02 ACID Properties - KirkYagami

Atomicity, Consistency, Isolation and Durability in Relational Database Systems.

ATMOCITY

-> All queries in a transaction must succeed.
-> If one query fails, all prior successful queries in the transaction should rollback.
-> If the database went down prior to a commit of a transaction, all the successful queries in the transactions should rollback

EG. if money was debited from account 1 and then database went down (money was not deposited to account 2).
An atomic transaction is a transaction that will rollback all queries if one or more queries failed.

CONSISTENCY

Ensures the database moves from one consistent state to another, maintaining integrity constraints.

Consistency ensures that any transaction will bring the database from one valid state to another, adhering to all defined rules such as primary keys, foreign keys, unique constraints, and check constraints.

-> **Integrity Constraints**
-> **Business Rule**

```sql
BEGIN TRANSACTION;

-- Ensure the accounts have enough balance
SELECT balance FROM accounts WHERE account_id = 1; -- Account A's balance
-- Assume it returns 50 (less than 100)

SELECT balance FROM accounts WHERE account_id = 2; -- Account B's balance
-- Assume it returns 300

-- Perform the transfer only if Account A has sufficient funds
IF (SELECT balance FROM accounts WHERE account_id = 1) >= 100
BEGIN
    -- Deduct from Account A
    UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;

    -- Add to Account B
    UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
    -- Log the transaction
    INSERT INTO transactions (transaction_id, account_id, amount) VALUES (1, 1, -100);
    INSERT INTO transactions (transaction_id, account_id, amount) VALUES (2, 2, 100);

    COMMIT TRANSACTION;
END
ELSE
BEGIN
    -- If there's not enough balance, rollback
    ROLLBACK TRANSACTION;
END
```

ISOLATION

Read Phenomena

-> Dirty Reads
-> Non-repeatable Reads
-> Phantom Reads
-> Lost Updates

1. Dirty Reads

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
-- Transaction 1 has not yet committed

SELECT balance FROM accounts WHERE account_id = 1;
-- If the isolation level is Read Uncommitted, this SELECT can see the uncommitted change
made by Transaction 1

ROLLBACK; --it will undo all uncommitted changes
-- The database will revert to its state before the transaction started.
```

To avoid dirty reads, use a higher isolation level like Read Committed:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

2. Non-repeatable Reads

It occurs when a transaction reads the same row multiple times and gets different values each time due to modifications by another concurrent transaction.

1. *Transaction A* starts and reads the balance:

```sql
BEGIN; -- need to explicitly mention so that the queries become part of this one transaction.
SELECT balance FROM accounts WHERE account_id = 1;
-- Returns: 100
```

2. **Transaction B** starts and updates the balance:

```sql
BEGIN;
UPDATE accounts SET balance = 200 WHERE account_id = 1;
COMMIT;
```

3. **Transaction A** reads the balance again:

```sql
SELECT balance FROM accounts WHERE account_id = 1;
-- Returns: 200
```

Avoiding Non-repeatable Reads

1. **Repeatable Read**:
   - This isolation level ensures that if a transaction reads a row, it will see the same data for that row throughout the transaction. No other transaction can modify that row until the transaction completes.

     > *COMMIT the read transaction so that it is treated complete!*

```sql
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

2. **Serializable**:
   - The highest isolation level which ensures complete isolation from other transactions, preventing non-repeatable reads, phantom reads, and ensuring serializable execution of transactions.

```sql
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

3. Phantom Read

A **phantom read** occurs in SQL when a transaction re-executes a query returning a set of rows that satisfies a certain condition and finds that the set of rows satisfying the condition has changed due to another recently committed transaction. This anomaly typically occurs under the

Read Committed and Repeatable Read isolation levels but is prevented by the Serializable isolation level.

Example Scenario

1. **Initial State**:
   - The `accounts` table has rows:

     ```
     +----+---------+
     | ID | BALANCE |
     +----+---------+
     |  1 |     100 |
     |  2 |     200 |
     +----+---------+
     ```

2. **Transaction A** starts and reads rows with balance greater than 150:

   ```sql
   BEGIN;
   SELECT * FROM accounts WHERE balance > 150;
   -- Returns: (ID: 2, BALANCE: 200)
   ```

3. **Transaction B** inserts a new row and commits:

   ```sql
   BEGIN;
   INSERT INTO accounts (ID, BALANCE) VALUES (3, 300);
   COMMIT;
   ```

4. **Transaction A** re-executes the same query:

   ```sql
   SELECT * FROM accounts WHERE balance > 150;
   -- Returns: (ID: 2, BALANCE: 200), (ID: 3, BALANCE: 300)
   ```

In this scenario, Transaction A sees a different set of rows when it re-executes the same query due to the insertion performed by Transaction B. This is a phantom read because the set of rows that satisfy the condition has changed.

Preventing Phantom Reads

To prevent phantom reads, you can use the Serializable isolation level, which ensures that transactions are executed in a manner that guarantees complete isolation from other transactions. No other transaction can insert, update, or delete rows that would affect the current transaction until it completes.

Setting Isolation Levels

- **Read Committed**: Prevents dirty reads but allows non-repeatable reads and phantom reads.

  ```
  SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
  ```

- **Repeatable Read**: Prevents dirty reads and non-repeatable reads but allows phantom reads.

  ```
  SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
  ```

- **Serializable**: Prevents dirty reads, non-repeatable reads, and phantom reads.

  ```
  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  ```

Example with Serializable Isolation Level

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;

-- First read operation
SELECT * FROM accounts WHERE balance > 150;
-- Returns: (ID: 2, BALANCE: 200)

-- Another transaction cannot insert a new row until this transaction commits

COMMIT;
```

Lost Updates

\A **lost update** occurs when two or more transactions simultaneously read and update the same data, and the final result reflects only the last update, effectively "losing" the earlier updates.

Example Scenario

1. **Initial State**:
   - The `accounts` table has a row:

     ```
     +----+---------+
     | ID | BALANCE |
     +----+---------+
     ```

```
|  1 |      100 |
+----+---------+
```

2. **Transaction A** starts and reads the balance:

```sql
BEGIN;
SELECT balance FROM accounts WHERE id = 1;
-- Returns: 100
```

3. **Transaction B** starts and reads the balance:

```sql
BEGIN;
SELECT balance FROM accounts WHERE id = 1;
-- Returns: 100
```

4. **Transaction A** updates the balance:

```sql
UPDATE accounts SET balance = balance + 50 WHERE id = 1;
-- New balance: 150
COMMIT;
```

5. **Transaction B** updates the balance:

```sql
UPDATE accounts SET balance = balance - 30 WHERE id = 1;
-- New balance: 70 (Based on the initial read balance of 100)
COMMIT;
```

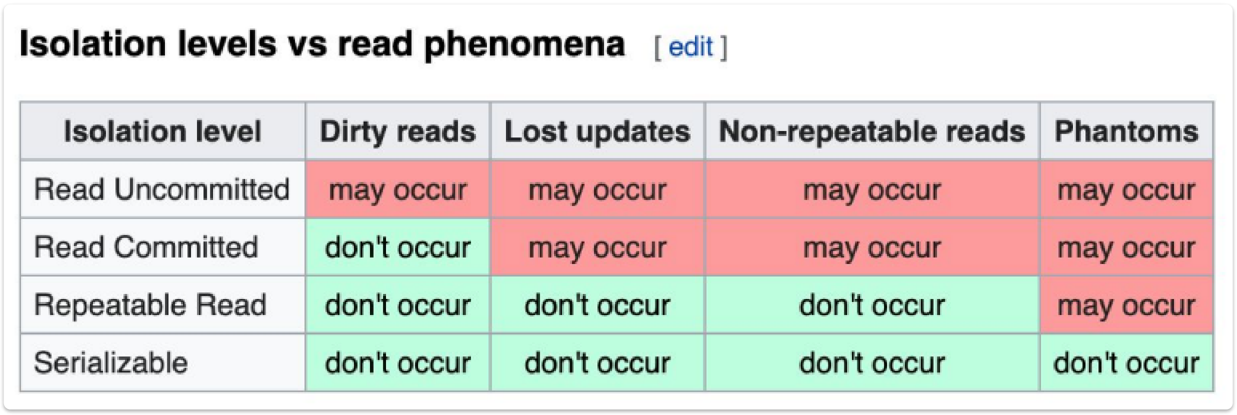
6. **Final State**:
   - The balance in the account is now 70, reflecting only Transaction B's update and "losing" the update made by Transaction A.

Preventing Lost Updates

- **Serializable Isolation Level**: Use the highest isolation level to ensure complete isolation from other transactions.

```sql
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
-- This isolation level ensures no other transactions can interfere
SELECT balance FROM accounts WHERE id = 1;
UPDATE accounts SET balance = balance + 50 WHERE id = 1;
COMMIT;
```

- Lost Update: Occurs when simultaneous transactions overwrite each other's updates, resulting in only the last update being preserved.
- Prevention: Use techniques such as pessimistic locking, optimistic locking, or the Serializable isolation level to ensure transactions are properly isolated and updates are not lost.



Isolation levels vs read phenomena [edit]

Isolation level	Dirty reads	Lost updates	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur	may occur
Read Committed	don't occur	may occur	may occur	may occur
Repeatable Read	don't occur	don't occur	don't occur	may occur
Serializable	don't occur	don't occur	don't occur	don't occur

Read phenomenon Isolation level	Dirty read	Non-repeatable read	Phantom read
Serializable	no	no	no
Repeatable read	no	no	yes
Read committed	no	yes	yes
Read uncommitted	yes	yes	yes

Isolation - Isolation Levels for inflight transactions

• Read uncommitted - No Isolation, any change from the outside is visible to the transaction, committed or not.
• Read committed - Each query in a transaction only sees committed changes by other transactions
• Repeatable Read - The transaction will make sure that when a query reads a row, that row will remain unchanged while its running. (Default for MySQL)
• Snapshot - Each query in a transaction only sees changes that have been committed up to the start of the transaction. It's like a snapshot version of the database at that moment.
• Serializable - Transactions are run as if they serialized one after the other. • Each DBMS implements Isolation level differently

Database Implementation of Isolation

- Each DBMS implements Isolation level differently
- Pessimistic - Row level locks, table locks, page locks to avoid lost updates
- Optimistic - No locks, just track if things changed and fail the transaction if so
- Repeatable read "locks" the rows it reads but it could be expensive if you read a lot of rows, postgres implements RR as snapshot. That is why you don't get phantom reads with postgres in repeatable read
- Serializable are usually implemented with optimistic concurrency control, you can implement it pessimistically with SELECT FOR UPDATE

dd

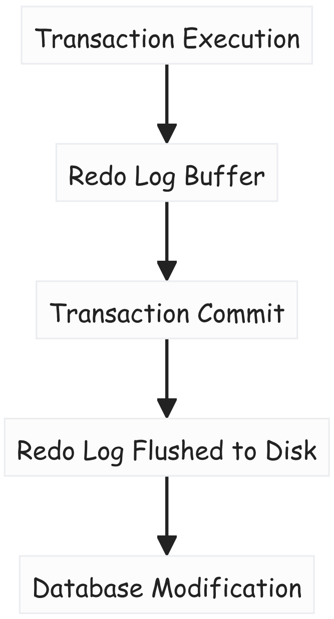
DURABILITY

> *Changes made by committed transactions must be persisted in a durable non-volatile storage.*

Once a transaction is committed, the changes it made to the database persist even in the event of system failures, crashes, or power outages. In other words, the durability property guarantees the permanence and persistence of committed transactions.

Key Components of WAL:

1. Log Files: Database systems maintain a separate log file or set of log files specifically for recording changes made by transactions. These log files contain records of actions such as insertions, updates, and deletions performed by transactions.
2. Redo Log and Undo Log: Within the log files, there are typically two types of logs: redo log and undo log.
   - Redo Log: Records changes made by transactions before they are applied to the database. Redo logs ensure that committed transactions can be replayed and their changes reapplied in the event of a crash or failure.
   - Undo Log: Records the inverse of changes made by transactions, enabling the database system to rollback transactions or undo their changes if necessary.
3. Log Sequence Number (LSN): Each log record is assigned a unique Log Sequence Number (LSN) to maintain the order of operations. This allows the database system to ensure that log records are processed in the correct sequence during recovery.


Transaction Execution
Redo Log Buffer
Transaction Commit
Redo Log Flushed to Disk
Database Modification