Birla Institute of Technology and Science, Pilani

CS F212 Database Systems

Lab No # 6

****

1 Transaction

Databases are all about sharing data, so it is common for multiple users to be accessing and even changing the same data at the same time. The simultaneous execution of operations is called concurrency. Sometimes concurrency can get us into trouble if our changes require multiple SQL statements. In general, if two or more users access the same data and one or more of the statements changes the data, we have a conflict. This is a classic problem in database systems; it is called the isolation or serializability problem. If the users perform multiple steps, conflicts can cause incorrect results to occur. To deal with this problem, databases allow the grouping of a sequence of SQL statements into an indivisible unit of work called a transaction. A transaction ends with either a commit or a rollback:

Commit—A commit permanently stores all the changes performed by the transaction.

Rollback—A rollback removes all the updates performed by the transaction, no matter how many rows have been changed. A rollback can be executed either by the DBMS to prevent incorrect actions or explicitly by the user.

The DBMS provides the following guarantees for a transaction, called the ACID properties: Atomicity, consistency, Isolation, and durability. These properties will be covered in the course in detail.

SQL starts a transaction automatically when a new statement is executed if there is no currently active transaction. This means that a new transaction begins automatically with the first statement after the end of the previous transaction or the beginning of the session.

MySQL Transaction:

```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]

transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
}
```

```
BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]

SET autocommit = {0 | 1}
```

Start Transaction

- MySQL provides a START TRANSACTION statement to begin the transaction. It also offers a "BEGIN" and "BEGIN WORK" as an alias of the START TRANSACTION.

Commit

- We will use a COMMIT statement to commit the current transaction. It allows the database to make changes permanently.

Rollback

- We will use a ROLLBACK statement to roll back the current transaction. It allows the database to cancel all changes and goes into their previous state.

Try following set of SQL query and observe the results.

```
START TRANSACTION;
SELECT * FROM `restaurant`.`items`;
update items
set price = price *2
where items.itemid = 'CHKSD';
SELECT * FROM `restaurant`.`items`;
ROLLBACK;
SELECT * FROM `restaurant`.`items`;


START TRANSACTION;
SELECT * FROM `restaurant`.`items`;
update items
set price = price *2
where items.itemid = 'CHKSD';
SELECT * FROM `restaurant`.`items`;
commit;
rollback;
SELECT * FROM `restaurant`.`items`;


START TRANSACTION;
SELECT * FROM `restaurant`.`items`;
update items
set price = price *0.5
where items.itemid = 'CHKSD';
```

```
SELECT * FROM `restaurant`.`items`;
ROLLBACK;
SELECT * FROM `restaurant`.`items`;
update items
set price = price *2
where items.itemid = 'CHKSD';
SELECT * FROM `restaurant`.`items`;
ROLLBACK;
SELECT * FROM `restaurant`.`items`;
```

Any transaction started by “START TRANSACTION” is ended with the “COMMIT” or a “ROLLBACK” statement.

Auto Commit: We will use a SET auto-commit statement to disable/enable the auto-commit mode for the current transaction. By default, the COMMIT statement executed automatically. So, if we do not want to commit changes automatically, use the below statement:

```
SET autocommit = 0;
OR,
SET autocommit = OFF:
```

Again, use the below statement to enable auto-commit mode:

```
SET autocommit = 1;
OR,
SET autocommit = ON:
```

2 Save Point

InnoDB supports the SQL statements SAVEPOINT, ROLLBACK TO SAVEPOINT, RELEASE SAVEPOINT and the optional WORK keyword for ROLLBACK.

```
SAVEPOINT identifier
ROLLBACK [WORK] TO [SAVEPOINT] identifier
RELEASE SAVEPOINT identifier
```

The SAVEPOINT statement sets a named transaction savepoint with a name of identifier. If the current transaction has a savepoint with the same name, the old savepoint is deleted and a new one is set.

2.1 Rollback to Save Point

The ROLLBACK TO SAVEPOINT statement rolls back a transaction to the named savepoint without terminating the transaction. Modifications that the current transaction made to rows after the savepoint was set are undone in the rollback, but InnoDB does not release the row locks that were stored in memory after the savepoint. (For a new inserted row, the lock information is carried by the transaction ID stored in the row; the lock is not separately stored in memory. In this case, the row lock is released in the undo.) Savepoint that were set at a later time than the named savepoint are deleted.

If the ROLLBACK TO SAVEPOINT statement returns the following error, it means that no savepoint with the specified name exists:

```
ERROR 1305 (42000): SAVEPOINT identifier does not exist
```

2.2 Release Save Point

- ☞ The RELEASE SAVEPOINT statement removes the named savepoint from the set of savepoints of the current transaction. No commit or rollback occurs. It is an error if the savepoint does not exist.
- ☞ All savepoints of the current transaction are deleted if you execute a COMMIT, or a ROLLBACK that does not name a savepoint.
- ☞ A new savepoint level is created when a stored function is invoked, or a trigger is activated. The savepoints on previous levels become unavailable and thus do not conflict with savepoints on the new level. When the function or trigger terminates, any savepoints it created are released and the previous savepoint level is restored.

Try following set of SQL queries and observe the results.

```
START TRANSACTION;
SELECT * FROM `restaurant`.`items`;
update items
set price = price *0.5
where items.itemid = 'CHKSD';
savepoint point1;
SELECT * FROM `restaurant`.`items`;
update items
set price = price *2
where items.itemid = 'CHKSD';
SELECT * FROM `restaurant`.`items`;
savepoint point2;
SELECT * FROM `restaurant`.`items`;
rollback to savepoint point1;
SELECT * FROM `restaurant`.`items`;
commit;
SELECT * FROM `restaurant`.`items`;
rollback to savepoint point1;
SELECT * FROM `restaurant`.`items`;
```

- ☞ When you execute second last query; it gives error, because after commit all savepoints are released.

3 Statements That Cause an Implicit Commit

Some statements cannot be rolled back. In general, these include data definition language (DDL) statements, such as those that create or drop databases, those that create, drop, or alter tables or stored routines.

You should design your transactions not to include such statements. If you issue a statement early in a transaction that cannot be rolled back, and then another statement later fails, the full effect of the transaction cannot be rolled back in such cases by issuing a ROLLBACK statement.

3.1 Data definition language (DDL) statements that define or modify database objects

Statements use keywords like ALTER, CREATE, DROP, INSTALL, RENAME, TRUNCATE, and UNINSTALL use implicit commit.

CREATE TABLE and DROP TABLE statements do not commit a transaction if the TEMPORARY keyword is used. (This does not apply to other operations on temporary tables such as ALTER TABLE and CREATE INDEX, which do cause a commit.) However, although no implicit commit occurs, neither can the statement be rolled back, which means that the use of such statements causes transactional atomicity to be violated

3.2 Statements that implicitly use or modify tables in the MySQL database

Statements use keywords like ALTER USER, CREATE USER, DROP USER, GRANT, RENAME USER, REVOKE, SET PASSWORD use implicit commit.

3.3 Transaction-control and locking statements

- BEGIN, LOCK TABLES, SET autocommit = 1 (if the value is not already 1), START TRANSACTION, UNLOCK TABLES.
- UNLOCK TABLES commits a transaction only if any tables currently have been locked with LOCK TABLES to acquire non-transactional table locks. A commit does not occur for UNLOCK TABLES following FLUSH TABLES WITH READ LOCK because the latter statement does not acquire table-level locks.
- Transactions cannot be nested. This is a consequence of the implicit commit performed for any current transaction when you issue a START TRANSACTION statement or one of its synonyms.

3.4 Data loading statements

- LOAD DATA. LOAD DATA causes an implicit commit only for tables using the NDB storage engine.

3.5 Administrative statements

- ☞ ANALYZE TABLE, CACHE INDEX, CHECK TABLE, FLUSH, LOAD INDEX INTO CACHE, OPTIMIZE TABLE, REPAIR TABLE, RESET (but not RESET PERSIST).

3.6 Replication control statements

- ☞ START REPLICA | SLAVE, STOP REPLICA | SLAVE, RESET REPLICA | SLAVE, CHANGE REPLICATION SOURCE TO, CHANGE MASTER TO.

**********************************END***************************************