Introduction to SQL with MySQL and Workbench

## Day 02

At this point you should have a good understanding of databases, tables, records and fields. We have used the UI to interact with these different parts of the DB ecosystem. Now we transition to using more SQL. This is the code we use to work with almost every part of a database system. We will perform CRUD operations using just SQL commands. If you are asked to truncate the table use this command: TRUNCATE `products`; or use the UI.

**Content for Day02**

**Part 1 – Inserting and Editing Records**

**Part 2 – Introduction to the Where Clause**

**Part 3 – Using SELECT as a Filter**

**Part 4 – Adding a New Table and a new Constraint**

**Part 5 – Special Cases of Insert**

**Part 6 – Special Cases of Update**

**Part 7 – Deleting Rows**

# Part 1 – Inserting and Editing Records

In this section we add several more records to our Products table. Before we do that we will change the structure of the table a bit.

1. In this part, we add a new column, but not just any column, we add a column with a default value already in place. We will add a *Boolean* column to indicate whether the product is negotiable or not. We will automatically insert the False value and then allow employees to change to true as they see fit.

|  |
| --- |
| **ALTER TABLE**  **`KarlsBikes`.`products`**  **ADD COLUMN**  **negotiable**  **BOOL NULL DEFAULT**  **false;** |

So, now when we add a new record, the negotiable column will automatically get the value of false.

1. Add a new record to the table:

|  |
| --- |
| **INSERT INTO `KarlsBikes`.`products`**  **(pName, pCategory, pVendor, pDescription, quantity, costPrice, MSRP)**  **VALUES**  **('Treckie Pedal Assist 260','EAssist','Quantom Bicycles','Precision-engineered from original Jaguar Big Cycle specification.',10,472.53,908.57);** |

1. Add another record to the table but this time include the negotiable field and pass the value of true to it:

|  |
| --- |
| **INSERT INTO `products`**  **(pName, pCategory, pVendor, pDescription, quantity, costPrice, MSRP, negotiable)**  **VALUES**  **('Horch 48V','EAssist','Autoart Movement','Pedal assist electric bicycle off road capable',11,263.30,450.00, true);** |

1. By now we should have three records. What if we wanted to update **all** values at once, well we can execute an SQL statement like this:

|  |
| --- |
| **UPDATE**  **`KarlsBikes`.`products`**  **SET**  **pCategory = 'EAssisted';** |

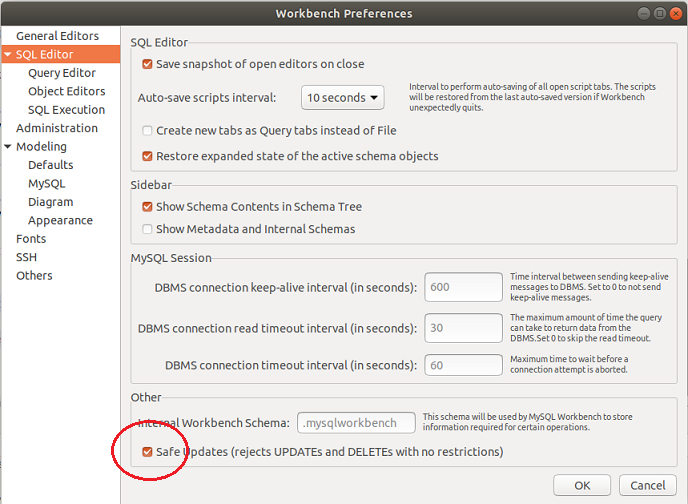
This may or may not produce an error, see next point.

1. If you did get an error like the one below, we may need to make a change in the setup of our MySQL via the UI

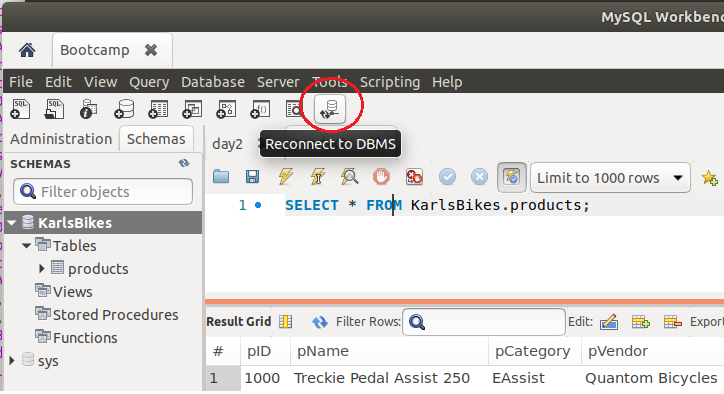
|  |
| --- |
|  |

Notice the eror message is telling us exactly what to do. Also we are doing this since we have not yet met the WHERE clause. If you know about the WHERE clause, then use it and move on otherwise go to point 6 below.

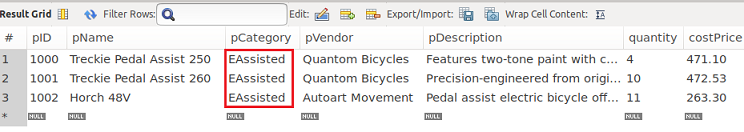
1. Click on the Edit menu item then choose Preferences and on the first dialog box, uncheck the last checkbox at the bottom of that window, Safe Updates:



Click the OK button at the bottom right to confirm. Remember to do the opposite after you have inserted the record.

1. In order for the changes to take effect you must close this connection and re-open it. There is no easy way to do this other than to close MySQL Workbench and then re-open it. However I found that if you renew the connection, it may work:  
   

The UI will respond with a confirmation page of its own, just clik on the Close button.

1. So, once the connection has been closed and reopened, try the SQL statement from Part 4 again, see my results below:   
   Originally we had EAssist, but now all three rows are EAssisted.

|  |
| --- |
| End of section |

# Part 2 – Introduction to the Where Clause

In this section we take a look at one of the most used and most powerful keyword in SQL, the ‘WHERE’ clause. This is the key code that restricts the dataset (result) of many queries. It also keeps the data safe as we will soon demonstrate.

1. In this next demo, we will execute another .sql file. Before we do that, lets truncate the records we have now, so right click on the products table and choose the option Truncate table. Confirm when asked to proceed.
2. In the code for today there should be a file called D2P2\_Products.sql. Import or just execute the SQL statements that are in that file. If you look in the Action Output window you should see something like 51 rows returned.
3. Run the select all rows on the products table. In the Result Grid window the pCategory column is now a mix of EAassisted and EAssist values. Lets change the EAssisted to just EAssist for all the fields. We could execute an update like we did in Part1. That will solve the problem of changing all the fields, but this is also a good opportunity to introduce the WHERE clause. Execute the following code and you will see that only three rows will be affected:

|  |
| --- |
| **UPDATE**  **`KarlsBikes`.`products`**  **SET**  **pCategory = 'EAssist'**  **WHERE pCategory = 'EAssisted';** |

1. This WHERE keyword is so versatile, it can be used in almost all CRUD operations. For example if we wanted to view all the records that had City in the pCategory column, we could execute this statement:

|  |
| --- |
| **SELECT**  **\***  **FROM**  **`KarlsBikes`.`products`**  **WHERE**  **pCategory = 'City';** |

The above code should return 7 rows. Note the asterisk indicates ALL records that match the criteria we asked via the WHERE clause. This means that in the future we can use the SELECT part to also manipulate the data we get back from the SQL code.

1. In a similar fashion we can get or view all the products that the vendor Quantom Bicycles sell:

|  |
| --- |
| **SELECT**  **\***  **FROM**  **`KarlsBikes`.`products`**  **WHERE**  **pVendor = 'Quantom Bicycles';** |

This code should return 14 rows.

1. We can also add mathematical comparisons to the end of the WHERE clause to further qualify the view. In the case below we want to see all inventory levels above 10 in stock:

|  |
| --- |
| **SELECT**  **\***  **FROM**  **`KarlsBikes`.`products`**  **WHERE**  **quantity > 10;** |

This should return 9 rows.

1. In fact you can do a range:

|  |
| --- |
| **SELECT**  **\***  **FROM**  **`KarlsBikes`.`products`**  **WHERE**  **quantity BETWEEN 11 AND 20;** |

This statement should return 5 rows. However we did have to employ an   
SQL Logical Operator in the keyword BETWEEN.

1. We could also expand the simple WHERE clause. We can use operators like AND, OR and NOT to fine tune our resultset:

|  |
| --- |
| **SELECT**  **\***  **FROM**  **`KarlsBikes`.`products`**  **WHERE**  **quantity NOT BETWEEN 11 AND 20;** |

This query will return 46 rows. Considering the opposite returned 5 rows and the simple SQL without any WHERE clause in it will return 51 rows, it means the queries are correct.

1. Here is an example using the AND operator:

|  |
| --- |
| **SELECT**  **\***  **FROM**  **`KarlsBikes`.`products`**  **WHERE**  **pVendor = 'Gearbox Handler'**  **AND quantity > 20;** |

This should return just two rows. Notice we are using two different fields. Using the same field for different vendors for example, involves a bit more manipulation.

1. You could also use the OR operator to get multiple vendors:

|  |
| --- |
| **SELECT**  **\***  **FROM**  **`KarlsBikes`.`products`**  **WHERE**  **pVendor = 'Gearbox Handler'**  **OR pVendor = 'Autoart Movement';** |

This should return six rows.

|  |
| --- |
| End of section |

# Part 3 – Using SELECT as a Filter

In this section we take a look at using the SELECT as a filter for the resultset. So far we have used the WHERE clause and for very complicated queries, it is possile to combine these two parts. So we can have one criteria after SELECT and a different criteria after the WHERE clause. Lets start with the SELECT clause.

1. One of the simplest SELECT is to just return one or more columns like this:

|  |
| --- |
| **SELECT**  **pName, pDescription**  **FROM**  **`KarlsBikes`.`products`;** |

This should return 51 rows and show only two columns.

1. Another useful keyword is the DISTINCT keyword. It is used in addition to SELECT in order to eliminate duplicate rows.

|  |
| --- |
| **SELECT DISTINCT**  **pDescription**  **FROM**  **`KarlsBikes`.`products`;** |

This should return 16 rows and no duplicates. Without DISTINCT you would get 51 rows.

1. If you have lots of records, you can use the TOP keyword to get just the first *n* records. The following code is typical in many vendor database software but it wont work in MySQL:

|  |
| --- |
| **SELECT TOP 3 \* FROM `KarlsBikes`.`products`;** |

For MySQL remove TOP3 and add LIMIT 3 just before the semi-colon:

|  |
| --- |
| **SELECT**  **\***  **FROM**  **`KarlsBikes`.`products`**  **LIMIT 3;** |

1. For those numeric columns, it is possible to use the SELECT clause to get minimums or maximums from the data:

|  |
| --- |
| **SELECT**  **MIN(quantity)**  **FROM**  **`KarlsBikes`.`products`;** |

Since we did have three models that did not have any inventory, we get zero.

1. Now, change the MIN to a MAX and it should return model #11. If you wanted to see more columns then the entire query would have to be restructured. The MIN and MAX keywords are actually functions and they are part of the **aggregation**, or grouping features of most databases. If you wanted to see the actual row that has that minimum value, then a sub-query would work. Of course you can also do a sort based on that particular column, coming up shortly.
2. Other functions similar to MIN and MAX are COUNT(), AVG() and SUM(). For example we can get the count of all records by running this query:

|  |
| --- |
| **SELECT**  **COUNT(quantity)**  **FROM**  **`KarlsBikes`.`products`;** |

We have to use a column name, but I could have used any column name, pName or pCategory would have returns the same result, 51. The COUNT function is used to count the number of rows based on a SELECT statement

1. On the other hand, AVG() and SUM() will need some kind of numeric data to operate on:

|  |
| --- |
| **SELECT**  **SUM(quantity)**  **FROM**  **`KarlsBikes`.`products`;** |

The above should return 393 bicycles in stock at this moment in time.

1. There are several situations where we combine keywords in both the SELECT and FROM clauses to return even more detailed views. Below we test out the GROUP BY clause:

|  |
| --- |
| **SELECT**  **COUNT(pCategory), pVendor**  **FROM**  **products**  **GROUP BY pVendor;** |

The above should return 9 rows. However it tells us theat Ryderz has three different model. Follow Me Bikes has 7 different models. In fact without more coding, we could substitute any of the columns inside of the COUNT() parameters and you would get the same results.

1. Continuing from #8, copy the name Autoart Movement and replace the field for row #28. You can do this on the query window where you got all records. Now run the same code as in #8 you will now see that Autoart Movement has two instead of one from the count() function:

|  |
| --- |
| **SELECT**  **COUNT(pCategory), pVendor**  **FROM**  **products**  **GROUP BY pVendor;** |

|  |
| --- |
| End of section |

# Part 4 – Adding a New Table and a new Constraint

Now we will add a new table to the Products table we currently have. Adding this table will help us understand relations between tables. We will establish a foreign key constraint to start with. This constraint will show that upon insertion or deletion, our data is protected and reliable.

1. This next code will create a category table. Continuing the theme of bicycle sales, each bicycle should belong to a category or two. Some bicycles can be classified as dual purpose:

|  |
| --- |
| **CREATE TABLE `productCat` (**  **`category` varchar(50) NOT NULL,**  **`textDescription` varchar(4000) DEFAULT NULL,**  **`negotiable` mediumtext,**  **PRIMARY KEY (`category`)**  **);** |

Create this table in the KarlsBikes database. Remember to refresh afterwards.

1. Once we have the table established we can insert a few categories:

|  |
| --- |
| **INSERT INTO**  **`productCat`**  **VALUES**  **('City','Made specially for Urban areas','false'),**  **('EAssist','Comes with an electric motor but pedal must engage the motor','true'),**  **('Electric','Fully electric','true'),('Hybrid','Mountain or Road','false'),**  **('Mountain','Bicycles made for rough riding over hills and inclines','false'),**  **('Road','Suberbs and light off roading','false'),('Tandem','Two or more adults or one adult and one child','false'),**  **('Urban','City bike used for delivery','true');** |

1. Since we did not add a constraint to the products table when we created it, we can do this **after the fact**. So this statement will add a tie the products and productCat tables together:

|  |
| --- |
| **ALTER TABLE products**  **ADD FOREIGN KEY (pCategory)**  **REFERENCES productCat(category);** |

1. Since we already added records in #2 lets try to delete one of those records:

|  |
| --- |
| **DELETE FROM**  **productCat**  **WHERE**  **category = 'Road';** |

Note that this statement was successful. If you look at all the records in the products table, none of them have Road in the entire column, so there is no problem to delete Road from the productCat table.

1. Now try to remove a category from the productCat table that is being used by the current products table:

|  |
| --- |
| **DELETE FROM**  **productCat**  **WHERE**  **category =**  **'City';** |

Note that this statement failed. It failed because the system realized that City is already used in the products table. Removing this field will affect several records in the product table.

1. Lets try the other side of this equation. Attempt to insert a record into the products table that does not yet have a corresponding category in the productCat table:

|  |
| --- |
| **INSERT INTO `products`**  **(pName, pCategory, pVendor, pDescription, quantity, costPrice, MSRP)**  **VALUES**  **('Treckie Road Warrior','Road','Gearbox Handler','Versatile road bike with 14 gears',2,360.00,420.00);** |

Note that this statement will also fail. If you still see Road as part of the Result Grid, remember to refresh the Grid itself.

1. Now insert the Road value into the category of the productCat table:

|  |
| --- |
| **INSERT INTO**  **`productCat`**  **VALUES**  **('Road','Used mainly in Urban areas but light off-road capable','false');** |

Note that this statement will work AND it now gives us the ability to insert the record into the products table from #6 above.

1. Lets try the insert from #6 again:

|  |
| --- |
| **INSERT INTO `products`**  **(pName, pCategory, pVendor, pDescription, quantity, costPrice, MSRP)**  **VALUES**  **('Treckie Road Warrior','Road','Gearbox Handler','Versatile road bike with 14 gears',2,360.00,420.00);** |

Now this statement will pass, since we now have a Road category

|  |
| --- |
| End of section |

# Part 5 – Special Cases of Insert

In this section we will examine how the *insert* statement can be used. We have already seen two versions, inserting with all the field columns named and inserting without the column names. You have also seen inserting records with an auto increment field and one with a timestamp. In this section we will look at inserting a value into a default field, using the null value, inserting multiple rows and inserting an apostrophe.

1. For this part, *drop* the products table you have now. The code will succeed since the constraint is the other way. We cant remove a table if parts of that table is being used in a different table.
2. Create a new one based on the following code. Notice the constraint is being created at the same time the table is being created:

|  |
| --- |
| **CREATE TABLE `products` (**  **`pID` INT NOT NULL AUTO\_INCREMENT,**  **`pName` varchar(70) NOT NULL,**  **`pCategory` varchar(10) NOT NULL,**  **`pVendor` varchar(50) NULL,**  **`pDescription` text NULL,**  **`quantity` int NOT NULL,**  **CONSTRAINT `QTY\_CHECK` CHECK ((`quantity` > 0)),**  **PRIMARY KEY (`pID`)**  **);** |

This constraint says that when we insert a new record, we must have at least one bicycle to put into inventory. Also pay attention to which columns do not need a value. These are indicated with a NULL In the creation code.

1. We will now attempt to insert a record with the four values the table requires and we are also complying with the check constraint.

|  |
| --- |
| **INSERT INTO `products`**  **VALUES ( 'Treckie Pedal Assist 250','EAssist', 'Features two-tone paint and kick stand',4);** |

This insert will fail since we did not match the number of fields exactly. The problem is the *auto increment* field, it needs a value. Also even though the vendor field does not need a value, it still needs to be accounted for in the INSERT statement.

1. If the query above is not successful, repeat with the *null* values in place where they are expected:

|  |
| --- |
| **INSERT INTO `products`**  **VALUES (null, 'Treckie Pedal Assist 250','EAssist',null, 'Features two-tone paint and kick stand',4);** |

This should succeed and shows one of the ways of using null.

1. Sometimes you may want to insert a value into a field where the value itself has an apostrophe. Under normal circumstances this will fail:

|  |
| --- |
| **INSERT INTO `products`**  **VALUES (null, 'Saunder's Big Bike','City','Large frame bicycle using nitrogen',2);** |

1. In order to fix the statement above we need to escape the apostrophe at the end of Saunder’s:

|  |
| --- |
| **INSERT INTO `products`**  **VALUES (null, 'Saunder\'s Big Bike','City','Saunders 2-Wheels','Large frame bicycle using nitrogen',2);** |

1. Using the MySQL Workbench UI, right click on the products table and choose the truncate table option. This option will delete all records, it won’t delete or drop the table itself.
2. Now we will insert the same two records but in a bulk way, so both records at the same time:

|  |
| --- |
| **INSERT INTO `products`**  **VALUES**  **(null, 'Saunder\'s Big Bike','City','Saunders 2-Wheels','Large frame bicycle using nitrogen',2),**  **(null, 'Treckie Pedal Assist 250','EAssist',null, 'Features two-tone paint and kick stand',4);** |

Notice that each record is enclosed in parenthesis and separate by commas.

1. Lets now add a new column to the products table with a default value:

|  |
| --- |
| **ALTER TABLE   `products`**  **ADD COLUMN   `costPrice` DECIMAL(8,2) NULL DEFAULT 1.00 AFTER `quantity`;** |

1. Execute the *truncate* command and now insert the two products again, this time utilizing the default keyword:

|  |
| --- |
| **INSERT INTO `products`**  **VALUES**  **(null, 'Saunder\'s Big Bike','City','Saunders 2-Wheels','Large frame bicycle using nitrogen', null, 2),**  **(null, 'Treckie Pedal Assist 250','EAssist',null, 'Features two-tone paint and kick stand', default, 3);** |

After the query runs, execute a select all and you will see that the first record went in with a null value and the second went in with the default value of 1.00

|  |  |  |
| --- | --- | --- |
| 1. Execute the *truncate* command one more time. After that edit the table either using the UI or the SQL below:  |  | | --- | | **ALTER TABLE `KarlsBikes`.`products`**  **ADD COLUMN `orderDate` DATE NULL AFTER `costPrice`,**  **ADD COLUMN `shipDate` DATETIME NULL AFTER `orderDate`;** |   This code establishes two more columns. For the orderDate column the default is the date and time that the record was inserted (see #11). For the shipDate column, we will perform a small calculation.   1. Execute the SQL below:  |  | | --- | | **INSERT INTO `products`**  **VALUES**  **(null, 'Saunder\'s Big Bike','City','Saunders 2-Wheels','Large frame bicycle using nitrogen',200, CURDATE(), CURDATE()+5, 2),**  **(null, 'Treckie Pedal Assist 250','EAssist',null, 'Features two-tone paint and kick stand',450, CURDATE(), CURDATE()+5, 1);** |   Notice that we are able to insert the date just like a timestamp that we used in Day01 Part6 . However with this approach we can actually do date arithmetic.  End of section |

# Part 6 – Special Cases of Update

The keywords used to modify data in a MySQL database are UPDATE and SET. For most updates the WHERE clause is used to restrict the updating to certain rows. Other keywords used are limit, Order by ASC and DESC.

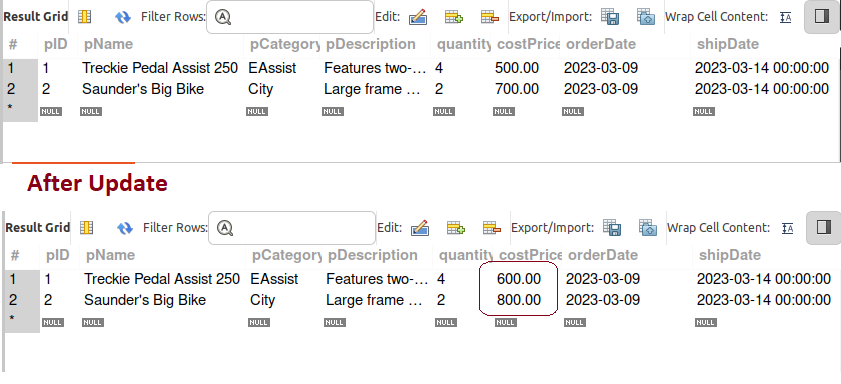
1. Lets continue using the database and table from Part 5. If you have any records in the products table, truncate them and insert just these two:

|  |
| --- |
| **INSERT INTO `products`**  **VALUES**  **(null, 'Treckie Pedal Assist 250','EAssist','Features two-tone paint and kick stand',4, 500, CURDATE(), CURDATE()+5),**  **(null, 'Saunder\'s Big Bike','City','Large frame bicycle using nitrogen',2, 700, CURDATE(), CURDATE()+5);** |

1. Management has ordered that due to inflation, we will add $100 to each bicycle in stock. To accomplish this we can do an update and arithmetic, in one go:

|  |
| --- |
| **UPDATE products**  **SET costPrice = costPrice + 100;** |

Notice that you may get a safe update error message. You can turn this off in the UI for this example.

1. Below is a before and after image. Notice that the costPrice values have all risen by $100:  
   
2. Its quite possible to update just one or maybe just a group of records. This is where that WHERE clause comes in handy again. Lets say that we need to increase the *electric assist* bicycles even more due to parts shortage, we can employ the where clause like this:

|  |
| --- |
| **UPDATE products**  **SET costPrice = costPrice + 100**  **WHERE pCategory = "EAssist";** |

1. For the next few exercises lets create a new table and insert a few records:

|  |
| --- |
| **CREATE TABLE `customers` (**  **`customerNumber` int NOT NULL,**  **`customerName` varchar(50) NOT NULL,**  **`contactLastName` varchar(50) NOT NULL,**  **`contactFirstName` varchar(50) NOT NULL,**  **`phone` varchar(50) NOT NULL,**  **`addressLine1` varchar(50) NOT NULL,**  **`addressLine2` varchar(50) DEFAULT NULL,**  **`city` varchar(50) NOT NULL,**  **`state` varchar(50) DEFAULT NULL,**  **`postalCode` varchar(15) DEFAULT NULL,**  **`country` varchar(50) NOT NULL,**  **`salesRepEmployeeNumber` int DEFAULT NULL,**  **`creditLimit` decimal(10,2) DEFAULT NULL,**  **PRIMARY KEY (`customerNumber`)**  **);** |

1. Now insert the customers using the D2P6.sql file.

|  |
| --- |
|  |

1. Lets increase the credit limit to all customers in France by 10%

|  |
| --- |
| UPDATE customers  SET creditLimit = (creditLimit \* 1.10)  WHERE country = "France"; |

1. At this point, we can become a bit more complex. Lets say that we wanted to increase the limits of customers in the state of California, in the US but we want to do so in some kind of order. Perhaps sales reps with lower rep numbers have been with the company longer than others. We also want to do this increase on a trial basis so we limit the number of records being changed to just 5. This is what such an update will look like:

|  |
| --- |
| UPDATE customers  SET creditLimit = (creditLimit \* 1.10)  WHERE country = "USA" AND state = "CA"  ORDER BY salesRepEmployeeNumber ASC  LIMIT 5; |

Note: this may not be exactly what we do in the real life, but it is very close.

1. As you might have guessed by now, it is possible to update multiple fields. For example as we are updating the credit limit in California, maybe management has decided that just one rep is enough for the state, so we can change reps within the update:

|  |
| --- |
| UPDATE customers  SET creditLimit = (creditLimit \* 1.10), salesRepEmployeeNumber = 1165  WHERE country = "USA" AND state = "CA"  ORDER BY salesRepEmployeeNumber ASC  LIMIT 5; |

So, to update multiple columns, just add a comma then the name of the column and the new value. Of course this second updated field will be based on the where clause.

# Part 7 – Deleting Rows

DELETE works similarly to UPDATE. Of course you should pay closer attention to your delete statements as each one can eliminate the entire table. It is absolutely essential that you use a WHERE clause with your statement. Of course without a WHERE clause you are essentially running a TRUNCATE statement, all records will be deleted.

1. Using the same database and tables from parts 6 and 7 execute the following statement to delete all records from the products table:

|  |
| --- |
| **DELETE FROM products;** |

If you had the previous two rows, they will be deleted.

1. Now do the same thing for customers:

|  |
| --- |
| **DELETE FROM customers;** |

1. Insert the customers using the D2P6.sql file. You did this for part6 #6
2. Lets say the company is no longer doing business in Norway, we can delete all customers from there:

|  |
| --- |
| **DELETE FROM customers**  **WHERE country = "Norway";** |

Remember constraints, luckily we do not have any for our customers table.

1. We are limiting our business in France, we want to delete all customers who do not have a credit limit greater than 100,000:

|  |
| --- |
| **DELETE FROM customers**  **WHERE country = "France" AND creditLimit < 100000;** |

After running this query, customer number 103 should be eliminated

1. Insert the two records into the products table like we did in part6 #9.
2. Now add a CHECK constraint using ALTER TABLE. I explain the check constraint in Appendix A.

|  |
| --- |
| **ALTER TABLE**  **`products`**  **ADD CHECK**  **(quantity > 0);** |

1. Now try to edit the row like this:

|  |
| --- |
| **UPDATE products**  **SET quantity = 0;** |

This statement should fail, but the delete will pass, see #9. Also check that the table actually have records, if not insert the two from Part 6 #1.

1. Now try to delete one of the rows:

|  |
| --- |
| **DELETE FROM products**  **WHERE pID = 3;** |

This statement should pass

# Appendix A – The Check Constraint

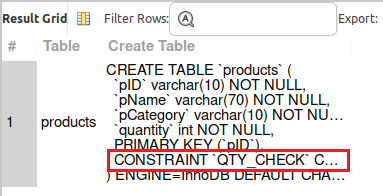
In this section we will examine another constraint called the *check constraint*. With this constraint we perform a function on the column and based on the result of that function the transaction will continue or not. For example we may not want to sell a bicycle if the inventory is less than 2.

1. In the following code we establish a products table with the check constraint already in the statement:

|  |
| --- |
| **CREATE TABLE `products` (**  **`pID` varchar(10) NOT NULL,**  **`pName` varchar(70) NOT NULL,**  **`pCategory` varchar(10) NOT NULL,**  **`quantity` int NOT NULL,**  **CONSTRAINT `QTY\_CHECK` CHECK ((`quantity` > 0)),**  **PRIMARY KEY (`pID`)**  **);** |

1. In order to verify that the constraint is there, you can run the *create table* command:

|  |
| --- |
| **SHOW CREATE TABLE products;** |

Note: this command can be used in many situations, to check the structure of a table. Below is an image of how the constraint showed up after running this statement:  
  


1. We will now attempt to insert a record with 0 quantity just to see the error code we get:

|  |
| --- |
| **INSERT INTO**  **products**  **VALUES**  **(100,"Super Cycle", 0);** |

1. If the query above is not successful, you should see error code 3819:

|  |
| --- |
| **);** |

1. Notice that when we created this constraing at #1 above, we gave it a name. That name also showed up in the error message. We can therefore use that name to remove the constraint:

|  |
| --- |
| **ALTER TABLE products DROP CHECK QTY\_CHECK;** |

1. In order to verify that the constraint is gone, you can run the create table command:

|  |
| --- |
| **show create table products;** |

1. This is what this statement revealed, notice that there are no constraints:

|  |
| --- |
|  |

1. Lets now add back in the constraint now that the table is present. Notice that I did not give the constraint a name, we will see what the software give us for a name:

|  |
| --- |
| **ALTER TABLE**  **`products`**  **ADD CHECK**  **(quantity > 0);** |

1. Below is the same command we ran in #6 , notice that the software gave us a default constraint name:

|  |
| --- |
|  |

1. Try adding a record with a zero quantity, then add a record with any number above 0. The second attempt will pass but not the first.

|  |
| --- |
| **insert into**  **products**  **values**  **(100,"Super Cycle", 2);** |

|  |
| --- |
| End of section |