**SQL Pre preparation content**

SQL Syntax

SQL Statements

Most of the actions you need to perform on a database are done with SQL statements.

SQL statements consists of keywords that are easy to understand.

The following SQL statement returns all records from a table named "Customers":

In this tutorial we will teach you all about the different SQL statements.

Database Tables

A database most often contains one or more tables. Each table is identified by a name (e.g. "Customers" or "Orders"), and contain records (rows) with data.

In this tutorial we will use the well-known Northwind sample database (included in MS Access and MS SQL Server).

Below is a selection from the [**Customers**](https://www.w3schools.com/sql/trysql.asp?filename=trysql_customers) table used in the examples:

The table above contains five records (one for each customer) and seven columns (CustomerID, CustomerName, ContactName, Address, City, PostalCode, and Country).

Keep in Mind That...

* SQL keywords are NOT case sensitive: select is the same as SELECT

In this tutorial we will write all SQL keywords in upper-case.

Semicolon after SQL Statements?

Some database systems require a semicolon at the end of each SQL statement.

Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.

In this tutorial, we will use semicolon at the end of each SQL statement.

Some of The Most Important SQL Commands

* SELECT - extracts data from a database
* UPDATE - updates data in a database
* DELETE - deletes data from a database
* INSERT INTO - inserts new data into a database
* CREATE DATABASE - creates a new database
* ALTER DATABASE - modifies a database
* CREATE TABLE - creates a new table
* ALTER TABLE - modifies a table
* DROP TABLE - deletes a table
* CREATE INDEX - creates an index (search key)
* DROP INDEX - deletes an index

**SQL Commands with Examples**

**Example table**

Throughout this tutorial, we will use the **films and soundtracks** tables.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **films** | | | | | | | |
| movie\_name | director | revenue | | date | genre | language | |
| Avengers: Age of Ultron | J. Whedon | 1400000000 | | 2015 | Action | English | |
| Amelie | Jean-Pierre Jeunet | 174000000 | | 2002 | Drama | French | |
| **soundtracks** | | | | | | | |
| soundtrack | | | composer | | | | date |
| Inception: Music from the Motion Picture | | | Hans Zimmer | | | | 2010 |
| Le Fabuleux Destin D’Amelie Poulain | | | Yann Tiersen | | | | 2001 |

**1. SELECT**

The SELECT statement is used to specify which columns of a database table should be included in the result.

This example selects only the columns movie\_name and director; other columns are not returned.

SELECT movie\_name, director

FROM films;

This code is written in SQL, which is a programming language used for managing and manipulating data stored in relational databases.

• The code is a SQL query that selects the columns "movie\_name" and "director" from the "films" table.

• The "SELECT" keyword is used to specify the columns to be retrieved, while the "FROM" keyword is used to specify the table from which the data is to be retrieved.

• In summary, this code retrieves the movie names and directors from the "films" table.

For selecting all the columns from a table, you can run SELECT \* (pronounced "star"). This example selects all the columns from the **films** table.

SELECT \*

FROM films;

This code is written in SQL, which is a language used for managing and manipulating relational databases.

• The code is a SQL query that selects all columns and rows from the "films" table in the database.

• The asterisk (\*) is a wildcard character that represents all columns in the table.

• When this code is executed, the database will return all the data stored in the "films" table.

**2. LIMIT**

Limiting the number of rows returned from a table is a useful trick for speeding query time. To limit the number of rows, you can use the LIMIT command.  This example selects all columns from the **films** table and then limits the results to the first ten rows.

SELECT \*

FROM films

LIMIT 10;

This code is written in SQL, which is a programming language used for managing and manipulating data stored in relational databases.

• The code is selecting all columns (indicated by the asterisk symbol) from the "films" table in the database.

• The "LIMIT 10" statement limits the output to the first 10 rows of the table.

• In summary, this code is retrieving the first 10 rows of data from the "films" table in the database.

**3. AS**

To rename a column or a table when returning results, you can use the AS command to set an alias for your outputs. This example selects the movie\_name column and gives it the movie\_title alias.

SELECT movie\_name AS movie\_title

FROM films;

This code is written in SQL.

• The code selects the column "movie\_name" from the table "films" and renames it as "movie\_title" using the "AS" keyword.

• The result of this query will be a list of movie titles from the "films" table.

**4. SELECT DISTINCT**

Datasets often contain duplicate rows or values in a column. Combining SELECT with DISTINCT drops duplicates. This example returns the unique values in the director column.

SELECT DISTINCT director

FROM films;

This code is written in SQL.

• The code selects all distinct values of the "director" column from the "films" table.

• The DISTINCT keyword ensures that only unique values are returned, so if there are multiple films directed by the same person, their name will only appear once in the result set.

• Overall, this code is retrieving a list of all unique directors from the "films" table.

**5. COUNT**

COUNT() returns the number of rows in the table or group. This example returns the number of rows in the **films** table, then renames the result as number\_of\_films.

SELECT COUNT(\*) AS number\_of\_films

FROM films;

This code is written in SQL.

• It selects the count of all rows in the "films" table and aliases the result as "number\_of\_films".

• The asterisk (\*) is a wildcard character that represents all columns in the table.

• So, this query will return a single value representing the total number of films in the "films" table.

**6. MIN**

MIN() returns the minimum value in a numeric column. For text columns, MIN() returns the first value alphabetically. This example returns the movie with the lowest amount of revenue.

SELECT MIN(revenue) AS minimum\_revenue,

FROM films;

This code is written in SQL.

• The code selects the minimum value of the "revenue" column from the "films" table and assigns it the alias "minimum\_revenue".

• The result will be a single row with one column, which will display the minimum revenue value in the "minimum\_revenue" column.

**7. MAX**

MAX() returns the maximum value in a numeric column. For text columns, MAX() returns the last value alphabetically. This example returns the movie with the maximum amount of revenue.

SELECT MAX(revenue) AS maximum\_revenue,

FROM films;

This code is written in SQL.

• The code selects the maximum value of the "revenue" column from the "films" table and assigns it the alias "maximum\_revenue".

• The "MAX" function is used to find the highest value in the "revenue" column.

• The comma after "maximum\_revenue" is a syntax error and should be removed.

**8. SUM**

SUM() returns the total of numeric values. This example returns the total revenue of all the films listed in the **films** table.

SELECT SUM(revenue) AS total\_revenue

FROM films;

This code is written in SQL.

• It selects the sum of the revenue column from the films table and aliases the result as "total\_revenue".

• The result will be a single row with a single column containing the total revenue of all the films in the table.

**9. AVERAGE**

AVERAGE calculates the arithmetic mean of a column. This example returns the average revenue of all the films listed in the **films** table.

SELECT AVERAGE(revenue) AS average\_earned

FROM films;

This code is written in SQL.

• It selects the average value of the "revenue" column from the "films" table and renames the result as "average\_earned".

• The "AS" keyword is used to assign a new name to the result of the aggregation function.

• In summary, this code calculates the average revenue earned from all the films in the "films" table and returns it as a single value with the name "average\_earned".

**10. WHERE**

The WHERE clause filters rows that match a certain condition. For example, below, we are filtering films that earned more than 500 million dollars

SELECT revenue

FROM films

WHERE revenue > 500000000;

This is a SQL code snippet.

• This code selects the revenue column from the films table where the revenue is greater than 500,000,000.

• The ">" symbol is used to compare the revenue column with the value of 500,000,000.

• The "WHERE" clause is used to filter the results based on the condition specified.

• In summary, this code retrieves the revenue of films that have generated more than 500 million dollars in revenue.

Other conditional operators can be used such as <, >, =>, <=, == (equals), != (not equals) for filtering.

**11. HAVING**

HAVING clause is similar to the WHERE clause, but it can only be used with aggregate functions while WHERE can’t. For example, in the below query, we are choosing all the movie genres that have at least 50 movies in their category:

SELECT movie\_name, director, date, COUNT(genre)

FROM films

GROUP BY genre

HAVING COUNT(genre) >= 50;

This is a SQL code that selects the movie name, director, and date from the "films" table and counts the number of times each genre appears in the table.

• It then groups the results by genre and filters the results to only show genres that appear 50 or more times.

• The "GROUP BY" clause is used to group the results by genre, which means that the results will be grouped together based on the genre column.

• The "HAVING" clause is used to filter the results based on the count of each genre.

• The ">=" symbol is an HTML entity that represents the greater than or equal to sign (>=) in SQL.

• Overall, this code is used to find the genres that appear frequently in the "films" table.

Here is another example that groups films by age rating and only choose the ratings with average revenue of over 100 million:

SELECT movie\_name, director, date, revenue

FROM films

GROUP BY age\_rating

HAVING AVERAGE(genre) >= 100;

This code is written in SQL.

• The code selects the columns "movie\_name", "director", "date", and "revenue" from the table "films".

• It then groups the results by the column "age\_rating".

• The "HAVING" clause is used to filter the results of the "GROUP BY" clause.

• In this case, it filters the results to only include groups where the average value of the "genre" column is greater than or equal to 100.

• Note that the syntax for the comparison operator in the "HAVING" clause is ">" which is the HTML entity for the ">" symbol.

• In SQL, the ">" symbol is used instead.

**12. AND**

AND operator is used when filtering rows that match more than one condition. In the below example, we will filter for English films that grossed more than 500 million dollars.

SELECT \*

FROM films

WHERE revenue > 500000000 AND

language == "English";

This is a SQL query that selects all columns from the "films" table where the revenue is greater than 500,000,000 and the language is English.

• The ">" symbol is used to compare the revenue column to the value of 500,000,000.

• The "AND" operator is used to combine the two conditions, so both must be true for a row to be selected.

• The "==" operator is used to compare the language column to the string "English".

• The semicolon at the end of the query indicates the end of the statement.

**13. OR**

OR is another conditional operator that allows you to subset rows if any of the conditions separated by OR are true. This example returns English films that earned less than 100 million dollars or French films that earned more than 500 million dollars.

SELECT \*

FROM films

WHERE (revenue < 100000000 AND language == "English") OR

(revenue > 500000000 AND language == “French”);

**14. BETWEEN**

BETWEEN allows you to subset rows within a certain range, which makes WHERE clauses simpler and easier to read. In the example above, we choose all the films released between 2020 and 2022.

SELECT \*

FROM films

WHERE date BETWEEN 2020 AND 2022;

This code is written in SQL.

• The code selects all columns (**\***) from the **films** table where the **date** column falls between the years 2020 and 2022.

• The **BETWEEN** operator is used to specify a range of values for the **date** column.

• This query will return all rows from the **films** table where the **date** column has a value between January 1, 2020 and December 31, 2022.

**15. IN**

The IN operator is a shorthand for multiple OR statements. This example returns all films that were released in either of these years — 1998, 1966, 2001, and 2012.

SELECT movie\_name, date, revenue

FROM films

WHERE date IN (1998, 1966, 2001, 2012);

This code is written in SQL.

• The code selects the movie\_name, date, and revenue columns from the films table.

• The WHERE clause filters the results to only include rows where the date column matches one of the specified values (1998, 1966, 2001, or 2012).

• In other words, this code retrieves the movie name, date, and revenue for any films that were released in 1998, 1966, 2001, or 2012.

**16. LIKE**

The LIKE operator lets you search for patterns in a text column using special strings called wildcards. Wildcards let you find text strings that fit a specific pattern. For example, using “A% W%” wildcard on the director column will find all directors with a first name starting with A and the last name beginning with W.

SELECT \*

FROM films

WHERE director LIKE "A% W%";

This is a SQL code that selects all columns from the "films" table where the "director" column starts with the letter "A" and has a "W" somewhere later in the name.

• The "%" symbol is a wildcard character that can represent any number of characters.

• So, in this case, the query will return all films where the director's name starts with "A" and has a "W" somewhere later in the name, regardless of the number of characters in between.

**17. GROUP BY**

GROUP BY lets you group rows based on column values. GROUP BY is usually used with aggregate functions like COUNT, MIN, MAX, AVERAGE, and more. In the example below, we find the average revenue of each movie genre.

SELECT AVERAGE(revenue), movie\_name, director, date

FROM films

GROUP BY genre;

This code is written in SQL.

• It selects the average revenue, movie name, director, and date from the "films" table.

• The data is then grouped by genre using the "GROUP BY" clause.

• This means that the results will be grouped by each unique value in the "genre" column.

• The "AVERAGE" function is used to calculate the average revenue for each genre.

• The result will show the average revenue, movie name, director, and date for each genre in the "films" table.

We could have used MIN, MAX, or COUNT(revenue) to find the lowest earning, highest earning, or the number of movies in each genre. Note that the GROUP BY statement practically does not affect the query if it isn’t used with an aggregate function.

**18. ORDER BY**

ORDER BY lets you order rows based on a column value. You can order by ascending (default) or descending order by adding the ASC or DESC. This example orders revenue by ascending order.

SELECT \*

FROM films

ORDER BY revenue ASC;

This code is written in SQL.

• It selects all columns from the "films" table and orders the results in ascending order based on the "revenue" column.

• This means that the films with the lowest revenue will appear first in the results.

**19. UPDATE**

UPDATE is used to change the values of individual cells in an existing table. It is used with the SET keyword. This example updates the director “J. Whedon”, to “Joss Whedon”.

UPDATE films

SET director = "Joss Whedon"

WHERE director = "J. Whedon"

This is SQL code that updates the "director" column in the "films" table.

• The code sets the value of the "director" column to "Joss Whedon" for all rows where the current value of the "director" column is "J.

• Whedon".

• In other words, it replaces all instances of "J.

• Whedon" with "Joss Whedon" in the "director" column of the "films" table.

The WHERE clause is crucial when writing UPDATE statements. Without it, the above query would have made all films directed by Joss Whedon.

**20. ALTER TABLE**

You can use the ALTER TABLE statement to modify the properties of the table and its columns (not actual cell values). For example, changing column names, adding new columns, dropping them, or changing their data type. The examples below showcase dropping the date column and adding the age\_rating column.

ALTER TABLE films

DROP COLUMN date;

This code is written in SQL.

• It is an ALTER TABLE statement that modifies the structure of a table named "films".

• Specifically, it drops (removes) the column named "date" from the table.

• This means that any data previously stored in that column will be permanently deleted.

Was this helpful? Yes No

ALTER TABLE films

ADD COLUMN age\_rating;

This code is written in SQL.

• The code is adding a new column called "age\_rating" to the "films" table using the ALTER TABLE statement.

• This new column will be empty and will need to be populated with data.

**21. CREATE TABLE**

CREATE TABLE creates a new table in a database. Below, we are creating a “libraries” table with four columns —an integer column called  id, a character column called name, a character column called version, and an integer column called num\_downloads.

CREATE TABLE libraries (

lib\_id int,

name varchar(100),

version varchar(100),

num\_downloads int

)

This code is written in SQL and creates a new table called "libraries" with four columns: "lib\_id" (an integer), "name" (a string of up to 100 characters), "version" (a string of up to 100 characters), and "num\_downloads" (an integer).

• The "CREATE TABLE" statement is used to define the structure of a new table in a database.

• In this case, the "libraries" table is being created with the specified columns and data types.

**22. INSERT INTO**

INSERT INTO statement can be used to add new rows to a table. In this example, we add the movie “Doctor Strange” to the **films** table.

INSERT INTO films (movie\_name, director, revenue, date, genre, language)

VALUES ("Doctor Strange 2", “Sam Raimi”, 409000000, 2022, “Action”, “English”)

**23. INNER JOIN**

The INNER JOIN command selects rows that have matching values in both tables. For example, in the query below, we are the films and soundtracks tables on a common date column which returns films and soundtracks that were released in the same years.

SELECT \*

FROM films

INNER JOIN soundtracks

ON films.date = soundtracks.date

LIMIT 10;

This is a SQL code that selects all columns from the "films" table and the "soundtracks" table where the "date" column in both tables match.

• The INNER JOIN keyword is used to combine the two tables based on the matching "date" column.

• The result is limited to the first 10 rows using the LIMIT keyword.

**24. LEFT JOIN**

A LEFT JOIN keeps all of the original records in the left table and returns missing values for any columns from the right table where the joining field did not find a match.

SELECT \*

FROM films

LEFT JOIN soundtracks

ON films.date = soundtracks.date

LIMIT 10;

This is a SQL code that selects all columns from the "films" table and joins it with the "soundtracks" table using the "date" column as the join condition.

• The "LEFT JOIN" keyword ensures that all records from the "films" table are included in the result set, even if there is no matching record in the "soundtracks" table.

• The "LIMIT 10" clause limits the result set to the first 10 rows.

**25. RIGHT JOIN**

A RIGHT JOIN keeps all of the original records in the right table and returns missing values for any columns from the left table where the joining field did not find a match.

SELECT \*

FROM films

LEFT JOIN soundtracks

ON films.date = soundtracks.date

LIMIT 10;

This is a SQL code that selects all columns from the "films" table and joins it with the "soundtracks" table using the "date" column as the join condition.

• The "LEFT JOIN" keyword ensures that all records from the "films" table are included in the result set, even if there is no matching record in the "soundtracks" table.

• The "LIMIT 10" clause limits the result set to the first 10 rows.