Database Security: Transactions, Access Control, and SQL Injection

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 };
```

Access Control

SQL uses a user based security model where each user is granted explicit privileges on specific databases and tables. It could be as general as all permissions on every table in each database (as a root user would have), or as specific as only SELECT on a select few tables in a specific database.

The general syntax for the command than grants permissions is:

GRANT <privilege> ON <database>.<table> TO <user>@<host> [IDENTIFIED BY '<password>'];

The different possible privileges[1] are:

Privilege	Meaning
ALL [PRIVILEGES]	Grant all privileges at specified access level except GRANT OPTION
ALTER	Enable use of ALTER TABLE
ALTER ROUTINE	Enable stored routines to be altered or dropped
CREATE	Enable database and table creation
CREATE ROUTINE	Enable stored routine creation
CREATE TEMPORARY TABLES	Enable use of CREATE TEMPORARY TABLE
CREATE USER	Enable use of CREATE USER, DROP USER, RENAME USER, and REVOKE ALL PRIVILEGES
CREATE VIEW	Enable views to be created or altered
DELETE	Enable use of DELETE
DROP	Enable databases, tables, and views to be dropped
EXECUTE	Enable the user to execute stored routines
FILE	Enable the user to cause the server to read or write files
GRANT OPTION	Enable privileges to be granted to or removed from other accounts
INDEX	Enable indexes to be created or dropped
INSERT	Enable use of INSERT
LOCK TABLES	Enable use of LOCK TABLES on tables for which you have the SELECT privilege
PROCESS	Enable the user to see all processes with SHOW PROCESSLIST
REFERENCES	Not implemented
RELOAD	Enable use of FLUSH operations
REPLICATION CLIENT	Enable the user to ask where master or slave servers are
REPLICATION SLAVE	Enable replication slaves to read binary log events from the master
SELECT	Enable use of SELECT
SHOW DATABASES	Enable SHOW DATABASES to show all databases
SHOW VIEW	Enable use of SHOW CREATE VIEW
SHUTDOWN	Enable use of mysqladmin shutdown
SUPER	Enable use of other administrative operations such as CHANGE MASTER TO, KILL, PURGE BINARY LOGS, SET GLOBAL, and mysqladmin debug command
UPDATE	Enable use of UPDATE
USAGE	Synonym for "no privileges"

You can also use special characters in the GRANT statement.

[1] These are privileges in MySQL. They differ based on the DBMS. Oracle SQL has hundreds of privileges.

***** When specifying the database/table, a '*' may be used to match anything.

- *.* — matches all tables in all databases (use with caution).
- Foo.* — matches all tables in the Foo database.
- *.Bar — matches the Bar tables in every database.

% As with the LIKE string operation, the '%' can be used when specifying a user to match any (or no) characters.

- '%'@'localhost' — matches all users connecting from the local machine.
- 'bob'@'%' — matches the user 'bob' coming from any machine.
- '%'@'192.168.0.%' — matches any user coming from a machine with a local ip.

REVOKE

The converse of GRANT is REVOKE. The general syntax[2] for REVOKE is:

REVOKE <privilege> ON <database>.<table> FROM <user>@<host>;

SQL Injection

SQL Injection is a technique used to attack applications that are backed by a database. Most of the different SQL Injection techniques revolve around putting SQL statements where numbers or string literals are expected.

SQL injection attacks are considered one of the top 10 web application vulnerabilities of 2007 and 2010 by the Open Web Application Security Project. In operational environments, it has been noted that applications experience an average of 71 attempts an hour (`http://blog.imperva.com/2011/09/sql-injection-by-the-numbers.html`).

Examples

SELECT Injection Consider the following situation where a query is dynamically built:

```
String inputAccount = getUserInput();
String query = "SELECT customerId, balance FROM Accounts" +
               " WHERE accountNum = '" + inputAccount + "'" +
               " AND secretSecurityCheck = TRUE";
```

A normal user would properly enter in their account number, resulting in the following well-formed query:

```
SELECT customerId, balance
FROM Accounts
WHERE accountNum = 'abc123' AND secretSecurityCheck = TRUE;
```

However, a malicious users could give an input value like:

```
abc123' OR 1 = 1 --
```

This attack contains three parts:

- "abc123'" — A logical input that forcibly ends the string literal with a terminating "'".

[2] Both GRANT and REVOKE have more options and variants in their syntax.

- "OR 1 = 1" — A statement that makes the entire logical expression evaluate to TRUE.
- "--" — End the input with a comment to ignore any final parts of the query.

The final malicious query would be:

```
SELECT customerId, balance
FROM Accounts
WHERE accountNum = 'abc123' OR 1 = 1 -- AND secretSecurityCheck = TRUE;
```

Which essentially becomes:

```
SELECT customerId, balance FROM Accounts;
```

The malicious user was able to get the customer identifiers along with all their account balances.

SELECT Injection Consider the following dynamic query the represents a withdrawal from a bank:

```
String inputValue = getUserInput();
String query = "UPDATE Accounts" +
               " SET balance = balance - " + inputValue +
               " WHERE accountNum = 'abc123'";
```

A malicious user can input:

```
100, balance = 1000000
```

The resulting update becomes:

```
UPDATE Accounts SET balance = balance - 100, balance = 1000000;
```

Instead of withdrawing $100, the malicious user set their balance to $1,000,000.

Prevention

There are multiple ways to prevent against SQL Injection:

Sanitization. Sanitizing your input means removing all invalid input and escaping all SQL syntax.

Sanitizing the string from the first injection example would be as easy as escaping the single quote:

```
abc123'' OR 1 = 1 --
```

You should **NOT** write your own sanitization code. There is SQL sanitization code written in every language.

Prepared Statements. Prepared statements (in most connectors) will do automatic sanitization for you.

Restrict User Input. Restrict the types of input that the user is allowed to enter. Notice that in the second example, the host language programmer was lazy and used a String where a number would have been more correct. Requiring that numeric values ONLY be numbers can help prevent against injection.

Proper Permissions. An often overlooked injection prevention technique is to make sure that your database permissions are set correctly. If the host application is highly restricted in what operations it can perform, the window for injection is narrowed.

References

1. http://www.postgresql.org/docs/9.1/static/transaction-iso.html
2. http://dev.mysql.com/doc/refman/5.0/en///grant.html
3. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project