Transactions

Transactions

A transaction is a sequence of SQL operations treated as a single atomic command.

Why Transactions?

Transactions in a database context have two main purposes:

1. To provide atomic work units that can guarantee the database always remains in a consistent state.
2. To provide isolation between different database users.

Example. Imagine a database that handles banking information. Let the following simple table handle personal bank accounts: Accounts(accountNum, customerId, balance).

A transfer $100 from account 1 to account 2 could be represented in the following two SQL commands:

```
UPDATE Accounts SET balance = balance - 100 WHERE accountsNum = 1;
UPDATE Accounts SET balance = balance + 100 WHERE accountsNum = 2;
```

If there is a database crash after the first command, then what is the result? The database is left in an **inconsistent** state. Although no referential integrity has been broken, the database no longer maintains a consistent view of reality. Account 1 lost $100 that was never gained by Account 2.

ACID

ACID is an acronym that represents a set of database properties that guarantee all database transactions are processed reliably.

Atomicity. Atomicity requires that all database transactions are "all or nothing". Either all of the transaction goes through, or none of it happens.

Consistency. Consistency ensures that any successful transaction will leave the database in a valid state. A valid database is determined by all the defined rules, including but not limited to constraints, triggers, and cascades.

Isolation. Isolation ensures that all transactions are unaware of all other transactions, they are independent. They act as though they all happen one after another instead of at the same time.

Durability. Durability means that once a transaction is committed, it will remain so even if the database crashes or power is lost. This means that the transaction must be committed to persistent memory.

Isolation Levels

Of all the ACID constraints, `Isolation` is the one most often relaxed. This is because performing full isolation can mean obtaining and holding expensive locks on the database.

Isolation generally comes in four different levels:

Serializable

This is the highest (most strict) isolation level. No two transactions are allowed to work on the same table at the same time. One transaction may be blocked until another completes.

```
> START TRANSACTION;
> SELECT balance FROM Accounts WHERE accountNum = 1;
1000
> SELECT balance FROM Accounts WHERE accountNum = 1;
1000
> COMMIT;
```

```
> START TRANSACTION;
*BLOCKED*
*BLOCKED*
*BLOCKED*
*BLOCKED*
*BLOCKED*
> SELECT balance FROM Accounts WHERE accountNum = 1;
1000
> UPDATE Accounts SET balance = balance - 100 WHERE accountNum
= 1;
> SELECT balance FROM Accounts WHERE accountNum = 1;
900
> COMMIT;
```

Repeatable Read

This is the next strictest isolation level. The same value will always be read (even if the value is incorrect).

```
> START TRANSACTION;
> SELECT balance FROM Accounts WHERE accountNum = 1;
1000



> SELECT balance FROM Accounts WHERE accountNum = 1;
1000
> COMMIT;
```

```
> START TRANSACTION;
> UPDATE Accounts SET balance = balance - 100 WHERE accountNum
= 1;
> SELECT balance FROM Accounts WHERE accountNum = 1;
900
> COMMIT;
```

Read Committed

The most recently committed version of the value will always be read (even if the value was updated in the middle of the transaction).

```
> START TRANSACTION;
> SELECT balance FROM Accounts WHERE accountNum = 1;
1000



> SELECT balance FROM Accounts WHERE accountNum = 1;
1000



> SELECT balance FROM Accounts WHERE accountNum = 1;
900
> COMMIT;
```

```
> START TRANSACTION;
> UPDATE Accounts SET balance = balance - 100 WHERE accountNum
= 1;
> SELECT balance FROM Accounts WHERE accountNum = 1;
900



> COMMIT;
```

Read Uncommitted

Uncommitted values (from other transactions) may be read.

```
> START TRANSACTION;
> SELECT balance FROM Accounts WHERE accountNum = 1;
1000

> SELECT balance FROM Accounts WHERE accountNum = 1;
900

> SELECT balance FROM Accounts WHERE accountNum = 1;
900
> COMMIT;
```

```
> START TRANSACTION;
> UPDATE Accounts SET balance = balance - 100 WHERE accountNum
= 1;
> SELECT balance FROM Accounts WHERE accountNum = 1;
900

> COMMIT;
```

Comparison

- Dirty Read — A transaction reads data written by a concurrent uncommitted transaction.
- Non-Repeatable Read — A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).
- Phantom Read — A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

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

Syntax

To start a transaction, you use:

```
START TRANSACTION;
```

To commit a transaction, use:

```
COMMIT;
```

To rollback a transaction, just use ROLLBACK instead of COMMIT:

```
ROLLBACK;
```

```
> START TRANSACTION;
> SELECT balance FROM Accounts WHERE accountNum = 1;
1000
> UPDATE Accounts SET balance = 2 WHERE accountNum = 1;
> SELECT balance FROM Accounts WHERE accountNum = 1;
2
> ROLLBACK;
> SELECT balance FROM Accounts WHERE accountNum = 1;
1000
```

If you want to use transactions in MySQL, you should set your session `autocommit` variable to false. This can also be set on a server-wide basis.

```
SET autocommit = 0;
```

To set the transaction level of a transaction, you use the following command:

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
    { REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE };
```

References

1. `http://www.postgresql.org/docs/9.1/static/transaction-iso.html`