# MySQL

- A MySQL database server contains many databases (or schemas).
- Each database consists of one or more tables.
- A table is made up of columns (or fields) and rows (records).

First let see how to check the databases available in the server. You can use **`SHOW DATABASES`** command to list all the existing databases in the server.

Please note that the SQL keywords and commands are NOT case-sensitive.

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
+--------------------+
4 rows in set (0.00 sec)
```

- The databases "`mysql`", "`information_schema`" and "`performance_schema`" are system databases used internally by MySQL.
- A "`test`" database is provided during installation for your testing.

### 1) Creating & Deleting a Database:

- Now let us create our own database, and I would like to begin with a simple example - a *product sales database*.
- A product sales database typically consists of many tables, e.g., products, customers, suppliers, orders, payments, employees, among others.
- Let's name our database as "`product_sales`". For example:
- You can create a new database using SQL command "`CREATE DATABASE databaseName`";

```
mysql> CREATE DATABASE product_sales;
Query OK, 1 row affected (0.03 sec)

It creates a new database called product_sales.
```

- and delete a database using "`DROP DATABASE databaseName`".

```
mysql> DROP DATABASE product_sales;
Query OK, 0 rows affected (0.11 sec)
This command removes the product_sales database.
```

- You could optionally apply condition "IF EXISTS" or "IF NOT EXISTS" to these commands.

```
mysql> CREATE DATABASE IF NOT EXISTS product_sales;
Query OK, 1 row affected (0.01 sec)

This will create a database only if a product_sales database does not exist
in the server.
```

```
Similarly you can add IF EXISTS to the command. Lets see on example.
mysql> DROP DATABASE IF EXISTS product_sales;
Query OK, 0 rows affected (0.00 sec)

This command will delete the database only if product_sales database exists in the
db server.
```

One important point to notice here is use SQL DROP commands with extreme care, because once you delete the entity, then there is no way to recover the database as the deleted entities are irrecoverable. **THERE IS NO UNDO here**

**2) Setting the default database:**

- Now let us see how to select a default database

```
-- Let me Create the database " product_sales "
mysql> CREATE DATABASE product_sales;
Query OK, 1 row affected (0.01 sec)

-- Lets confirm that " product_sales " database has been created.
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| product_sales          |
| ......             |
+--------------------+



-- lets Set " product_sales " as the default database

mysql> USE product_sales;
Database changed
```

- The command "`USE databaseName`" selects a particular database as the default (or current) database.
- Now you can refer to a table in the default database using `tableName` directly; but you need to use the fully-qualified `databaseName.tableName` to refer to a table NOT in the default database.
- In our example, we have a database named "`product_sales`" with a table named "`products`".
- If we issue "`USE product_sales`" to set `product_sales` as the default database, we can simply call the table as "`products`".
- Otherwise, we need to reference the table as "`product_sales.products`".
- To display the current default database, issue command "`SELECT DATABASE()`".

```
-- it Shows the current (default) database
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| product_sales |
+------------+

-- In order to show all the tables in the current database. Use “show tables” command.
mysql> SHOW TABLES;
Empty set (0.00 sec)
-- It shows empty set because "product_sales" has no tables.
```

### 3) Creating and deleting a table:

- Now lets create a new table in product_sales database
- You can create a new table *in the default database* using command "`CREATE TABLE tableName`" and you can delete the table using the command "`DROP TABLE tableName`".
- You can also apply condition "`IF EXISTS`" or "`IF NOT EXISTS`".
- To create a table, you need to define all its columns, by providing the columns' *name*, *type*, and *attributes*.

```
-- Lets create the table "products". Let me just write the command for creating a table now, I will explain you the command in a minute.

mysql> CREATE TABLE products (
         productID    INT UNSIGNED  NOT NULL AUTO_INCREMENT,
         productCode  CHAR(3)       NOT NULL DEFAULT '',
```

```
          name         VARCHAR(30)   NOT NULL DEFAULT '',
          quantity     INT UNSIGNED  NOT NULL DEFAULT 0,
          price        DECIMAL(5,2)  NOT NULL DEFAULT 0.00,
          PRIMARY KEY  (productID)
        );

create table products (productID int unsigned not null auto_increment, productCode
char(3) not null default '', name varchar(30) not null default '', quantity int
unsigned not null default 0, price decimal(5,2) not null default 0.00, primary
key(productID));

Query OK, 0 rows affected (0.08 sec)

-- lets confirm that the "products" table has been created
mysql> SHOW TABLES;
+---------------------+
| Tables_in_ product_sales |
+---------------------+
| products            |
+---------------------+
```

```
-- You can use “Describe” or “Desc” command to describe the fields (columns) of
the "products" table
mysql> DESCRIBE products;
+-------------+------------------+------+-----+------------+----------------+
| Field       | Type             | Null | Key | Default    | Extra          |
+-------------+------------------+------+-----+------------+----------------+
| productID   | int(10) unsigned | NO   | PRI | NULL       | auto_increment |
| productCode | char(3)          | NO   |     |            |                |
| name        | varchar(30)      | NO   |     |            |                |
| quantity    | int(10) unsigned | NO   |     | 0          |                |
| price       | decimal(5,2)     | NO   |     | 0.00   |                |
+-------------+------------------+------+-----+------------+----------------+
```

We define 5 columns in the table products: productID, productCode, name, quantity and price. The data types are:

- productID is INT UNSIGNED (INT means integer, it accepts only integer values for productid, otherwise it will throw you an error. And the number 10 after INT represents the size of the integer, here in this case productID accepts an integer of maximum size 10. And the attribute UNSIGNED means non-negative integers, which means the productid will accept only positive integers. If you don’t specify the attribute UNSIGNED in the command, by default it will take SIGNED attribute which accepts both positive and negative integers.

- productCode is CHAR(3) – CHAR(3) means a fixed-length alphanumeric string of 3 characters. The productCode accepts only 3 characters.

- `name` is `VARCHAR(30)` - a variable-length string of up to 30 characters. We use fixed-length string for `productCode`, as we assume that the `productCode` contains exactly 3 characters. On the other hand, we use variable-length string for `name`, as its length varies – `VARCHAR` is more efficient than `CHAR`.
- `quantity` is also `INT`.
- `price` is `DECIMAL(10,2)` - a decimal number with 2 decimal places. `DECIMAL` type is recommended for currency.

- The attribute "`NOT NULL`" specifies that the column cannot contain the `NULL` value. `NULL` is a special value indicating "no value", "unknown value" or "missing value".
- We also set the default values of all the columns. The column will take on its default value, if no value is specified during the record creation.
- We set the column `productID` as *primary key*.
- Values of the primary-key column must be unique.
- Every table shall contain a primary key. This ensures that every row can be distinguished from other rows.
- We set the column `productID` to `AUTO_INCREMENT`. with default starting value of 1.
- When you insert a `NULL` (recommended) (or 0, or a missing value), into an `AUTO_INCREMENT` column, the maximum value of that column plus 1 would be inserted.

**4) Inserting the data:**

We can use the `INSERT INTO` statement to insert a new row with all the column values

- Let's fill up our "`products`" table with rows.

```
mysql> INSERT INTO products VALUES (1001, 'IPH', 'Iphone 5S Red', 5000,
600.23);
Query OK, 1 row affected (0.04 sec)
```

- You need to list the values in the same order in which the columns are defined in the `CREATE TABLE`, separated by commas.
- For columns of string data type (`CHAR`, `VARCHAR`), enclosed the value with a pair of single quotes (or double quotes).
- For columns of numeric data type (`INT`, `DECIMAL`, `FLOAT`, `DOUBLE`), simply place the number. You don't have to enclose them with quotes.
- We set the `productID` of the first record to 1001, and use `AUTO_INCREMENT` for the rest of records by inserting a `NULL`, or with a missing column value.

```
-- Lets Insert multiple rows at a time
mysql> INSERT INTO products VALUES(NULL, 'IPH', 'Iphone 5S Black',  8000,
655.25),(NULL, 'IPH', 'Iphone 5S Blue',  2000, 625.25);

Query OK, 2 rows affected (0.03 sec)
Records: 2  Duplicates: 0  Warnings: 0
 -- Please note that Inserting NULL to an auto_increment column results in
max_value + 1
```

```
-- Now lets see how to Insert value to selected columns
mysql> INSERT INTO products (productCode, name, quantity, price) VALUES ('SNY',
'Xperia Z1', 10000, 555.48),('SNY', 'Xperia S', 8000, 400.49);

Query OK, 2 row affected (0.03 sec)
 -- Here we didn't specified the productID, as Missing value for an auto_increment
column also results in max_value + 1

-- now lets check another scenario
mysql> INSERT INTO products (productCode, name) VALUES ('SNY', 'Xperia U');
Query OK, 1 row affected (0.04 sec)
Here all the missing columns get their default values

-- Now lets query the table to check the rows inserted into the table.
mysql> SELECT * FROM products;
# productID, productCode, name, quantity, price
'1001', 'IPH', 'Iphone 5S Red', '5000', '600.23'
'1002', 'IPH', 'Iphone 5S Black', '8000', '655.25'
'1003', 'IPH', 'Iphone 5S Blue', '2000', '625.25'
'1004', 'SNY', 'Xperia Z1', '10000', '555.48'
'1005', 'SNY', 'Xperia S', '8000', '400.49'
'1006', 'SNY', 'Xperia U', '0', '0.00'

I will explain about the select command in a minute. We have inserted 6 rows in to
the products table.

-- We use delete command to remove a row from the table.
mysql> DELETE FROM products WHERE productID = 1006;

It removes the productID 1006 row from the table.
```

**5) Querying the database:**

Coming back to the select query, we use it to query the database. The syntax for SELECT command is:

```
SELECT column1Name, column2Name, ... FROM tableName
   -- to List all the rows of the specified columns
(Or)
SELECT * FROM tableName
-- to List all the rows of ALL columns, * is a wildcard denoting all
columns

(or)

SELECT column1Name, column2Name,... FROM tableName WHERE criteria
SELECT * FROM tableName WHERE criteria
```

```
-- to List rows that meet the specified criteria in WHERE clause
```

For examples,

```
-- to List all rows for the specified columns, here is the command
mysql> SELECT name, price FROM products;
```

```
-- to List all rows of ALL the columns. The wildcard * denotes ALL columns
mysql> SELECT * FROM products;
```

**5.1) Comparison Operators:**

Now lets see how to use comparison operators in SQL commands. For numbers (`INT`, `DECIMAL`, `FLOAT`), you could use comparison operators: `'='` (equal to), `'<>'` or `'!='` (not equal to), `'>'` (greater than), `'<'` (less than), `'>='` (greater than or equal to), `'<='` (less than or equal to), to compare two numbers.

For example, `price > 400`, `quantity <= 500`.

```
mysql> SELECT name, price FROM products WHERE price < 600;

# name, price
'Xperia Z1', '555.48'
'Xperia S', '400.49'
2 rows in set (0.00 sec)

this gives you the name and price of the products whose price is less than 600.



mysql> SELECT name, quantity FROM products WHERE quantity <= 2000;
# name, quantity
'Iphone 5S Blue', '2000'
1 row in set (0.00 sec)

this gives you the name and quantity of the products whose quantity is less than
2000.
```

For strings, you could also use '=', '<>', '>', '<', '>=', '<=' to compare two strings (e.g., productCode = 'PEC').

```
Mysql> SELECT name, price FROM products WHERE productCode = 'IPH';

# name, price
'Iphone 5S Red', '600.23'
'Iphone 5S Black', '655.25'
'Iphone 5S Blue', '625.25'
3 rows in set (0.00 sec)

Please note that the string values should always be quoted in the command

This command gives you the name and price of the products whose productCode is “IPH”
```

**5.2) String Pattern Matching:**

Now lets discuss about how to use string pattern matching in SQL commands
For strings, in addition to full matching using operators like '=' and '<>', we can perform *pattern matching* using operator LIKE (or NOT LIKE) with wildcard characters. The wildcard '_' matches any single character; '%' matches any number of characters (including zero). For example,

- 'abc%' matches strings beginning with 'abc';
- '%xyz' matches strings ending with 'xyz';
- '%aaa%' matches strings containing 'aaa';
- '___' matches strings containing exactly three characters; and
- 'a_b%' matches strings beginning with 'a', followed by any single character, followed by 'b', followed by zero or more characters.

```
-- lets check in products table, for the "name" which begins with 'Iphone'
mysql> SELECT name, price FROM products WHERE name LIKE 'Iphone%';

# name, price
'Iphone 5S Red', '600.23'
'Iphone 5S Black', '655.25'
'Iphone 5S Blue', '625.25'
```

```
-- to check for the "name" which contains the word ‘Blue’
mysql> SELECT name, price FROM products WHERE name LIKE '%Blue%';

# name, price
'Iphone 5S Blue', '625.25'
```

### 5.3)Logical Operators

Now lets see how to use Boolean operators like AND, OR in SQL commands.
Using the boolean operators `AND`, `OR` You can combine multiple conditions. For example

```
mysql> SELECT * FROM products WHERE quantity >= 5000 AND name LIKE 'Iphone%';
# productID, productCode, name, quantity, price
'1001', 'IPH', 'Iphone 5S Red', '5000', '600.23'
'1002', 'IPH', 'Iphone 5S Black', '8000', '655.25'

This gives you all the details of products whose quantity is >=5000 and the name like ‘Iphone’

mysql> SELECT * FROM products WHERE quantity >= 5000 AND price > 650 AND name LIKE 'Iphone%';
# productID, productCode, name, quantity, price
'1002', 'IPH', 'Iphone 5S Black', '8000', '655.25'
```

### 5.4) IN, NOT IN

You can select from members of a set with `IN` (or `NOT IN`) operator. This is easier and clearer than the equivalent `AND-OR` expression.

```
mysql> SELECT * FROM products WHERE name IN ('Iphone 5S Red', 'Iphone 5S Black');

# productID, productCode, name, quantity, price
'1001', 'IPH', 'Iphone 5S Red', '5000', '600.23'
'1002', 'IPH', 'Iphone 5S Black', '8000', '655.25'

This gives the product details for the names provided in the list specified in the command. i.e 'Iphone 5S Red', 'Iphone 5S Black'
```

### 5.5) BETWEEN, NOT BETWEEN

To check if the value is within a range, you could use `BETWEEN ... AND ...` operator. Again, this is easier and clearer than the equivalent `AND-OR` expression.

```
mysql> SELECT * FROM products WHERE (price BETWEEN 400 AND 600) AND (quantity BETWEEN 5000 AND 10000);

# productID, productCode, name, quantity, price

'1004', 'SNY', 'Xperia Z1', '10000', '555.48'
```

```
'1005', 'SNY', 'Xperia S', '8000', '400.49'


This command gives you the product details whose price is in between 400 and 600
and quantity in between 5000 and 10000
```

### 5.6) ORDER BY Clause

Now let see how to retrieve the data from the table in a specific order.

You can order the rows selected using `ORDER BY` clause, and its syntax is:

```
SELECT ... FROM tableName
WHERE criteria
ORDER BY columnA ASC|DESC, columnB ASC|DESC, ...
```

The selected row will be ordered according to the values in *columnA*, in either ascending (`ASC`) or descending (`DESC`) order. The default is ASC order. If several rows have the same value in *columnA*, it will be ordered according to *columnB*, and so on. For strings, the ordering could be case-sensitive or case-insensitive, depending on the so-called character collating sequence used. For examples,

```
-- Lets Order the results by price in descending order
mysql> SELECT * FROM products WHERE name LIKE 'Iphone%' ORDER BY price DESC;

# productID, productCode, name, quantity, price
'1002', 'IPH', 'Iphone 5S Black', '8000', '655.25'
'1003', 'IPH', 'Iphone 5S Blue', '2000', '625.25'
'1001', 'IPH', 'Iphone 5S Red', '5000', '600.23'
```

### 5.7) LIMIT Clause

A `SELECT` query on a large database may produce many rows. You could use the `LIMIT` clause to limit the number of rows displayed, e.g.,

```
-- Display the first two rows
mysql> SELECT * FROM products ORDER BY price LIMIT 2;

# productID, productCode, name, quantity, price
'1005', 'SNY', 'Xperia S', '8000', '400.49'
'1004', 'SNY', 'Xperia Z1', '10000', '555.48'
```

```
SELECT column FROM table LIMIT 18 OFFSET 8 – It gives 18 records from the
table starting from row 9
```

### 5.8) AS - Alias

You could use the keyword `AS` to define an *alias* for an identifier (such as column name, table name). The alias will be used in displaying the name. It can also be used as reference. For example,

```
mysql> SELECT productID AS ID, productCode AS Code, name AS Description, price AS
Unit_Price
          -- Define aliases to be used as display names
       FROM products
       ORDER BY ID;
          -- Use alias ID as reference

# ID, Code, Description, Unit_Price
'1001', 'IPH', 'Iphone 5S Red', '600.23'
'1002', 'IPH', 'Iphone 5S Black', '655.25'
'1003', 'IPH', 'Iphone 5S Blue', '625.25'
'1004', 'SNY', 'Xperia Z1', '555.48'
'1005', 'SNY', 'Xperia S', '400.49'
```

# 6)Producing Summary Reports

### 6.1) DISTINCT

A column may have duplicate values, we could use keyword `DISTINCT` to select only distinct values. We can also apply `DISTINCT` to several columns to select distinct combinations of these columns. For examples,

```
-- Without DISTINCT
mysql> select productCode from products;

# productCode
IPH
IPH
IPH
SNY
SNY

-- With DISTINCT on price
mysql> select distinct productCode from products;

# productCode
'IPH'
'SNY'
```

### 6.2) GROUP BY Clause

The `GROUP BY` clause allows you to *collapse* multiple records with a common value into groups. For example,

```
mysql> SELECT * FROM products ORDER BY productCode, productID;
# productID, productCode, name, quantity, price
'1001', 'IPH', 'Iphone 5S Red', '5000', '600.23'
```

```
'1002', 'IPH', 'Iphone 5S Black', '8000', '655.25'
'1003', 'IPH', 'Iphone 5S Blue', '2000', '625.25'
'1004', 'SNY', 'Xperia Z1', '10000', '555.48'
'1005', 'SNY', 'Xperia S', '8000', '400.49'

mysql> SELECT * FROM products GROUP BY productCode;
       -- Only first record in each group is shown
# productID, productCode, name, quantity, price
'1001', 'IPH', 'Iphone 5S Red', '5000', '600.23'
'1004', 'SNY', 'Xperia Z1', '10000', '555.48'
```

We can apply `GROUP BY` Aggregate functions to each group to produce group summary report.

The function `COUNT(*)` returns the rows selected; `COUNT(columnName)` counts only the non-`NULL` values of the given column. For example,

```
mysql> SELECT COUNT(*) AS `Count` FROM products;

# Count
'5'
-- Function COUNT(*) returns the number of rows selected

-- Now lets get the count using GROUP BY clause

mysql> SELECT productCode, COUNT(*) FROM products GROUP BY
productCode;

# productCode, COUNT(*)
'IPH', '3'
'SNY', '2'

We got 'IPH' count as 3 because we have 3 entries in our table with
the product code 'IPH' and similarly 2 entries for the product code
'SNY'
```

Besides `COUNT()`, there are many other `GROUP BY` aggregate functions such as `AVG()`, `MAX()`, `MIN()` and `SUM()`. For example,

```
mysql> SELECT MAX(price), MIN(price), AVG(price), SUM(quantity) FROM products;

       -- Without GROUP BY - All rows
# MAX(price), MIN(price), AVG(price), SUM(quantity)
```

```
'655.25', '400.49', '567.340000', '33000'

This gives you MAX price, MIN price, AVG price and total quantities of all the
products available in our products table

Now lets use GROUP BY clause

mysql> SELECT productCode, MAX(price) AS `Highest Price`, MIN(price) AS `Lowest
Price` FROM products GROUP BY productCode;

# productCode, Highest Price, Lowest Price
'IPH', '655.25', '600.23'
'SNY', '555.48', '400.49'

Which means, the highest price of a iphone product available in our database is
655.25 and the lowest price is 600.23.
Similarly the highest price of a sony product is 555.48 and lowest price is 400.49
```

## 2.7 Modifying Data

To modify existing data, use `UPDATE ... SET` command, with the following syntax:

```
UPDATE tableName SET columnName = {value|NULL|DEFAULT}, ... WHERE criteria
```

For example,

```
-- You can modify more than one values
mysql> UPDATE products SET quantity = quantity + 50, price = 600.5 WHERE name =
'Xperia Z1';

Lets check the modification in products table

mysql> SELECT * FROM products WHERE name = 'Xperia Z1';
# productID, productCode, name, quantity, price
'1004', 'SNY', 'Xperia Z1', '10050', '600.50'

You can see that the quantity of Xperia Z1 is increased by 50
```

## 2.8 Deleting Rows

Use the `DELELE FROM` command to delete row(s) from a table, the syntax is:

```
DELETE FROM tableName
-- It deletes all rows from the table.

DELETE FROM tableName WHERE criteria
```

```
-- It deletes only the row(s) that meets the criteria
```

For example,

```
mysql> DELETE FROM products WHERE name LIKE 'Xperia%';
Query OK, 2 row affected (0.00 sec)

mysql> SELECT * FROM products;
# productID, productCode, name, quantity, price
'1001', 'IPH', 'Iphone 5S Red', '5000', '600.23'
'1002', 'IPH', 'Iphone 5S Black', '8000', '655.25'
'1003', 'IPH', 'Iphone 5S Blue', '2000', '625.25'

mysql> DELETE FROM products;
Query OK, 3 rows affected (0.00 sec)

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

Beware that "`DELETE FROM tableName`" without a `WHERE` clause deletes ALL records from the table. Even with a `WHERE` clause, you might have deleted some records unintentionally. It is always advisable to issue a `SELECT` command with the same `WHERE` clause to check the result set before issuing the `DELETE` (and `UPDATE`).

## 3.1 One-To-Many Relationship

Suppose that each product has one supplier, and each supplier supplies one or more products. We could create a table called `suppliers` to store suppliers' data (e.g., name, address and phone number). We create a column with unique value called `supplierID` to identify every suppliers. We set `supplierID` as the *primary key* for the table `suppliers` (to ensure uniqueness and facilitate fast search).

To relate the `suppliers` table to the `products` table, we add a new column into the `products` table - the `supplierID`.

We then set the `supplierID` column of the `products` table as a foreign key references the `supplierID` column of the `suppliers` table to ensure the so-called *referential integrity*.

We need to first create the `suppliers` table, because the `products` table references the `suppliers` table.

```
mysql> CREATE TABLE suppliers (supplierID  INT UNSIGNED  NOT NULL AUTO_INCREMENT,
name VARCHAR(30) NOT NULL DEFAULT '', phone CHAR(8) NOT NULL DEFAULT '', PRIMARY
KEY (supplierID));

mysql> DESCRIBE suppliers;
# Field, Type, Null, Key, Default, Extra
'supplierID', 'int(10) unsigned', 'NO', 'PRI', NULL, 'auto_increment'
'name', 'varchar(30)', 'NO', '', '', ''
'phone', 'char(8)', 'NO', '', '', ''

Lets insert some data in to the suppliers table.

mysql> INSERT INTO suppliers VALUE (501, 'ABC Traders', '88881111'), (502, 'XYZ
Company', '88882222'), (503, 'QQ Corp', '88883333');

mysql> SELECT * FROM suppliers;
+------------+-------------+----------+
| supplierID | name        | phone    |
+------------+-------------+----------+
|        501 | ABC Traders | 88881111 |
|        502 | XYZ Company | 88882222 |
|        503 | QQ Corp     | 88883333 |
+------------+-------------+----------+
```

### ALTER TABLE

The syntax for `ALTER TABLE` is as follows:

```
ALTER TABLE tableName
  {ADD [COLUMN] columnName columnDefinition}
  {ALTER|MODIFY [COLUMN] columnName columnDefinition
    {SET DEFAULT columnDefaultValue} | {DROP DEFAULT}}
  {DROP [COLUMN] columnName [RESTRICT|CASCADE]}
  {ADD tableConstraint}
  {DROP tableConstraint [RESTRICT|CASCADE]}
```

Instead of deleting and re-creating the `products` table, we shall use the statement "`ALTER TABLE`" to add a new column `supplierID` into the `products` table.

```
mysql> ALTER TABLE products ADD COLUMN supplierID INT UNSIGNED NOT NULL;
Query OK, 4 rows affected (0.13 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> DESCRIBE products;

# Field, Type, Null, Key, Default, Extra
'productID', 'int(10) unsigned', 'NO', 'PRI', NULL, 'auto_increment'
'productCode', 'char(3)', 'NO', '', '', ''
```

```
'name', 'varchar(30)', 'NO', '', '', ''
'quantity', 'int(10) unsigned', 'NO', '', '0', ''
'price', 'decimal(5,2)', 'NO', '', '0.00', ''
'supplierID', 'int(10) unsigned', 'NO', '', NULL, ''
```

Next, we shall add a *foreign key constraint* on the `supplierID` columns of the `products` child table to the `suppliers` parent table, to ensure that every `supplierID` in the `products` table always refers to a*valid* `supplierID` in the `suppliers` table - this is called *referential integrity*.
Before we can add the foreign key, we need to set the `supplierID` of the existing records in the `products` table to a valid `supplierID` in the `suppliers` table (say `supplierID=501`).

```
-- Now lets set the supplierID of the existing records to a valid supplierID of supplier table
mysql> UPDATE products SET supplierID = 501;

-- Lets Add a foreign key constrain
mysql> ALTER TABLE products ADD FOREIGN KEY (supplierID) REFERENCES suppliers (supplierID);
Query OK, 4 rows affected (0.26 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> DESCRIBE products;
# Field, Type, Null, Key, Default, Extra
'productID', 'int(10) unsigned', 'NO', 'PRI', NULL, 'auto_increment'
'productCode', 'char(3)', 'NO', '', '', ''
'name', 'varchar(30)', 'NO', '', '', ''
'quantity', 'int(10) unsigned', 'NO', '', '0', ''
'price', 'decimal(5,2)', 'NO', '', '0.00', ''
'supplierID', 'int(10) unsigned', 'NO', 'MUL', NULL, ''

mysql> UPDATE products SET supplierID = 502 WHERE productID  = 1003;

mysql> SELECT * FROM products;
# productID, productCode, name, quantity, price, supplierID
'1001', 'IPH', 'Iphone 5S Red', '5000', '600.23', '501'
'1002', 'IPH', 'Iphone 5S Black', '8000', '655.25', '501'
'1003', 'IPH', 'Iphone 5S Blue', '2000', '625.25', '502'
```

## SELECT with JOIN

`SELECT` command can be used to query and join data from two related tables. For example, to list the product's `name` (in `products` table) and supplier's `name` (in `suppliers` table), we could join the two table via the two common `supplierID` columns:

```
mysql> SELECT products.name, price, suppliers.name FROM products JOIN suppliers ON products.supplierID = suppliers.supplierID WHERE price < 650;

# name, price, name
'Iphone 5S Red', '600.23', 'ABC Traders'
'Iphone 5S Blue', '625.25', 'XYZ Company'

    -- Here we need to use products.name and suppliers.name to differentiate the two "names"




-- Join via WHERE clause (lagacy and not recommended)
mysql> SELECT products.name, price, suppliers.name FROM products, suppliers WHERE products.supplierID = suppliers.supplierID AND price < 650;

# name, price, name
'Iphone 5S Red', '600.23', 'ABC Traders'
'Iphone 5S Blue', '625.25', 'XYZ Company'
```

In the above query result, two of the columns have the same heading "`name`". We could create *aliases* for headings.

```
-- Lets use aliases for column names for display
mysql> SELECT products.name AS `Product Name`, price, suppliers.name AS `Supplier Name` FROM products JOIN suppliers ON products.supplierID = suppliers.supplierID WHERE price < 650;

# Product Name, price, Supplier Name
'Iphone 5S Red', '600.23', 'ABC Traders'
'Iphone 5S Blue', '625.25', 'XYZ Company'
```