Objectives:

To understand and use commit and rollback, and to learn and use the basic syntax of MySQL's Data Manipulation Language (DML), to insert, update and delete data from existing tables.

PART I – Transaction Control with Commit, Rollback and Savepoint

One of the first things a new MySQL user must understand is that data added to, modified in, or deleted from a table are saved immediately, this option can be disabled when using the command-line interface and can be very useful for implementing save points and rolling back changes. MySQL allows users to make a series of data-changing statements together as one logical unit of work. This unit of work, referred to as a transaction, begins when a new transaction is started and ends when the transaction is either committed or rolled back. Because of MySQL's transaction control logic, a user may make numerous changes to the data in tables. If the user is satisfied with the changes made, the data can be saved in the database. Alternatively, if a problem is discovered with the changes made, all changes made can be discarded. Either alternative ends the transaction.

DISABLE AUTOCOMMIT

By default MySQL starts in autocommit mode, what this means is any time you run an SQL query any changes that take place are automatically changed. You can disable autocommit by issuing the following command during your MySQL session.

```
SET autocommit = 0;
```

START A NEW TRANSACTION

You can start a new transaction in MySQL by issuing the BEGIN command or, when you start a transaction even if you set autocommit to 0, the value is automatically set to 0 until you COMMIT or ROLLBACK at which point autocommit reverts to its previous value.

```
START TRANSCATION;
```

COMMIT

When the user issues a commit statement, it is a signal to the database that all changes made by the user during the current transaction contain no mistakes, and that the user is ready to save the work in the database. The syntax couldn't be simpler:

```
COMMIT;
```

When executed, the commit statement ends the current transaction and begins a new transaction.

ROLLBACK

When the user issues a rollback statement, it is a signal to the database that all changes made by the user during the current transaction should be discarded. Again, the syntax is simple:

```
ROLLBACK;
```

You may be asking, "Can changes be rolled back after a commit?". The answer is no. Committed data can be updated or deleted, however, as you will see in the next Part of this Lab.

SAVEPOINT

Occasionally, a user working with long transactions may wish to designate the statements completed to that point as being correct, but to continue working on the transaction without pausing to commit. To do so, the user would issue a savepoint command:

```
SAVEPOINT ok_so_far;
```

which consists of the 'savepoint' keyword, followed by a space and a user-selected name for the savepoint. It may be useful for you to think of a savepoint as a kind of bookmark that you can set as a placeholder.

The use of commit, rollback and savepoint is illustrated in the following example.

```
UPDATE shipments
SET ship_cost = 10.95
WHERE to_city = 'Salt Lake City';

SAVEPOINT ok_so_far;

UPDATE shipments
SET ship_cost = 14.95;

ROLLBACK TO ok_so_far;
COMMIT;
```

PART II – The Data Manipulation Language

The Data Manipulation Language (DML) allows data in a table to be modified or *manipulated*. There are three principal elements of DML: INSERT, UPDATE and

DELETE. In addition, the SELECT keyword, is classified as an element of DML.

INSERT

It is important to recognize that when a MySQL table is created, there is no data in the table at all. Data has to be inserted into the table. You will recognize the following output as being the result of a describe of the shipment_items table:

```
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| item_id     | int(8)       | NO   | PRI | NULL    | auto_increment |
| shipment_id | int(8)       | NO   | MUL | NULL    |                |
| quantity    | int(4)       | NO   |     | NULL    |                |
| description | varchar(40)  | YES  |     | NULL    |                |
| height      | decimal(6,2) | YES  |     | NULL    |                |
| width       | decimal(6,2) | YES  |     | NULL    |                |
| weight      | decimal(6,2) | YES  |     | NULL    |                |
| ship_cost   | decimal(7,2) | YES  |     | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
```

There are two ways to insert data into an MySQL table. The first way does not require the columns of the table to be explicitly mentioned in the INSERT statement, but does require that a value be inserted for each column. Note also that each value inserted must be of the correct data type (refer to the describe output). So, let’s insert a new record into the table:

```
INSERT INTO shipment_items
VALUES (0, 20, 3, ‘Our Best New Widget’, 20, 30, 20, 12.50);
```

Notice the keywords INSERT INTO followed by the table_name, which is followed by the keyword VALUES. There are 8 data elements enclosed by parentheses in the example, one data element for each column in the table. Notice also that each data element is of the correct datatype (numbers and a text string). The data are entered by MySQL into the table in the order listed in the command. For example, shipment_id is set to 21, quantity is set to 3, etc. Note: ITEM_ID is set to 0 in the insert command and is updated to appropriate value by MySQL.

The second way of inserting data into a table requires that each column in the table be explicitly mentioned in the INSERT statement, but allows for missing data in certain circumstances. Note that the circumstances under which missing data are tolerated by MySQL are limited to columns having no NOT NULL setting. NOT NULL is an integrity constraint requiring data to be entered in each field that is so declared.

The following example illustrates the second way to insert data into a table. Note that only a portion of the columns are represented in the example, but that all the NOT NULL columns have appropriate data inserts:

```
INSERT INTO shipment_items
(item_id, shipment_id, quantity, description, weight, ship_cost)
VALUES (0, 20, 3, ‘Our Other New Widget’, 20, 12.50);
```

Also note that the columns to be INSERTed are listed in parentheses immediately after the table_name (i.e., before the VALUES keyword). And, finally, note that no data were entered for the height and width fields for this record. As long as the there is no NOT NULL constraint on the field, and the columns into which data will be placed are explicitly named in the statement, this will work fine. Data for the missing fields can always be added later with the UPDATE command.

UPDATE

The UPDATE command is used to modify or change existing data in a table. If you INSERT data into a table, commit the change, and discover later that there in an error with your INSERTed data, you can always UPDATE the data. Assume that the shipping cost for 'Our Other New Widget', INSERTed in the above section is found to be in error. We can fix it with UPDATE. First we need to get the ITEM_ID for our inserted data. This can be done with the following select command:

```
SELECT item_id, description
FROM shipment_items
WHERE description = 'Our Best New Widget';
```

In my case shipment_id is 42. In your case it might have a different value. Now we can use the update command to update the cost:

```
UPDATE shipment_items
SET ship_cost = 12.75
WHERE item_id = 42;
```

The UPDATE command works with the SET keyword to modify the value of the indicated field. However, and this is extremely important, the WHERE clause is needed to limit the effect of the UPDATE command. The UPDATE command will work without the WHERE clause, but if WHERE is not specified the UPDATE command will modify all ship_cost fields to the value specified!

If you unintentionally UPDATE each field, issue a ROLLBACK (or ROLLBACK TO SAVEPOINT). If you COMMIT such an unintentional modification, you will have no choice but to manually (or with a script) UPDATE each field to its correct value. This is the problem identified in section on SAVEPOINTs above.

DELETE

The DELETE command works with the FROM keyword, and can also be limited with the WHERE clause. DELETE removes an entire record (row) from the database. DELETE is not used to remove a value from a single field (if you wish to remove a single value, UPDATE that field and SET its value to NULL). The following example illustrates the use of DELETE:

```
DELETE FROM shipment_items
WHERE ship_cost = 12.75;
```

The above statement will remove the entire row wherever ship_cost = 12.75. Is that acceptable? Perhaps it is given our present shipment_items table. However, assuming we want to DELETE the row we INSERTed and UPDATed in the above sections, a safer and better approach would be to specify a WHERE parameter unique to the specific record (row). Setting the WHERE clause to the value of the primary key for the record will ensure that only that record is affected. The primary key uses the UNIQUE and NOT NULL integrity constraints (again, more later). On that basis, a better DELETE statement would be:

```
DELETE FROM shipment_items
WHERE item_id = 42;
```

By the way, it was mentioned above that WHERE limits DELETE just as it does UPDATE. Will DELETE work without a WHERE clause? Yes it will, but:

```
DELETE FROM shipment_items;
```

without a WHERE clause will DELETE every row from the table!!!

PART III – Problems

PAY ATTENTION to the instructions! DO NOT commit or rollback an operation until instructed to do so.

1. Establish connection to mysql with user student and database ict4300

2. Enable spooling using “tee”

3. Set autocommit to 0.

4. Start a new transaction.

5. Describe the cost_table.

6. Write INSERT statements to place the following data into the cost_table:

SHIPPER	MIN_SIZE	MAX_SIZE	MAX_WEIGHT	TIME_FRAME	COST_FACTOR
US Mail	2	20	25	20	2
Fly By Night	10	100	150	5	10
Mail Box, Etc.	2	40	30	5	30
Box Right	5	40	30	5	30
Ship Aid	5	40	30	5	30

7. Write a single SELECT statement that will verify your INSERTS.

8. Write a DELETE statement to remove just the new Ship Aid record.

9. Write a SELECT statement to display the distinct SHIPPERs in the cost_table to verify that the single Ship Aid record was DELETED.

10. Write a statement to UPDATE the cost_factor to 5 for the US Mail record.

11. Write a SELECT statement to verify the UPDATED cost_factor field for the US Mail.

12. Rollback all changes to the cost_table for this transaction.

13. Write a single SELECT statement that will verify that all changes to the cost_table have been rolled back.

14. Open Notepad. Repeat Exercise (2), typing the INSERT statements in Notepad. Save the file as cost_table_inserts.sql in the same folder that you spool your output to. You have just built a script to INSERT data into the cost_table.

15. Execute the cost_table_inserts.sql script from SQL. Recall the execute command is SOURCE, and make to sure to specify the correct path the script's folder.

16. Issue a SAVEPOINT command. Name the savepoint "costs_ok";

17. Write statements to make the following UPDATEs to cost_table (be careful with WHERE):

    (1) Make max_size = 50 for US Mail and Box Right.

    (2) Make min_size = 100 for Fly By Night.

    (3) Make cost_factor = 300 for Ship Aid.

18. Write a SELECT statement to display the Shipper and each modified field to verify the changes.

19. Rollback the changes made to your savepoint.

20. Commit the changes from this transaction.

21. Write DELETE statements to remove the records INSERTed by your cost_table_inserts.sql script.

22. Cost_table should now be back to its original condition! Write a SELECT statement to verify that the new Shippers added by running cost_table_inserts.sql have been removed.