Using Transactions In MySQL (Part 1)
By icarus
2003−11−03

Moving Up
Back in the good old days, the only people who took MySQL seriously were the geeks, who used it to power their Web sites, and the evangelists, who held it up to the public at large as a shining example of what the open−source community was capable of producing when left to its own devices. Despite MySQL's numerous strengths at the time − speed being the primary one − very few people looked at it as a viable alternative to enterprise−scale RDBMSs like Oracle and Microsoft SQL Server; rather, it was considered competent for small and medium−scale applications, but inadequate for large−scale enterprise deployment.

Today, however, things have changed. MySQL is now in use in more than 30,000 installations across the planet and, with large corporate houses like SAP taking an interest in its future development, seems slated to give commercial RDBMS vendors a serious run for their money in the enterprise market. The reasons for this gradual shift are not hard to find: while earlier versions of MySQL (the 3.23.x series) were pretty basic, lacking support for advanced features like transaction processing, data replication, subqueries and multi−table operations, newer versions (the 4.x series) have moved aggressively to close the feature set gap and now include many new capabilities (including all the items mentioned above).

One of the newest features in MySQL, and one of the most frequently−requested ones on the MySQL wishlist, is transactions. Transactions are found in almost all commercial RDBMSs, and their omission from the MySQL canon has been a source of much hand−wringing amongst MySQL enthusiasts for the last few years. Despite initial resistance to the idea from MySQL HQ (which was understandably concerned about reducing the speed of its RDBMS engine by adding transactional support to it), transactions have finally made an appearance in MySQL 4.0.

That's where this article comes in. Over this two−part tutorial, I will be introducing you to transactions, which allow you to build more robust Web applications and simultaneously reduce the possibility of data corruption in your DBMS. Regardless of whether you've a novice who's never heard of transactions before, or someone who's been tinkering with MySQL for a while (like me), I think you'll find the next few pages interesting. So come on in, and let's get started.

Money, Money, Money...
We'll start, as always, with getting the terminology down: what the heck is a transaction when it's at home?

In the SQL world, the term "transaction" refers to a series of SQL statements which are treated as a single unit by the RDBMS. Typically, a transaction is used to group together SQL statements which are interdependent on each other; a failure in even one of them is considered a failure of the group as a whole. Thus, a transaction is said to be successful only if "all" the individual statements within it are executed successfully.

You might find it hard to think of situations where this "all−for−one, one−for−all" approach would be useful. In reality, transactions abound all around us, in bank transfers, stock trades, Web−based shopping carts, inventory control... the list goes on and on. In all these cases, the success of the transaction depends on a number of interdependent actions executing successfully and in harmony with each other; a failure in any of them must cancel the transaction and return the system back to its earlier, pre−transaction state.

Consider, for example, the simple example of a bank account transfer. Let's assume Joe's just moved to a new city and his Mum wants to give him 00 to help him get settled. She strolls down to her neighborhood bank and asks them to transfer 00 to Joe's account, debiting her own account for the same. The bank, suspicious like all good banks are, first checks to make sure she has 00 in her account. When they find that she does, they subtract 00 from her account balance and add 00 to Joe's account.

If you were to diagram the transaction above, it might look like this.

A failure in any of the steps above − say, for example, if the bank debited Joe's Mum's account but wasn't able to credit the same amount to Joe's account − would imply the failure of the entire transaction. In the event of such a failure, the transaction would be cancelled, and the system (in this example, the bank balances of Joe and his mother) would be
Thus, by offering a way to group a series of database actions together into a single unit, and by adding intelligence at the database level to "undo" the effects of failed database operations and revert the system to a stable state, transactions play an important role in helping SQL developers build more robust applications.

The Acid Test
Any RDBMS which supports transactions must conform to the so−called "ACID rules", which specify the fundamental principles for truly secure transactions.

- ** Atomicity**: The transaction must be treated as an indivisible unit, and all the statements within it must be successful for the transaction to be considered successful. In the event of a transaction failure, the system should be returned to its pre−transaction state.

- **Consistency**: Once a transaction has been completed, the system must be in a consistent state, with all of its integrity constraints satisfied.

- **Isolation**: The changes made by a transaction should be invisible to other transactions while it is in progress.

- **Durability**: Once a transaction has been completed, the changes it has wrought must be remembered by the system even in the event of a system failure.

Is MySQL ACID−compliant? Well, yes and no.

You see, MySQL supports a number of different table formats. There's the ancient ISAM format, the newer MyISAM format (which is intended to supplant the ISAM format), the BerkeleyDB format, and the latest entrant, the InnoDB format. Each of these table formats has its own advantages and disadvantages, and not all of them support transactions in the ACID−recommended manner.

As of MySQL 4.0, native ACID−compliant transactions are only possible with InnoDB and BerkeleyDB tables; for other table types, transactional environments need to be implemented at the application level, through the use of table locks or other mechanisms. My focus in this tutorial will primarily be on transactions with the InnoDB table type; however, for users constrained to older table types, a brief discussion of how to simulate a transactional environment will appear in the second part of this tutorial.

Turning the Tables
Now that you know the basics, let's take a little break from the theory and dive into the gritty reality of transactions. The first step is to create some InnoDB tables to use as a base for development. Before you can do this, though, you need to check if your MySQL build supports InnoDB tables. You can verify this by checking the "have_innodb" variable on a running MySQL server, as below:

```
mysql> SHOW VARIABLES LIKE 'have_innodb';
+--------------------------+
| Variable_name | Value |
+--------------------------+
| have_innodb | YES |
+--------------------------+
1 row in set (0.00 sec)
```

For BDB support, look for the "have_bdb" variable:

```
mysql> SHOW VARIABLES LIKE 'have_bdb';
+--------------------------+
| Variable_name | Value |
+--------------------------+
| have_bdb | YES |
+--------------------------+
1 row in set (0.00 sec)
```
Most recent MySQL binary distributions should support InnoDB out of the box. In case yours doesn't, or if you custom-built your MySQL server, you'll need to recompile it after adding the "−−with−innodb" parameter to the configure script.

Now, even if InnoDB support is enabled, it doesn't mean that MySQL will create new tables using that format automatically. By default, when MySQL creates a new table, it does so using the MyISAM table format. As noted on the previous page, this table type does not support transactions. In order to tell MySQL to create an InnoDB table, therefore, it becomes necessary to add the optional TYPE clause to your CREATE TABLE command.

The following example illustrates, by creating an InnoDB table to store user names and passwords:

```sql
mysql> CREATE TABLE users (  
    -> id int(8) NOT NULL auto_increment,  
    -> name varchar(255) NOT NULL default '',  
    -> pass varchar(255) NOT NULL default '',  
    -> PRIMARY KEY (id)  
    -> ) TYPE=InnoDB;
Query OK, 0 rows affected (0.16 sec)
```

In a similar manner, create two other tables, one for group memberships and the other for mailbox configuration.

```sql
mysql> CREATE TABLE groups (  
    -> uid int(8) NOT NULL default '0',  
    -> grp varchar(255) NOT NULL default ''  
    -> ) TYPE=InnoDB;
Query OK, 0 rows affected (0.09 sec)
```

```sql
mysql> CREATE TABLE mailboxes (  
    -> uid int(8) NOT NULL default '0',  
    -> host varchar(255) NOT NULL default '',  
    -> mboxname varchar(255) NOT NULL default '',  
    -> mboxpass varchar(255) NOT NULL default ''  
    -> ) TYPE=InnoDB;
Query OK, 0 rows affected (0.10 sec)
```

In case you already have MyISAM tables that you need to convert to InnoDB format for transactional usage, you might think that the only way is to manually back up the table data, drop the table, recreate it in InnoDB format and reinser the records. You'd be wrong – MySQL allows you to alter the table type on the fly using the very cool ALTER TABLE command, as below:

```sql
mysql> ALTER TABLE mailboxes TYPE=INNODB;
Query OK, 1 row affected (0.13 sec)
Records: 1 Duplicates: 0 Warnings: 0
```

Obviously, you can also do this with the BDB (or any other) table type.

With the InnoDB tables created, let's move on to an actual transaction.

**A Question of Commitment**

There are three main phases in the life cycle of a transaction. Here's a flowchart illustrating these phases.

Let's now look at each of these in detail.

1. Starting the transaction: In order to initiate a transaction, MySQL offers the BEGIN WORK command. When you issue this command, MySQL assumes that everything following it is part of a transaction, and therefore subject to ACID rules.
mysql> BEGIN WORK;
Query OK, 0 rows affected (0.00 sec)

You can also abbreviate this command to just BEGIN, or use the equivalent START TRANSACTION command.

2. Performing the transaction: Once the transaction has been initiated, you can use regular SQL commands to manipulate the database. The isolation principle ensures that the changes you make are not visible to other users of the database. Actually, that's not completely true, but accept it for the moment and I'll discuss the caveats a little further along.

In this example, let us assume that a transaction consists of adding a new user to the system, using the following steps:

1. Create a new user record in the "users" table with the user's name and password.

mysql> INSERT INTO users (name, pass) VALUES ('alan',
mysql> PASSWORD('ferret'));
Query OK, 1 row affected (0.06 sec)

mysql> SELECT * FROM users;
+---------+--------+--------+
| id      | name   | pass   |
|---------+--------+--------+
| 1       | alan   | 5af23f026beddb81 |
+---------+--------+--------+
1 row in set (0.02 sec)

2. Set the user's group memberships using the ID generated by the first step.

mysql> INSERT INTO groups (uid, grp) VALUES (LAST_INSERT_ID(), 'hr'),
(LAST_INSERT_ID(), 'admin');
Query OK, 2 rows affected (0.00 sec)
Records: 2 Duplicates: 0 Warnings: 0

mysql> SELECT * FROM groups;
+-+-----+
| uid | grp |
|-----+-----+
| 1   | hr  |
| 1   | admin |
+-----+
2 rows in set (0.00 sec)

3. Set up the user's mailbox using the ID generated by the first step.

mysql> INSERT INTO mailboxes (uid, host, mboxname, mboxpass) VALUES
(LAST_INSERT_ID(), 'my.pop.server', 'my.name', 'my.pass'); Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM mailboxes;
+-------+-------+--------+--------+
| uid   | host  | mboxname | mboxpass |
|-------+-------+--------+--------+
| 1     | my.pop.server | my.name | my.pass |
+-------+-------+--------+--------+
1 row in set (0.01 sec)
3. Ending the transaction: Once the steps involved in the transaction have been completed, you have a choice. You can either save the changes made by the entire transaction, or you can undo everything you just did and revert the tables back to the state they were in before you issued the BEGIN WORK command.

Let’s try cancelling first:

```
mysql> ROLLBACK;
Query OK, 0 rows affected (0.03 sec)
```

The ROLLBACK command cancels the transaction and reverts the system back to its initial state. This can be clearly verified by checking the various tables with SELECT queries – you’ll see that the changes made by the INSERT statements above have not registered at all:

```
mysql> SELECT * FROM users;
Empty set (0.01 sec)
```

```
mysql> SELECT * FROM groups;
Empty set (0.01 sec)
```

```
mysql> SELECT * FROM mailboxes;
Empty set (0.00 sec)
```

What about swinging the other way and saving the changes to disk once the transaction is done? Start the transaction again, insert the records as above, and when you’re done, save the changes with the COMMIT command:

```
mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)
```

And now, when you check the various tables again, you’ll see that the user records have all been saved to the system:

```
mysql> SELECT * FROM users;
+-------+---------+--------+
| id    | name    | pass   |
|-------+---------+--------+
| 3     | alan    | 5af23f026beddb81 |
+-------+---------+--------+
1 row in set (0.01 sec)
```

```
mysql> SELECT * FROM groups;
+-------+------+
| uid   | grp   |
|-------+------|
| 3     | hr    |
| 3     | admin |
+-------+------+
2 rows in set (0.01 sec)
```

```
mysql> SELECT * FROM mailboxes;
+-------+--------+--------+---------+
| uid   | host   | mboxname | mboxpass |
|-------+--------+--------+---------+
| 3     | my.pop.server | my.name | my.pass |
+-------+--------+--------+---------+
1 row in set (0.00 sec)
```

Typically, an application developer would write code to check for errors during a transaction and generate a
ROLLBACK if any occur (an example of such an application can be found in the second part of this article). The COMMIT and ROLLBACK commands thus play an important role in satisfying the atomicity principle, and in ensuring the integrity of the database at all times.

Rules of the Game

It is important to note that MySQL does not permit nested transactions. As in the example below, if you start a new transaction without ending the previous one with a COMMIT or ROLLBACK, MySQL will automatically commit the previous transaction’s data to disk before beginning a new transaction.

```sql
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.03 sec)

mysql> INSERT INTO users (name, pass) VALUES ('john', PASSWORD('john'));
Query OK, 1 row affected (0.57 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM users;
+---+----+--------------+
| id | name | pass         |
+---+----+--------------+
| 3 | alan| 5af23f026beddb81 |
| 4 | john| 2ca0ede551581d29 |
+---+----+--------------+
2 rows in set (0.26 sec)
```

A number of other SQL commands also perform such an implicit COMMIT – dropping, creating and altering tables; dropping and creating indexes; and dropping and creating databases.

That said, it is interesting to note that if you begin a transaction, but exit the session before issuing a COMMIT or ROLLBACK, MySQL does *not* perform an automatic COMMIT; instead, it issues a ROLLBACK. The following example demonstrates:

```
[me@host]$ mysql
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 5 to server version: 4.0.12-max-debug
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE master;
Database changed

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM users;
+---+----+--------------+
| id | name | pass         |
+---+----+--------------+
| 3 | alan| 5af23f026beddb81 |
| 4 | john| 2ca0ede551581d29 |
+---+----+--------------+
2 rows in set (0.00 sec)

mysql> INSERT INTO users (name, pass) VALUES ('tim', PASSWORD('hoo'));
```

Now, when you open a new session, and check the "users" table, you'll see that there is no record for Tim, since MySQL issued a ROLLBACK when the previous session exited.

MySQL operates in what is known as "autocommit mode". Simply, this means that MySQL treats every single SQL command as a single-statement transaction, and internally issues a COMMIT after each query to save it to disk. If this is not what you require, you can turn this feature off, by setting the special AUTOCOMMIT variable to 0, as below:

mysql> SET AUTOCOMMIT=0;
Query OK, 0 rows affected (0.03 sec)

Once this is done, you will need to explicitly issue a COMMIT after every command to have its modifications saved to disk. A failure to do so will result in all your changes automatically being rolled back when you exit the session.

mysql> SET AUTOCOMMIT=0;
Query OK, 0 rows affected (0.03 sec)

mysql> SELECT * FROM users;
+---------+-------+--------+
| id | name   | pass   |
+---------+-------+--------+
| 3 | alan   | 5af23f026beddb81 |
| 4 | john   | 2ca0ede551581d29 |
+---------+-------+--------+
2 rows in set (0.00 sec)

mysql> INSERT INTO users (name, pass) VALUES ('tim', PASSWORD('hoo'));
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO users (name, pass) VALUES ('jim', PASSWORD('i283kh'));

This is of particular relevance to the next example – so flip the page and let's see how.
You can always restore the default setting by setting the AUTOCOMMIT variable back to 1, as below:

```
mysql> SET AUTOCOMMIT=1;
Query OK, 0 rows affected (0.03 sec)
```

Obviously, the AUTOCOMMIT variable has no impact on non-transactional table types like MyISAM; every change made to those tables is saved to disk immediately and cannot be rolled back.

**Time Out**

And that's about it for the moment. Over the last few pages, I introduced you to transactions in MySQL, explaining the theory behind the transactional model and showing you how to implement a transactional environment with MySQL's InnoDB tables. I also demonstrated one of MySQL's quirks – automatic COMMITs and ROLLBACKs in particular situations – and showed you how to control automatic commits with the AUTOCOMMIT variable.

In the second part of this article, I'll be exploring the MySQL transactional model in a multi-user scenario, illustrating some of the data corruption problems that raise their ugly heads in this environment and showing you how you can use MySQL's isolation levels to reduce their likelihood. I'll also show you how to leverage off the transaction model to build more robust SQL applications, with a sample Perl/DBI application, and explain how to use transactions with non-transactional MyISAM tables. All that and more, next time...so make sure you come back!

Note: All examples in this article have been tested on MySQL 4.0.14. Examples are illustrative only, and are not meant for a production environment. Neither Melonfire nor DevShed provide warranties or support for the source code described in this article.

This article copyright Melonfire 2000–2003. All rights reserved.