# A Primer on SQL Transactions

In SQL, transactions are essential for maintaining database integrity when dealing with multiple related operations as well as when multiple users are interacting with a database concurrently. In this article we'll cover the concept of SQL Transactions, what they are, and how to work with them.

## What are SQL Transactions?

SQL **Transactions** is a grouping of logic statements that can contain one or more SQL statements that interact within a database. A transaction in its entirety can be **committed** to a database as a single logical unit, or **rolled back** (undone) as a single logical unit.

## Why are SQL Transactions Necessary?

In certain instances, a database application has to account for every possible failure scenario while writing and reading from a database. While ensuring that every aspect of database integrity is not violated, application code will be extremely complex, convoluted, and expensive to develop and maintain.

Fortunately, via the use of SQL Transactions, code/maintenance can be simplified. Through the use of committing and rolling back transactions as necessary, database integrity can easily be maintained.

## Understanding With an Example

Using a simple logical example, let us suppose a financial brokerage maintains a database for its clients. The brokerage is required to generate a quarterly financial statement for each of its clients that reports on statement balance, current values of holdings, and any transactions that occurred during the quarter. What sort of processing steps might that hypothetically require?

1) We may have to loop through each client's transaction history account to ascertain and calculate each of the transactions that occurred during the quarter (both purchases and sales);

2) We will need to calculate each client's total portfolio's return for the quarter, along with year-to-date returns;

3) We will need to calculate each client's taxable and non-taxable returns for the quarter;

4) We will need to **insert** a record into a Statements table of some sort to record the prior 3 steps;

5) We will need to **update** the current portfolio holdings values and investment totals for the quarter in a Quarterly Values table of some sort;

6) We will need to **update** the statement balance in an Accounts table of some sort;

Obviously, many read and write operations are required for the above steps. What if a transaction that falls within a reported quarter is backed out or changed after calculations have already been made? What if one of the updates noted above fails after we have already inserted a record into our Statements table? What if the total statement balance cannot be updated?

These are all valid concerns and scenarios that can occur in a database application. To have to account for every possible permutation of error to occur in a database application would obviously be a coding/maintenance nightmare.

The answer here is to make use of SQL Transactions which can dramatically simplify coding, allow all statements in a transaction block to either succeed or fail, and most importantly, ensure database integrity.

## SQL Transaction Delimiter Syntax

**NOTE**: It should be noted that the implementation of transactions is very similar in concept across most database implementations, but syntax can vary slightly.

Typically, within the confines of an SQL Stored Procedure (or via Command-line statement entries), we may define the beginning of a transaction with the following command in SQL Server:

**BEGIN TRANSACTION** NameOfTransaction;

In MySQL, the syntax is slightly different, but has the same meaning:

**START TRANSACTION**;

## Committing Transactions

If a database application decides that all the processing of a block of change activity has succeeded, the application can use the **COMMIT TRANSACTION** statement at the end of the block to Commit the requisite changes to the database. In SQL Server, the command is:

**COMMIT TRANSACTION**; OR
**COMMIT**;

In MySQL, the command is:

```
COMMIT;
```

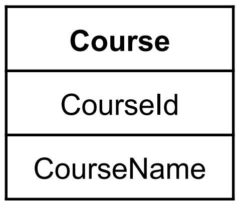
## Rolling Back Transactions

If a database application decides that something in a change activity code block has failed, the application can use the **ROLLBACK** statement to effectively decommit any statements that have already been executed since the beginning of the transaction. In SQL Server and MySQL the syntax of the command to do this is:

```
ROLLBACK;
```

## Database Transaction Examples

For the purposes of highlighting the mechanisms behind transaction processing, let us assume a simple database named School that includes a table named Course (among other tables), that is defined as follows:



```
CREATE TABLE Course)
        CourseId       SMALLINT            NOT NULL;
        CourseName     VARCHAR(40)         NOT NULL);
```

**<u>Example #1</u>**: (Committing a Transaction)

This example (using SQL Server syntax) obtains the largest ‘CourseId’ value from table Course, adds 1 to it, and inserts a row into the Course table, and then Commits the Transaction. If any part of the transaction CourseAdd fails to execute properly, prior to the Commit, none of the transaction will be processed. That means if the Select or Insert statement fails in some capacity, the entire transaction will be null and void.

```
BEGIN TRANSACTION CourseAdd;

DECLARE @CrsId    SMALLINT;

SELECT @CrsId = MAX(CourseId) + 1
FROM    Course;

INSERT Course VALUES (@CrsId, ‘Biology 101’);
```

```
COMMIT TRANSACTION;
```

**<u>Example #2</u>**: (Rolling Back of a Transaction)

In this example, we are manually adding (inserting) rows into table Course from the Command Line. Rather than just adding rows directly (which are assumed Commits), we decide to wrap all of our Inserts statements into a Transaction. That way, if we make a mistake, we can rollback the entire transaction easily. Using MySQL syntax:

```
START TRANSACTION;
INSERT Course VALUES (1, 'Biology 101');
INSERT Course VALUES (2, 'Computer Science 101');
ROLLBACK;
```

Here we see that, by wrapping our Insert statements inside of a transaction, we have ensured that the Insert statements have not committed the Insert data into the table Course until a Commit command is received. By virtue of issuing a Rollback statement, we've effectively undone the 2 prior Insert statements, and not Committed either of these 2 rows to the database.

**<u>Example # 3</u>**: (Combining Commit and Rollback both in a Transaction)

This simplistic example combines the ability to both Commit and Rollback transactions in the same transaction code block. Consider the following SQL Server code segment:

```
BEGIN TRANSACTION InsertCourse;

DECLARE @CrsId    SMALLINT;

SELECT @CrsId = MAX(CourseId) + 1
FROM   Course;

INSERT Course VALUES (@CrsId, 'Biology 101');

IF (SELECT COUNT(CourseName)
   FROM Course
   WHERE CourseName = 'Biology 101') > 1
   ROLLBACK;
END IF;

COMMIT TRANSACTION;
```

In this SQL code segment, we are Inserting a row in table Course with the next highest CourseId. Before committing the transaction, we check to see if we have more than one row where the CourseName is “Biology 101”. If we do, we do not want to commit the transaction (that is, commit that Insert to the database), and at that point we Rollback the transaction, and the code segment aborts from further processing. If it is the first instance of a CourseName of “Biology 101”, then the transaction will proceed, and eventually be Committed to the database.

## More Benefits of Using Transactions

When should you use Transactions?  Should you use them all the time?

The simple answer is to use them all the time, especially when you are dealing with multiple groups of statements (i.e. a scenario where all the statements in a sequence of statements must succeed for the associated data to be Committed to the database). In this case, a failure of a component within the transaction necessitates a Rollback.

Moreover, use of transactions are extremely beneficial when you run the risk of things like power failures, server crashes, disk drive failure, database crashes, etc.

In the event of any of these failures, if transactions have not yet been committed, you are guaranteed to have maintained the integrity of the database. Without the use of transactions, any atomic statements that have been applied to the database remain intact, regardless of whether associated statements have been executed or not (hence, a data integrity issue may result).

## Conclusion

SQL Transactional logic is a fundamental mechanism to ensure database integrity, and to minimize the amount of error control logic that is required in a multi-user database application. In short, use of transactions in SQL environments guarantee accuracy and completeness.