8

Transactions and Locking

Objectives

After completing this lesson, you should be able to:

- Use transaction control statements to run multiple SQL statements concurrently
- Explain the ACID properties
- Describe the transaction isolation levels
- Use locking to protect transactions

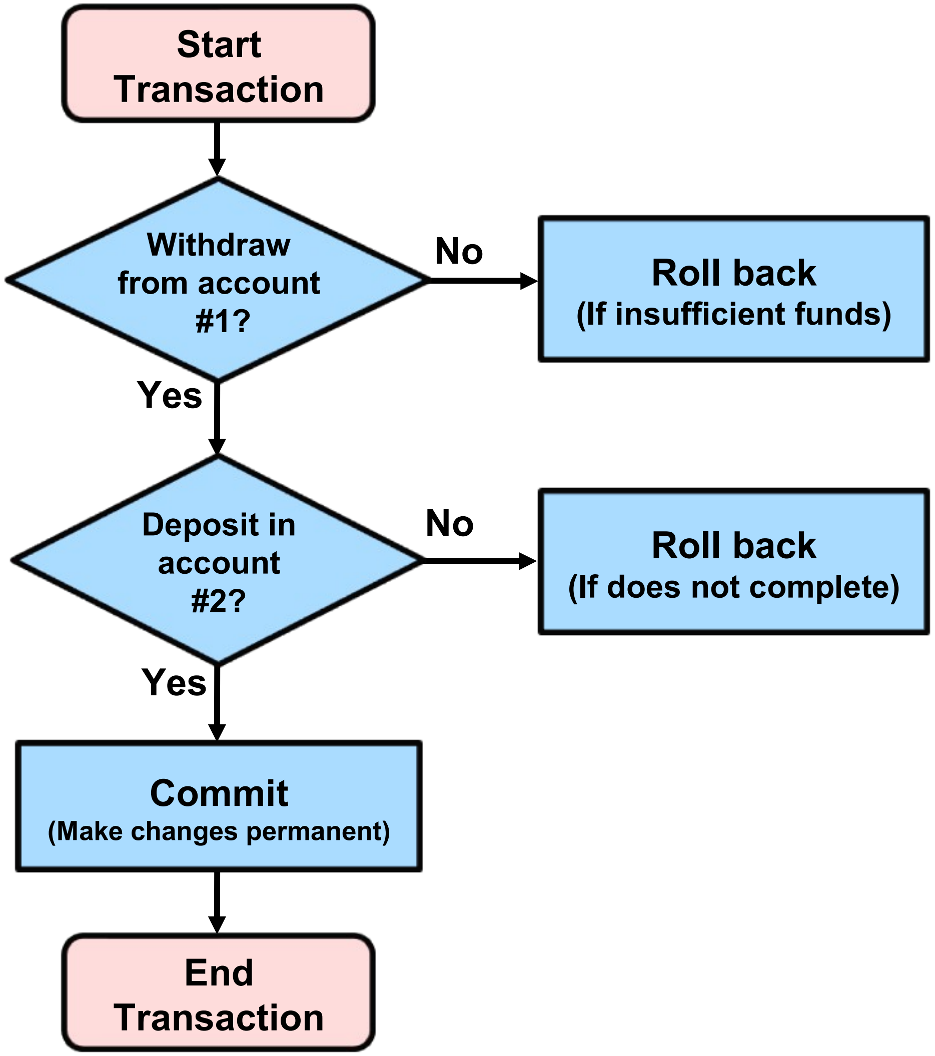
Transactions

- A collection of data manipulation execution steps that are treated as a single unit of work
  - `Use to group multiple statements`
  - `Use when multiple clients are accessing data from the same table concurrently`
- All or none of the steps succeed
  - `Execute if all steps are good`
  - `Cancel if steps have error or are incomplete`
- **ACID** compliant

Transaction Diagram

Example:

A banking transaction


Start Transaction
Withdraw from account #1?
No
Roll back (If insufficient funds)
Yes
Deposit in account #2?
No
Roll back (If does not complete)
Yes
Commit (Make changes permanent)
End Transaction

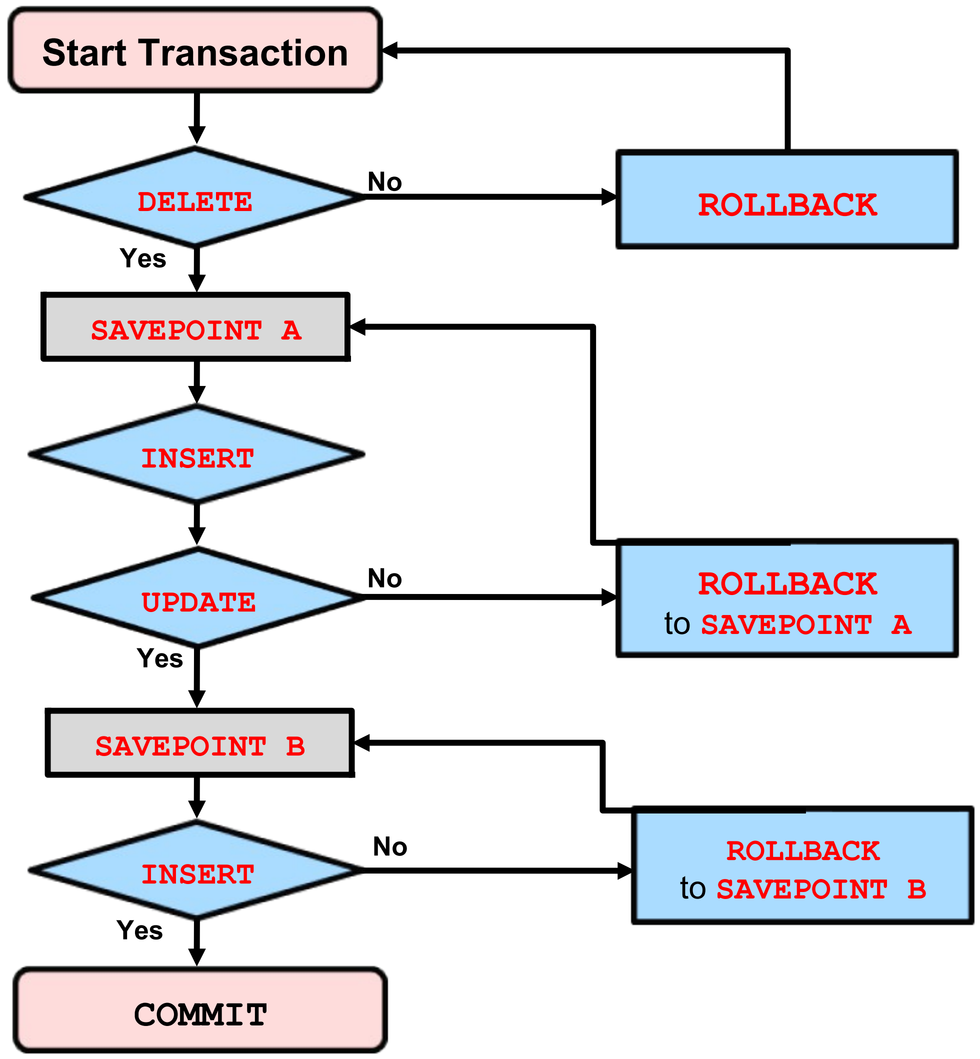
ACID

- Atomic
  - All statements execute successfully or are canceled as a unit
- Consistent
  - Database that is in a consistent state when a transaction begins, is left in a consistent state by the transaction
- Isolated
  - One transaction does not affect another
- Durable
  - All changes made by transaction that complete successfully are recorded properly in database
    - Changes are not lost

Transaction SQL Control Statements

- `START TRANSACTION` (or `BEGIN`): Explicitly begins a new transaction
- `SAVEPOINT`: Assigns a location in the process of a transaction for future reference
- `COMMIT`: Makes the changes from the current transaction permanent
- `ROLLBACK`: Cancels the changes from the current transaction
- `ROLLBACK TO SAVEPOINT`: Cancels the changes executed after the savepoint
- `RELEASE SAVEPOINT`: Removes the savepoint identifier
- `SET AUTOCOMMIT`: Disables or enables the default autocommit mode for the current connection

SQL Control Statements Flow: Example



AUTOCOMMIT Mode

- Determine how and when new transactions are started.
- `AUTOCOMMIT` mode enabled by default:
  - Implicitly commits each statement as a transaction
- Disable **AUTOCOMMIT** mode:

```
SET AUTOCOMMIT=0;
SET SESSION AUTOCOMMIT=0;
SET @@AUTOCOMMIT :=0;
```

  - When AUTOCOMMIT is disabled, transactions span multiple statements by default.
  - You can end a transaction with COMMIT or ROLLBACK.
- Check the **AUTOCOMMIT** setting with **SELECT**:

```
SELECT @@AUTOCOMMIT;
```

Implicit Commit

- Implicit commit terminates current transaction.
- SQL statements that implicitly commit:
  - `START TRANSACTION`
  - `SET AUTOCOMMIT = 1`
- Nontransactional statements that cause a commit:
  - `Data definition statements (ALTER, CREATE, DROP)`
  - `Administrative statements (GRANT, REVOKE, SET PASSWORD)`
  - `Locking statements (LOCK TABLES, UNLOCK TABLES)`
- Example of statements that cause an implicit commit:
  - `TRUNCATE TABLE`
  - `LOAD DATA INFILE`

Transactional Storage Engines

List the engine characteristics with **SHOW ENGINES**:

```
mysql> SHOW ENGINES\G
********************* 2. row **********************
      Engine: InnoDB
     Support: DEFAULT
     Comment: Default engine as of MySQL 5.5, Supports
   transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
********************* 1. row **********************
      Engine: MyISAM
     Support: YES
     Comment: Great Performance ...
Transactions: NO
          XA: NO
  Savepoints: NO
...
```

Transaction Isolation Problems

Three common problems:

- “Dirty” read
  - When a transaction reads the changes made by another uncommitted transaction
- Non-repeatable read
  - When another transaction commits changes causing the read operation to be non-repeatable
- Phantom read (or phantom row)
  - A row that appears but was not previously visible within the same transaction

A non-repeatable read occurs when a transaction re-reads data it has previously read and finds that data has been modified by another transaction.

Isolation Levels

Four isolation levels:

- `READ UNCOMMITTED`
  - Allows a transaction to see uncommitted changes made by other transactions
- `READ COMMITTED`
  - Allows a transaction to see committed changes made by other transactions
- `REPEATABLE READ`
  - Ensures consistent `SELECT` output for each transaction, regardless of committed or uncommitted changes
  - Default level for InnoDB
- `SERIALIZABLE`
  - Completely isolates the effects of a transaction from others

Isolation Level Problems

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read Uncommitted	Possible	Possible	Possible
Read Committed	Not possible	Possible	Possible
Repeatable Read	Not possible	Not possible	Possible*
Serializable	Not possible	Not possible	Not possible

* Not possible for InnoDB

Setting the Isolation Level

- Set the level at server startup.
  - Use the --transaction-isolation option with the mysqld command.
  - Or set transaction-isolation in the configuration file:

```
[mysqld]
transaction-isolation = <isolation_level>
```

- Set for a running server by using a SET TRANSACTION ISOLATION LEVEL statement.
  - Syntax examples:

```
SET GLOBAL TRANSACTION ISOLATION LEVEL <isolation_level>;
SET SESSION TRANSACTION ISOLATION LEVEL <isolation_level>;
SET TRANSACTION ISOLATION LEVEL <isolation_level>;
```

Global Isolation Level

tx_isolation: Default transaction isolation level. Removed in MySQL 8.0.3.

Use: @@transaction_isolation;

Requires the **SUPER** privilege:

```
mysql> SELECT @@tx_isolation;
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |
+-----------------+

mysql> SELECT @@global.tx_isolation,
              @@session.tx_isolation;
+-----------------------+------------------------+
| @@global.tx_isolation | @@session.tx_isolation |
+-----------------------+------------------------+
| READ-UNCOMMITTED      | REPEATABLE-READ        |
+-----------------------+------------------------+
```

Transaction Example: Isolation

Sesion 1	Sessin 2
`mysql> PROMPT s1>`	
`s1> SET SESSION TRANSACTION`	
` -> ISOLATION LEVEL READ COMMITTED;`	
`s1> SELECT @@global.tx_isolation;`	
`+------------------------+`	
`| @@global.tx_isolation |`	
`+------------------------+`	
`| READ-COMMITTED         |`	
`+------------------------+`	
	`mysql> PROMPT s2>`
	`s2> START TRANSACTION;`
	`s2> INSERT INTO City`
	` -> (Name, CountryCode,Population)`
	` -> VALUES ('Sakila', 'SWE', 1);`
`s1> SELECT Name, CountryCode`	
` -> FROM City`	
` -> WHERE Name = 'Sakila';`	
`Empty Set (0.0 sec)`	

Transaction Example: Isolation

Sesion 1	Session 2
	`s2> COMMIT;`

`s1> SELECT Name, CountryCode`
` -> FROM City`
` -> WHERE Name = 'Sakila';`
`+---------+-------------+`
`| Name    | CountryCode |`
`+---------+-------------+`
`| Sakila | SWE         |`
`+---------+-------------+`	

Locking Concepts

- MySQL uses a multi-threaded architecture.
  - `Problems arise with multiple client access to a table.`
  - `Client coordination is necessary.`
- Locking is a mechanism to prevent concurrency problems.
  - `Managed by server`
  - `Locks for one client, to restrict others`
- Types of locks:
  - `Shared lock`
  - `Exclusive lock`

Explicit Row Locks

InnoDB supports two types of row locking:

- **LOCK IN SHARE MODE**
  - Locks each row with a shared lock

```
SELECT * FROM Country WHERE Code='AUS'
  LOCK IN SHARE MODE\G
```

- **FOR UPDATE**
  - Locks each row with an exclusive lock

```
SELECT counter_field INTO @@counter_field
  FROM child_codes FOR UPDATE;

UPDATE child_codes SET counter_field = @@counter_field
  + 1;
```

Deadlocks

- When multiple transactions each require data that the other has already locked exclusively
- InnoDB detects and aborts (rollback) one of the transactions and allows the other one to complete.
- To reduce deadlocks:
  - `Use small transactions (low number of rows inserted, updated, or deleted) to avoid rolling back too many changes`
  - `Use SHOW ENGINE INNODB STATUS to find cause`
  - `Be prepared to re-issue a transaction if it deadlocks`
  - `Commit your transactions often`
  - `Access your tables and rows in a fixed order`
  - `Add well-chosen indexes to your tables`
  - `Use less locking and lower isolation level, such as READ COMMITTED`

Transaction Example: Deadlock

Session 1	Session 2
s1> START TRANSACTION; s1> UPDATE Country -> SET Name = 'Sakila' -> WHERE Code = 'SWE';	
	s2> START TRANSACTION; s2> UPDATE Country -> SET Name = 'World Cup Winner' -> WHERE Code = 'ITA';
s1> DELETE FROM Country -> WHERE Code = 'ITA';	
	s2> UPDATE Country -> SET population=1 -> WHERE Code = 'SWE'; ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
Query OK, 1 row affected (0.0 sec)	

Implicit Locks

The MySQL server locks the table (or row) based on the commands issued and the storage engines being used:

Operation	InnoDB	MyISAM
`SELECT`	No lock*	Table-level shared lock
`UPDATE/DELETE`	Row-level exclusive lock	Table-level exclusive lock
`ALTER TABLE`	Table-level shared lock	Table-level shared lock

* No lock unless **`SERIALIZABLE`** level, **`LOCK IN SHARE MODE`**, or **`FOR UPDATE`** is used

Summary

In this lesson, you should have learned how to:

- Use transaction control statements to run multiple SQL statements concurrently
- Explain the ACID properties
- Describe the transaction isolation levels
- Use locking to protect transactions