|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Database Management System Laboratory** | **Semester:** | **IV** |
| **Date of Performance:** | **04 / 04 / 2023** | **Batch No:** | **A – 2** |
| **Faculty Name:** | **Prof. Shila Dhande** | **Roll No.:** | **16014022050** |
| **Faculty Sign & Date:** |  | **Grade / Marks:** | **\_\_\_ / 25** |

**Experiment No.: 8**

**Title: Implementing TCL / DCL**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| **Objective:** To be able to implement TCL and DCL. |

|  |
| --- |
| **COs to be achieved:** |
| **CO2:** 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.  **CO4:** Demonstrate the concept of transaction, concurrency control and recovery techniques. |

|  |
| --- |
| **Books / Journals / Websites Referred:** |
| 1. G. K. Gupta:” Database Management Systems”, McGraw – Hill 2. Korth, Slberchatz, Sudarshan: “Database Systems Concept”, 6th Edition, McGraw Hill 3. Elmasri and Navathe, “Fundamentals of Database Systems”, 5th Edition, PEARSON Education. |

|  |
| --- |
| **Tools Required:** |
| * Postgresql Software |

|  |
| --- |
| **Theory:** |
| **DCL** stands for **Data Control Language** and 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 withdraws 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:   * **COMMIT** * **SAVEPOINT** * **ROLLBACK** * **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 roll back 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 cannot modify or lock the rows changed by other processes after the repeatable read transaction has begun.  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 behavior, 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 Details (Problem Statement, Query and Screenshots of Results):** |
| **Demonstrate DCL and TCL language commands on your database.**  **DCL**  -- creating roles  CREATE ROLE admin\_user WITH LOGIN PASSWORD 'admin\_user';  CREATE ROLE staff\_user WITH LOGIN PASSWORD 'staff\_user';  -- grants for admin\_usr  -- grant all privileges on all tables to the administrative user  GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO admin\_user;  -- grant permission to create schemas and roles  GRANT CREATE ON SCHEMA public TO admin\_user;  -- grants for staff\_user  -- grant necessary permissions for hotel staff user  GRANT SELECT, INSERT, UPDATE, DELETE ON booking TO staff\_user;  GRANT SELECT, INSERT, UPDATE, DELETE ON customer TO staff\_user;  GRANT SELECT, INSERT, UPDATE, DELETE ON customer\_services TO staff\_user;  GRANT SELECT, INSERT, UPDATE, DELETE ON rooms TO staff\_user;  GRANT SELECT, INSERT, UPDATE, DELETE ON services TO staff\_user;  -- grant specific permissions for payment (assuming staff can only view payments)  GRANT SELECT ON payment TO staff\_user;  -- revoking staff\_user permissions  -- revoke necessary permissions from hotel staff user  REVOKE SELECT, INSERT, UPDATE, DELETE ON public.booking FROM staff\_user;  REVOKE SELECT, INSERT, UPDATE, DELETE ON public.customer FROM staff\_user;  REVOKE SELECT, INSERT, UPDATE, DELETE ON public.customer\_services FROM staff\_user;  REVOKE SELECT, INSERT, UPDATE, DELETE ON public.rooms FROM staff\_user;  REVOKE SELECT, INSERT, UPDATE, DELETE ON public.services FROM staff\_user;  -- revoke SELECT permission on the payment table  REVOKE SELECT ON public.payment FROM staff\_user;  Creating roles and password:    Using GRANT for admin\_user:    Testing GRANT for admin\_user:  Creating connection –    Viewing booking table as admin\_user –    Using GRANT for staff\_usr:    Testing GRANT for staff staff\_user:  Creating connection –    Viewing customer table as staff\_user –    Using REVOKE for staff\_user:    Tetsing REVOKE for staff\_user:    **TCL**  -- starting transaction  BEGIN;  -- inserting data values  INSERT INTO customer (customer\_id, customer\_name, customer\_email, customer\_phone) VALUES  (2001, 'John Doe', 'john@example.com', '1234567890'),  (2002, 'Alice Smith', 'alice@example.com', '9876543210'),  (2003, 'Bob Johnson', 'bob@example.com', '4561237890');    -- creating savepoint  SAVEPOINT before\_payment;    -- commit to save changes  COMMIT;  -- intentional error to check rollback  INSERT INTO customer (customer\_id, customer\_name, customer\_email, customer\_phone) VALUES  (2000, 'John Doe', 'john@example.com', '1234567890'),  (2000, 'Alice Smith', 'alice@example.com', '9876543210');  COMMIT;  -- using rollback to go back before the error  ROLLBACK;  -- end transaction  END;  SELECT \* FROM customer  Starting a transaction:    Inserting values in customer table:    Creating SAVEPOINT:    Using COMMIT to save changes:    Intentional error:    Commit:    Using ROLLBACK:    Ending transaction:    Updated customer table: |

|  |
| --- |
| **Post Lab Subjective / Objective Type Questions:** |
| **Discuss ACID properties of transaction with suitable examples.**  ACID (Atomicity, Consistency, Isolation, Durability) properties are essential characteristics that ensure the reliability and integrity of transactions in a database system. Here's a discussion of each property along with suitable examples:   1. **Atomicity:**   Atomicity ensures that a transaction is treated as a single unit of operation, either all of its operations are completed successfully, or none are performed at all. In other words, a transaction is either fully completed or fully aborted.  Example: Consider a bank transfer where money is being moved from one account to another. Atomicity ensures that if the debit operation (withdrawal from one account) is successful, the credit operation (deposit into another account) will also be successful. If either operation fails, the entire transaction will be rolled back to ensure data consistency.   1. **Consistency:**   Consistency ensures that the database remains in a consistent state before and after the execution of a transaction. It enforces integrity constraints, domain constraints, and referential integrity to maintain the correctness of data.  Example: Suppose a database maintains a constraint that every order must have a valid customer associated with it. If a transaction attempts to insert an order without specifying a customer or with a non-existent customer ID, consistency ensures that the transaction is aborted, and the database state remains consistent.   1. **Isolation:**   Isolation ensures that the concurrent execution of transactions does not result in interference or concurrency-related problems such as dirty reads, non-repeatable reads, and phantom reads. Each transaction should appear to execute in isolation, without being affected by other concurrently executing transactions.  Example: Consider two transactions T1 and T2 operating on the same bank account. If T1 is transferring funds from the account while T2 is querying the account balance, isolation ensures that T2 sees a consistent state of the account unaffected by the changes made by T1 until T1 commits its changes.   1. **Durability**:   Durability guarantees that once a transaction has been committed, its effects persist even in the event of system failure. The changes made by committed transactions are permanently stored and cannot be lost.  Example: After a successful funds transfer transaction in a banking system, the changes made to the account balances are durable. Even if the system crashes immediately after the transaction commits, the updated balances will be preserved in the database and will not be lost. |

|  |
| --- |
| **Conclusion:** |
| The experiment demonstrated the significance of Transaction Control Language (TCL) and Data Control Language (DCL) in ensuring transactional integrity and data security within the DBMS, highlighting their crucial roles in maintaining ACID properties and enforcing access control. |

**Signature of faculty in-charge with Date:**