

# Transactions



In almost all applications that access MySQL databases, multiple users concurrently attempt to view and modify data.

The simultaneous operations may result in data that is inconsistent and inaccurate.

Using transactions avoid these problems by isolating each operation.

**In the context of SQL and relational databases,**

*a transaction* is a set of one or more SQL statements that perform a set of related actions.

The statements are grouped together and treated as a single unit whose success or failure depends on the successful execution of each statement in the transaction.

**In addition to ensuring the proper execution of a set of SQL statements,**

a transaction locks the tables involved in the transaction so that other users cannot modify any rows that are involved in the transaction.

# ACID – generally, every transaction is ACID



(Atomicity-Consistency-Isolation-Durability) compliant in accordance with concurrency control and transparency in the DS. It is NOT required, but the best dbase transactions are 'ACID friendly'.

**ATOMICITY** – ALL or NO operations in a transaction are performed.

Transaction Recovery System – ensures atomicity and enables the all or nothing output.

**CONSISTENCY** – consistent state is maintained before a transaction starts and after it concludes.

**ISOLATION** – concurrent transactions DO NOT interfere with each other.

Partial results of incomplete transactions are not visible to others before the transactions are committed.

**DURABILITY** – transactions results are locked/ permanent after being committed.

System guarantees that results of a committed transaction will be permanent even if a failure occurs after the commit.

# Starting a Transaction



MySQL provides the

- ❑ `START TRANSACTION`  
to create a transaction, and
- ❑ `COMMIT`, `ROLLBACK` statements  
to end it.
- ❑ The `COMMIT` statement  
saves changes to the database
- ❑ The `ROLLBACK` statement  
will undo any changes made during the transaction and database is  
reverted to the pre-transaction state.

## The START TRANSACTION Statement



The START TRANSACTION statement requires no clauses or options:

**START TRANSACTION**

START TRANSACTION statement notifies MySQL that the statements that follow should be treated as a unit, until the transaction ends, successfully or otherwise.

*Note:* A BEGIN statement can also be used to start a transaction.

# Committing a Transaction




## The COMMIT Statement

The COMMIT statement is used to terminate a transaction and to save all changes made by the transaction to the database. There are no additional clauses or options:

COMMIT

The following transaction is made of two INSERT statements, followed by a COMMIT statement:



```
START TRANSACTION;  
INSERT INTO Studio VALUES (101, 'MGM Studios');  
INSERT INTO Studio VALUES (102, 'Wannabe  
    Studios');  
COMMIT;  
SELECT * FROM Studio;
```

Ensure the table type is as of InnoDB before this trying this example.





```
START TRANSACTION;
```

```
UPDATE Studio SET title = 'Temporary Studios'  
WHERE id = 101;
```

```
UPDATE Studio SET title = 'Studio with no buildings'  
WHERE id = 102;
```

```
SELECT * FROM Studio;
```

```
ROLLBACK;
```

```
SELECT * FROM Studio;
```

## Adding Savepoints to Your Transaction



The **SAVEPOINT** and **ROLLBACK TO SAVEPOINT** statements isolate portions of a transaction. The **SAVEPOINT** statement defines a marker in a transaction, and the **ROLLBACK TO SAVEPOINT** statement allows you to roll back a transaction to a predetermined marker (savepoint).



Start Transactions;

SAVEPOINT savepoint1

INSERT INTO Studio VALUES (105, 'Noncomformant Studios');

INSERT INTO Studio VALUES (106, 'Studio Cartel');

SELECT \* FROM Studio;

ROLLBACK TO SAVEPOINT savepoint1;

INSERT INTO Studio VALUES (105, 'Moneymaking Studios');

INSERT INTO Studio VALUES (106, 'Studio Mob');

SELECT \* FROM Studio;

COMMIT;



A *dirty read* can take place when:


Transaction A modifies data in a table.

Around the same time, another Transaction B reads the table, *before* those modifications are committed to the database.

Transaction A rolls back (cancels) the changes, returning the database to its original state.

Transaction B now has data inconsistent with the database.

Worse, Transaction B may modify the data based on its initial read, which is incorrect or *dirty read*.




The transaction isolation level determine the degree to which other transactions can “see” details inside an in-progress transaction and are arranged in hierarchical order.

Serializable

Repeatable Read

Read Committed


Read Uncommitted



**SERIALIZABLE** — In the **SERIALIZABLE** isolation level of MySQL, data reads are implicitly run with a read lock

## **REPEATABLE READ**

provides more isolation from a transaction, ensuring that data reads are the same throughout the transaction even if the data has been changed and committed by a different transaction. The **SERIALIZABLE** isolation level provides the slowest performance but also the most isolation



READ COMMITTED provides some isolation and slightly slower performance, because only committed data changes are seen by other transactions. However, READ COMMITTED does not address the issue of data changing in the middle of a transaction.

READ UNCOMMITTED is the easiest isolation level to implement and provides the fastest performance. The problem with READ UNCOMMITTED is that it provides no isolation between transactions.

# Locking Nontransactional Tables

- ❑ MySQL supports the use of transactions only for InnoDB and BDB tables.
- ❑ There might be times, though, when you want to lock other types of tables that are included in your database.
- ❑ By locking nontransactional tables manually, you can group SQL statements together and set up a transaction-like operation in order to prevent anyone from changing the tables that participate in your operation.
- ❑ To lock a nontransactional table, you must use the LOCK TABLES statement.
- ❑ Once you've completed updating the tables, you should use the UNLOCK TABLES statement to release them.



## The **LOCK TABLES** Statement

To lock a table in a MySQL database, you should use the `LOCK TABLES` statement, as shown in the following syntax:

```
LOCK {TABLE | TABLES}
<table name> [AS <alias>] {READ [LOCAL] | [LOW_PRIORITY] WRITE}
[[, <table name> [AS <alias>] {READ [LOCAL] | [LOW_PRIORITY] WRITE}]...]
```

To use the statement, you must specify the `LOCK` keyword, the `TABLE` or `TABLES` keyword, one or more tables, and the type of lock for each table. The `TABLE` and `TABLES` keywords are synonymous, and you can use either one, whether you're locking one table or multiple tables. Generally, `TABLE` is used for one table, and `TABLES` is used for multiple tables, but either can be used in either situation.

For each table that you specify, you have the option of providing an alias for the table. In addition, you must specify a `READ` lock type or a `WRITE` lock type. If you specify `READ`, any connection can read from the table, but no connection can write to the table. If you specify `READ LOCAL`, nonconflicting `INSERT` statements can be executed by any connection. If you specify `WRITE`, the current connection can read or write to the table, but no other connections can access the table until the lock has been removed. If you specify `LOW_PRIORITY WRITE`, other connections can obtain `READ` locks while the current session is waiting for the `WRITE` lock.

*Once you lock a table, it remains locked until you explicitly unlock the table with the `UNLOCK TABLES` statement (described in the text that follows) or end your current session.*

Now take a look at an example of a `LOCK TABLE` statement. The following statement places a lock on the `Books` table:

```
LOCK TABLE Books READ;
```

As you can see, a `READ` lock has been placed on the table. Now only read access is available to all connections. You are not limited, however, to placing a lock on only one table. For example, the following `LOCK TABLES` statement places locks on the `Books` and `BookOrders` tables:

```
LOCK TABLES Books READ, BookOrders WRITE;
```

In this case, a `READ` lock is placed on the `Books` table, and a `WRITE` lock is placed on the `BookOrders` table. As a result, other connections can read from the `Books` table, but they cannot access the `BookOrders` table.

## The **UNLOCK TABLES** Statement

Once you've completed accessing a locked table, you should explicitly unlock the table or end your current session. To unlock one or more locked tables, you must use the `UNLOCK TABLES` statement, shown in the following syntax:

```
UNLOCK {TABLE | TABLES}
```

As you can see, you must specify the `UNLOCK` keyword along with the `TABLE` or `TABLES` keyword. As with the `LOCK TABLES` statement, the `TABLE` and `TABLES` keywords are synonymous, which means that you can use either one, regardless of the number of tables that you're unlocking.

One thing to notice about the `UNLOCK TABLES` statement is that no table names are specified. When you use this statement, all tables that have been locked from within the current session are unlocked. For example, to unlock the `Books` and `BookOrders` tables, you would use a statement similar to the following:

```
UNLOCK TABLES;
```

If only one table is locked, you can still use this statement, although you can also use the `TABLE` keyword, rather than `TABLES`. Either way, any locked tables are now unlocked.

In general, you should try to use InnoDB tables when setting up your system to support transactions. In the case of the `DVDRentals` database, all tables are defined as InnoDB. Should you ever run into a situation in which you want to lock another type of table manually, you can use the `LOCK TABLES` and `UNLOCK TABLES` statements.

# Stored Procedures

A stored procedure is a routine made up of a set of predefined SQL statements that take some sort of action on the data in your database. The SQL statements are bundled together in a named package that is stored on the MySQL server. Once the stored procedure has been created, you can invoke the SQL statements by calling the procedure name. MySQL then executes the statements as though you had issued them interactively or through an application (via an API).

By using stored procedures, client applications don't need to issue the same statements repeatedly. They can simply call the stored procedure. For example, suppose that you are creating an application that must issue the same `SELECT` statement numerous times in the course of a user session. If the statement is embedded in the application, it must be sent from the application to the database each time you need to retrieve data. If you create a stored procedure that contains the `SELECT` statement, however, you can simply call that procedure from your application.

To create a stored procedure, you must use the `CREATE PROCEDURE` statement, provide a name for the procedure, and add the necessary SQL statements. For example, the following SQL statement creates the `BookAmount` stored procedure:

```
CREATE PROCEDURE BookAmount (@bookId INT)
BEGIN
    SELECT BookName, Quantity FROM Books WHERE BookID=@bookId;
END
```

```
CREATE PROCEDURE BookAmount (@bookId INT)
BEGIN
    SELECT BookName, Quantity FROM Books WHERE BookID=@bookId;
END
```

As you can see, you specify the `CREATE PROCEDURE` keywords, followed by a name for the procedure, and then followed by any necessary parameter definitions. In this case, there is one parameter definition: `@bookId`, which is defined as an `INT` type. If you do not need to include any parameter definitions, you must still include the parentheses. The actual SQL statement is enclosed in a `BEGIN/END` block. Notice that that `SELECT` statement is the only component of the stored procedure definition that ends with a semi-colon. You should terminate each SQL statement in the `BEGIN/END` block with a semi-colon.

There are other elements that you can add to a stored procedure definition, but this example provides you with the basic ones. All the elements shown in the example are required, except for the parameter definition. You do not have to define a parameter in a stored procedure, but you do have to include the `BEGIN/END` block and one or more SQL statements.

Once you've created your stored procedure, you can invoke it simply by using the `CALL` statement:

```
CALL BookAmount(101);
```

Notice that all you need to specify is the `CALL` keyword, then the name of the stored procedure, and then a parameter value, enclosed in parentheses. If no parameters are defined for the stored procedure, you must still include the parentheses, with no value enclosed in them.

Despite the ease of creating and using stored procedures, they do have their down side. Because they're stored on the MySQL server, using them increases the load on your database system because more work has to be done on the server side and less on the presentation side. This can be an important consideration if numerous front-end servers are being supported by relatively few back-end database servers.

In some situations, stored procedures can be particularly helpful, especially when security is a critical concern (such as in banking applications). In this case, you can set up MySQL so that applications and users can only execute the stored procedures, but cannot access the database directly. In addition, each operation is properly logged, so you have a more complete record of everything that has transpired, should you need to track events.

As you can see, the decision to use a stored procedure depends on the needs of your application or how you want to distribute the workload across the server tiers. When you do decide to implement stored procedures, you'll find them to be a very useful tool in developing data-driven applications.

# Views

Views are yet another database feature that has been available to most RDBMSs for many years. A view is a virtual table whose definition is stored in the database but that does not actually contain any data. Instead, a view points to one or more tables and presents data from those tables in a structure similar to any other type of table. A view is basically a `SELECT` statement stored as a named object. You can then access a view as you would another table, and the result set is displayed as though you had invoked the `SELECT` statement directly.

Suppose that you are creating a database for a bookstore. The database includes three tables: `Books`, `Authors`, and `AuthorBook`, which matches books to their authors. If you want view the name of the books with a copyright date before 1980 and the books' authors, you can issues a `SELECT` statement similar to the following:

```
SELECT BookTitle, Copyright, CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Books AS b, AuthorBook AS ab, Authors AS a
WHERE b.BookID=ab.BookID AND ab.AuthID=a.AuthID AND Copyright<1980
ORDER BY BookTitle;
```

The `SELECT` statement retrieves the necessary information and displays it in a format similar to the following:

BookTitle	Copyright	Author
Black Elk Speaks	1932	Black Elk
Black Elk Speaks	1932	John G. Neihardt
Hell's Angels	1966	Hunter S. Thompson
Letters to a Young Poet	1934	Rainer Maria Rilke

4 rows in set (0.00 sec)