Transactions

Introduction on Transactions

- A transaction is a logical unit of work that contains one or more SQL statements.
- **Transactions are atomic units of work that can be committed or rolled back**.
- When a transaction makes multiple changes to the database, either all the changes succeed when the transaction is committed, or all the changes are undone when the transaction is rolled back.

- A transaction begins with the first executable SQL statement.
- A transaction ends when it is committed or rolled back,
  - either explicitly with a COMMIT or ROLLBACK statement
  - or implicitly when a DDL (Data Definition Language (DDL) is used to manage table and index structure and CREATE, ALTER, RENAME, DROP and TRUNCATE statements are to name a few data definition elements) statement is issued.

Example

- Consider a banking database.
- Suppose a bank customer transfers money from his savings account (SB a/c) to his current account (CA a/c), the statement will be divided into four blocks :
  - Debit SB a/c.
  - Credit OD a/c.
  - Record in Transaction Journal
  - End Transaction

- The SQL statement to debit SB a/c is as follows:
- UPDATE sb_accounts SET balance = balance – 1000 WHERE account_no = 932656 ;

- The SQL statement to credit OD a/c is as follows:
- UPDATE od_accountsSET balance = balance + 1000 WHERE account_no = 933456 ;

- The SQL statement for record in transaction journal is as follows:
- INSERT INTO journal VALUES (100896, 'Tansaction on Benjamin Hampshair a/c', '26-AUG-08', 932656, 933456, 1000);

- The SQL statement for End Transaction is as follows :
- COMMIT WORK;

Auto commit

- To force MySQL not to commit changes automatically, you can use the following statement:

SET autocommit = 0;

MySQL and the ACID Model

- **ACID (Atomicity, Consistency, Isolation, Durability)** is a set of properties that guarantee that database transactions are processed reliably.

- In MySQL, InnoDB storage engine supports ACID-compliant features.

Atomicity

- The atomicity aspect of the ACID model mainly involves InnoDB transactions.

- Related MySQL features include :
  - Autocommit setting.
  - COMMIT statement.
  - ROLLBACK statement.
  - Operational data from the INFORMATION_SCHEMA tables.

Consistency

- The consistency aspect of the ACID model mainly involves internal InnoDB processing to protect data from crashes.

- Related MySQL features include :
  - InnoDB doublewrite buffer.
  - InnoDB crash recovery.

Isolation

- The isolation aspect of the ACID model mainly involves InnoDB transactions, in particular, the isolation level that applies to each transaction.

- Related MySQL features include :
  - Autocommit setting.
  - SET ISOLATION LEVEL statement.
  - The low-level details of InnoDB locking. During performance tuning, you see these details through INFORMATION_SCHEMA tables.

- **Durability:**
- The durability aspect of the ACID model involves MySQL software features interacting with your particular hardware configuration.
- Because of the many possibilities depending on the capabilities of your CPU, network, and storage devices, this aspect is the most complicated to provide concrete guidelines for.
- Related MySQL features include:
  - InnoDB doublewrite buffer turned on and off by the innodb_doublewrite configuration option.
  - Write buffer in a storage device, such as a disk drive, SSD, or RAID array.
  - Battery-backed cache in a storage device.
  - The operating system used to run MySQL, in particular, its support for the fsync() system call.
  - Uninterruptible power supply (UPS) protecting the electrical power to all computer servers and storage devices that run MySQL servers and store MySQL data.
  - Your backup strategy, such as frequency and types of backups, and backup retention periods.

MySQL Transaction

- MySQL supports local transactions (within a given client session) through statements such as SET autocommit, START TRANSACTION, COMMIT, and ROLLBACK.
- Syntax of START TRANSACTION, COMMIT, and ROLLBACK:

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}

- These statements provide control over use of transactions :
- The **START TRANSACTION or BEGIN** statement begins a new transaction.
- **COMMIT** commits the current transaction, making its changes permanent.
- **ROLLBACK** rolls back the current transaction, canceling its changes.
- The **SETautocommit** statement disables or enables the default autocommit mode for the current session.
- By default, MySQL runs with autocommit mode enabled. This means that as soon as you execute a statement that updates (modifies) a table, MySQL stores the update on disk to make it permanent. The change cannot be rolled back.
- Roll back is not possible if MySQL runs with autocommit mode enabled.
- To disable autocommit mode, use the START TRANSACTION statement.

Example

```
SELECT * FROM STUD_MARKS;

UPDATE stud_marks set NAME="bobby" WHERE STUDENT_ID=2;

SELECT * FROM STUD_MARKS;

ROLLBACK;

SELECT * FROM STUD_MARKS;
```

Example

```
START TRANSACTION;

SELECT * FROM STUD_MARKS;

UPDATE stud_marks set NAME="bob" WHERE STUDENT_ID=2;

SELECT * FROM STUD_MARKS;

ROLLBACK;

SELECT * FROM STUD_MARKS;
```

Rollback constraints

- In MySQL, some statements **cannot be rolled back**.
- DDL statements such as CREATE or DROP databases, CREATE, ALTER or DROP tables or stored routines.
- You should design a transaction without these statements.

- The statements listed in this section (and any synonyms for them) implicitly end any transaction active in the current session, as if you had done a COMMIT before executing the statement.
  - Data definition language (DDL) statements that define or modify database objects. ALTER DATABASE ... UPGRADE DATA DIRECTORY NAME, ALTER EVENT, ALTER PROCEDURE, ALTER SERVER, ALTER TABLE, ALTER VIEW, CREATE DATABASE, CREATE EVENT, CREATE INDEX, CREATE PROCEDURE, CREATE SERVER, CREATE TABLE, CREATE TRIGGER, CREATE VIEW, DROP DATABASE, DROP EVENT, DROP INDEX, DROP PROCEDURE, DROP SERVER, DROP TABLE, DROP TRIGGER, DROP VIEW, RENAME TABLE, TRUNCATE TABLE.
  - ALTER FUNCTION, CREATE FUNCTION, and DROP FUNCTION also cause an implicit commit when used with stored functions, but not with UDFs. (ALTER FUNCTION can only be used with stored functions.)

- ALTER TABLE, CREATE TABLE, and DROP TABLE do not commit a transaction if the **TEMPORARY keyword** is used.
- Statements that implicitly use or modify tables in the MySQL database. CREATE USER, DROP USER, GRANT, RENAME USER, REVOKE, SET PASSWORD.
- Transaction-control and locking statements. BEGIN, LOCK TABLES, SET autocommit = 1 (if the value is not already 1), START TRANSACTION, UNLOCK TABLES.
- Data loading statements. LOAD DATA INFILE. LOAD DATA INFILE causes an implicit commit only for tables using the NDB storage engine.
- Administrative statements. ANALYZE TABLE, CACHE INDEX, CHECK TABLE, LOAD INDEX INTO CACHE, OPTIMIZE TABLE, REPAIR TABLE.
- Replication control statements. Beginning with MySQL 5.6.7: START SLAVE, STOP SLAVE, RESET SLAVE, CHANGE MASTER TO.

SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT

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

- 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.
- Syntax:

SAVEPOINT identifier

ROLLBACK [WORK] TO [SAVEPOINT] identifier

RELEASE SAVEPOINT identifier

LOCK and UNLOCK Tables

- MySQL enables client sessions to acquire table locks explicitly for the purpose of cooperating with other sessions for access to tables or to prevent other sessions from modifying tables during periods when a session requires exclusive access to them.
- A session can acquire or release locks only for itself.
- One session cannot acquire locks for another session or release locks held by another session.
- LOCK TABLES explicitly acquires table locks for the current client session.
- Table locks can be acquired for base tables or views.
- You must have the LOCK TABLES privilege, and the SELECT privilege for each object to be locked.
- UNLOCK TABLES explicitly releases any table locks held by the current session.
- LOCK TABLES implicitly releases any table locks held by the current session before acquiring new locks.

- Syntax:

LOCK TABLES *tbl_name* [[AS] *alias*] *lock_type*

[, *tbl_name* [[AS] *alias*] *lock_type*] ...

***lock_type*:**

READ [LOCAL]

| [LOW_PRIORITY] WRITE

UNLOCK TABLES

SET TRANSACTION Syntax

```
SET [GLOBAL | SESSION] TRANSACTION
transaction_characteristic [, transaction_characteristic] ...

transaction_characteristic:
        ISOLATION LEVEL level
        | READ WRITE
        | READ ONLY

level:
        REPEATABLE READ
        | READ COMMITTED
        | READ UNCOMMITTED
        | SERIALIZABLE
```

- With the GLOBAL keyword, the statement applies globally for all subsequent sessions. Existing sessions are unaffected.

- With the SESSION keyword, the statement applies to all subsequent transactions performed within the current session.

- Without any SESSION or GLOBAL keyword, the statement applies to the next (not started) transaction performed within the current session.

Example

- Start a transaction using START TRANSACTION statement.
- Get latest sales order number from the orders table, and use the next sales order number as the new sales order number.
- Insert a new sales order into the orders table for a given customer.
- Insert new sales order items into the orderdetails table.
- Commit changes using COMMIT statement.
- Get data from both table orders and orderdetails tables to confirm the changes.

```
START TRANSACTION;

SELECT @orderNumber := max(orderNumber)  FROM TR_ORDERS;

SET @orderNumber = @orderNumber+1;

SELECT @orderNumber;

INSERT INTO
tr_orders(orderNumber,orderDate,requiredDate,shippedDate,status,
customerNumber) VALUES(@orderNumber, now(), date_add(now(),
INTERVAL 5 DAY), date_add(now(), INTERVAL 2 DAY),'In
Process',145);

INSERT INTO tr_orderdetails(orderNumber,
productCode,quantityOrdered,priceEach,orderLineNumber)
values(@orderNumber,'S18_1749', 30, '136',
1),(@orderNumber,'S18_2248', 50, '55.09', 2);

COMMIT;

SELECT * FROM tr_orders a inner join tr_orderdetails b on
a.ordernumber = b.ordernumber where a.ordernumber =
@ordernumber;
```

References

- https://www.w3resource.com/mysql/mysql-transaction.php
- http://www.mysqltutorial.org/mysql-transaction.aspx