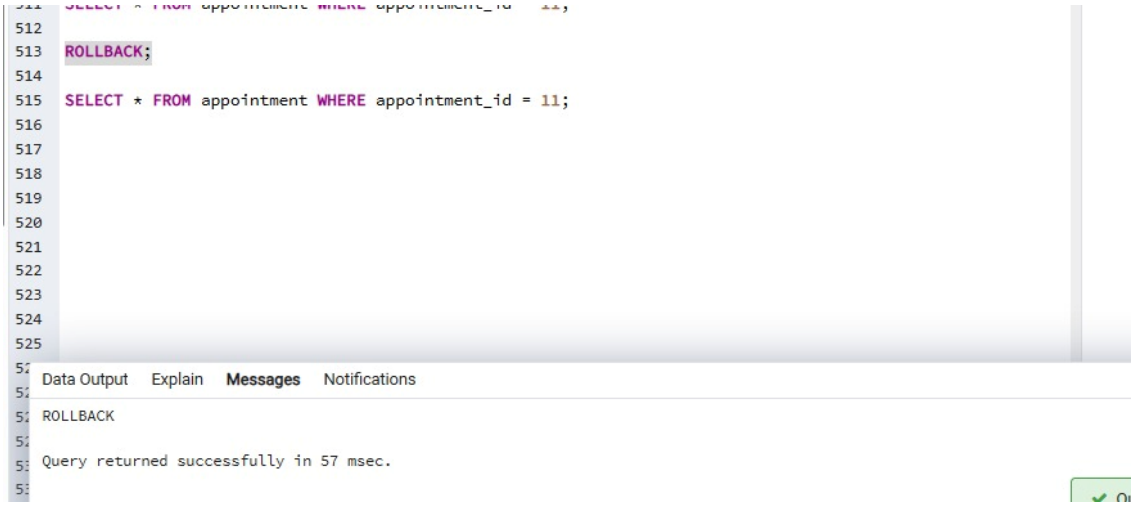
TCL

Rollback

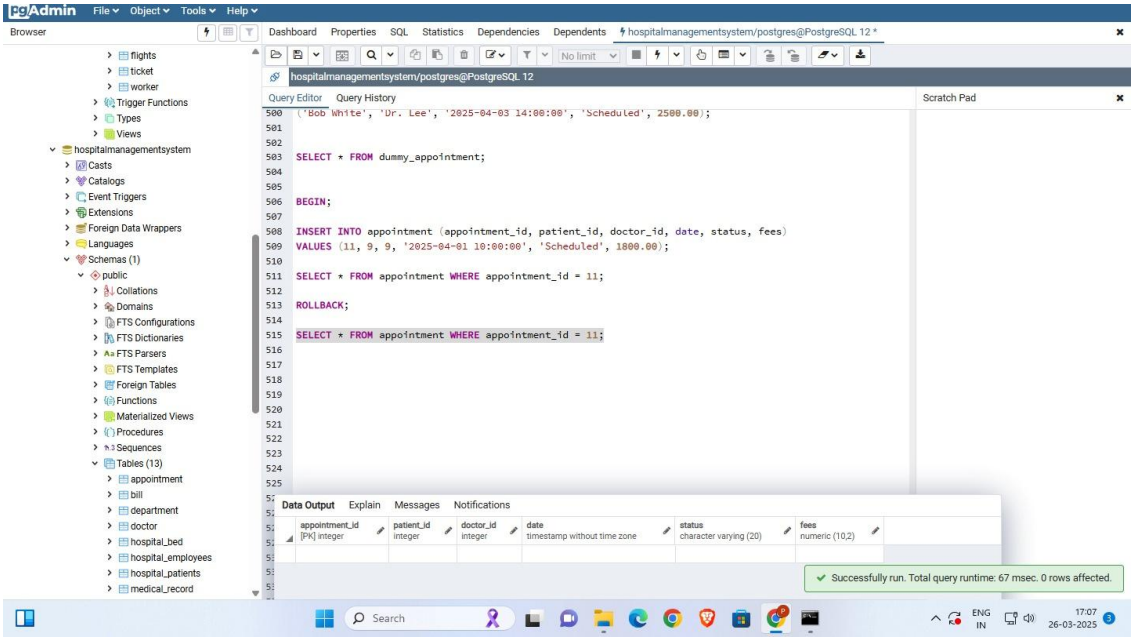
Begin Transaction & Select Appointment_id=11



RollBack

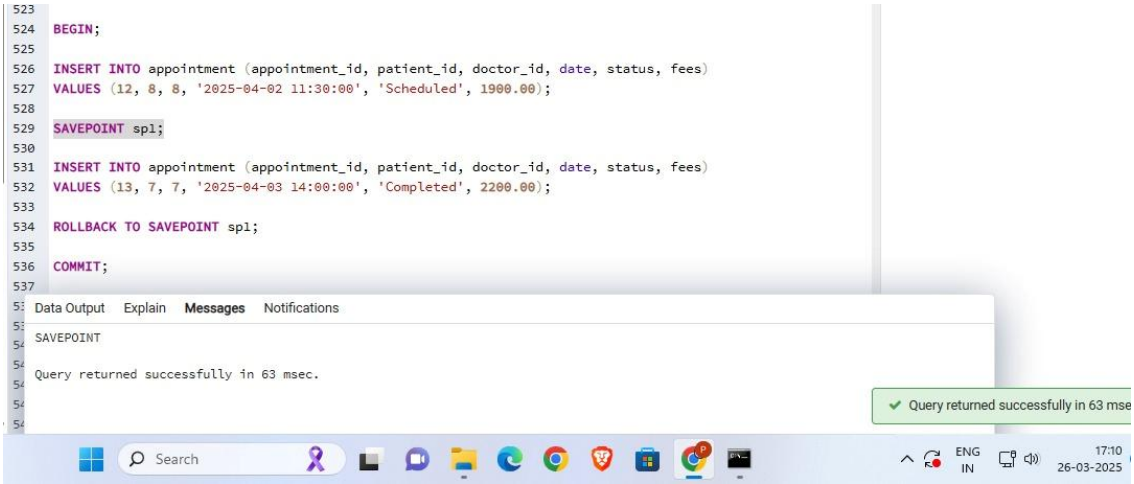


Rollback delete Operation



Commit

Savepoint S1



Rollback To Savepoint S1

```
524 BEGIN;
525
526 INSERT INTO appointment (appointment_id, patient_id, doctor_id, date, status, fees)
527 VALUES (12, 8, 8, '2025-04-02 11:30:00', 'Scheduled', 1900.00);
528
529 SAVEPOINT sp1;
530
531 INSERT INTO appointment (appointment_id, patient_id, doctor_id, date, status, fees)
532 VALUES (13, 7, 7, '2025-04-03 14:00:00', 'Completed', 2200.00);
533
534 ROLLBACK TO SAVEPOINT sp1;
535
536 COMMIT;
537
```

Data Output Explain **Messages** Notifications

```
ROLLBACK

Query returned successfully in 74 msec.
```

Commit

```
523
524 BEGIN;
525
526 INSERT INTO appointment (appointment_id, patient_id, doctor_id, date, status, fees)
527 VALUES (12, 8, 8, '2025-04-02 11:30:00', 'Scheduled', 1900.00);
528
529 SAVEPOINT sp1;
530
531 INSERT INTO appointment (appointment_id, patient_id, doctor_id, date, status, fees)
532 VALUES (13, 7, 7, '2025-04-03 14:00:00', 'Completed', 2200.00);
533
534 ROLLBACK TO SAVEPOINT sp1;
535
536 COMMIT;
537
538
539
540
541
542
```

Data Output Explain **Messages** Notifications

```
COMMIT

Query returned successfully in 56 msec.
```

Conclusion:

The above experiment highlights Implementation of TCL and DCL commands on our database 'Hospital Management System'. In TCL the rollback and commit queries were used. DCL involved user management and use of SQL shell to perform queries by that user.

Post Lab question:

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

   ACID properties ensure reliable and consistent transactions in a database.

   It stands for:

   Atomicity – Ensures that a transaction is all-or-nothing.
   Consistency – Ensures the database remains valid before and after a transaction.
   Isolation – Ensures transactions don't interfere with each other.
   Durability – Ensures that committed transactions persist, even after system failures.

   Example-

```
BEGIN;  -- Start transaction

-- Atomicity: Either both insertions happen, or none

INSERT INTO Customer (U_id, F_name, L_name, Phone_no, Age)

VALUES (101, 'John', 'Doe', '1234567890', 30);

INSERT INTO Payment (Transaction_id, Amount, Booking_id,
Payment_mode)

VALUES (5001, 2500.00, 101, 'Credit Card');

COMMIT;  -- Save changes permanently
```

   If an error occurs in one query, the transaction fails entirely (Atomicity)
   After commit, the changes are permanent (Durability)

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

   SAVEPOINT allows setting a checkpoint inside a transaction. If needed, we can ROLLBACK to that SAVEPOINT instead of rolling back the entire transaction.

   Key Benefits:

   - Helps in error handling.
   - Allows undoing specific parts of a transaction.
   - Improves flexibility in transaction control.