Data Modelling 4:

Transactions Overview

Definition:
A transaction in SQL is a sequence of one or more SQL statements executed as a single unit. The key concept is that all statements in a transaction are treated as a whole—either they all succeed (commit) or they all fail (rollback). This ensures data integrity and consistency.

Transaction Access Modes

1. READ-ONLY:

- **Definition**: Restricts the transaction to perform only SELECT statements. No data modifications are allowed (INSERT, UPDATE, DELETE).

- **Purpose**: Ensures data is not accidentally modified, allowing unlimited reads.

- **Use Case**: Ideal for reporting, data analytics, or scenarios where no data alteration is needed.

  SQL:

  ```
  START TRANSACTION READ ONLY;

  -- Fetch some data (this is allowed in READ ONLY mode)

  SELECT * FROM employees;

  INSERT INTO employees (employee_id, employee_name, department_id,
  manager_id, salary) VALUES (1, 'John Doe', 101, 10, 50000);

  COMMIT;
  ```

2. READ-WRITE:

- **Definition**: Allows the transaction to perform both read (SELECT) and write (INSERT, UPDATE, DELETE) operations. This is the default mode.

- **Purpose**: Enables data modifications within the transaction.

- **Use Case**: Suitable for typical database operations where data changes are necessary.

SQL:

```
START TRANSACTION READ WRITE;

-- Fetch data from the table (allowed in READ-WRITE mode)

SELECT * FROM employees;

INSERT INTO employees (employee_id, employee_name, department_id,
manager_id, salary) VALUES (12, 'John Doe', 101, 10, 50000);

UPDATE employees SET salary = 55000 WHERE employee_id = 1;

COMMIT;
```

Transaction Isolation Levels

1. Read Uncommitted:

- **Scenario**: Transaction 1 modifies data, and Transaction 2 reads the uncommitted changes before Transaction 1 commits or rolls back.
- **Outcome**: Transaction 2 may see "dirty data" that could be rolled back, leading to inconsistencies.

SQL:

```
TRANSACTION 1:
SELECT * FROM accounts;

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

START TRANSACTION;

UPDATE accounts SET balance = balance - 500 WHERE user_id = 1;

-- Start Transaction 2

SELECT * FROM accounts;

ROLLBACK;
```

```
TRANSACTION 2:

SELECT * FROM accounts;

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

START TRANSACTION;

SELECT * FROM accounts; //uncommitted changes are getting reflected here

-- End Transaction 1

COMMIT;

SELECT * FROM accounts; //committed changes are getting reflected here as transaction1 is executed
```

2. Read Committed

- **Scenario**: Transaction 1 modifies data but has not yet committed. Transaction 2 can only see data committed by other transactions.
- **Outcome**: Prevents dirty reads. Transaction 2 reads only committed data, ensuring data consistency.

SQL:

```
TRANSACTION 1:
SELECT * FROM accounts;

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

START TRANSACTION;

UPDATE accounts SET balance = balance - 500 WHERE user_id = 1;

-- Start Transaction 2

SELECT * FROM accounts;

COMMIT;


TRANSACTION 2:

SELECT * FROM accounts;

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

```
START TRANSACTION;

SELECT * FROM accounts; //only committed changes are getting reflected here

-- End Transaction 1

COMMIT;

SELECT * FROM accounts; //committed changes are getting reflected
```

3. Repeatable Read

- **Scenario**: Transaction 1 reads a record, then Transaction 2 modifies the same record. Transaction 1 reads the record again and sees the initial value, even if Transaction 2 committed changes.

- **Outcome**: Prevents non-repeatable reads. Transaction 1 will continue to see the old value, preserving read consistency.

  SQL:

```
// Only Committed values are being read
TRANSACTION 1:
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

UPDATE accounts SET balance = balance + 500 WHERE user_id = 1;

SELECT * FROM accounts;

-- Start Transaction 2

COMMIT;


TRANSACTION 2:

-- Reads the original value since Transaction 1 hasn't committed yet

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

SELECT balance FROM accounts WHERE user_id = 1;

COMMIT;
```

// Once read, the snapshot is being maintained through out the transaction

TRANSACTION 1.1:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

-- Start Transaction 2.1

UPDATE accounts SET balance = balance + 500 WHERE user_id = 1;

SELECT * FROM accounts;

COMMIT;
```

TRANSACTION 2.1:

```
-- Reads the original value from DB and retains it through out the Transaction

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

SELECT balance FROM accounts WHERE user_id = 1; //read the original DB value

-- Execute Transaction 1.1

SELECT balance FROM accounts WHERE user_id = 1; //still showing original value although transaction 1.1 has committed the new value in DB

COMMIT;
```

// Blocking Phantom Reads in Repeatable Read mode

TRANSACTION 1.2:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

SELECT * FROM accounts WHERE department_id = 101;

-- do one more transaction which inserts some new data in this table with department_id 101

-- Start Transaction 2.2

SELECT * FROM accounts WHERE department_id = 101; // data is getting reflected here

COMMIT;
```

TRANSACTION 2.2:

```
-- New insertion is gonna be made in this transaction, but the data will not be reflected in another already running txn.

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

INSERT INTO accounts (user_id, user_name, account_type, balance, department_id)

VALUES (7, 'Sachin Tendulkar', 'All Rounder', 30000, 101);

COMMIT;

-- check the data in already running Transaction 1.2
```

4. Serializable

- **Scenario**: Transaction 1 queries a set of rows, then Transaction 2 inserts a new row matching the query. Transaction 1 does not see the new row, preventing phantom reads.

- **Outcome**: Provides the strictest isolation, ensuring no phantom reads occur by completely isolating the transaction.

  SQL:

```
// Only Committed values are being read
TRANSACTION 1:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

START TRANSACTION;

SELECT * FROM accounts WHERE department_id = 101;

-- This query will lock all rows with department_id = 101

-- and also prevent any other inserts, updates, or deletes in this range.

-- Start Transaction 2

COMMIT;
```

TRANSACTION 2:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

START TRANSACTION;

INSERT INTO accounts (user_id, user_name, account_type, balance,
department_id)

VALUES (8, 'VVS LAXMAN', 'Batting', 20000, 101);

// its gonna wait as lock has been taken by transaction 1 which is not yet completed.

ROLLBACK;

SELECT * FROM accounts;
```

Rollbacks and Cascading Rollbacks

- **Rollback**: Reverts all changes made during a transaction, preserving database consistency if an error occurs. Rollbacks ensure the database adheres to the ACID principles.

- **Cascading Rollbacks**: Occur when the rollback of one transaction triggers rollbacks in other dependent transactions, maintaining overall consistency.

Autocommit Modes

1. Autocommit ON:

- **Definition**: Each SQL statement is a separate transaction that is immediately committed upon success.

- **Outcome**: If a statement fails, only the unsuccessful statement is rolled back, while prior successful statements remain committed.

SQL:

SESSION 1:

```
SET autocommit = 1;

INSERT INTO accounts (user_id, user_name, account_type, balance,
department_id) VALUES (15, 'Rinku Singh', 'Batsman', 25000, 106);

UPDATE accounts SET balance = 75000 WHERE user_id = 1;

-- Start Another session to read from accounts table

select * from accounts;
```

SESSION 2:

```
select * from accounts; // reads updated values as auto commit is on
```

2. Autocommit OFF:

- **Definition**: Transactions require explicit management with `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK` commands.
- **Outcome**: Ensures that if one statement fails, the entire transaction can be rolled back, preventing partial commits.

SQL:

SESSION 1:

```
SET autocommit = 0;

UPDATE accounts SET balance = 50000 WHERE user_id = 1;

-- Start Another session to read from accounts table

select * from accounts;

COMMIT;
```

SESSION 2:

```
select * from accounts; // reads older value as Session1’s update is not committed yet
```

NOTE: After setting auto commit, if we start the transaction its gonna over ride the auto commit conifg, as transaction starts with START TRANSACTION and ends with either COMMIT or ROLLBACK.

Transactional Logs in MySQL

MySQL uses three main types of logs to ensure data durability and consistency during transactions:

1. BinLog (Binary Log)

- Logs modifications from INSERT, UPDATE, and DELETE statements, but only for committed transactions.
- **Purpose**: Used for replication and point-in-time recovery, not for rollbacks.

2. Redo Log

- Stores changes in a redo log buffer before writing to disk.
- **Purpose**: Assists with crash recovery by applying committed changes not yet written to data files.

3. Undo Log

- Maintains "before" versions of data for rollback purposes.
- **Purpose**: Supports Multi-Version Concurrency Control (MVCC), allowing consistent snapshots for transactions.

Locking Mechanisms:

1. Pessimistic Locking

- **Definition**: Locks data immediately to prevent other transactions from reading or modifying it.

- **Implementation**:
  - `SELECT ... FOR UPDATE`: Exclusive lock, blocking both reads and writes from other transactions.
  - `LOCK IN SHARE MODE/ SELECT ... FOR SHARE`: Shared lock, allowing reads but blocking writes.
- **Pros**: Guarantees consistency by blocking conflicting operations.
- **Cons**: Can lead to performance issues and deadlocks in high-concurrency environments.

SQL:
SELECT… FOR UPDATE:

SESSION 1:

```
SELECT * FROM accounts WHERE user_id = 1;

-- Set isolation level to REPEATABLE READ for the transaction

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

-- Start Another session to do operation on this table.

SELECT * FROM accounts WHERE department_id = 101 LIMIT 1 FOR UPDATE;


COMMIT;
```

SESSION 2: as we are only reading the lock will not be applied

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

SELECT * FROM accounts WHERE department_id = 101;

COMMIT;
```

SESSION 2.1: lock will be applied as we are reding in Select.. For Update mode

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

SELECT * FROM accounts WHERE department_id = 101 limit 1 FOR UPDATE;

COMMIT;
```

SESSION 2.2: lock will be applied as we are updating the data

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;
```

```
UPDATE accounts SET balance = 50000 WHERE user_id = 1;

COMMIT;
```

SELECT… FOR SHARE:

SESSION 1:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

SELECT * FROM accounts WHERE department_id = 101 LIMIT 1 FOR SHARE;

-- Start Another session to do operation on this table.

COMMIT;
```

SESSION 2: as we are only reading the lock will not be applied

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

SELECT * FROM accounts WHERE department_id = 101;

COMMIT;
```

SESSION 2.1: lock will not be applied as we are reading in share mode.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

SELECT * FROM accounts WHERE department_id = 101 limit 1 FOR SHARE;

COMMIT;
```

SESSION 2.2: lock will be applied as we are updating the data

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;
```

```
UPDATE accounts SET balance = 50000 WHERE department_id = 101;

ROLLBACK;
```

2. Optimistic Locking:

- **Definition**: Assumes conflicts are rare, checking for conflicts only at commit time.

- **Process**:
  1. **Read**: The transaction reads data and its version/timestamp.
  2. **Update**: Checks for version changes before committing.
  3. **Commit**: Proceeds if no changes were detected; otherwise, retries.

- **Pros**: High concurrency and low blocking.

- **Cons**: Increased retries in conflict-heavy environments.

NOTE: SQL do not implement this feature, this feature gets implemented either in application layer or in query layer.

SQL:

SESSION 1:

```
START TRANSACTION;

SELECT * FROM accounts WHERE id = 1;

-- Update the balance if the version is still 0

UPDATE accounts  SET balance = balance + 500, version = version + 1  WHERE id = 1
AND version = 0;

SELECT * FROM accounts WHERE id = 1;

-- Start Another session to do operation on this table.

-- Commit the transaction

COMMIT;
```

SESSION 2: as we are only reading the lock will not be applied

```
START TRANSACTION;

SELECT * FROM accounts WHERE id = 1;
```

```
-- Commit Transaction 1

UPDATE accounts  SET balance = balance + 10000, version = version + 1  WHERE id = 1 AND version = 1;

select * from accounts where id = 1;

COMMIT;

select * from accounts where id = 1;

// this transaction's update will not be reflected in DB as version 1 is not there at the time of executing this update in DB the version was 2.
```

Gap Locks and Next Key Locks

- **Gap Locks**: Lock gaps between rows in index ranges, preventing insertions to avoid phantom reads.
  - Example: In a table with entries (1, 3, 5), a SELECT on values > 1 would lock gaps between these rows to block insertions at 2 and 4.

- **Next Key Locks**: Lock both the current row and the gap before it, blocking changes within the scanned range.
  - Example: Selecting row 3 locks rows and gaps between 1 and 3, preventing changes that could affect future reads.

Auto Increment Lock Modes

1. **Mode 0 (Traditional)**: Table-level lock for the full insert, blocking other insertions.

2. **Mode 1 (Increment Locking)**: Short table lock only while generating the next ID.

3. **Mode 2 (No Locking)**: Lightweight mutex with no table or row locks, allowing higher concurrency.

Locking and Isolation Levels Mapping

Isolation Level	Locking Mechanism	Prevents
READ UNCOMMITTED	No locks	Nothing (allows dirty reads)
READ COMMITTED	Shared for reads, exclusive for writes	Dirty reads
REPEATABLE READ	Shared, exclusive, gap, and next-key locks	Dirty reads, non-repeatable, phantom reads
SERIALIZABLE	Full isolation with range locks	Dirty, non-repeatable, phantom reads, full serializability

TABLES REQUIRED:

```
-- Creating accounts table
CREATE TABLE accounts (
    id INT PRIMARY KEY AUTO_INCREMENT,
    user_id INT UNIQUE KEY NOT NULL,
    user_name VARCHAR(50) NOT NULL,
    account_type VARCHAR(20) NOT NULL,
    balance INT(20),
    department_id VARCHAR(20) NOT NULL,
);


CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(50) NOT NULL,
    department_id INT,
    manager_id INT,
    salary INT
);

ALTER TABLE accounts ADD COLUMN version INT DEFAULT 0;
```