**Batch: A1 Roll No.: 1911004**

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

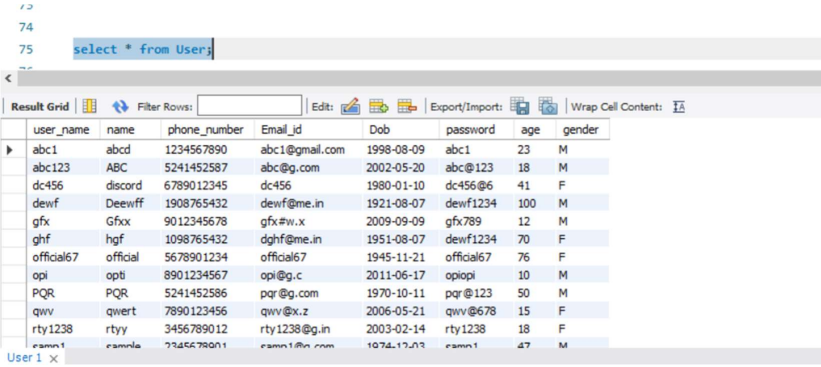
Demonstrate DCL and TCL language commands on your database.

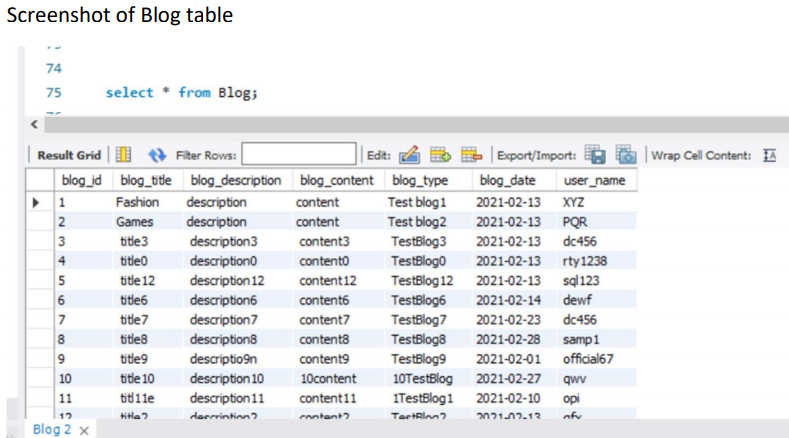
**Creating User and Granting Privileges to user:**

**Database use: blog\_management**

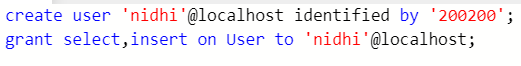
**Tables it has: User, Blog, shares, likes, comments, follow, deactivate, deactivated\_data**

**Screenshot of User table:**



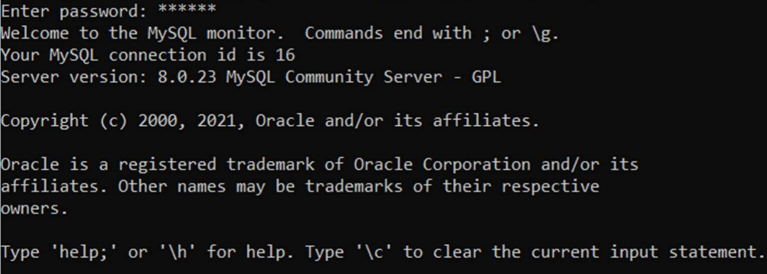


1. **First we create a new user in MySQL**

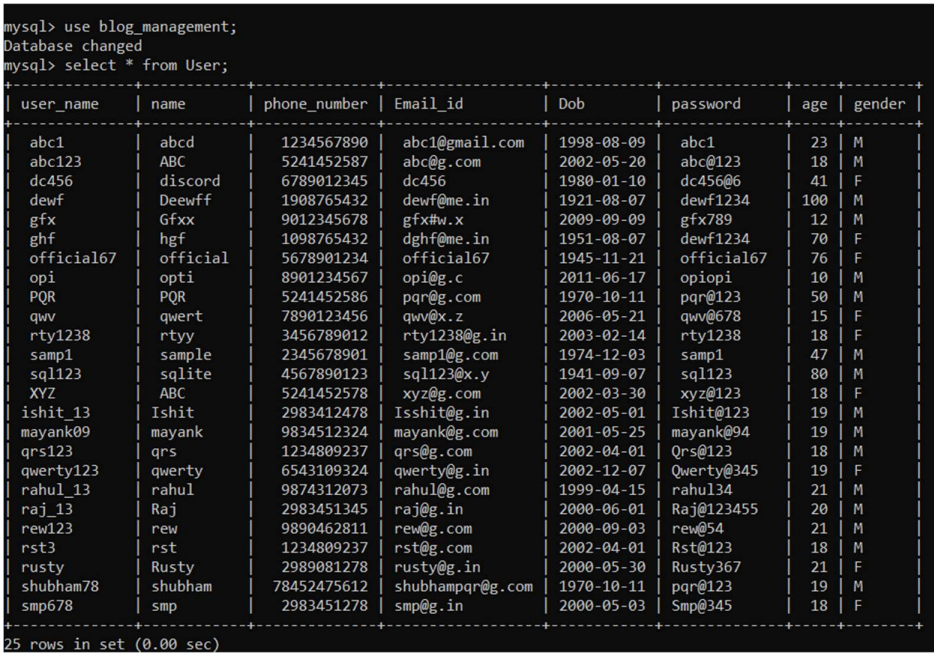
****

**We have granted the user the rights to select and insert on the User table**

**2. Now we will go to command line client and login to mysql with username and password provided for the user nidhi**



**3. Now executing the granted query**



**4. Inserting values of a new user**



**5. Trying to execute select query on blog table**



**So the user can only execute the query which have been granted to them. This can be achieved using DCL commands.**

**6. Revoking the rights from the user nidhi**

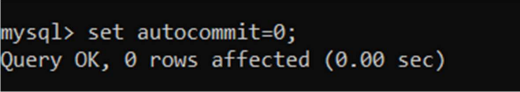


**7. after revoking**

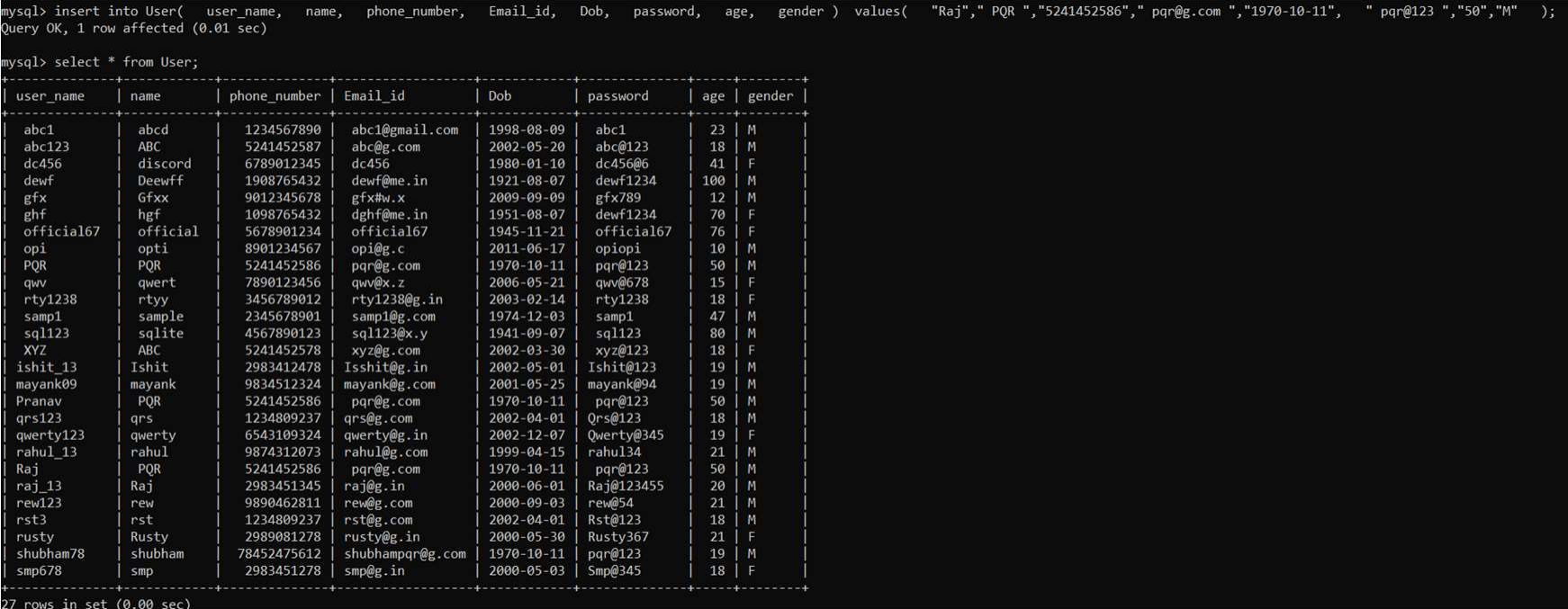


**TCL Commands:**

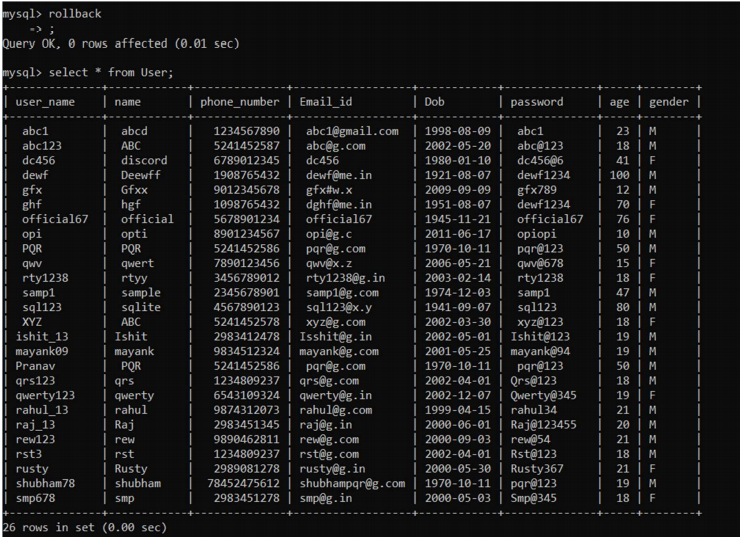
1. **Setting AUTOCOMMIT = “false” and using ROLLBACK to revert the changes done:**



**Inserting a User with user\_name = “Raj”**

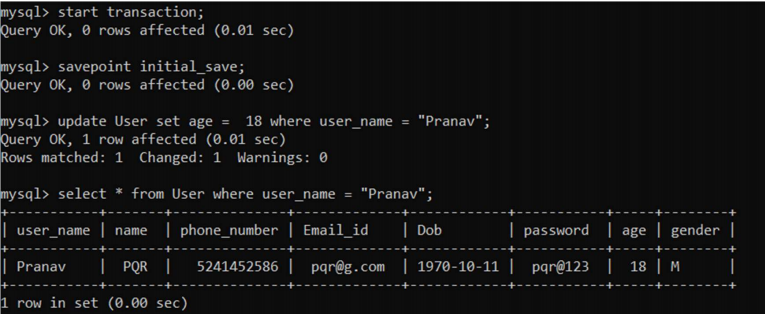


**Using rollback to revert the changes**

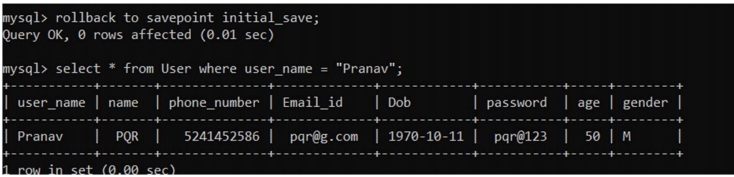


**Using the transaction and savepoint commands creating a savepoint;**

**Updating age of user = “Pranav” to 18**



**Rollback to the savepoint;**

****

**As you can see the original age of the user has been restored**

# Conclusion:

# In this experiment we have learnt about the DCL and TCL commands and implemented them on our database blog\_management.

# Post lab question:

# Discuss ACID properties of transaction with suitable example

# A transaction is a very small unit of a program and it may contain several low level tasks. A transaction in a database system must maintain Atomicity, Consistency, Isolation, and Durability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

# Atomicity − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction. 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 − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well. 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.

# Durability − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action. 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

# Isolation − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction. 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.