

Database Management System

IT 261

Bachelor of Information Technology

Prepared by Dr. Khaled Eskaf

Lecture 8



Revision

In SQL,

The MAX() function returns the maximum value of a column.
The MIN() function returns the minimum value of a column.

SQL MAX() Function

The syntax of the SQL MAX() function is:

```
SELECT MAX(columnn)  
FROM table;  
Here,
```

column is the name of the column you want to filter
table is the name of the table to fetch the data from

```
SELECT MAX(age)  
FROM Customers;
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT MAX(age)  
FROM Customers;
```

MAX(age)

31

The syntax of the SQL MIN() function is:

```
SELECT MIN(columnn)
```

```
FROM table;
```

Here,

column is the name of the column you want to filter

table is the name of the table to fetch the data from

```
SELECT MIN(age)
FROM Customers;
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT MIN(age)
FROM Customers;
```

MIN(age)

22

The SQL COUNT() function returns the number of records returned by a query.

Example

--returns the number of rows in the Customers table

```
SELECT COUNT(*)  
FROM Customers;
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SELECT COUNT(*)
FROM Customers;

COUNT(*)

5

In SQL, the SUM() and AVG() functions are used to calculate total and average values in numeric columns.

```
--select the sum of amount from Orders table  
SELECT SUM(amount) AS total_sales  
FROM Orders;
```

Table: Orders

order_id	item	amount	customer_id
1	Keyboard	400	4
2	Mouse	300	4
3	Monitor	12000	3
4	Keyboard	400	1
5	Mousepad	250	2

SELECT SUM(amount) AS total_sales
FROM Orders;

total_sales

13350

--select the sum of the amount of customer_id 4 from orders
SELECT SUM(amount) AS total_of_cus4
FROM Orders
WHERE customer_id = 4;

Table: Orders

order_id	item	amount	customer_id
1	Keyboard	400	4
2	Mouse	300	4
3	Monitor	12000	3
4	Keyboard	400	1
5	Mousepad	250	2

SELECT SUM(amount) AS total_of_cus4
FROM Orders
WHERE customer_id = 4;

total_of_cus4

700

```
-- get average age of customers  
SELECT AVG(age) AS average_age  
FROM Customers;
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT AVG(age) AS average_age  
FROM Customers;
```

average_age
25.6

The ORDER BY clause in SQL is used to sort the result set in ascending or descending order.

Example

```
-- orders all rows from Customers in ascending order by country  
SELECT *  
FROM Customers  
ORDER BY country;
```

Here, the SQL command selects all rows from the Customers table and then sorts them in ascending order by country.

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SELECT *
FROM Customers
ORDER BY first_name;

customer_id	first_name	last_name	age	country
5	Betty	Doe	28	UAE
3	David	Robinson	22	UK
1	John	Doe	31	USA
4	John	Reinhardt	25	UK
2	Robert	Luna	22	USA

We can use the **ASC** keyword to explicitly sort selected records in ascending order. For example,

```
-- orders all rows from Customers in ascending order by age
SELECT *
FROM Customers
ORDER BY age ASC;
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SELECT *
FROM Customers
ORDER BY age ASC;

customer_id	first_name	last_name	age	country
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE
1	John	Doe	31	USA

We use the **DESC** keyword to sort the selected records in descending order. For example,

```
-- order all rows from Customers in descending order by age
SELECT *
FROM Customers
ORDER BY age DESC;
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT *
FROM Customers
ORDER BY age DESC;
```

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
5	Betty	Doe	28	UAE
4	John	Reinhardt	25	UK
2	Robert	Luna	22	USA
3	David	Robinson	22	UK

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

In SQL, we use the GROUP BY clause to **group rows based on the value of columns.**

Example

-- count the number of customers in each country

```
SELECT country, COUNT(*) AS number  
FROM Customers  
GROUP BY country;
```

SELECT country, COUNT(*) AS number
FROM Customers
GROUP BY country;

country	number
UAE	1
UK	2
USA	2

Table: Orders

order_id	item	amount	customer_id
1	Keyboard	400	4
2	Mouse	300	4
3	Monitor	12000	3
4	Keyboard	400	1
5	Mousepad	250	2

Example: GROUP BY Amount Spent By Each Customer

-- calculate the total amount spent by each customer

```
SELECT customer_id, SUM(amount) AS total  
FROM Orders  
GROUP BY customer_id;
```

```
SELECT customer_id, SUM(amount) AS total  
FROM Orders  
GROUP BY customer_id;
```

customer_id	total
1	400
2	250
3	12000
4	700

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

We use the SQL LIKE operator with the WHERE clause to get a result set that matches the given string pattern.

Example

```
-- select customers who live in the UK
```

```
SELECT *  
FROM Customers  
WHERE country LIKE 'UK';
```

```
SELECT *  
FROM Customers  
WHERE country LIKE 'UK';
```

Note: Although the LIKE operator behaves similarly to the = operator in this example, they are not the same. The = operator is used to check **equality**, whereas the LIKE operator is used to **match string patterns only**.

customer_id	first_name	last_name	age	country
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK

We can also invert the working of the LIKE operator by using the **NOT** operator with it. This returns a result set that doesn't match the given string pattern.

-- select customers who **don't** live in the USA

```
SELECT *  
FROM Customers  
WHERE country NOT LIKE 'USA';
```

SQL LIKE With Multiple Values

We can use the LIKE operator with **multiple string patterns** using the **OR operator**. For example,

```
-- select customers whose last_name starts with R and ends with t  
-- or customers whose last_name ends with e
```

```
SELECT *  
FROM Customers  
WHERE last_name LIKE 'R%' OR last_name LIKE '%e';
```

Here, the SQL command selects customers whose last_name starts with **R** and ends with **t** or customers whose last_name ends with **e**.

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
-- select rows where the last name  
-- of customers start with R  
SELECT *  
FROM Customers  
WHERE last_name LIKE 'R%';
```

```
SELECT *  
FROM Customers  
WHERE last_name LIKE 'R%';
```

customer_id	first_name	last_name	age	country
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK

In SQL, we use the INSERT INTO statement to insert new row(s) into a database table.

```
INSERT INTO table_name(column1, column2, column3, ...)  
VALUES  
(value1, value2, value3, ...);
```

- table_name is the table where the new row(s) are inserted
- column1, column2, column3, ... are the columns where the values are to be inserted
- value1, value2, value3, ... are the values to be inserted

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK

INSERT INTO Customers(customer_id, first_name,
last_name, age, country)
VALUES (5, 'Harry', 'Potter', 31, 'USA');

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Harry	Potter	31	USA

Insert Data Only in Specified Column

If we skip column names during row insertion, the values of those columns will be NULL.

Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
INSERT INTO Customers(first_name, last_name, age)
VALUES
('Brad', 'Pitt', 58);
```

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE
6	Brad	Pitt	58	NULL

```
INSERT INTO Customers(first_name, last_name, age, country)
VALUES
('Harry', 'Potter', 31, 'USA'),
('Chris', 'Hemsworth', 43, 'USA'),
('Tom', 'Holland', 26, 'UK');
```

Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK

```
INSERT INTO Customers(first_name, last_name, age,
country)
VALUES
('Harry', 'Potter', 31, 'USA'),
('Chris', 'Hemsworth', 43, 'USA'),
('Tom', 'Holland', 26, 'UK');
```

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Harry	Potter	31	USA
6	Chris	Hemsworth	43	USA
7	Tom	Holland	26	UK

In SQL, the UPDATE statement is used to modify **existing** records in a database table.

```
UPDATE table_name  
SET column1 = value1, column2 = value2, ...  
[WHERE condition];
```

- **table_name** is the name of the table to be modified
- **column1, column2, ...** are the names of the columns to be modified
- **value1, value2, ...** are the values to be set to the respective columns
- **[WHERE condition]** is an optional clause specifying which rows should be updated

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

UPDATE Customers
SET first_name = 'Johnny'
WHERE customer_id = 1;

customer_id	first_name	last_name	age	country
1	Johnny	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Harry	Potter	31	USA

-- update multiple values in the given row

```
UPDATE Customers
```

```
SET first_name = 'Johnny', last_name = 'Depp'
```

```
WHERE customer_id = 1;
```

the SQL command changes the value of the first_name column to **Johnny** and last_name to **Depp** if customer_id is equal to **1**.

In SQL, the DELETE clause is used to delete row(s) from a database table.

Example

```
DELETE FROM Customers  
WHERE customer_id = 4;
```

Here, the SQL command will delete a row from the Customers table if its customer_id is 4.

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

DELETE FROM Customers
WHERE customer_id = 5;

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK

The WHERE clause determines which rows to delete. However, we can delete all rows at once if we omit the WHERE clause. For example,

```
DELETE FROM Customers;
```



MySQL Workbench

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 call our database "**southwind**"

We shall begin with the first table called "products" with the following columns (having data types as indicated) and rows:

Database: southwind Table: products				
productID INT	productCode CHAR(3)	name VARCHAR(30)	quantity INT	price DECIMAL(10,2)
1001	PEN	Pen Red	5000	1.23
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49

You can create a new database using SQL command "**CREATE DATABASE** databaseName"; and delete a database using "DROP DATABASE databaseName". You could optionally apply condition "IF EXISTS" or "IF NOT EXISTS" to these commands. For example,

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

```
mysql> DROP DATABASE southwind;  
Query OK, 0 rows affected (0.11 sec)
```

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

```
mysql> DROP DATABASE IF EXISTS southwind;  
Query OK, 0 rows affected (0.00 sec)
```

IMPORTANT: Use SQL DROP (and DELETE) commands with extreme care, as the deleted entities are irrecoverable. THERE IS NO UNDO!!!

-- Remove the database "southwind", if it exists.

-- Beware that DROP (and DELETE) actions are irreversible and not recoverable!

```
mysql> DROP DATABASE IF EXISTS southwind;
```

Query OK, 1 rows affected (0.31 sec)

-- Create the database "southwind"

```
mysql> CREATE DATABASE southwind;
```

Query OK, 1 row affected (0.01 sec)

-- Create the table "products". Read "explanations" below for the column definitions

```
mysql> CREATE TABLE IF NOT EXISTS 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(7,2) NOT NULL DEFAULT 99999.99,  
    PRIMARY KEY (productID)  
);
```

-- Show all the tables to confirm that the "products" table has been created

```
mysql> SHOW TABLES;
```

The types are:

- productID is INT UNSIGNED - non-negative integers.
- productCode is CHAR(3) - a fixed-length alphanumeric string of 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 UNSIGNED (non-negative integers).
- price is DECIMAL(10,2) - a decimal number with 2 decimal places.
- DECIMAL is precise (represented as integer with a fix decimal point). On the other hand, FLOAT and DOUBLE (real numbers) are not precise and are approximated. DECIMAL type is recommended for currency.

-- 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(7,2)	NO		99999.99	

Let's fill up our "products" table with rows. 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. Take note that strings must be enclosed with a pair of single quotes (or double quotes).

-- Insert a row with all the column values

```
mysql> INSERT INTO products VALUES (1001, 'PEN', 'Pen Red', 5000, 1.23);
```

-- Insert multiple rows in one command

-- Inserting NULL to the auto_increment column results in max_value + 1

```
mysql> INSERT INTO products VALUES  
      (NULL, 'PEN', 'Pen Blue', 8000, 1.25),  
      (NULL, 'PEN', 'Pen Black', 2000, 1.25);
```

Query OK, 2 rows affected (0.03 sec)

Records: 2 Duplicates: 0 Warnings: 0

-- Insert value to selected columns

-- Missing value for the auto_increment column also results in max_value + 1

```
mysql> INSERT INTO products (productCode, name, quantity, price) VALUES
```

```
    ('PEC', 'Pencil 2B', 10000, 0.48),
```

```
    ('PEC', 'Pencil 2H', 8000, 0.49);
```

Query OK, 2 row affected (0.03 sec)

-- Missing columns get their default values

```
mysql> INSERT INTO products (productCode, name) VALUES ('PEC', 'Pencil HB');
```

Query OK, 1 row affected (0.04 sec)

-- 2nd column (productCode) is defined to be NOT NULL

```
mysql> INSERT INTO products values (NULL, NULL, NULL, NULL, NULL);
```

ERROR 1048 (23000): Column 'productCode' cannot be null

-- Query the table

```
mysql> SELECT * FROM products;
```

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.23
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49
1006	PEC	Pencil HB	0	9999999.99

6 rows in set (0.02 sec)

-- List all rows for the specified columns

```
mysql> SELECT name, price FROM products;
```

```
+-----+-----+
| name   | price |
+-----+-----+
| Pen Red | 1.23  |
| Pen Blue | 1.25  |
| Pen Black | 1.25  |
| Pencil 2B | 0.48  |
| Pencil 2H | 0.49  |
+-----+-----+
5 rows in set (0.00 sec)
```

-- List all rows of ALL the columns. The wildcard * denotes ALL columns

```
mysql> SELECT * FROM products;
```

```
+-----+-----+-----+-----+-----+
| productID | productCode | name   | quantity | price |
+-----+-----+-----+-----+-----+
| 1001 | PEN | Pen Red | 5000 | 1.23 |
| 1002 | PEN | Pen Blue | 8000 | 1.25 |
| 1003 | PEN | Pen Black | 2000 | 1.25 |
| 1004 | PEC | Pencil 2B | 10000 | 0.48 |
| 1005 | PEC | Pencil 2H | 8000 | 0.49 |
+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

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 > 1.0, quantity <= 500.

```
mysql> SELECT name, price FROM products WHERE price < 1.0;
```

```
+-----+-----+
| name      | price |
+-----+-----+
| Pencil 2B | 0.48  |
| Pencil 2H | 0.49  |
+-----+-----+
2 rows in set (0.00 sec)
```

```
mysql> SELECT name, quantity FROM products WHERE quantity <= 2000;
```

```
+-----+-----+
| name      | quantity |
+-----+-----+
| Pen Black | 2000     |
+-----+-----+
1 row in set (0.00 sec)
```

CAUTION: Do not compare FLOATs (real numbers) for equality ('=' or '<>'), as they are not precise. On the other hand, DECIMAL are precise.

For strings, you could also use '=', '<>', '>', '<', '>=', '<=' to compare two strings (e.g., productCode = 'PEC'). The ordering of string depends on the so-called collation chosen. For example,

SELECT name, price FROM products WHERE productCode = 'PEN';

-- String values are quoted

```
+-----+-----+
| name      | price |
+-----+-----+
| Pen Red   | 1.23  |
| Pen Blue  | 1.25  |
| Pen Black | 1.25  |
+-----+-----+
3 rows in set (0.00 sec)
```

```
+-----+-----+-----+-----+-----+
| productID | productCode | name      | quantity | price      |
+-----+-----+-----+-----+-----+
| 1001      | PEN         | Pen Red   | 5000     | 1.23       |
| 1002      | PEN         | Pen Blue  | 8000     | 1.25       |
| 1003      | PEN         | Pen Black | 2000     | 1.25       |
| 1004      | PEC         | Pencil 2B | 10000    | 0.48       |
| 1005      | PEC         | Pencil 2H | 8000     | 0.49       |
| 1006      | PEC         | Pencil HB | 0         | 9999999.99 |
+-----+-----+-----+-----+-----+
6 rows in set (0.02 sec)
```

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.

-- "name" begins with 'PENCIL'

mysql> SELECT name, price FROM products WHERE name LIKE 'PENCIL%';

name	price
Pencil 2B	0.48
Pencil 2H	0.49

-- "name" begins with 'P', followed by any two characters,
-- followed by space, followed by zero or more characters

mysql> SELECT name, price FROM products WHERE name LIKE 'P__ %';

name	price
Pen Red	1.23
Pen Blue	1.25
Pen Black	1.25

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.23
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49
1006	PEC	Pencil HB	0	9999999.99

6 rows in set (0.02 sec)

You can combine multiple conditions with boolean operators AND, OR. You can also invert a condition using operator NOT. For examples,

```
SELECT * FROM products WHERE quantity >= 5000 AND name LIKE 'Pen %';
```

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.23
1002	PEN	Pen Blue	8000	1.25

```
SELECT * FROM products WHERE quantity >= 5000 AND price < 1.24 AND name LIKE 'Pen %';
```

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.23

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.

```
SELECT * FROM products
```

```
WHERE (price BETWEEN 1.0 AND 2.0) AND (quantity BETWEEN 1000 AND 2000);
```

productID	productCode	name	quantity	price
1003	PEN	Pen Black	2000	1.25

NULL is a special value, which represent "no value", "missing value" or "unknown value". You can checking if a column contains NULL by IS NULL or IS NOT NULL. For example,

```
SELECT * FROM products WHERE productCode IS NULL;
```

Using comparison operator (such as = or <>) to check for NULL is a mistake - a very common mistake. For example,

```
SELECT * FROM products WHERE productCode = NULL;  
-- This is a common mistake. NULL cannot be compared.
```

-- Order the results by price in descending order

```
SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC;
```

productID	productCode	name	quantity	price
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1001	PEN	Pen Red	5000	1.23

-- Order by price in descending order, followed by quantity in ascending (default) order

```
mysql> SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC, quantity;
```

productID	productCode	name	quantity	price
1003	PEN	Pen Black	2000	1.25
1002	PEN	Pen Blue	8000	1.25
1001	PEN	Pen Red	5000	1.23

```
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	PEN	Pen Red	1.23
1002	PEN	Pen Blue	1.25
1003	PEN	Pen Black	1.25
1004	PEC	Pencil 2B	0.48
1005	PEC	Pencil 2H	0.49

```
SELECT CONCAT(productCode, ' - ', name) AS `Product Description`, price FROM products;
```

Product Description	price
PEN - Pen Red	1.23
PEN - Pen Blue	1.25
PEN - Pen Black	1.25
PEC - Pencil 2B	0.48
PEC - Pencil 2H	0.49

DISTINCT

-- Without DISTINCT

SELECT price FROM products;

price
1.23
1.25
1.25
0.48
0.49

-- With DISTINCT on price

SELECT DISTINCT price AS `Distinct Price` FROM products;

Distinct Price
1.23
1.25
0.48
0.49

-- DISTINCT combination of price and name

SELECT DISTINCT price, name FROM products;

price	name
1.23	Pen Red
1.25	Pen Blue
1.25	Pen Black
0.48	Pencil 2B
0.49	Pencil 2H

-- Increase the price by 10% for all products

UPDATE products SET price = price * 1.1;

SELECT * FROM products;

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.35
1002	PEN	Pen Blue	8000	1.38
1003	PEN	Pen Black	2000	1.38
1004	PEC	Pencil 2B	10000	0.53
1005	PEC	Pencil 2H	8000	0.54

-- Modify selected rows

mysql> UPDATE products SET quantity = quantity - 100 WHERE name = 'Pen Red';

mysql> SELECT * FROM products WHERE name = 'Pen Red';

productID	productCode	name	quantity	price
1001	PEN	Pen Red	4900	1.35

NOTE: don't forget to use SET SQL_SAFE_UPDATES = 0;

```
DELETE FROM products WHERE name LIKE 'Pencil%';
```

```
SELECT * FROM products;
```

productID	productCode	name	quantity	price
1001	PEN	Pen Red	4950	1.23
1002	PEN	Pen Blue	8000	1.38
1003	PEN	Pen Black	2000	1.38



Thank You