SHOW DATABASES:

The SQL **SHOW** statement displays information contained in the database and its tables. This helpful tool lets you keep track of your database contents and remind yourself about the structure of your tables.
For example, the **SHOW DATABASES** command lists the databases managed by the server.

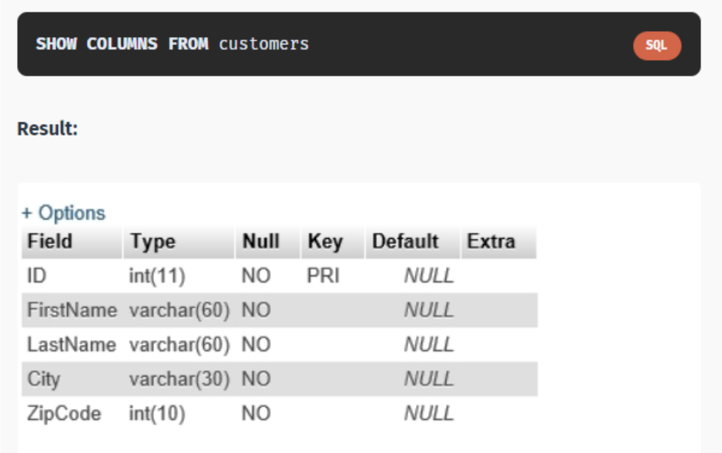
SHOW TABLES:

The **SHOW TABLES** command is used to display all of the tables in the currently selected MySQL database.

SHOW COLUMNS:

SHOW COLUMNS displays information about the columns in a given table.

The following example displays the columns in our **customers** table:



```
SHOW COLUMNS FROM customers
```

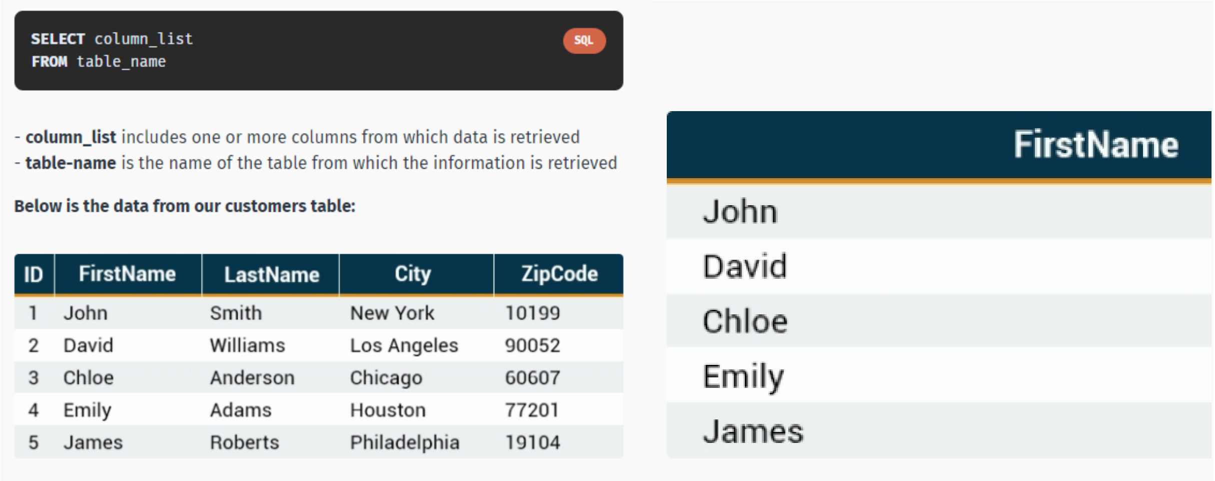
Field	Type	Null	Key	Default	Extra
ID	int(11)	NO	PRI	*NULL*	
FirstName	varchar(60)	NO		*NULL*	
LastName	varchar(60)	NO		*NULL*	
City	varchar(30)	NO		*NULL*	
ZipCode	int(10)	NO		*NULL*	

SELECT STATEMENT:

The **SELECT** statement is used to select data from a database.
The result is stored in a result table, which is called the **result-set**.

A **query** may retrieve information from selected columns or from all columns in the table.
To create a simple SELECT statement, specify the name(s) of the column(s) you need from the table.

SELECT Firstname FROM customers



```sql
SELECT column_list
FROM table_name
```

- **column_list** includes one or more columns from which data is retrieved
- **table-name** is the name of the table from which the information is retrieved

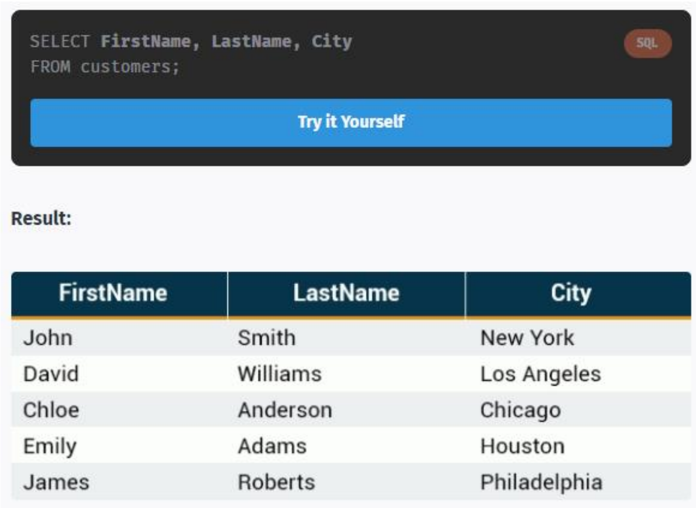
Below is the data from our customers table:

ID	FirstName	LastName	City	ZipCode
1	John	Smith	New York	10199
2	David	Williams	Los Angeles	90052
3	Chloe	Anderson	Chicago	60607
4	Emily	Adams	Houston	77201
5	James	Roberts	Philadelphia	19104

FirstName
John
David
Chloe
Emily
James

Selecting multiple columns:

As previously mentioned, the SQL SELECT statement retrieves records from tables in your SQL database.
You can select multiple table columns at once.
Just list the column names, separated by **commas**



```sql
SELECT FirstName, LastName, City
FROM customers;
```

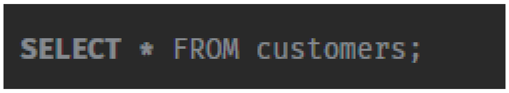
Result:

FirstName	LastName	City
John	Smith	New York
David	Williams	Los Angeles
Chloe	Anderson	Chicago
Emily	Adams	Houston
James	Roberts	Philadelphia

Selecting all columns:

To retrieve all of the information contained in your table, place an **asterisk (*)** sign after the SELECT command, rather than typing in each column names separately.

The following SQL statement selects all of the columns in the **customers** table:



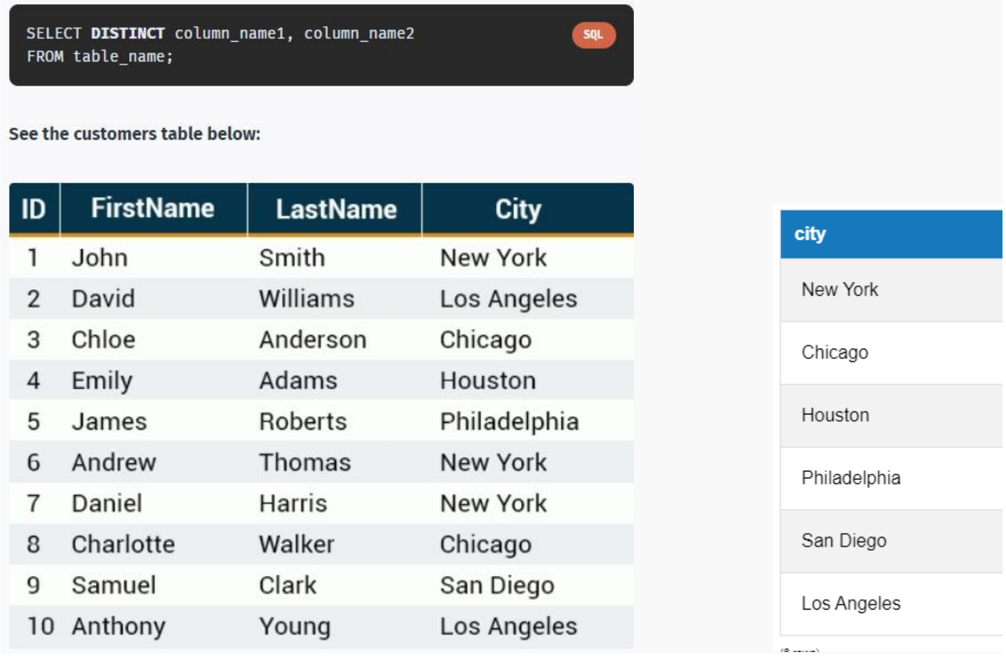
```sql
SELECT * FROM customers;
```

DISTINCT:

In situations in which you have multiple duplicate records in a table, it might make more sense to return only unique records, instead of fetching the duplicates.

The SQL **DISTINCT** keyword is used in conjunction with SELECT to eliminate all duplicate records and return only unique ones.

SELECT DISTINCT City FROM customers;



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

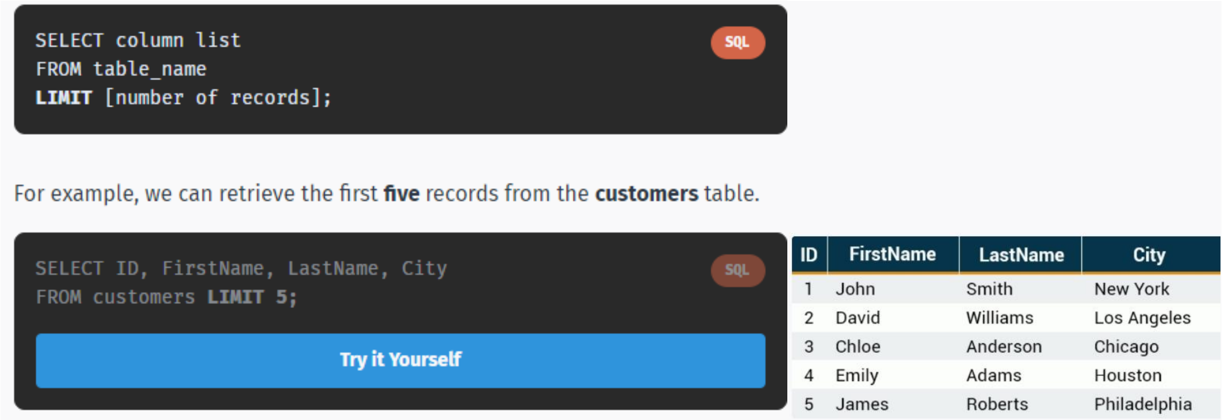
See the customers table below:

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

city
New York
Chicago
Houston
Philadelphia
San Diego
Los Angeles

LIMIT:

By default, all results that satisfy the conditions specified in the SQL statement are returned. However, sometimes we need to retrieve just a subset of records. In MySQL, this is accomplished by using the **LIMIT** keyword.



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

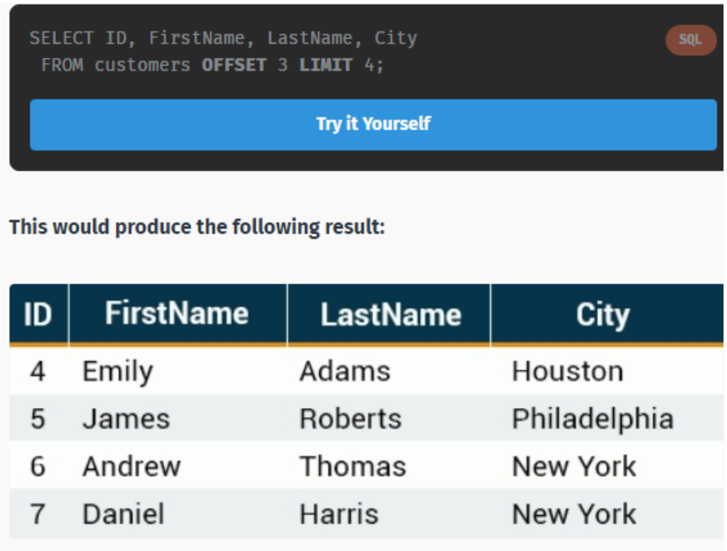
For example, we can retrieve the first **five** records from the **customers** table.

```sql
SELECT ID, FirstName, LastName, City
FROM customers LIMIT 5;
```

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

OFFSET -(starting from):

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:



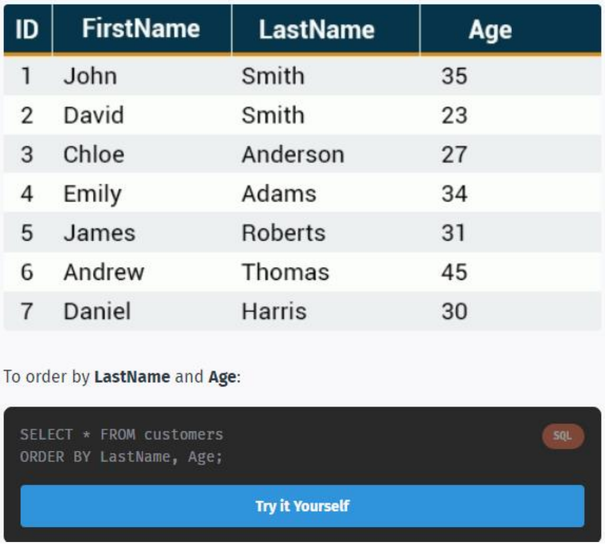
```
SELECT ID, FirstName, LastName, City
 FROM customers OFFSET 3 LIMIT 4;
```

This would produce the following result:

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

ORDER BY:

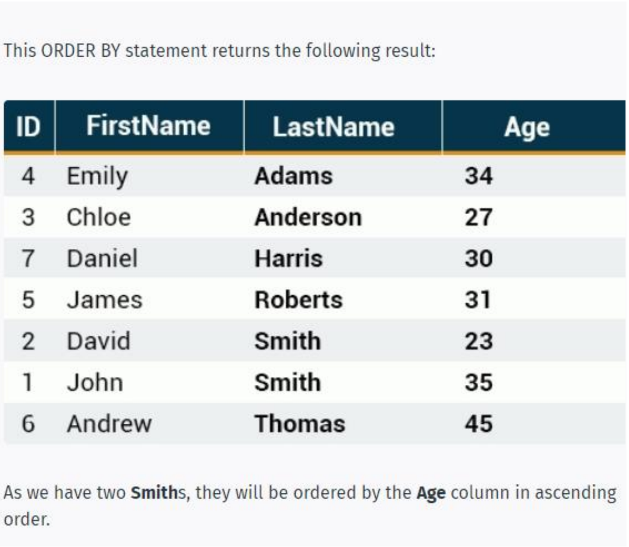
ORDER BY can sort retrieved data by multiple columns. When using ORDER BY with more than one column, separate the list of columns to follow ORDER BY with **commas**.
Here is the **customers** table, showing the following records:



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;
```



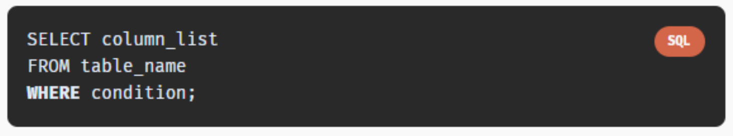
This ORDER BY statement returns the following result:

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**

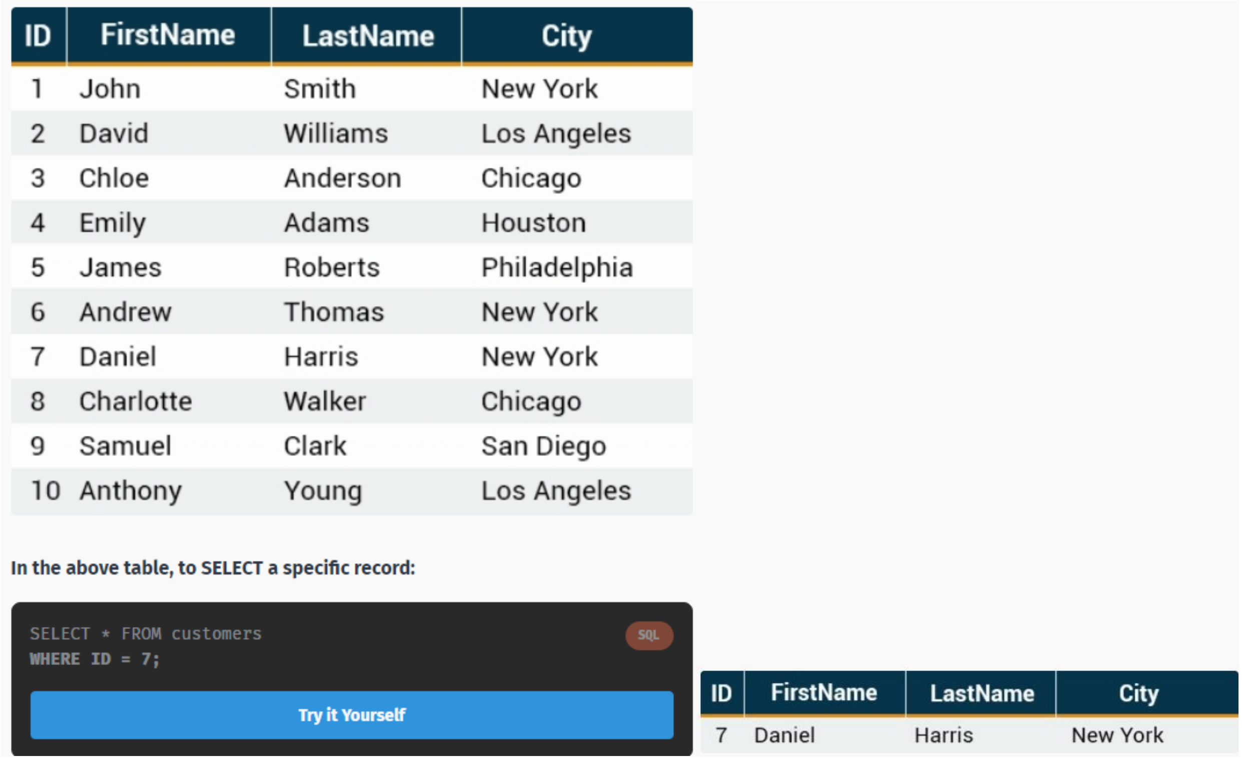
As we have two **Smith**s, they will be ordered by the **Age** column in ascending order.

WHERE:

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



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



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

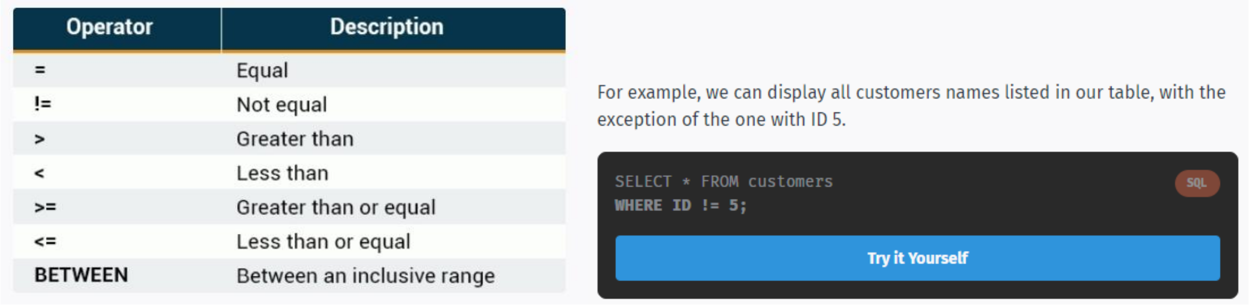
In the above table, to SELECT a specific record:

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

ID	FirstName	LastName	City
7	Daniel	Harris	New York

SQL Operations:

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

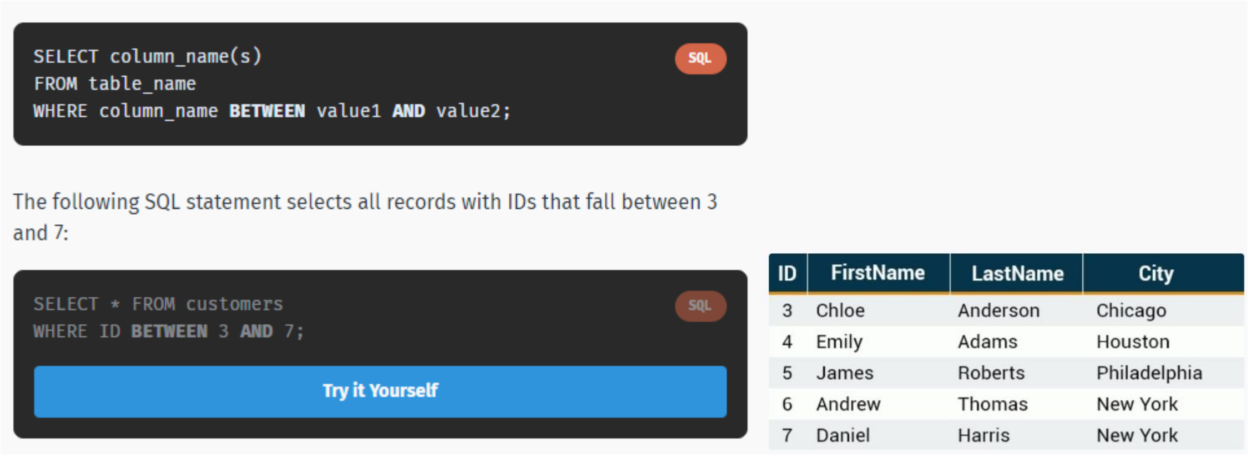
For example, we can display all customers names listed in our table, with the exception of the one with ID 5.

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

Row no:5 is excluded from the list.

BETWEEN:

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



```sql
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:

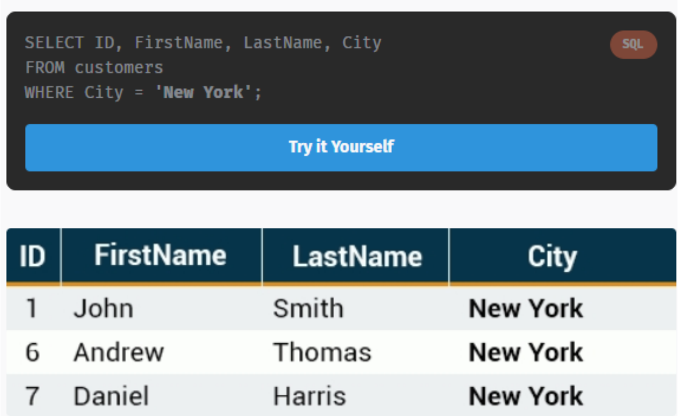
```sql
SELECT * FROM customers
WHERE ID BETWEEN 3 AND 7;
```

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

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

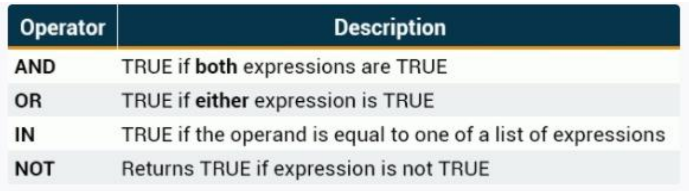


```sql
SELECT ID, FirstName, LastName, City
FROM customers
WHERE City = 'New York';
```

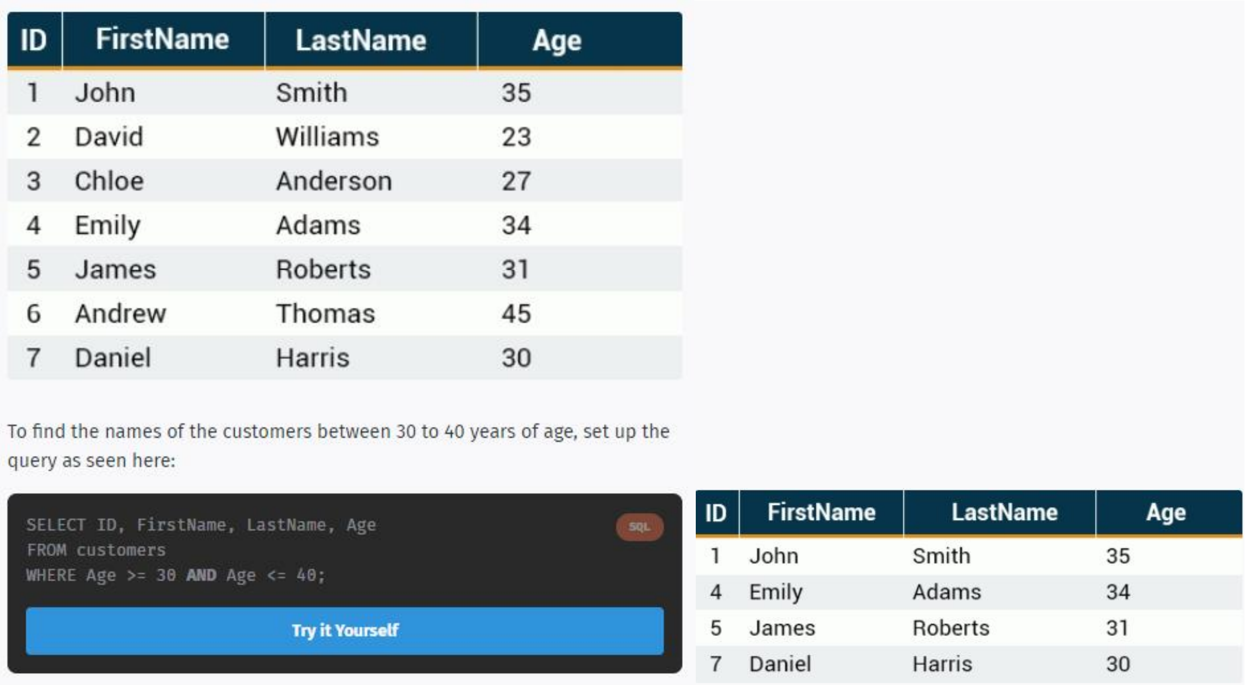
ID	FirstName	LastName	City
1	John	Smith	**New York**
6	Andrew	Thomas	**New York**
7	Daniel	Harris	**New York**

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



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

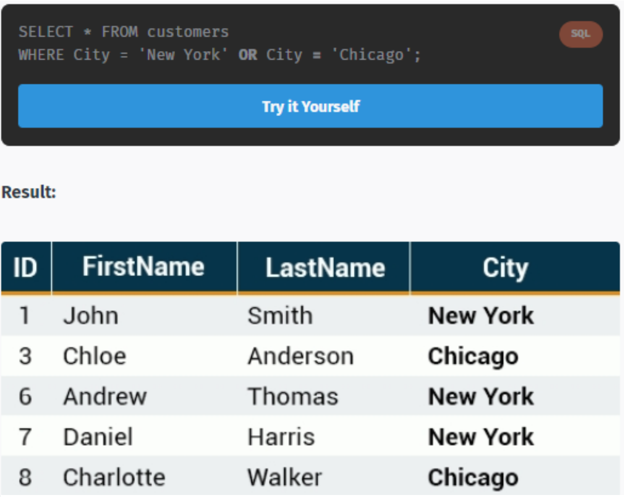
To find the names of the customers between 30 to 40 years of age, set up the query as seen here:

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

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

OR:

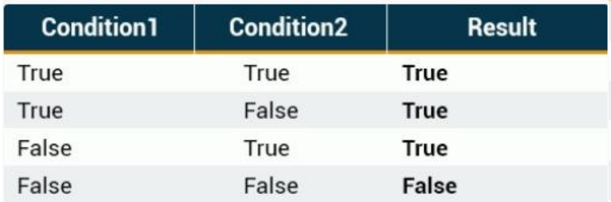
If you want to select rows that satisfy at least one of the given conditions, you can use the logical **OR** operator.



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

Result:

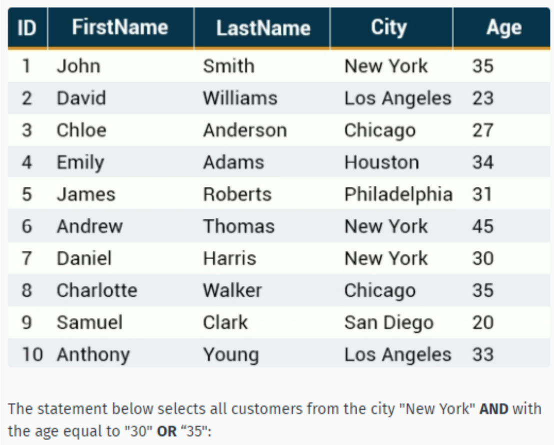
ID	FirstName	LastName	City
1	John	Smith	**New York**
3	Chloe	Anderson	**Chicago**
6	Andrew	Thomas	**New York**
7	Daniel	Harris	**New York**
8	Charlotte	Walker	**Chicago**



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

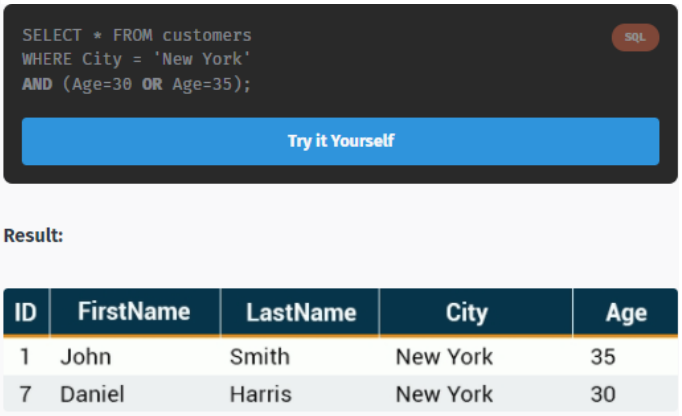
Combining AND & OR:

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.



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":



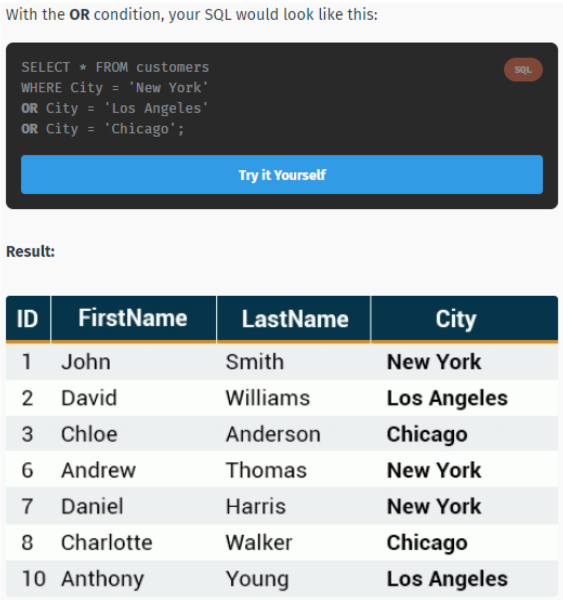
```
SELECT * FROM customers
WHERE City = 'New York'
AND (Age=30 OR Age=35);
```

Result:

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

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.

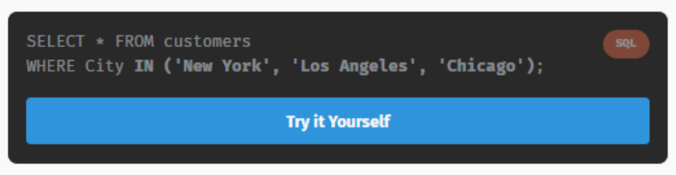


With the **OR** condition, your SQL would look like this:

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

Result:

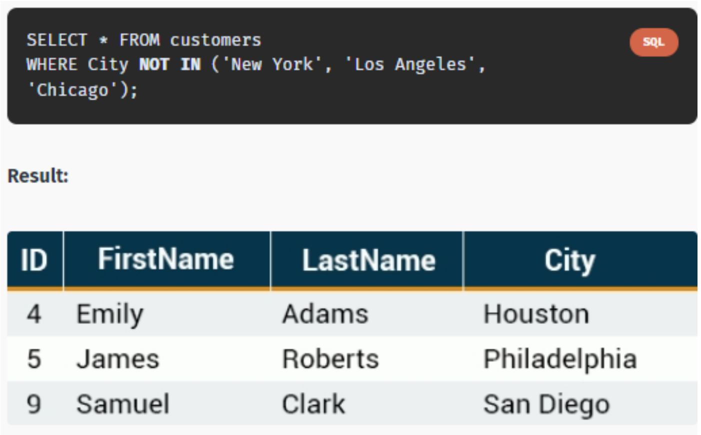
ID	FirstName	LastName	City
1	John	Smith	**New York**
2	David	Williams	**Los Angeles**
3	Chloe	Anderson	**Chicago**
6	Andrew	Thomas	**New York**
7	Daniel	Harris	**New York**
8	Charlotte	Walker	**Chicago**
10	Anthony	Young	**Los Angeles**



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

NOT IN operator:

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



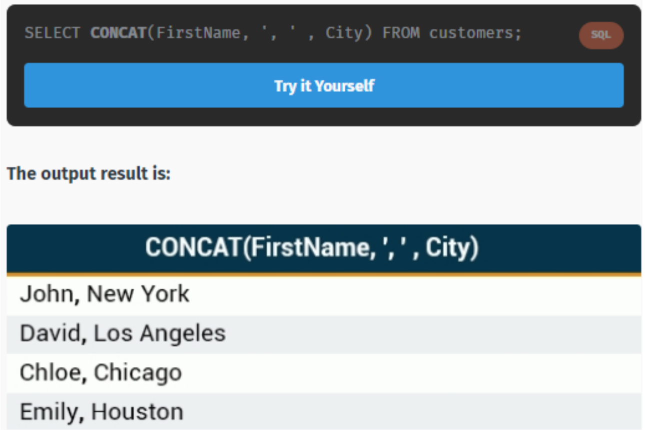
```sql
SELECT * FROM customers
WHERE City NOT IN ('New York', 'Los Angeles',
'Chicago');
```

Result:

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

CONCAT Function:

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



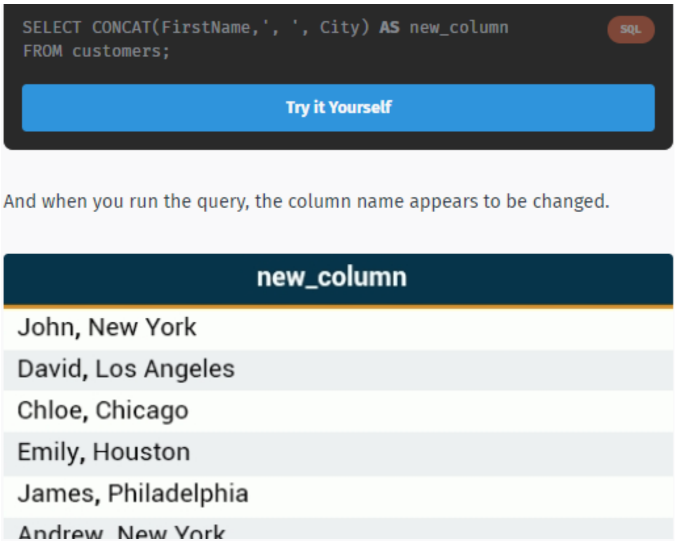
```sql
SELECT CONCAT(FirstName, ', ' , City) FROM customers;
```

The output result is:

CONCAT(FirstName, ', ' , City)
John, New York
David, Los Angeles
Chloe, Chicago
Emily, Houston

AS keyword:

A concatenation results in a new column. The default column name will be the CONCAT function.



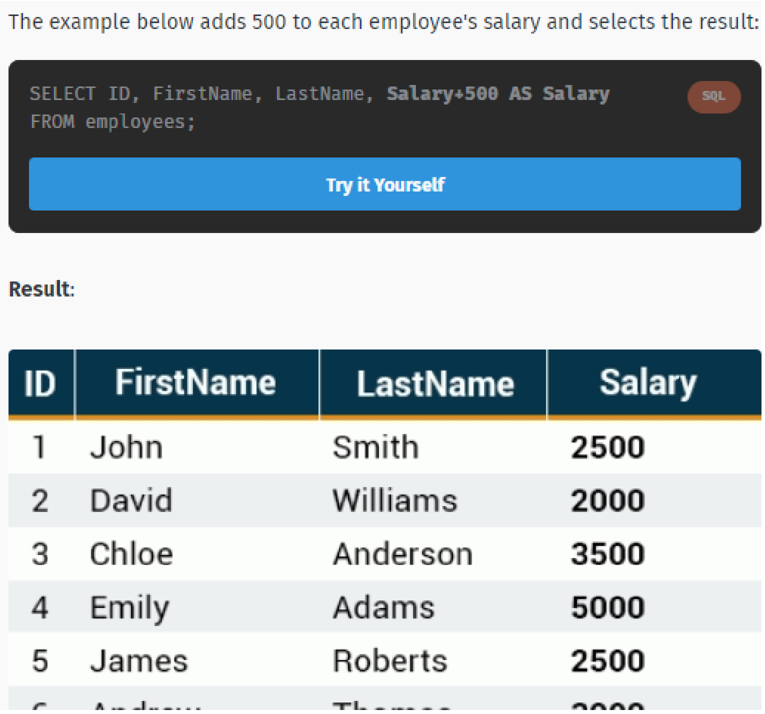
```
SELECT CONCAT(FirstName,', ', City) AS new_column
FROM customers;
```

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

Arithmetic Operators:

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



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

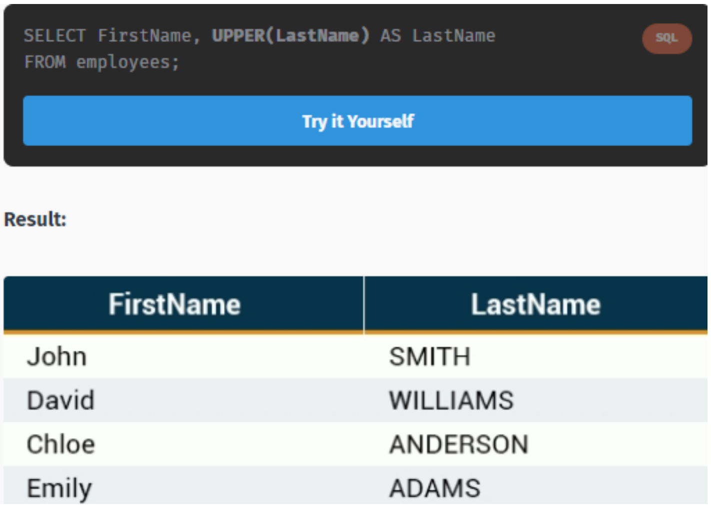
```
SELECT ID, FirstName, LastName, Salary+500 AS Salary
FROM employees;
```

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	[illegible]

UPPER and LOWER Function:

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



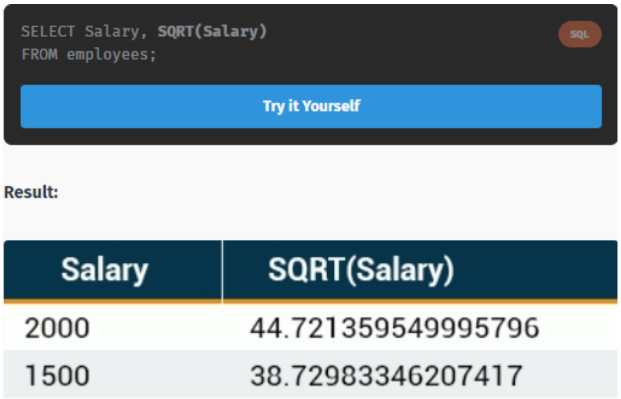
```
SELECT FirstName, UPPER(LastName) AS LastName
FROM employees;
```

Result:

FirstName	LastName
John	SMITH
David	WILLIAMS
Chloe	ANDERSON
Emily	ADAMS

SQRT and AVG functions:

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



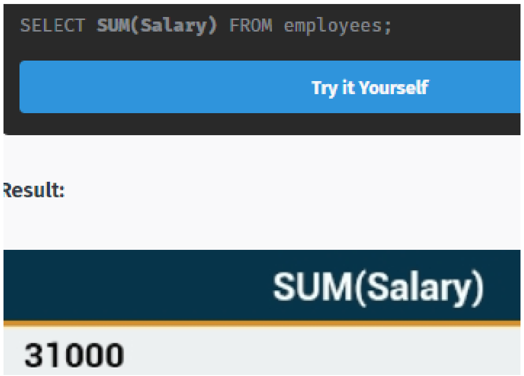
```
SELECT Salary, SQRT(Salary)
FROM employees;
```

Result:

Salary	SQRT(Salary)
2000	44.721359549995796
1500	38.72983346207417

SUM Function:

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



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

Result:

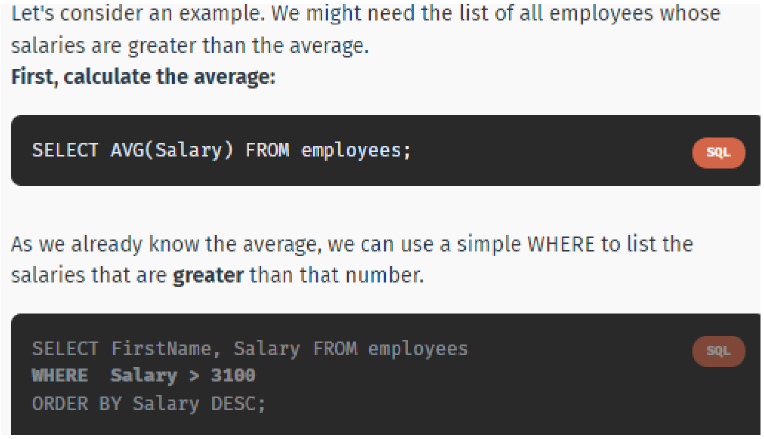
SUM(Salary)
31000

Subqueries:

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

DESC and ASC keyword:

The **DESC** keyword sorts results in **descending** order.
Similarly, **ASC** sorts the results in **ascending** order.



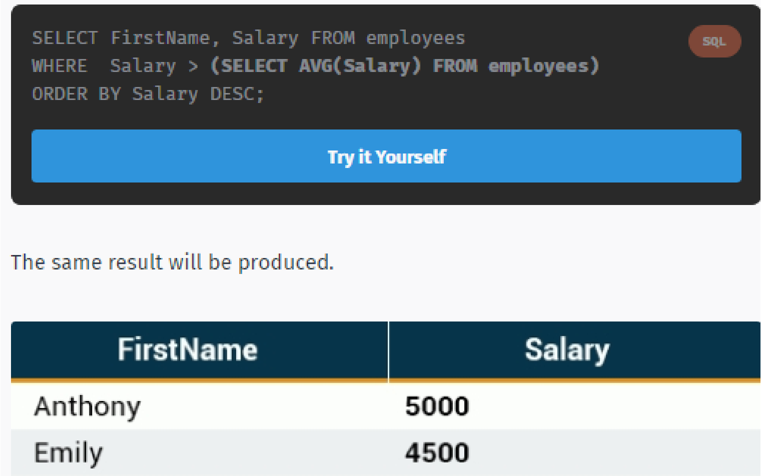
Let's consider an example. We might need the list of all employees whose salaries are greater than the average.
First, calculate the average:

```
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;
```

(instead of 2 codes, do this)



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

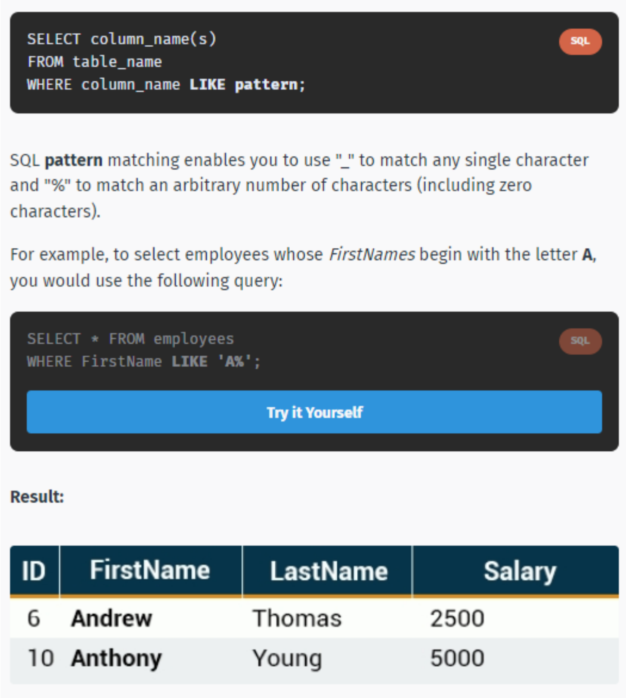
The same result will be produced.

FirstName	Salary
Anthony	5000
Emily	4500

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.

LIKE Operator:

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



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

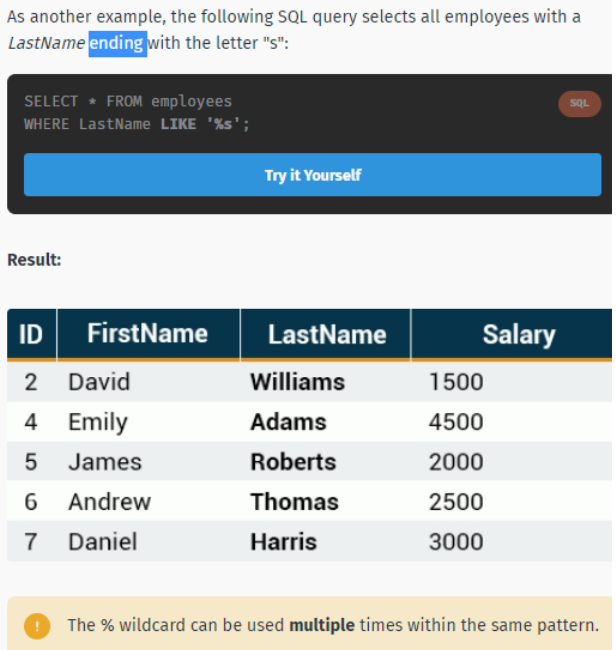
SQL **pattern** matching enables you to use "_" to match any single character and "%" to match an arbitrary number of characters (including zero characters).

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

```sql
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":

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

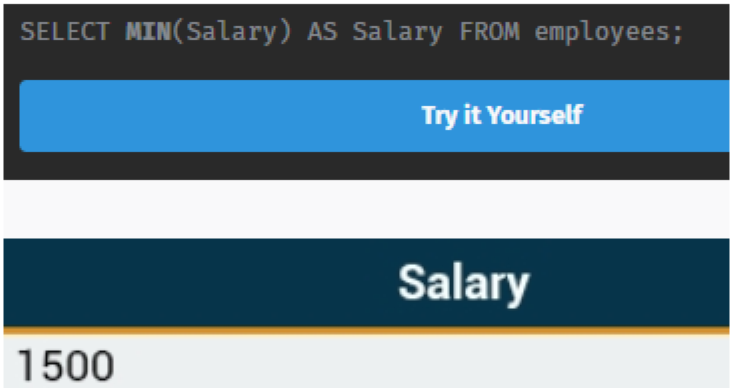
Result:

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.

MIN Function:

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



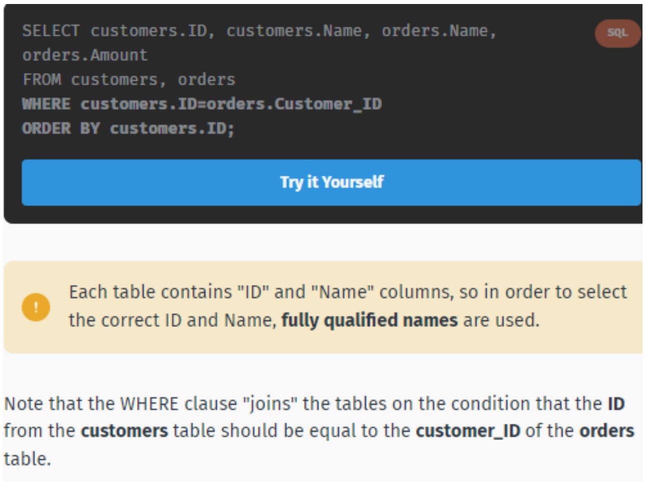
```sql
SELECT MIN(Salary) AS Salary FROM employees;
```

Salary
1500

TABLE OPERATIONS

Joining Tables:

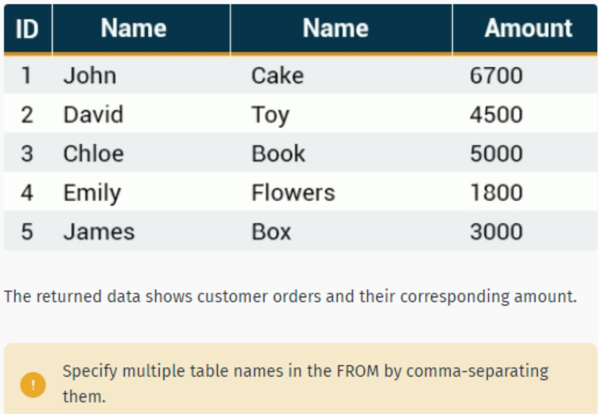
To join the two tables, specify them as a comma-separated list in the FROM clause:



```
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.



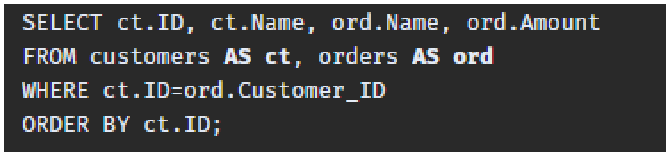
ID	Name	Name	Amount
1	John	Cake	6700
2	David	Toy	4500
3	Chloe	Book	5000
4	Emily	Flowers	1800
5	James	Box	3000

The returned data shows customer orders and their corresponding amount.

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

Custom Names:

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



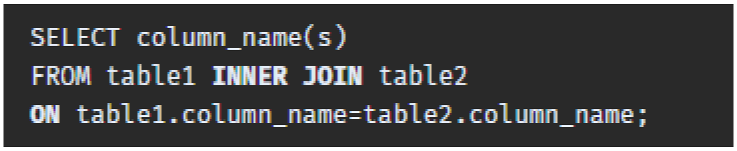
```
SELECT ct.ID, ct.Name, ord.Name, ord.Amount
FROM customers AS ct, orders AS ord
WHERE ct.ID=ord.Customer_ID
ORDER BY ct.ID;
```

Types of Joins:

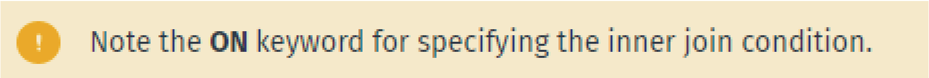
The following are the types of JOIN that can be used in MySQL:
- **INNER** JOIN
- **LEFT** JOIN
- **RIGHT** JOIN

INNER JOIN:

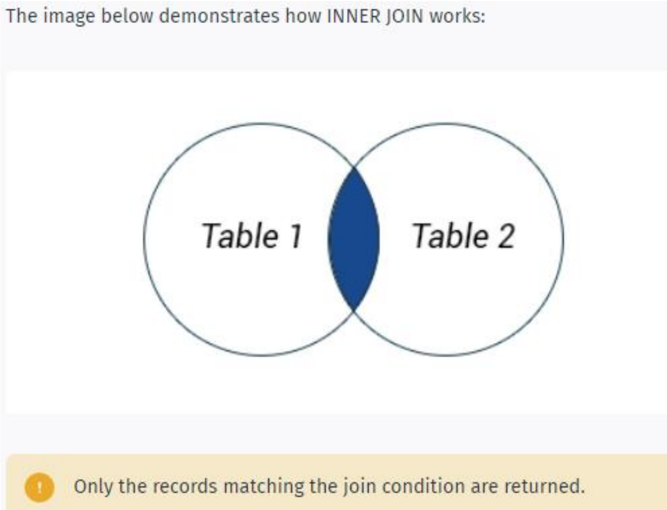
INNER JOIN is equivalent to JOIN. It returns rows when there is a match between the tables.



```
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.



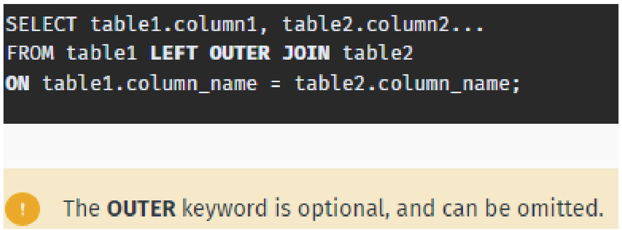
The image below demonstrates how INNER JOIN works:

Only the records matching the join condition are returned.

LEFT JOIN:

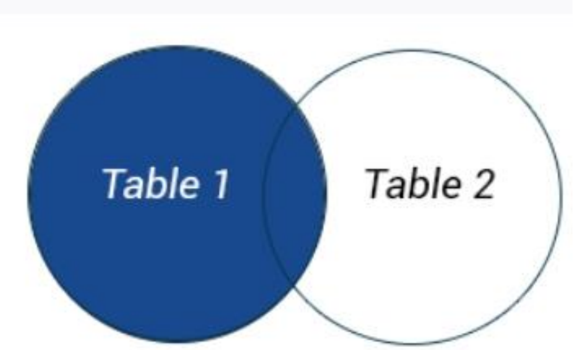
The **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table.

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.



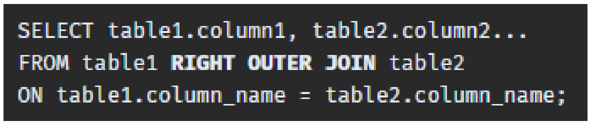
```
SELECT table1.column1, table2.column2...
FROM table1 LEFT OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

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

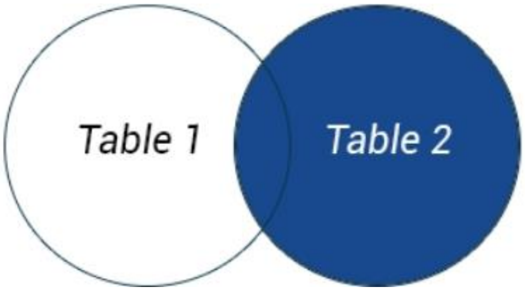


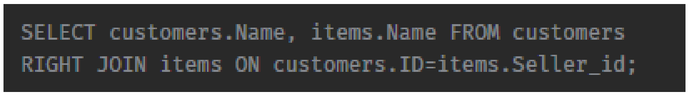
RIGHT JOIN:

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



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





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

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.

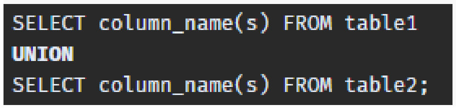
To accomplish this, use the **UNION** and **UNION ALL** operators.
UNION combines multiple datasets into a single dataset, and removes any existing duplicates.
UNION ALL combines multiple datasets into one dataset, but does not remove duplicate rows.

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

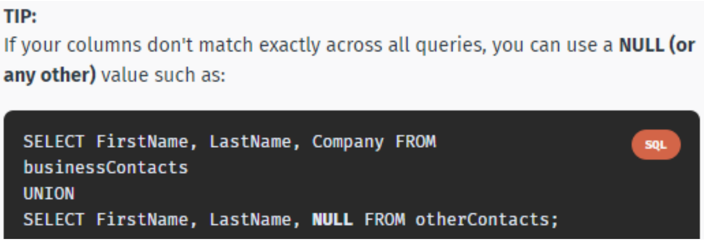
UNION:

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.

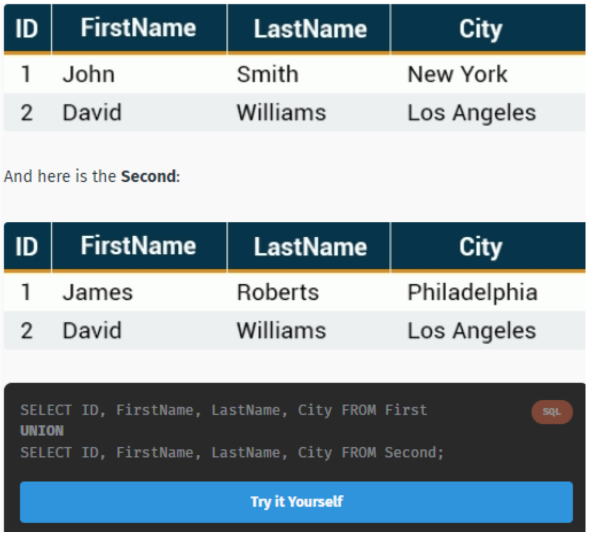


```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```



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;
```

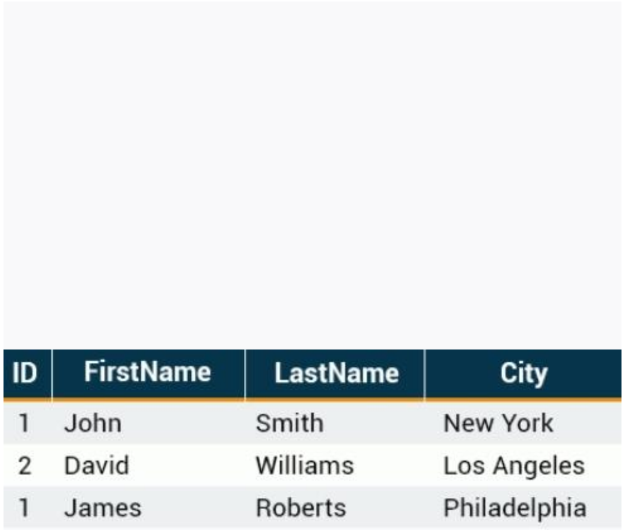


ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles

And here is the **Second**:

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

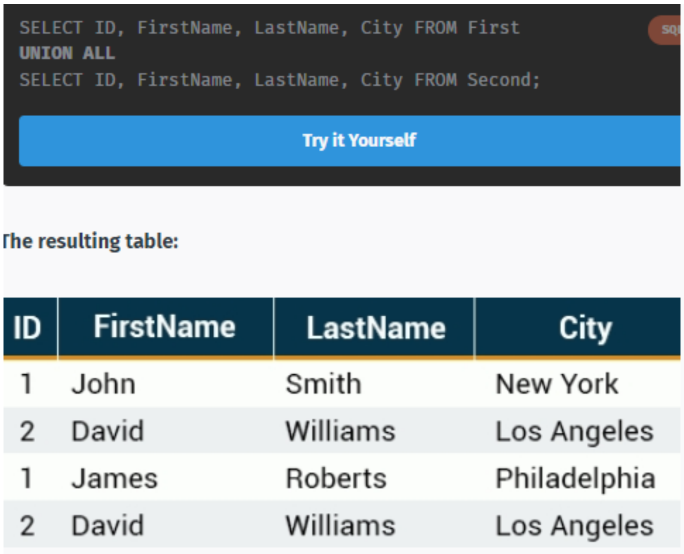
```
SELECT ID, FirstName, LastName, City FROM First
UNION
SELECT ID, FirstName, LastName, City FROM Second;
```



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

UNION ALL:

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



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

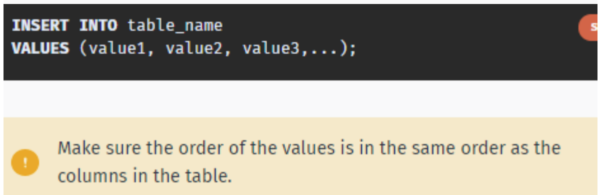
The resulting table:

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

DUPLICATES ARE INCLUDED!!!

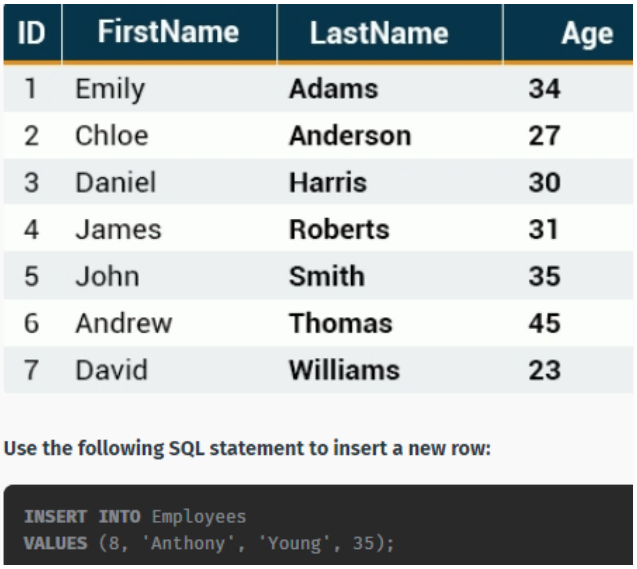
INSERT INTO:

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:



```sql
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.

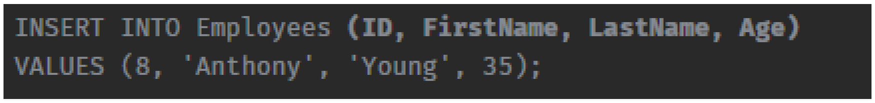


ID	FirstName	LastName	Age
1	Emily	Adams	34
2	Chloe	Anderson	27
3	Daniel	Harris	30
4	James	Roberts	31
5	John	Smith	35
6	Andrew	Thomas	45
7	David	Williams	23

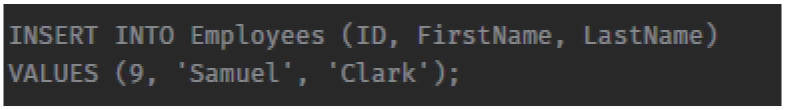
Use the following SQL statement to insert a new row:

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

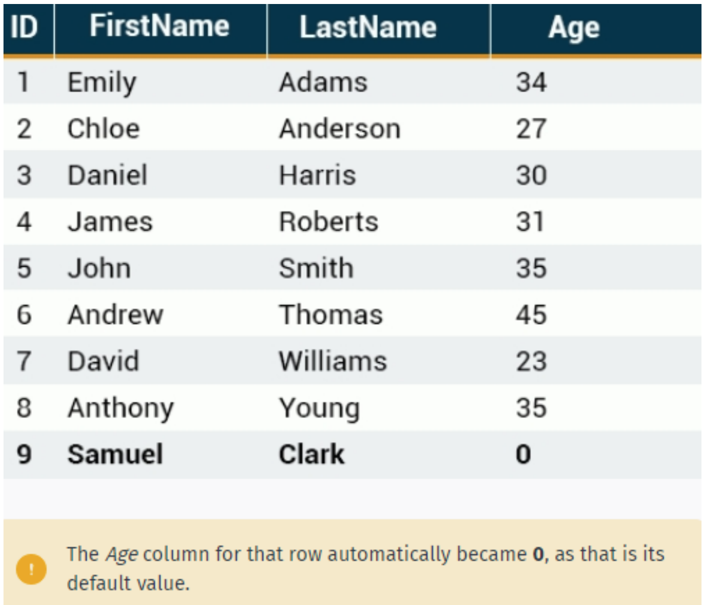
Alternatively, you can specify the table's column names in the INSERT INTO statement:



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



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

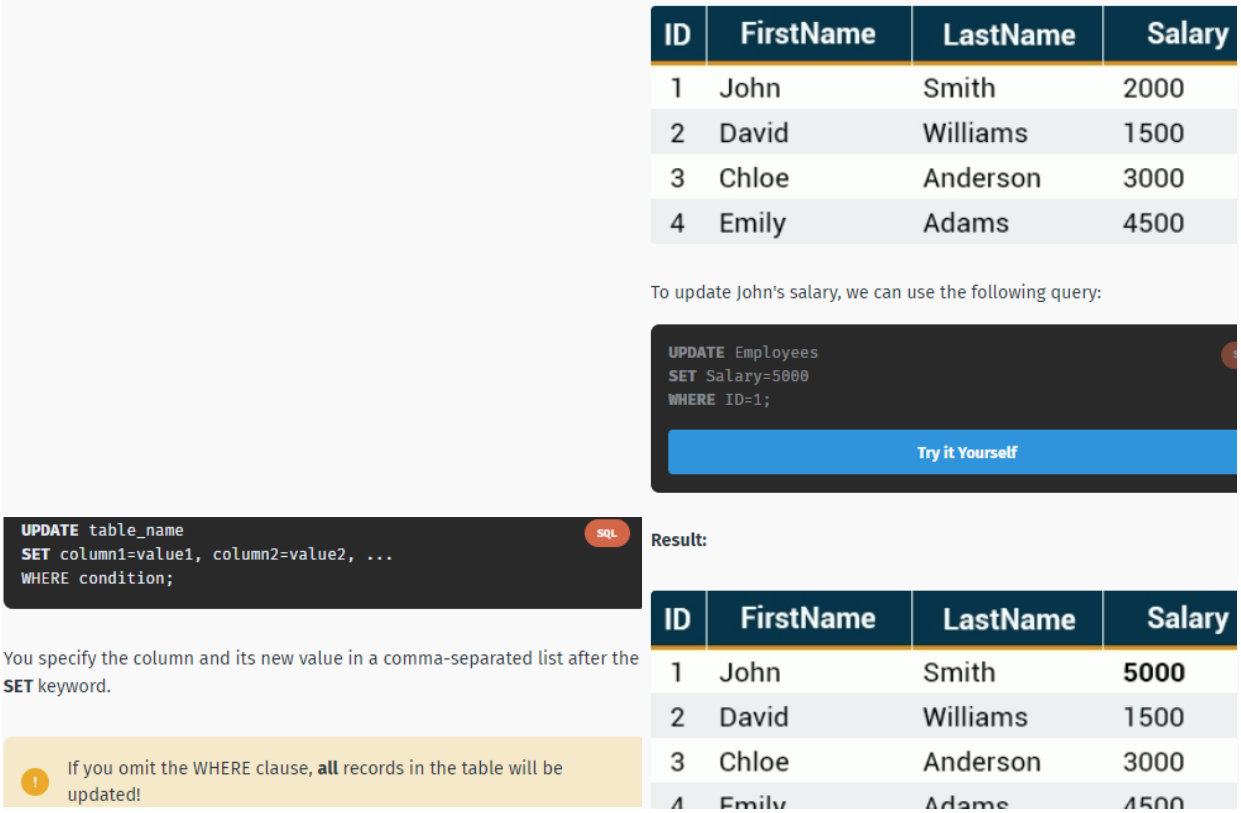


ID	FirstName	LastName	Age
1	Emily	Adams	34
2	Chloe	Anderson	27
3	Daniel	Harris	30
4	James	Roberts	31
5	John	Smith	35
6	Andrew	Thomas	45
7	David	Williams	23
8	Anthony	Young	35
9	**Samuel**	**Clark**	**0**

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

UPDATE statement:

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



```
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!

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

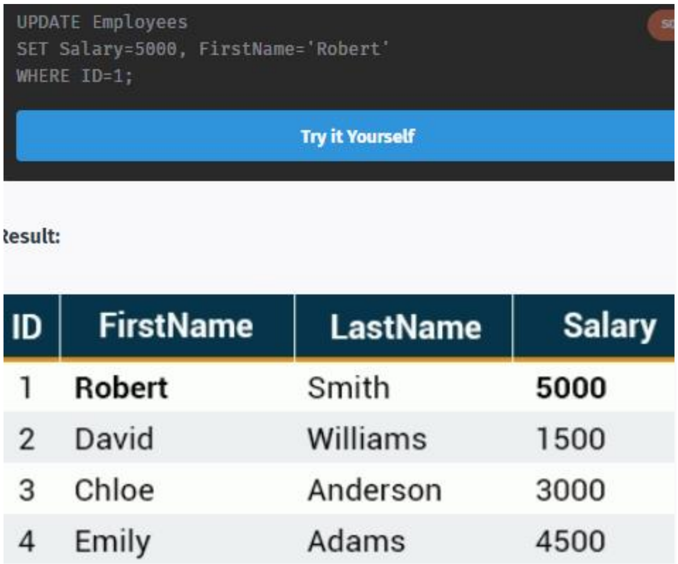
To update John's salary, we can use the following query:

```
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:



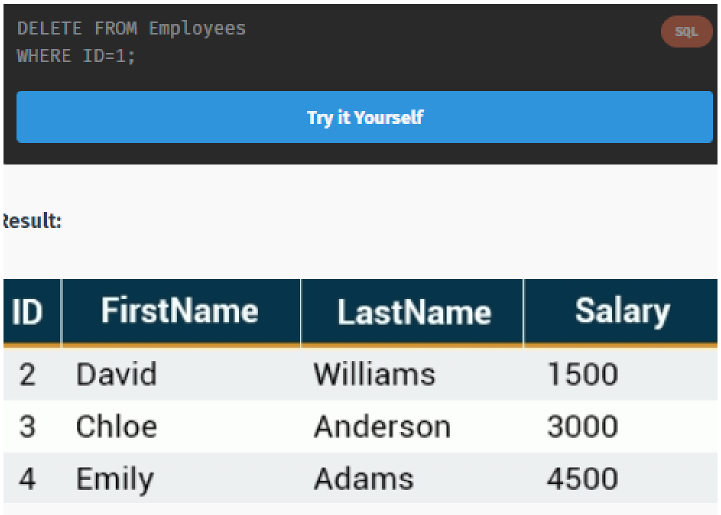
```
UPDATE Employees
SET Salary=5000, FirstName='Robert'
WHERE ID=1;
```

Result:

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

DELETE statement:

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



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

Result:

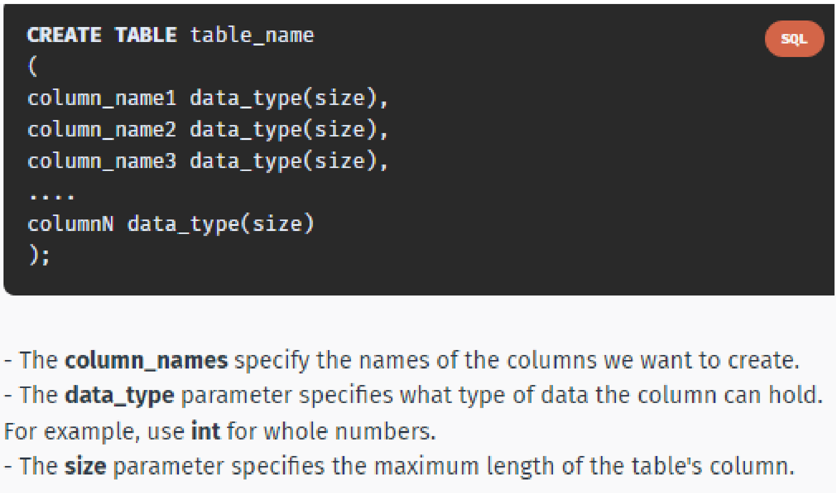
ID	FirstName	LastName	Salary
2	David	Williams	1500
3	Chloe	Anderson	3000
4	Emily	Adams	4500

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

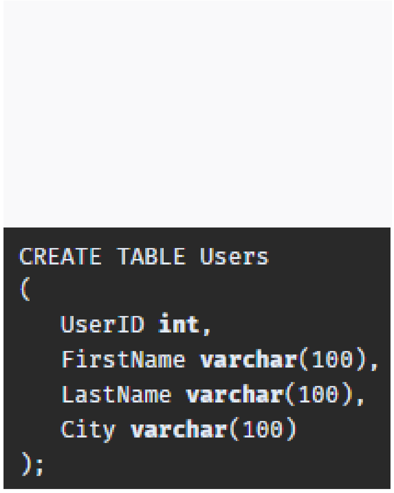
A single database can house hundreds of tables, each playing its own unique role in the database schema.
SQL tables are comprised of table rows and columns. Table columns are responsible for storing many different types of data, including numbers, texts, dates, and even files.

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



```sql
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.



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

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.

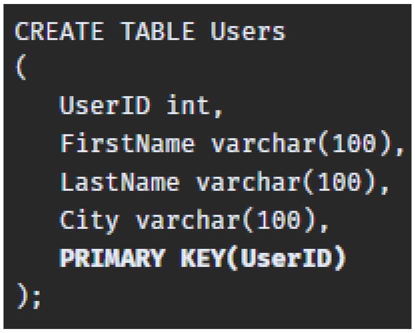
VARCHAR(M) - Variable-length character string. Max size is specified in parenthesis.

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.

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.



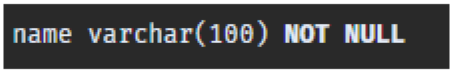
```
CREATE TABLE Users
(
   UserID int,
   FirstName varchar(100),
   LastName varchar(100),
   City varchar(100),
   PRIMARY KEY(UserID)
);
```

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.



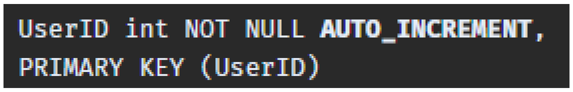
```
name varchar(100) NOT NULL
```

AUTO_INCREMENT:

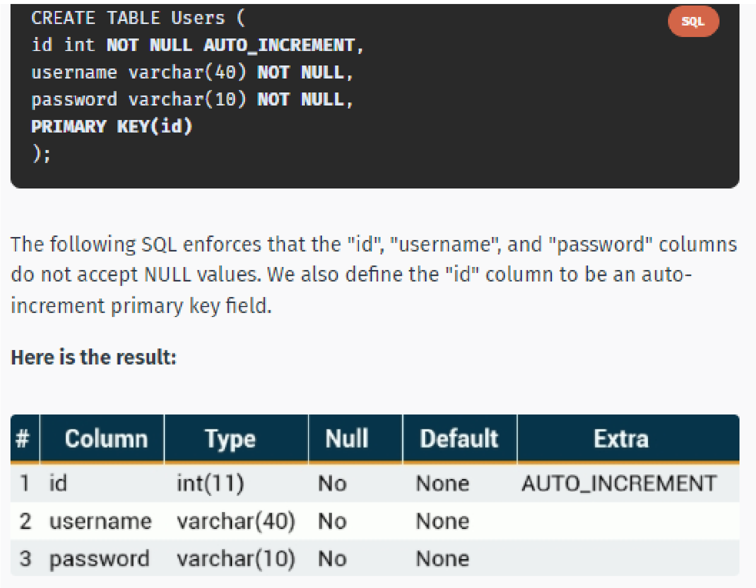
Auto-increment allows a unique number to be generated when a new record is inserted into a table.

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)
```



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

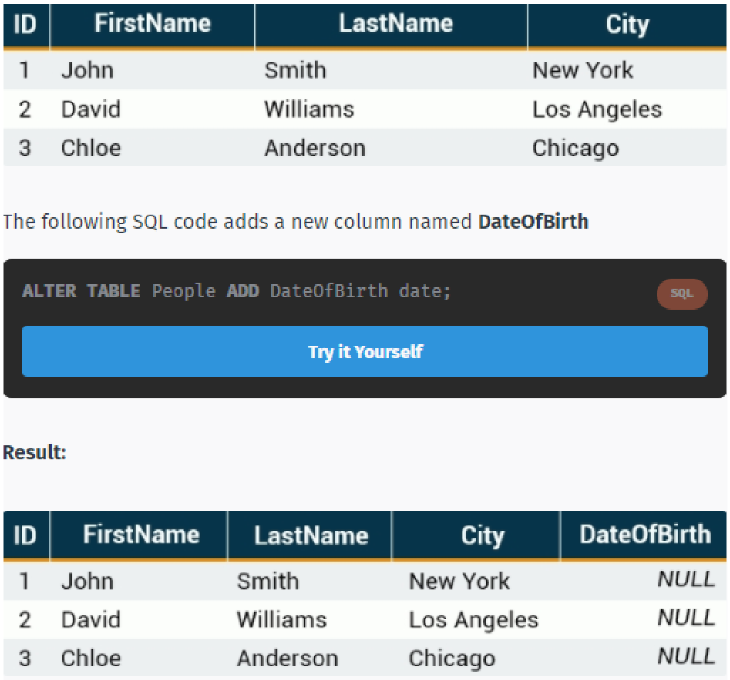
The following SQL enforces that the "id", "username", and "password" columns do not accept NULL values. We also define the "id" column to be an auto-increment primary key field.

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	

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.



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**

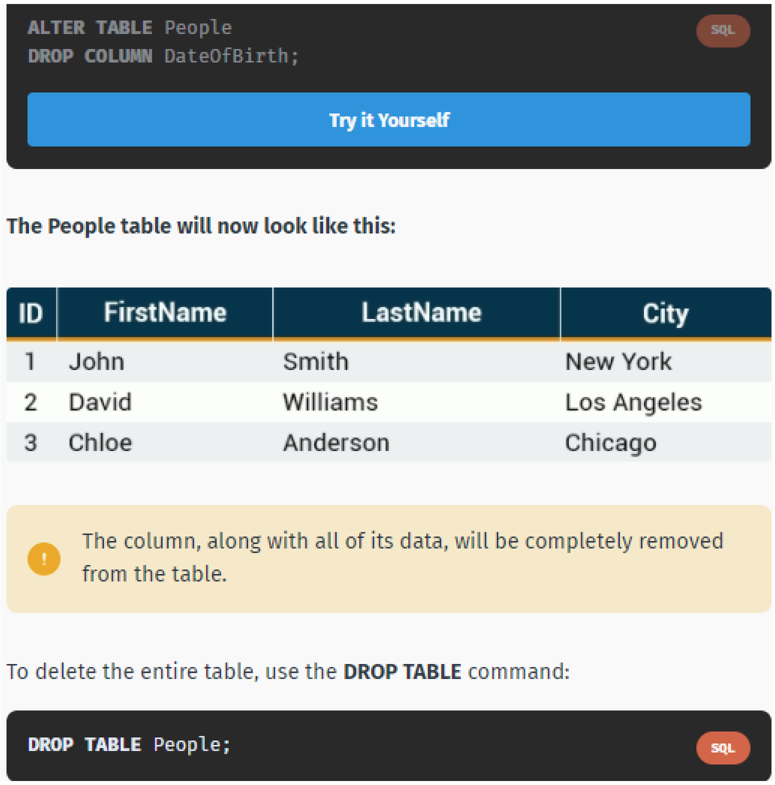
```
ALTER TABLE People ADD DateOfBirth date;
```

Result:

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

DROP:

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



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

The People table will now look like this:

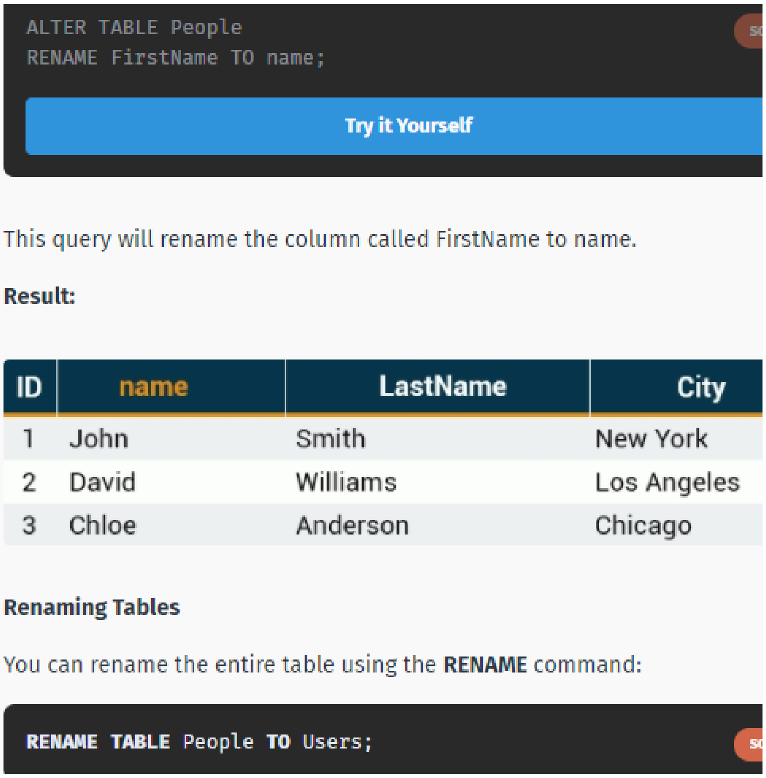
ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
3	Chloe	Anderson	Chicago

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:

```sql
DROP TABLE People;
```

RENAME:



```sql
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:

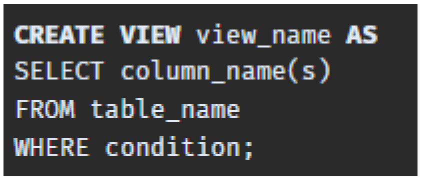
```sql
RENAME TABLE People TO Users;
```

VIEW:

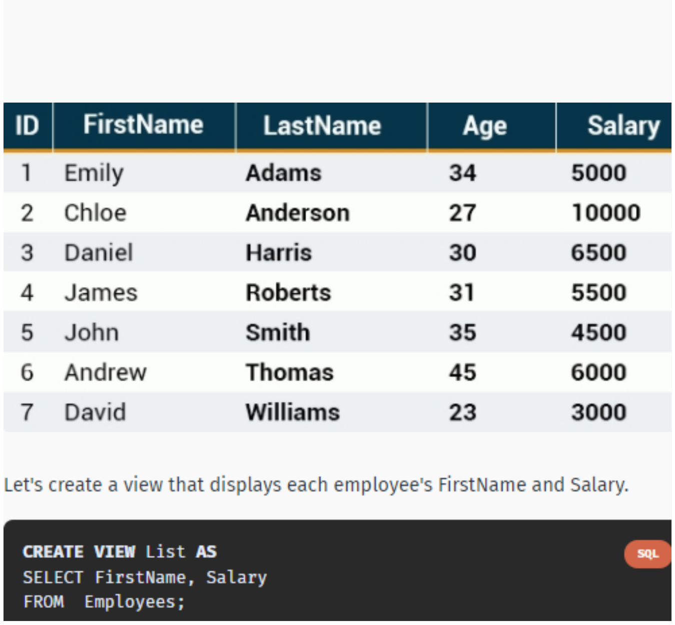
In SQL, a VIEW is a **virtual table** that is based on the result-set of an SQL statement.

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.



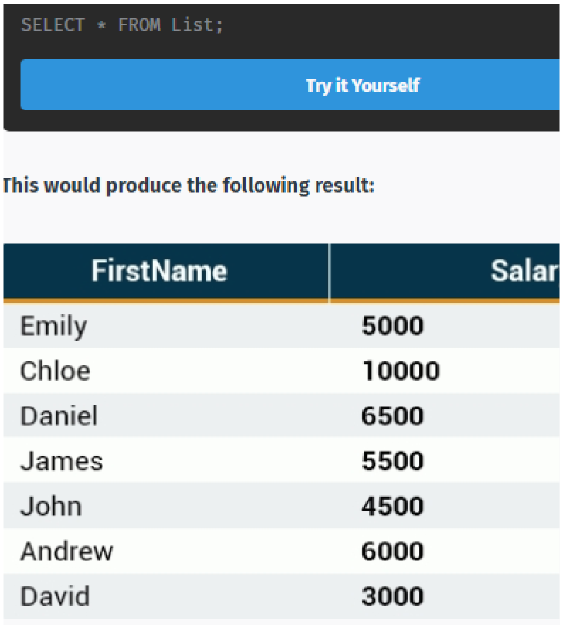
```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition;
```



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;
```

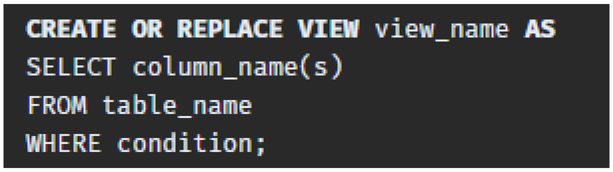


```
SELECT * FROM List;
```

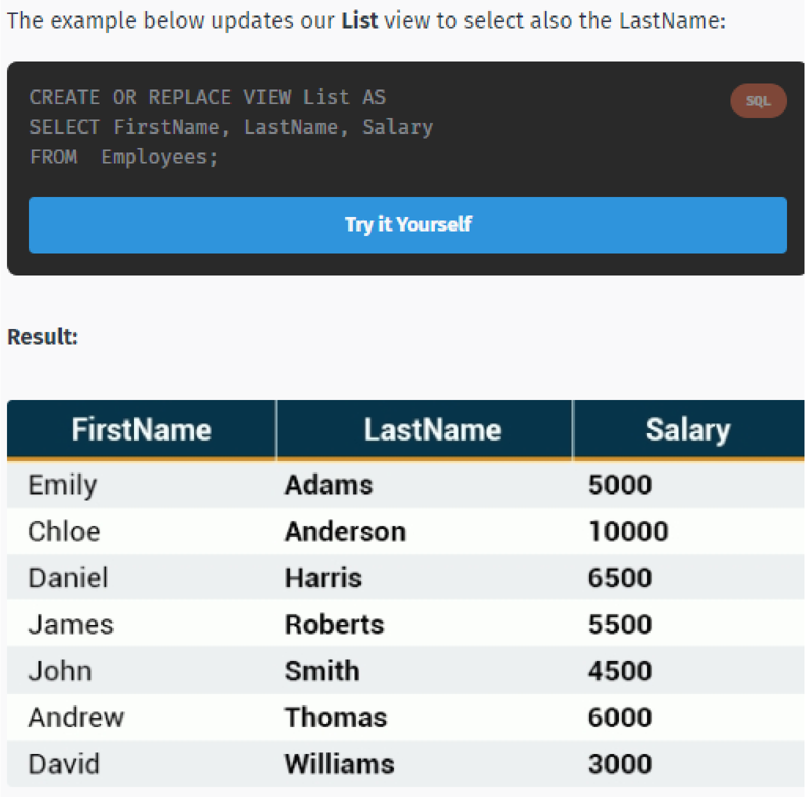
This would produce the following result:

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

REPLACE VIEW -aka updating a view:



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

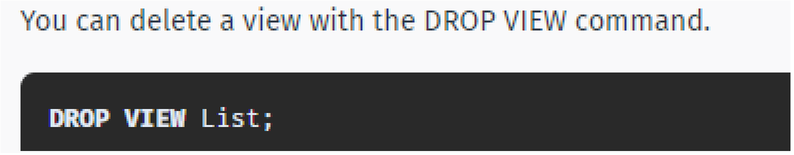


The example below updates our **List** view to select also the LastName:

```sql
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.

```sql
DROP VIEW List;
```