```
SELECT DISTINCT column_name1, column_name2
FROM table_name;
```

```
SELECT column list
FROM table_name
LIMIT [number of records];
```

The LIMIT Keyword

You can also pick up a set of records from a particular **offset**.
In the following example, we pick up **four** records, starting from the **third** position:

```
Select id, fname, lname, city

From studTable offset 3 limit 4;
```

Equivalent to

```
Select id, fname, lname, city

From studTable limit 3,4;
```

Order By

ORDER BY is used with SELECT to **sort** the returned data.

The following example sorts our **customers** table by the *FirstName* column.

```
SELECT * FROM customers
ORDER BY FirstName;
```

Build a query to select "name" and "city" from the "people" table, and order by the "id".

```
SELECT name, city
FROM people
ORDER BY id;
```

Sorting Multiple Columns

ORDER BY can sort retrieved data by multiple columns. When using ORDER BY with more than one column, separate the list of columns to follow ORDER BY with **commas**.

Here is the **customers** table, showing the following records:

ID	FirstName	LastName	Age
1	John	Smith	35
2	David	Smith	23
3	Chloe	Anderson	27
4	Emily	Adams	34
5	James	Roberts	31
6	Andrew	Thomas	45
7	Daniel	Harris	30

To order by **LastName** and **Age**:

```
SELECT * FROM customers
ORDER BY LastName, Age;
```

This ORDER BY statement returns the following result:

ID	FirstName	LastName	Age
4	Emily	**Adams**	**34**
3	Chloe	**Anderson**	**27**
7	Daniel	**Harris**	**30**
5	James	**Roberts**	**31**
2	David	**Smith**	**23**
1	John	**Smith**	**35**
6	Andrew	**Thomas**	**45**

As we have two **Smith**s, they will be ordered by the **Age** column in ascending order.

The ORDER BY command starts ordering in the same sequence as the columns. It will order by the first column listed, then by the second, and so on.

Fill in the blanks to order the query results by "name", and then by "state".

```
SELECT name, state, address

FROM customers

ORDER BY name, state;
```

The WHERE Statement

The **WHERE** clause is used to extract only those records that fulfill a specified criterion.

The syntax for the WHERE clause:

```
SELECT column_list
FROM table_name
WHERE condition;
```

Consider the following table:

ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
3	Chloe	Anderson	Chicago
4	Emily	Adams	Houston
5	James	Roberts	Philadelphia
6	Andrew	Thomas	New York
7	Daniel	Harris	New York
8	Charlotte	Walker	Chicago
9	Samuel	Clark	San Diego
10	Anthony	Young	Los Angeles

In the above table, to SELECT a specific record:

```
SELECT * FROM customers
WHERE ID = 7;
```

ID	FirstName	LastName	City
7	Daniel	Harris	New York

The WHERE clause is used to extract only those records that fulfill a specified criterion.

SQL Operators

Comparison Operators and **Logical Operators** are used in the WHERE clause to filter the data to be selected.

The following comparison operators can be used in the WHERE clause:

Operator	Description
=	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range

For example, we can display all customers names listed in our table, with the exception of the one with ID 5.

```
SELECT * FROM customers
WHERE ID != 5;
```

Result:

ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
3	Chloe	Anderson	Chicago
4	Emily	Adams	Houston
6	Andrew	Thomas	New York
7	Daniel	Harris	New York
8	Charlotte	Walker	Chicago
9	Samuel	Clark	San Diego
10	Anthony	Young	Los Angeles

As you can see, the record with ID=5 is excluded from the list.

The BETWEEN Operator

The BETWEEN operator selects values within a range. The first value must be lower bound and the second value, the upper bound.

The syntax for the BETWEEN clause is as follows:

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

The following SQL statement selects all records with IDs that fall between 3 and 7:

```
SELECT * FROM customers
WHERE ID BETWEEN 3 AND 7;
```

Result:

ID	FirstName	LastName	City
3	Chloe	Anderson	Chicago
4	Emily	Adams	Houston
5	James	Roberts	Philadelphia
6	Andrew	Thomas	New York
7	Daniel	Harris	New York

As you can see, the lower bound and upper bound are both included in the range.

Text Values

When working with text columns, surround any text that appears in the statement with **single quotation marks (')**.

The following SQL statement selects all records in which the *City* is equal to 'New York'.

```
SELECT ID, FirstName, LastName, City
FROM customers
WHERE City = 'New York';
```

ID	FirstName	LastName	City
1	John	Smith	**New York**
6	Andrew	Thomas	**New York**
7	Daniel	Harris	**New York**

If your text contains an apostrophe (single quote), you should use two single quote characters to escape the apostrophe. For example: 'Can''t'.

Logical Operators

Logical operators can be used to combine two Boolean values and return a result of **true**, **false**, or **null**.

The following operators can be used:

Operator	Description
AND	TRUE if **both** expressions are TRUE
OR	TRUE if **either** expression is TRUE
IN	TRUE if the operand is equal to one of a list of expressions
NOT	Returns TRUE if expression is not TRUE

When retrieving data using a SELECT statement, use logical operators in the WHERE clause to combine multiple conditions.

If you want to select rows that satisfy all of the given conditions, use the logical operator, **AND**.

ID	FirstName	LastName	Age
1	John	Smith	35
2	David	Williams	23
3	Chloe	Anderson	27
4	Emily	Adams	34
5	James	Roberts	31
6	Andrew	Thomas	45
7	Daniel	Harris	30

To find the names of the customers between 30 to 40 years of age, set up the query as seen here:

```
SELECT ID, FirstName, LastName, Age
FROM customers
WHERE Age >= 30 AND Age <= 40;
```

This results in the following output:

ID	FirstName	LastName	Age
1	John	Smith	35
4	Emily	Adams	34
5	James	Roberts	31
7	Daniel	Harris	30

You can combine as many conditions as needed to return the desired results.

OR

If you want to select rows that satisfy at least one of the given conditions, you can use the logical **OR** operator.

The following table describes how the logical OR operator functions:

Condition1	Condition2	Result
True	True	**True**
True	False	**True**
False	True	**True**
False	False	**False**

For example, if you want to find the customers who live either in New York or Chicago, the query would look like this:

```
SELECT * FROM customers
WHERE City = 'New York' OR City = 'Chicago';
```

Result:

ID	FirstName	LastName	City
1	John	Smith	New York
3	Chloe	Anderson	Chicago
6	Andrew	Thomas	New York
7	Daniel	Harris	New York
8	Charlotte	Walker	Chicago

You can OR two or more conditions.

OR

Drag and drop from the options below to select customers who live either in CA or in Boston.

```
SELECT name, state, city
FROM customers
WHERE state = 'CA'  OR city = 'Boston';
```

Combining AND & OR

The SQL **AND** and **OR** conditions may be combined to test multiple conditions in a query. These two operators are called **conjunctive operators**.

When combining these conditions, it is important to use **parentheses**, so that the order to evaluate each condition is known.

Consider the following table:

ID	FirstName	LastName	City	Age
1	John	Smith	New York	35
2	David	Williams	Los Angeles	23
3	Chloe	Anderson	Chicago	27
4	Emily	Adams	Houston	34
5	James	Roberts	Philadelphia	31
6	Andrew	Thomas	New York	45
7	Daniel	Harris	New York	30
8	Charlotte	Walker	Chicago	35
9	Samuel	Clark	San Diego	20
10	Anthony	Young	Los Angeles	33

The statement below selects all customers from the city "New York" **AND** with the age equal to "30" **OR** "35":

```
SELECT * FROM customers
WHERE City = 'New York'
AND (Age=30 OR Age=35);
```

Result:

ID	FirstName	LastName	City	Age
1	John	Smith	New York	35
7	Daniel	Harris	New York	30

You can nest as many conditions as you need.

The IN Operator

The **IN** operator is used when you want to compare a column with more than one value.

For example, you might need to select all customers from New York, Los Angeles, and Chicago. With the **OR** condition, your SQL would look like this:

```
SELECT * FROM customers
WHERE City = 'New York'
OR City = 'Los Angeles'
OR City = 'Chicago';
```

Result:

ID	FirstName	LastName	City
1	John	Smith	**New York**
2	David	Williams	**Los Angeles**
3	Chloe	Anderson	**Chicago**
6	Andrew	Thomas	**New York**
7	Daniel	Harris	**New York**
8	Charlotte	Walker	**Chicago**
10	Anthony	Young	**Los Angeles**

The IN operator is used when you want to compare a column with more than one value.

The IN Operator

You can achieve the same result with a single IN condition, instead of the multiple **OR** conditions:

```
SELECT * FROM customers
WHERE City IN ('New York', 'Los Angeles', 'Chicago');
```

This would also produce the same result:

ID	FirstName	LastName	City
1	John	Smith	**New York**
2	David	Williams	**Los Angeles**
3	Chloe	Anderson	**Chicago**
6	Andrew	Thomas	**New York**
7	Daniel	Harris	**New York**
8	Charlotte	Walker	**Chicago**
10	Anthony	Young	**Los Angeles**

Note the use of **parentheses** in the syntax.

The NOT IN Operator

The **NOT IN** operator allows you to exclude a list of specific values from the result set.

If we add the **NOT** keyword before **IN** in our previous query, customers living in those cities will be excluded:

```
SELECT * FROM customers
WHERE City NOT IN ('New York', 'Los Angeles', 'Chicago');
```

Result:

ID	FirstName	LastName	City
4	Emily	Adams	Houston
5	James	Roberts	Philadelphia
9	Samuel	Clark	San Diego

The NOT IN operator allows you to exclude a list of specific values from the result set.

The CONCAT Function

The **CONCAT** function is used to concatenate two or more text values and returns the concatenating string.

Let's concatenate the *FirstName* with the *City*, separating them with a *comma*:

```
SELECT CONCAT(FirstName, ', ' , City) FROM customers;
```

The output result is:

CONCAT(FirstName, ', ' , City)
John, New York
David, Los Angeles
Chloe, Chicago
Emily, Houston
James, Philadelphia
Andrew, New York
Daniel, New York
Charlotte, Chicago
Samuel, San Diego
Anthony, Los Angeles

The CONCAT() function takes two or more parameters.

The AS Keyword

A concatenation results in a new column. The default column name will be the CONCAT function.
You can assign a custom name to the resulting column using the **AS** keyword:

```
SELECT CONCAT(FirstName,', ', City) AS new_column
FROM customers;
```

And when you run the query, the column name appears to be changed.

new_column
John, New York
David, Los Angeles
Chloe, Chicago
Emily, Houston
James, Philadelphia
Andrew, New York
Daniel, New York
Charlotte, Chicago
Samuel, San Diego
Anthony, Los Angeles

A concatenation results in a new column.

Arithmetic Operators

Arithmetic operators perform arithmetical operations on numeric operands. The Arithmetic operators include addition (+), subtraction (-), multiplication (*) and division (/).

The following **employees** table shows employee names and salaries:

ID	FirstName	LastName	Salary
1	John	Smith	2000
2	David	Williams	1500
3	Chloe	Anderson	3000
4	Emily	Adams	4500
5	James	Roberts	2000
6	Andrew	Thomas	2500
7	Daniel	Harris	3000
8	Charlotte	Walker	3500
9	Samuel	Clark	4000
10	Anthony	Young	5000

The example below adds 500 to each employee's salary and selects the result:

```
SELECT ID, FirstName, LastName, Salary+500 AS Salary
FROM employees;
```

Result:

ID	FirstName	LastName	Salary
1	John	Smith	2500
2	David	Williams	2000
3	Chloe	Anderson	3500
4	Emily	Adams	5000
5	James	Roberts	2500
6	Andrew	Thomas	3000
7	Daniel	Harris	3500
8	Charlotte	Walker	4000
9	Samuel	Clark	4500
10	Anthony	Young	5500

Parentheses can be used to force an operation to take priority over any other operators. They are also used to improve code readability.

The UPPER Function

The **UPPER** function converts all letters in the specified string to uppercase.
The **LOWER** function converts the string to lowercase.

The following SQL query selects all *LastNames* as uppercase:

```
SELECT FirstName, UPPER(LastName) AS LastName
FROM employees;
```

Result:

FirstName	LastName
John	SMITH
David	WILLIAMS
Chloe	ANDERSON
Emily	ADAMS
James	ROBERTS
Andrew	THOMAS
Daniel	HARRIS
Charlotte	WALKER
Samuel	CLARK
Anthony	YOUNG

If there are characters in the string that are not letters, this function will have no effect on them.

SQRT and AVG

The **SQRT** function returns the square root of given value in the argument.

Let's calculate the square root of each Salary:

```
SELECT Salary, SQRT(Salary)
FROM employees;
```

Result:

Salary	SQRT(Salary)
2000	44.721359549995796
1500	38.729833462074170
3000	54.772255750516614
4500	67.082039324993690
2000	44.721359549995796
2500	50
3000	54.772255750516614
3500	59.160797830996160
4000	63.245553203367585
5000	70.710678118654760

Similarly, the **AVG** function returns the average value of a numeric column:

```
SELECT AVG(Salary) FROM employees;
```

Result:

AVG(Salary)
3100.0000

Another way to do the SQRT is to use POWER with the 1/2 exponent. However, SQRT seems to work faster than POWER in this case.

The SUM function

The **SUM** function is used to calculate the sum for a column's values.

For example, to get the sum of all of the salaries in the employees table, our SQL query would look like this:

```
SELECT SUM(Salary) FROM employees;
```

Result:

SUM(Salary)
31000

The sum of all of the employees' salaries is 31000.

Subqueries

A **subquery** is a query within another query.

Let's consider an example. We might need the list of all employees whose salaries are greater than the average.

First, calculate the average:

```
SELECT AVG(Salary) FROM employees;
```

As we already know the average, we can use a simple WHERE to list the salaries that are **greater** than that number.

```
SELECT FirstName, Salary FROM employees
WHERE Salary > 3100
ORDER BY Salary DESC;
```

Result:

FirstName	Salary
Anthony	5000
Emily	4500
Samuel	4000
Charlotte	3500

The **DESC** keyword sorts results in **descending** order.

Similarly, **ASC** sorts the results in **ascending** order.

Subqueries

A single subquery will return the same result more easily.

```
SELECT FirstName, Salary FROM employees
WHERE Salary > (SELECT AVG(Salary) FROM employees)
ORDER BY Salary DESC;
```

The same result will be produced.

FirstName	Salary
Anthony	5000
Emily	4500
Samuel	4000
Charlotte	3500

Enclose the subquery in **parentheses**.

Also, note that there is no semicolon at the end of the subquery, as it is part of our single query.

The Like Operator

The **LIKE** keyword is useful when specifying a **search condition** within your WHERE clause.

```
SELECT column_name(s)
FROM table_name
WHERE column_name LIKE pattern;
```

SQL **pattern** matching enables you to use "_" to match any single character and "%" to match an arbitrary number of characters (including zero characters).

For example, to select employees whose *FirstNames* begin with the letter **A**, you would use the following query:

```
SELECT * FROM employees
WHERE FirstName LIKE 'A%';
```

Result:

ID	FirstName	LastName	Salary
6	Andrew	Thomas	2500
10	Anthony	Young	5000

As another example, the following SQL query selects all employees with a *LastName* ending with the letter "s":

```
SELECT * FROM employees
WHERE LastName LIKE '%s';
```

Result:

ID	FirstName	LastName	Salary
2	David	**Williams**	1500
4	Emily	**Adams**	4500
5	James	**Roberts**	2000
6	Andrew	**Thomas**	2500
7	Daniel	**Harris**	3000

The % wildcard can be used **multiple** times within the same pattern.

The MIN Function

The **MIN** function is used to return the minimum value of an expression in a SELECT statement.

For example, you might wish to know the minimum salary among the employees.

```
SELECT MIN(Salary) AS Salary FROM employees;
```

Salary
1500

All of the SQL functions can be combined together to create a single expression.

Apartments

You want to rent an apartment and have the following table named **Apartments**:

id	city	address	price	status
1	Las Vegas	732 Hall Street	1000	Not rented
2	Marlboro	985 Huntz Lane	800	Not rented
3	Moretown	3757 Wines Lane	700	Not rented
4	Owatonna	314 Pritchard Court	500	Rented
5	Grayslake	3234 Cunningham Court	600	Rented
6	Great Neck	1927 Romines Mill Road	900	Not rented

Write a query to output the apartments whose prices **are greater than the average** and are also **not rented**, sorted by the 'Price' column.

Recall the **AVG** keyword.

Your Output

```
id,city,address,price,status
2,Marlboro,985 Huntz Lane,800,Not rented
6,Great Neck,1927 Romines Mill Road,900,Not rented
1,Las Vegas,732 Hall Street,1000,Not rented
```

Expected Output

```
id,city,address,price,status
2,Marlboro,985 Huntz Lane,800,Not rented
6,Great Neck,1927 Romines Mill Road,900,Not rented
1,Las Vegas,732 Hall Street,1000,Not rented
```

```
      select * from apartments
where price > (SELECT AVG(price) FROM apartments) and status='Not rented'
 order by price;
```

Joining Tables

All of the queries shown up until now have selected from just one table at a time.

One of the most beneficial features of SQL is the ability to combine data from two or more tables.

In the two tables that follow, the table named **customers** stores information about customers:

ID	Name	City	Age
1	John	New York	35
2	David	Los Angeles	23
3	Chloe	Chicago	27
4	Emily	Houston	34
5	James	Philadelphia	31

The **orders** table stores information about individual orders with their corresponding amount:

ID	Name	Customer_ID	Amount
1	Book	3	5000
2	Box	5	3000
3	Toy	2	4500
4	Flowers	4	1800
5	Cake	1	6700

In SQL, "**joining tables**" means combining data from two or more tables. A table join creates a **temporary table** showing the data from the joined tables.

Joining Tables

To join the two tables, specify them as a comma-separated list in the FROM clause:

```
SELECT customers.ID, customers.Name, orders.Name, orders.Amount
FROM customers, orders
WHERE customers.ID=orders.Customer_ID
ORDER BY customers.ID;
```

Each table contains "ID" and "Name" columns, so in order to select the correct ID and Name, **fully qualified names** are used.

Note that the WHERE clause "joins" the tables on the condition that the **ID** from the **customers** table should be equal to the **customer_ID** of the **orders** table.

Result:

ID	Name	Name	Amount
1	John	Cake	6700
2	David	Toy	4500
3	Chloe	Book	5000
4	Emily	Flowers	1800
5	James	Box	3000

The returned data shows customer orders and their corresponding amount.

Specify multiple table names in the FROM by comma-separating them.

Joining Tables

You are given the following **students** and **teachers** tables

students (with their teachers ID's):

id	firstname	lastname	teacherid
1	Melanie	Cavazos	1
2	Ted	Gray	2
3	Sandra	Dennis	1
4	John	Lockhart	3
5	Micheal	Cartwright	2
6	Henry	Suh	1

teachers:

id	firstname	lastname
1	Kenneth	Rogers
2	Julia	Marrero
3	Mary	Foulds

Write a query to output all of the students with their teachers' last names in one table, sorted by students ID.

The column with teachers' last names should be named "teacher" -- recall the **AS** keyword.

```
select  students.id, students.FirstName, students.lastname,
students.teacherid, teachers.lastname
from students, teachers
where students.teacherid=teachers.id
order by students.id;
```

Your Output

```
id,firstname,lastname,teacherid,lastname

 1,Melanie,Cavazos,1,Rogers

2,Ted,Gray,2,Marrero

3,Sandra,Dennis,1,Rogers

4,John,Lockhart,3,Foulds

5,Micheal,Cartwright,2,Marrero

 6,Henry,Suh,1,Rogers
```

Expected Output

```
id,firstname,lastname,teacher

1,Melanie,Cavazos,Rogers

 2,Ted,Gray,Marrero

3,Sandra,Dennis,Rogers

4,John,Lockhart,Foulds

 5,Micheal,Cartwright,Marrero

 6,Henry,Suh,Rogers
```

Custom Names

Let's help the pilots find out details about their flights.

You are given the following tables named **pilots** and **flights**:

Pilots

pilot_id	fullname	nationality	flight_id
1	John Ritchson	USA	2
2	Rayan Gomez	England	4
3	Omar Wallace	France	1
4	Brooklyn Austin	USA	3

Flights

id	duration	landing_country
1	4h	Germany
2	3h	USA
3	3h	Russia
4	2h	Mexico

Write a query to output the flight ID, full name of the pilot who is responsible for the flight, the country where the flight will land, and the duration of the flight (flight_id, fullname, landing_country, duration).

Order the table by flight ID.

```
select flights.id as flight_id, pilots.fullname,
flights.landing_country,
flights.duration
from pilots, flights
where pilots.flight_id=flights.id
order by flights.id;
```

Your Output

```
flight_id,fullname,landing_country,duration

1,Omar Wallace,Germany,4h

2,John Ritchson,USA,3h
```

```
3,Brooklyn Austin,Russia,3h
4,Rayan Gomez,Mexico,2h
```

Expected Output

```
flight_id,fullname,landing_country,duration
1,Omar Wallace,Germany,4h
2,John Ritchson,USA,3h
3,Brooklyn Austin,Russia,3h
4,Rayan Gomez,Mexico,2h
```

Custom names can be used for tables as well. You can shorten the join statements by giving the tables "nicknames":

```
SELECT ct.ID, ct.Name, ord.Name, ord.Amount
FROM customers AS ct, orders AS ord
WHERE ct.ID=ord.Customer_ID
ORDER BY ct.ID;
```

As you can see, we shortened the table names as we used them in our query.

Fill in the blanks to select item names and names of customers who bought the items. Use custom names to shorten the statement.

```
SELECT ct.name, [it].name
FROM customers [as] ct, items AS it
WHERE it.seller_id=[ct].id;
```

Types of Join

The following are the types of JOIN that can be used in MySQL:
- **INNER** JOIN
- **LEFT** JOIN
- **RIGHT** JOIN

INNER JOIN is equivalent to JOIN. It returns rows when there is a match between the tables.
Syntax:

```
SELECT column_name(s)
FROM table1 INNER JOIN table2
ON table1.column_name=table2.column_name;
```

Note the **ON** keyword for specifying the inner join condition.

The image below demonstrates how INNER JOIN works:

Only the records matching the join condition are returned.

LEFT JOIN

The **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table.

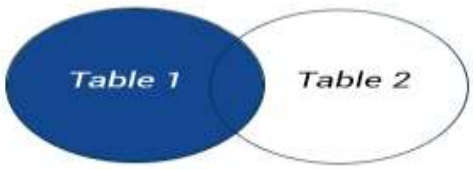
This means that if there are no matches for the **ON** clause in the table on the right, the join will still return the rows from the first table in the result.

The basic syntax of LEFT JOIN is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1 LEFT OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

The **OUTER** keyword is optional, and can be omitted.

The image below demonstrates how LEFT JOIN works:



Consider the following tables.

customers:

ID	Name	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
3	Chloe	Anderson	Chicago
4	Emily	Adams	Houston
5	James	Roberts	Philadelphia
6	Andrew	Thomas	New York
7	Daniel	Harris	New York

items:

ID	Name	Cost	Seller_id
39	Book	5.9	1
24	Box	2.99	1
72	Toy	23.7	2
36	Flowers	50.75	2
18	T-Shirt	22.5	3
16	Notebook	150.22	4
74	Perfume	90.9	6

The following SQL statement will return all **customers**, and the **items** they might have:

```
SELECT customers.Name, items.Name
FROM customers LEFT OUTER JOIN items
ON customers.ID=items.Seller_id;
```

Result:

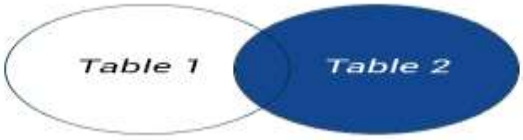
Name	Name
John	Book
John	Box
David	Toy
David	Flowers
Chloe	T-Shirt
Emily	Notebook
Andrew	Perfume
James	*NULL*
Daniel	*NULL*

The result set contains all the rows from the left table and matching data from the right table.

If no match is found for a particular row, **NULL** is returned.

RIGHT JOIN

The **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table.



The basic syntax of RIGHT JOIN is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1 RIGHT OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

Again, the **OUTER** keyword is optional, and can be omitted.

Consider the same example from our previous lesson, but this time with a RIGHT JOIN:

```
SELECT customers.Name, items.Name FROM customers
RIGHT JOIN items ON customers.ID=items.Seller_id;
```

Result:

Name	Name
John	Book
John	Box
David	Toy
David	Flowers
Chloe	T-Shirt
Emily	Notebook
Andrew	Perfume

The RIGHT JOIN returns all the rows from the right table (items), even if there are no matches in the left table (customers).

There are other types of joins in the SQL language, but they are not supported by MySQL.

Set Operation

Occasionally, you might need to combine data from multiple tables into one comprehensive dataset. This may be for tables with similar data within the same database or maybe there is a need to combine similar data across databases or even across servers.

To accomplish this, use the **UNION** and **UNION ALL** operators.

UNION combines multiple datasets into a single dataset, and removes any existing duplicates.
UNION ALL combines multiple datasets into one dataset, but does not remove duplicate rows.

UNION ALL is faster than UNION, as it does not perform the duplicate removal operation over the data set.

UNION

The **UNION** operator is used to combine the result-sets of two or more SELECT statements.

All SELECT statements within the UNION must have the **same number of columns**. The columns must also have the same **data types**. Also, the columns in each SELECT statement must be in the same order.
The syntax of UNION is as follows:

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

Here is the **First** of two tables:

ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles

And here is the **Second**:

ID	FirstName	LastName	City
1	James	Roberts	Philadelphia
2	David	Williams	Los Angeles

```
SELECT ID, FirstName, LastName, City FROM First
UNION
SELECT ID, FirstName, LastName, City FROM Second;
```

The resulting table will look like this one:

ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
1	James	Roberts	Philadelphia

As you can see, the duplicates have been removed.

TIP:
If your columns don't match exactly across all queries, you can use a **NULL (or any other)** value such as:

```
SELECT FirstName, LastName, Company FROM businessContacts
UNION
SELECT FirstName, LastName, NULL FROM otherContacts;
```

The UNION operator is used to combine the result-sets of two or more SELECT statements.

UNION ALL

UNION ALL selects all rows from each table and combines them into a single table.

The following SQL statement uses UNION ALL to select data from the **First** and **Second** tables:

```
SELECT ID, FirstName, LastName, City FROM First
UNION ALL
SELECT ID, FirstName, LastName, City FROM Second;
```

The resulting table:

ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
1	James	Roberts	Philadelphia
2	David	Williams	Los Angeles

As you can see, the result set includes the duplicate rows as well.

The Insert statement

Inserting Data

SQL tables store data in rows, one row after another. The **INSERT INTO** statement is used to add **new rows** of data to a table in the database.
The SQL **INSERT INTO** syntax is as follows:

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

Make sure the order of the values is in the same order as the columns in the table.

Consider the following **Employees** table:

ID	FirstName	LastName	Age
1	Emily	Adams	34
2	Chloe	Anderson	27
3	Daniel	Harris	30
4	James	Roberts	31
5	John	Smith	35
6	Andrew	Thomas	45
7	David	Williams	23

Use the following SQL statement to insert a new row:

```
INSERT INTO Employees
VALUES (8, 'Anthony', 'Young', 35);
```

The values are comma-separated and their order corresponds to the columns in the table.
Result:

ID	FirstName	LastName	Age
1	Emily	Adams	34
2	Chloe	Anderson	27
3	Daniel	Harris	30
4	James	Roberts	31
5	John	Smith	35
6	Andrew	Thomas	45
7	David	Williams	23
8	Anthony	Young	35

When inserting records into a table using the SQL INSERT statement, you must provide a value for every column that does not have a default value, or does not support NULL.

Alternatively, you can specify the table's column names in the INSERT INTO statement:

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

column1, column2, ..., columnN are the names of the columns that you want to insert data into.

```
INSERT INTO Employees (ID, FirstName, LastName, Age)
VALUES (8, 'Anthony', 'Young', 35);
```

This will insert the data into the corresponding columns:

ID	FirstName	LastName	Age
1	Emily	Adams	34
2	Chloe	Anderson	27
3	Daniel	Harris	30
4	James	Roberts	31
5	John	Smith	35
6	Andrew	Thomas	45
7	David	Williams	23
8	**Anthony**	**Young**	**35**

You can specify your own column order, as long as the values are specified in the same order as the columns.

It is also possible to insert data into **specific** columns only.

```
INSERT INTO Employees (ID, FirstName, LastName)
VALUES (9, 'Samuel', 'Clark');
```

Result:

ID	FirstName	LastName	Age
1	Emily	Adams	34
2	Chloe	Anderson	27
3	Daniel	Harris	30
4	James	Roberts	31
5	John	Smith	35
6	Andrew	Thomas	45
7	David	Williams	23
8	Anthony	Young	35
9	**Samuel**	**Clark**	**0**

The *Age* column for that row automatically became **0**, as that is its default value.

When inserting data into a table:

The number of columns in the insert statement and the number of columns of the table MUST be the same

We don't have to insert values for all columns in the table

Names of the columns MUST always be mentioned in the insert statement

Update and Delete Statement

Updating Data

The **UPDATE** statement allows us to alter data in the table.

The basic syntax of an **UPDATE** query with a **WHERE** clause is as follows:

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

You specify the column and its new value in a comma-separated list after the **SET** keyword.

If you omit the WHERE clause, **all** records in the table will be updated!

Consider the following table called "Employees":

ID	FirstName	LastName	Salary
1	John	Smith	2000
2	David	Williams	1500
3	Chloe	Anderson	3000
4	Emily	Adams	4500

To update John's salary, we can use the following query:

```
UPDATE Employees
SET Salary=5000
WHERE ID=1;
```

Result:

ID	FirstName	LastName	Salary
1	John	Smith	**5000**
2	David	Williams	1500
3	Chloe	Anderson	3000
4	Emily	Adams	4500

Updating Multiple Columns

It is also possible to UPDATE **multiple columns** at the same time by comma-separating them:

```
UPDATE Employees
SET Salary=5000, FirstName='Robert'
WHERE ID=1;
```

Result:

ID	FirstName	LastName	Salary
1	**Robert**	Smith	**5000**
2	David	Williams	1500
3	Chloe	Anderson	3000
4	Emily	Adams	4500

You can specify the column order any way you like in the SET clause.

Deleting Data

The **DELETE** statement is used to remove data from your table. DELETE queries work much like UPDATE queries.

```
DELETE FROM table_name
WHERE condition;
```

For example, you can delete a specific employee from the table:

```
DELETE FROM Employees
WHERE ID=1;
```

Result:

ID	FirstName	LastName	Salary
2	David	Williams	1500
3	Chloe	Anderson	3000
4	Emily	Adams	4500

If you omit the WHERE clause, **all** records in the table will be deleted!

The DELETE statement removes the data from the table permanently.

SQL Tables

A single database can house hundreds of tables, each playing its own unique role in the database schema.

SQL tables are comprised of table rows and columns. Table columns are responsible for storing many different types of data, including numbers, texts, dates, and even files.

The **CREATE TABLE** statement is used to create a new table.

Creating a basic table involves naming the table and defining its columns and each column's data type.

The basic syntax for the CREATE TABLE statement is as follows:

```
CREATE TABLE table_name
(
column_name1 data_type(size),
column_name2 data_type(size),
column_name3 data_type(size),
....
columnN data_type(size)
);
```

- The **column_names** specify the names of the columns we want to create.
- The **data_type** parameter specifies what type of data the column can hold. For example, use **int** for whole numbers.
- The **size** parameter specifies the maximum length of the table's column.

Note the **parentheses** in the syntax.

Assume that you want to create a table called "Users" that consists of four columns: UserID, LastName, FirstName, and City.
Use the following **CREATE TABLE** statement:

```
CREATE TABLE Users
(
UserID int,
FirstName varchar(100),
LastName varchar(100),
City varchar(100)
);
```

varchar is the datatype that stores characters. You specify the number of characters in the parentheses after the type. So in the example above, our fields can hold max **100** characters long text.

Data Types

Data types specify the type of data for a particular column.

If a column called "LastName" is going to hold names, then that particular column should have a "varchar" (variable-length character) data type.
The most common data types:
Numeric
INT -A normal-sized integer that can be signed or unsigned.
FLOAT(M,D) - A floating-point number that cannot be unsigned. You can optionally define the display length (M) and the number of decimals (D).
DOUBLE(M,D) - A double precision floating-point number that cannot be unsigned. You can optionally define the display length (M) and the number of decimals (D).

Date and Time
DATE - A date in *YYYY-MM-DD* format.
DATETIME - A date and time combination in YYYY-MM-DD HH:MM:SS format.
TIMESTAMP - A timestamp, calculated from midnight, January 1, 1970
TIME - Stores the time in HH:MM:SS format.

String Type
CHAR(M) - Fixed-length character string. Size is specified in parenthesis. Max 255 bytes.
VARCHAR(M) - Variable-length character string. Max size is specified in parenthesis.
BLOB - "Binary Large Objects" and are used to store large amounts of binary data, such as images or other types of files.
TEXT - Large amount of text data.

Choosing the correct data type for your columns is the key to good database design.

Primary Key

The **UserID** is the best choice for our Users table's primary key.
Define it as a primary key during table creation, using the **PRIMARY KEY** keyword.

```
CREATE TABLE Users
(
UserID int,
FirstName varchar(100),
LastName varchar(100),
City varchar(100),
PRIMARY KEY(UserID)
);
```

Specify the column name in the parentheses of the PRIMARY KEY keyword.

Creating a Table

Now, when we run the query, our table will be created in the database.

UserID	FirstName	LastName	City

You can now use **INSERT INTO** queries to insert data into the table.

SQL Constraints

SQL **constraints** are used to specify rules for table data.

The following are commonly used SQL constraints:
NOT NULL - Indicates that a column cannot contain any NULL value.
UNIQUE - Does not allow to insert a duplicate value in a column. The UNIQUE constraint maintains the uniqueness of a column in a table. More than one UNIQUE column can be used in a table.
PRIMARY KEY - Enforces the table to accept unique data for a specific column and this constraint create a unique index for accessing the table faster.
CHECK - Determines whether the value is valid or not from a logical expression.
DEFAULT - While inserting data into a table, if no value is supplied to a column, then the column gets the value set as DEFAULT.

For example, the following means that the **name** column disallows NULL values.

```
name varchar(100) NOT NULL
```

During table creation, specify column level constraint(s) after the data type of that column.

AUTO INCREMENT

Auto-increment allows a unique number to be generated when a new record is inserted into a table.

Often, we would like the value of the primary key field to be created automatically every time a new record is inserted.

By default, the starting value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record.
Let's set the UserID field to be a primary key that automatically generates a new value:

```
UserID int NOT NULL AUTO_INCREMENT,
PRIMARY KEY (UserID)
```

Auto-increment allows a unique number to be generated when a new record is inserted into a table.

Using Constraints

The example below demonstrates how to create a table using constraints.

```
CREATE TABLE Users (
id int NOT NULL AUTO_INCREMENT,
username varchar(40) NOT NULL,
password varchar(10) NOT NULL,
PRIMARY KEY(id)
);
```

SQL

The following SQL enforces that the "id", "username", and "password" columns do not accept NULL values. We also define the "id" column to be an auto-increment primary key field.

Here is the result:

#	Column	Type	Null	Default	Extra
1	id	int(11)	No	None	AUTO_INCREMENT
2	username	varchar(40)	No	None	
3	password	varchar(10)	No	None	

When inserting a new record into the Users table, it's not necessary to specify a value for the id column; a unique new value will be added automatically.

Alter, Drop, Rename a Table

ALTER TABLE

The **ALTER TABLE** command is used to add, delete, or modify columns in an existing table. You would also use the ALTER TABLE command to add and drop various constraints on an existing table.

Consider the following table called **People**:

ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
3	Chloe	Anderson	Chicago

The following SQL code adds a new column named **DateOfBirth**

```
ALTER TABLE People ADD DateOfBirth date;
```

Result:

ID	FirstName	LastName	City	DateOfBirth
1	John	Smith	New York	*NULL*
2	David	Williams	Los Angeles	*NULL*
3	Chloe	Anderson	Chicago	*NULL*

All rows will have the default value in the newly added column, which, in this case, is NULL.

Dropping

The following SQL code demonstrates how to delete the column named *DateOfBirth* in the People table.

```
ALTER TABLE People
DROP COLUMN DateOfBirth;
```

The People table will now look like this:

ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
3	Chloe	Anderson	Chicago

The column, along with all of its data, will be completely removed from the table.

To delete the entire table, use the **DROP TABLE** command:

```
DROP TABLE People;
```

Be careful when dropping a table. Deleting a table will result in the complete loss of the information stored in the table!

Renaming

The ALTER TABLE command is also used to rename columns:

```
ALTER TABLE People
RENAME FirstName TO name;
```

This query will rename the column called FirstName to name.

Result:

ID	name	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
3	Chloe	Anderson	Chicago

Renaming Tables
You can rename the entire table using the **RENAME** command:

```
RENAME TABLE People TO Users;
```
SQL

This will rename the table People to Users.

Views

In SQL, a VIEW is a **virtual table** that is based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

Views allow us to:
- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables and use it to generate reports.

To create a view:

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition;
```
SQL

The SELECT query can be as complex as you need it to be. It can contain multiple JOINS and other commands.

Creating Views

Consider the **Employees** table, which contains the following records:

ID	FirstName	LastName	Age	Salary
1	Emily	Adams	34	5000
2	Chloe	Anderson	27	10000
3	Daniel	Harris	30	6500
4	James	Roberts	31	5500
5	John	Smith	35	4500
6	Andrew	Thomas	45	6000
7	David	Williams	23	3000

Let's create a view that displays each employee's FirstName and Salary.

```
CREATE VIEW List AS
SELECT FirstName, Salary
FROM Employees;
```

Now, you can query the **List** view as you would query an actual table.

```
SELECT * FROM List;
```

This would produce the following result:

FirstName	Salary
Emily	5000
Chloe	10000
Daniel	6500
James	5500
John	4500
Andrew	6000
David	3000

A view always shows up-to-date data! The database engine uses the view's SQL statement to recreate the data each time a user queries a view.

Updating a View

You can update a view by using the following syntax:

```
CREATE OR REPLACE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition;
```

The example below updates our **List** view to select also the LastName:

```
CREATE OR REPLACE VIEW List AS
SELECT FirstName, LastName, Salary
FROM Employees;
```

Result:

FirstName	LastName	Salary
Emily	Adams	5000
Chloe	Anderson	10000
Daniel	Harris	6500
James	Roberts	5500
John	Smith	4500
Andrew	Thomas	6000
David	Williams	3000

You can delete a view with the DROP VIEW command.

```
DROP VIEW List;
```

It is sometimes easier to drop a table and recreate it instead of using the ALTER TABLE statement to change the table's definition.

Updating a View

Which statement in regard to views is correct?

Views need space in the database to be stored

Views must be updated manually

Views are being updated dynamically

Zoo

You manage a zoo. Each animal in the zoo comes from a different country. Here are the tables you have:

Animals

name	type	country_id
Candy	Elephant	3
Pop	Horse	1
Vova	Bear	2
Merlin	Lion	1
Bert	Tiger	3

Countries

id	country
1	USA
2	Russia
3	India

1) A new animal has come in, with the following details:
name - "Slim", type - "Giraffe", country_id - 1
Add him to the Animals table.
2) You want to make a complete list of the animals for the zoo's visitors. Write a query to output a new table with each animal's name, type and country fields, sorted by countries.

Recall **INSERT** and **INNER JOIN** keywords.

```
insert into animals (name, type, country_id)
values('slim','Giraffe',1);
select animals.name, animals.type, countries.country
from animals inner join countries
on animals.country_id=countries.id order by country

;
```

Your Output

```
name,type,country
 Bert,Tiger,India
 Candy,Elephant,India
 Vova,Bear,Russia
 slim,Giraffe,USA
Merlin,Lion,USA
Pop,Horse,USA
```

Expected Output

```
name,type,country

 Bert,Tiger,India

Candy,Elephant,India

 Vova,Bear,Russia

 Slim,Giraffe,USA

 Merlin,Lion,USA

Pop,Horse,USA
```

In the "users" table of website logins and passwords, select the first 10 records in the table.

```
SELECT * FROM users
LIMIT 10;
```

Drag and drop from the options below to create the table "users" to store website user logins and passwords.

```
CREATE TABLE users ( id INT NOT NULL
AUTO_INCREMENT,
login VARCHAR (100),
password VARCHAR(100) );
```

our boss asks you to print the list of the first one hundred customers who have balances greater than $1000 or who are from NY.

```
SELECT * FROM customers
WHERE balance > 1000  OR city = 'NY'
LIMIT 100;
```

You need the ages of all bears and lions. The first query shows the ages of bears and birds from zoo1, the other shows the ages of lions and crocodiles from zoo2.

```
SELECT age FROM zoo1
WHERE animal IN ('bear', 'bird')
UNION
SELECT age FROM zoo2
WHERE animal IN ('lion', 'crocodile')
```

Drag and drop from the options below to create a list of customers in the form "name is from city".

```
SELECT  CONCAT (name, ' is from ', city)
FROM customers;
```

The zoo administration wants a list of animals whose age is greater than the average age of all of the animals.

```
SELECT * FROM zoo
WHERE age >(SELECT AVG (age) FROM zoo);
```

Drag and drop from the options below to retrieve all students between the ages of 18 and 22.

```
SELECT name FROM students
WHERE age BETWEEN 18 AND 22;
```

Drag and drop from the options below to update the "students" table to set Jake's university to MIT. His id is 682.

```
UPDATE students
SET university='MIT' WHERE id=682;
```

When you inserted "elephant" as a new animal, you forgot to include the elephant's age. Correct this mistake by updating the "zoo" table.

```
UPDATE zoo
SET age=14
WHERE animal='elephant'
```

Drag and drop from the options below to update the food_balance to 23 for animals whose age is greater than the average age of the animals.

```
UPDATE zoo
SET food_balance=23
WHERE age > (SELECT AVG (age) FROM zoo);
```

You need your customer's names, along with the names of the cities in which they live. The names of the cities are stored in a separate table called "cities".

```
SELECT customers.name, cities.name
FROM customers
RIGHT OUTER JOIN cities
ON cities.id=customers.city_id;
```

the university's table containing student data, the students' last names have been omitted.

Correct this by adding a new column to the table.

```
ALTER TABLE students
ADD last_name VARCHAR(100);
```

```
SELECT name FROM students
WHERE university  IN ('MIT', 'Stanford', 'Harvard')
AND name='Jake';
```