K. J. Somaiya College of Engineering, Mumbai-77
(Autonomous College Affiliated to University of Mumbai)

Batch: A1 Roll No.:
16010120015-Yash
16010120006-Dikshita
16010120018-Jinay
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
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 ():

Creating a user

CREATE USER YASH WITH PASSWORD 'DATA1234';

```
23  CREATE USER YASH WITH PASSWORD 'DATA1234';
24
```

Data Output | Explain | Messages | Notifications

```
CREATE ROLE

Query returned successfully in 64 msec.
```

Grant access:

GRANT INSERT ON ITEMS TO YASH;

```
32
33  GRANT INSERT ON ITEMS TO YASH;
```

Data Output | Explain | Messages | Notifications

```
GRANT

Query returned successfully in 33 msec.
```

Grant access:

GRANT ALL PRIVILEGES ON ITEMS TO YASH;

```
25  GRANT ALL PRIVILEGES ON ITEMS TO YASH;
26
```

Data Output | Explain | Messages | Notifications

```
GRANT

Query returned successfully in 159 msec.
```

CREATE NEW USER AND GRANT ACCESS

```
CREATE USER DIKSHITA1 WITH PASSWORD 'DIKSHITA1234';
GRANT ALL PRIVILEGES ON WEAPONS TO DIKSHITA1;
```

```
37  CREATE USER DIKSHITA1 WITH PASSWORD 'DIKSHITA1234';
38  GRANT ALL PRIVILEGES ON WEAPONS TO DIKSHITA1;
39
```

Data Output | Explain | Messages | Notifications

```
GRANT

Query returned successfully in 52 msec.
```

```
CREATE USER JINAY WITH PASSWORD 'JINAY1234';
GRANT ALL PRIVILEGES ON DEPARTMENT TO JINAY;
```

```
42  CREATE USER JINAY WITH PASSWORD 'JINAY1234';
43  GRANT ALL PRIVILEGES ON DEPARTMENT TO JINAY;
44
```

Data Output | Explain | Messages | Notifications

```
GRANT

Query returned successfully in 41 msec.
```

```
GRANT SELECT ON ITEMS TO YASH;
GRANT SELECT, UPDATE, INSERT ON WEAPONS TO DIKSHITA;
GRANT SELECT (WNAME), UPDATE  ON WEAPONS TO JINAY;
```

```
46
47  GRANT SELECT ON ITEMS TO YASH;
48  GRANT SELECT, UPDATE, INSERT ON WEAPONS TO DIKSHITA;
49  GRANT SELECT (WNAME), UPDATE  ON WEAPONS TO JINAY;
50
```

Data Output | Explain | Messages | Notifications

```
GRANT

Query returned successfully in 42 msec.
```

REVOKE

REVOKE ALL PRIVILEGES ON SOLDIERS FROM YASH;

```
27  REVOKE ALL PRIVILEGES ON SOLDIERS FROM YASH;
28
```

Data Output | Explain | Messages | Notifications

```
REVOKE

Query returned successfully in 35 msec.
```

REVOKE ALL PRIVILEGES ON WEAPONS FROM DIKSHITA;

```
51  REVOKE ALL PRIVILEGES ON WEAPONS FROM DIKSHITA;
52
```

Data Output | Explain | Messages | Notifications

```
REVOKE

Query returned successfully in 178 msec.
```

REVOKE ALL PRIVILEGES ON DEPARTMENT FROM JINAY;

```
52
53  REVOKE ALL PRIVILEGES ON DEPARTMENT FROM JINAY;
```

Data Output | Explain | Messages | Notifications

```
REVOKE

Query returned successfully in 92 msec.
```

TCL COMMANDS

BEGIN;

```
INSERT INTO DEP VALUES ('1603','11614');

SAVEPOINT save_point;

INSERT INTO DEP VALUES ('1609','11614');

ROLLBACK TO SAVEPOINT save_point;

INSERT INTO DEP VALUES ('1610','11611');

COMMIT;
```

```
464  BEGIN;
465      INSERT INTO DEP VALUES ('1603','11614');
466      SAVEPOINT save_point;
467      INSERT INTO DEP VALUES ('1609','11614');
468      ROLLBACK TO SAVEPOINT save_point;
469      INSERT INTO DEP VALUES ('1610','11611');
470  COMMIT;
471
```

Data Output | Explain | Messages | Notifications

```
COMMIT

Query returned successfully in 33 msec.
```

CHANGE IN TH TABLE

Value inserted in 6th and 7th Position .

```
482  SELECT * FROM DEP
```

Data Output | Explain | Messages | Notifications

	dssno integer	dep_no integer
1	1601	11601
2	1602	11602
3	1603	11603
4	1604	11604
5	1605	11605
6	1603	11614
7	1610	11611

Conclusion:

The purpose of this experiment was to Learn and implement the TCL and DCL Commands.

TCL

- Transaction Control Language commands are used to manage transactions in the database.

- Transaction Control Language are used to manage the changes made by DML-statements. It also allows statements to be grouped together into logical transactions.

DCL

- A Data Control Language is used to control access to data stored in a database (Authorization). Data Control Language is a component of Structured Query Language (SQL).
- The operations for which privileges may be granted to or revoked from a user.

Hence, The experiment succeeded in showing the TCL and DCL Commands Sucessfully .

Postlab question:

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

Answer:

ACID refers to the four key properties of a transaction:
atomicity, consistency, isolation, and durability.

- **Atomicity**
  - All changes to data are performed as if they are a single operation.
  - For example : In an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.
- **Consistency**
  - Data is in a consistent state when a transaction starts and when it ends.
  - For example: In an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.
- **Isolation**
  - The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized.
  - For example: In an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.
- **Durability**
  - After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure.
  - For example: In an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.