SQL Basics Cheat Sheet

LearnSQL.com

SQL, or *Structured Query Language*, is a language to talk to databases. It allows you to select specific data and to build complex reports. Today, SQL is a universal language of data. It is used in practically all technologies that process data.

SAMPLE DATA

COUNTRY

id	name	population	area
1	France	66600000	640680
2	Germany	80700000	357000
...	...	...	...

CITY

id	name	country_id	population	rating
1	Paris	1	2243000	5
2	Berlin	2	3460000	3
...	...	...	...	...

QUERYING SINGLE TABLE

Fetch all columns from the country table:

```
SELECT *
FROM country;
```

Fetch id and name columns from the city table:

```
SELECT id, name
FROM city;
```

Fetch city names sorted by the rating column in the default ASCending order:

```
SELECT name
FROM city
ORDER BY rating [ASC];
```

Fetch city names sorted by the rating column in the DESCending order:

```
SELECT name
FROM city
ORDER BY rating DESC;
```

ALIASES

COLUMNS

```
SELECT name AS city_name
FROM city;
```

TABLES

```
SELECT co.name, ci.name
FROM city AS ci
JOIN country AS co
  ON ci.country_id = co.id;
```

FILTERING THE OUTPUT

COMPARISON OPERATORS

Fetch names of cities that have a rating above 3:

```
SELECT name
FROM city
WHERE rating > 3;
```

Fetch names of cities that are neither Berlin nor Madrid:

```
SELECT name
FROM city
WHERE name != 'Berlin'
  AND name != 'Madrid';
```

TEXT OPERATORS

Fetch names of cities that start with a 'P' or end with an 's':

```
SELECT name
FROM city
WHERE name LIKE 'P%'
  OR name LIKE '%s';
```

Fetch names of cities that start with any letter followed by 'ublin' (like Dublin in Ireland or Lublin in Poland):

```
SELECT name
FROM city
WHERE name LIKE '_ublin';
```

OTHER OPERATORS

Fetch names of cities that have a population between 500K and 5M:

```
SELECT name
FROM city
WHERE population BETWEEN 500000 AND 5000000;
```

Fetch names of cities that don't miss a rating value:

```
SELECT name
FROM city
WHERE rating IS NOT NULL;
```

Fetch names of cities that are in countries with IDs 1, 4, 7, or 8:

```
SELECT name
FROM city
WHERE country_id IN (1, 4, 7, 8);
```

QUERYING MULTIPLE TABLES

INNER JOIN

JOIN (or explicitly **INNER JOIN**) returns rows that have matching values in both tables.

```
SELECT city.name, country.name
FROM city
[INNER] JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	3	Iceland

LEFT JOIN

LEFT JOIN returns all rows from the left table with corresponding rows from the right table. If there's no matching row, **NULL**s are returned as values from the second table.

```
SELECT city.name, country.name
FROM city
LEFT JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	NULL	NULL

RIGHT JOIN

RIGHT JOIN returns all rows from the right table with corresponding rows from the left table. If there's no matching row, **NULL**s are returned as values from the left table.

```
SELECT city.name, country.name
FROM city
RIGHT JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
NULL	NULL	NULL	3	Iceland

FULL JOIN

FULL JOIN (or explicitly **FULL OUTER JOIN**) returns all rows from both tables – if there's no matching row in the second table, **NULL**s are returned.

```
SELECT city.name, country.name
FROM city
FULL [OUTER] JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	NULL	NULL
NULL	NULL	NULL	3	Iceland

CROSS JOIN

CROSS JOIN returns all possible combinations of rows from both tables. There are two syntaxes available.

```
SELECT city.name, country.name
FROM city
CROSS JOIN country;

SELECT city.name, country.name
FROM city, country;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
1	Paris	1	2	Germany
2	Berlin	2	1	France
2	Berlin	2	2	Germany

NATURAL JOIN

NATURAL JOIN will join tables by all columns with the same name.

```
SELECT city.name, country.name
FROM city
NATURAL JOIN country;
```

CITY			COUNTRY	
country_id	id	name	name	id
6	6	San Marino	San Marino	6
7	7	Vatican City	Vatican City	7
5	9	Greece	Greece	9
10	11	Monaco	Monaco	10

NATURAL JOIN used these columns to match rows: **city.id**, **city.name**, **country.id**, **country.name**.
NATURAL JOIN is very rarely used in practice.

Try out the interactive **SQL Basics** course at **LearnSQL.com**, and check out our other SQL courses.

LearnSQL.com is owned by Vertabelo SA
vertabelo.com | CC BY-NC-ND Vertabelo SA

SQL Basics Cheat Sheet

LearnSQL.com

AGGREGATION AND GROUPING

GROUP BY **groups** together rows that have the same values in specified columns. It computes summaries (aggregates) for each unique combination of values.

CITY		
id	name	country_id
1	Paris	1
101	Marseille	1
102	Lyon	1
2	Berlin	2
103	Hamburg	2
104	Munich	2
3	Warsaw	4
105	Cracow	4

→

CITY	
country_id	count
1	3
2	3
4	2

AGGREGATE FUNCTIONS

- `avg(expr)` – average value for rows within the group
- `count(expr)` – count of values for rows within the group
- `max(expr)` – maximum value within the group
- `min(expr)` – minimum value within the group
- `sum(expr)` – sum of values within the group

EXAMPLE QUERIES

Find out the number of cities:

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

Find out the number of cities with non-null ratings:

```
SELECT COUNT(rating)
FROM city;
```

Find out the number of distinctive country values:

```
SELECT COUNT(DISTINCT country_id)
FROM city;
```

Find out the smallest and the greatest country populations:

```
SELECT MIN(population), MAX(population)
FROM country;
```

Find out the total population of cities in respective countries:

```
SELECT country_id, SUM(population)
FROM city
GROUP BY country_id;
```

Find out the average rating for cities in respective countries if the average is above 3.0:

```
SELECT country_id, AVG(rating)
FROM city
GROUP BY country_id
HAVING AVG(rating) > 3.0;
```

SUBQUERIES

A subquery is a query that is nested inside another query, or inside another subquery. There are different types of subqueries.

SINGLE VALUE

The simplest subquery returns exactly one column and exactly one row. It can be used with comparison operators =, <, <=, >, or >=.
This query finds cities with the same rating as Paris:

```
SELECT name
FROM city
WHERE rating = (
  SELECT rating
  FROM city
  WHERE name = 'Paris'
);
```

MULTIPLE VALUES

A subquery can also return multiple columns or multiple rows. Such subqueries can be used with operators `IN`, `EXISTS`, `ALL`, or `ANY`.
This query finds cities in countries that have a population above 20M:

```
SELECT name
FROM city
WHERE country_id IN (
  SELECT country_id
  FROM country
  WHERE population > 20000000
);
```

CORRELATED

A correlated subquery refers to the tables introduced in the outer query. A correlated subquery depends on the outer query. It cannot be run independently from the outer query.
This query finds cities with a population greater than the average population in the country:

```
SELECT *
FROM city main_city
WHERE population > (
  SELECT AVG(population)
  FROM city average_city
  WHERE average_city.country_id = main_city.country_id
);
```

This query finds countries that have at least one city:

```
SELECT name
FROM country
WHERE EXISTS (
  SELECT *
  FROM city
  WHERE country_id = country.id
);
```

SET OPERATIONS

Set operations are used to combine the results of two or more queries into a single result. The combined queries must return the same number of columns and compatible data types. The names of the corresponding columns can be different.

CYCLING		
id	name	country
1	YK	DE
2	ZG	DE
3	WT	PL
...	...	...

SKATING		
id	name	country
1	YK	DE
2	DF	DE
3	AK	PL
...	...	...

UNION

`UNION` combines the results of two result sets and removes duplicates. **`UNION ALL`** doesn't remove duplicate rows.

This query displays German cyclists together with German skaters:

```
SELECT name
FROM cycling
WHERE country = 'DE'
UNION / UNION ALL
SELECT name
FROM skating
WHERE country = 'DE';
```

INTERSECT

`INTERSECT` returns only rows that appear in both result sets.

This query displays German cyclists who are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
INTERSECT
SELECT name
FROM skating
WHERE country = 'DE';
```

EXCEPT

`EXCEPT` returns only the rows that appear in the first result set but do not appear in the second result set.

This query displays German cyclists unless they are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
EXCEPT / MINUS
SELECT name
FROM skating
WHERE country = 'DE';
```

Try out the interactive **SQL Basics** course at **LearnSQL.com**, and check out our other SQL courses.

LearnSQL.com is owned by Vertabelo SA
vertabelo.com | CC BY-NC-ND Vertabelo SA

datacamp

MySQL Cheat Sheet

Learn SQL online at www.DataCamp.com

What is MySQL?

MySQL is an open-source relational database management system (RDBMS) known for its fast performance and reliability. Developed by Oracle Corporation, it's widely used for web applications and online publishing.

> Sample Data

The dataset contains details of the world's highest valued media franchises by gross revenue. Each row contains one franchise, and the table is named franchises.

Franchise	inception_year	total_revenue_busd	original_medium	owner	n_movies
Star Wars	1977	46.7	movie	The Walt Disney Company	12
Mickey Mouse and Friends	1928	52.2	cartoon	The Walt Disney Company	
Anpanman	1973	38.4	book	Froebel-kan	33
Winnie the Pooh	1924	48.5	book	The Walt Disney Company	6
Pokémon	1996	88	video game	The Pokémon Company	24
Disney Princess	2000	45.4	movie	The Walt Disney Company	

> Querying tables

Get all the columns from a table using `SELECT *`

```
SELECT *
  FROM franchises
```

Get a column from a table by name using `SELECT col`

```
SELECT franchise
  FROM franchises
```

Get multiple columns from a table by name using `SELECT col1, col2`

```
SELECT franchise, inception_year
  FROM franchises
```

Override column names with `SELECT col AS new_name`

```
SELECT franchise, inception_year AS creation_year
  FROM franchises
```

Arrange the rows in ascending order of values in a column with `ORDER BY col`

```
SELECT franchise, inception_year
  FROM franchises
  ORDER BY inception_year
```

Arrange the rows in descending order of values in a column with `ORDER BY col DESC`

```
SELECT franchise, total_revenue_busd
  FROM franchises
  ORDER BY total_revenue_busd DESC
```

Limit the number of rows returned with `LIMIT n`

```
SELECT *
  FROM franchises
  LIMIT 2
```

Get unique values with `SELECT DISTINCT`

```
SELECT DISTINCT owner
  FROM franchises
```

> Filtering Data

Filtering on numeric columns

Get rows where a number is greater than a value with `WHERE col > n`

```
SELECT franchise, inception_year
  FROM franchises
  WHERE inception_year > 1928
```

Get rows where a number is greater than or equal to a value with `WHERE col >= n`

```
SELECT franchise, inception_year
  FROM franchises
  WHERE inception_year >= 1928
```

Get rows where a number is less than a value with `WHERE col < n`

```
SELECT franchise, inception_year
  FROM franchises
  WHERE inception_year <= 1977
```

Get rows where a number is equal to a value with `WHERE col = n`

```
SELECT franchise, inception_year
  FROM franchises
  WHERE inception_year = 1996
```

Get rows where a number is not equal to a value with `WHERE col <> n` **or** `WHERE col != n`

```
SELECT franchise, inception_year
  FROM franchises
  WHERE inception_year <> 1996
```

Get rows where a number is between two values (inclusive) with `WHERE col BETWEEN m AND n`

```
SELECT franchise, inception_year
  FROM franchises
  WHERE inception_year BETWEEN 1928 AND 1977
```

Filtering on text columns

Get rows where text is equal to a value with `WHERE col = 'x'`

```
SELECT franchise, original_medium
  FROM franchises
  WHERE original_medium = 'book'
```

Get rows where text is one of several values with `WHERE col IN ('x', 'y')`

```
SELECT franchise, original_medium
  FROM franchises
  WHERE original_medium IN ('movie', 'video game')
```

Get rows where text contains specific letters with `WHERE col LIKE '%abc%'`
(% represents any characters)

```
SELECT franchise, original_medium
  FROM franchises
  WHERE original_medium LIKE '%oo%'
```

Filtering on multiple columns

Get the rows where one condition and another condition holds with `WHERE condn1 AND condn2`

```
SELECT franchise, inception_year, total_revenue_busd
  FROM franchises
  WHERE inception_year < 1950 AND total_revenue_busd > 50
```

Get the rows where one condition or another condition holds with `WHERE condn1 OR condn2`

```
SELECT franchise, inception_year, total_revenue_busd
  FROM franchises
  WHERE inception_year < 1950 OR total_revenue_busd > 50
```

Filtering on missing data

Get rows where values are missing with `WHERE col IS NULL`

```
SELECT franchise, n_movies
  FROM franchises
  WHERE n_movies IS NULL
```

Get rows where values are not missing with `WHERE col IS NOT NULL`

```
SELECT franchise, n_movies
  FROM franchises
  WHERE n_movies IS NOT NULL
```

> Aggregating Data

Simple aggregations

Get the total number of rows `SELECT COUNT(*)`

```
SELECT COUNT(*)
  FROM franchises
```

Get the total value of a column with `SELECT SUM(col)`

```
SELECT SUM(total_revenue_busd)
  FROM franchises
```

Get the mean value of a column with `SELECT AVG(col)`

```
SELECT AVG(total_revenue_busd)
  FROM franchises
```

Get the minimum value of a column with `SELECT MIN(col)`

```
SELECT MIN(total_revenue_busd)
  FROM franchises
```

Get the maximum value of a column with `SELECT MAX(col)`

```
SELECT MAX(total_revenue_busd)
  FROM franchises
```

Grouping, filtering, and sorting

Get summaries grouped by values with `GROUP BY col`

```
SELECT owner, COUNT(*)
  FROM franchises
  GROUP BY owner
```

Get summaries grouped by values, in order of summaries with `GROUP BY col ORDER BY smmry DESC`

```
SELECT original_medium, SUM(n_movies) AS total_movies
  FROM franchises
  GROUP BY original_medium
  ORDER BY total_movies DESC
```

Get rows where values in a group meet a criterion with `GROUP BY col HAVING condn`

```
SELECT original_medium, SUM(n_movies) AS total_movies
  FROM franchises
  GROUP BY original_medium
  ORDER BY total_movies DESC
  HAVING total_movies > 10
```

Filter before and after grouping with `WHERE condn_before GROUP BY col HAVING condn_after`

```
SELECT original_medium, SUM(n_movies) AS total_movies
  FROM franchises
  WHERE owner = 'The Walt Disney Company'
  GROUP BY original_medium
  ORDER BY total_movies DESC
  HAVING total_movies > 10
```

> MySQL-Specific Syntax

Not all code works in every dialect of SQL. The following examples work in MySQL, but are not guaranteed to work in other dialects.

Limit the number of rows returned, offset from the top with `LIMIT m, n`

```
SELECT *
  FROM franchises
  LIMIT 2, 3
```

By default, MySQL uses case insensitive matching in `WHERE clauses.`

```
SELECT *
  FROM franchises
  WHERE owner = 'THE WALT DISNEY COMPANY'
```

To get case sensitive matching, use `WHERE BINARY condn`

```
SELECT *
  FROM franchises
  WHERE BINARY owner = 'THE WALT DISNEY COMPANY'
```

Get the current date with `CURDATE()` **and the current** `datetime` **with** `NOW()` **or** `CURTIME()`

```
SELECT CURDATE(), NOW(), CURTIME()
```

List available tables with `show tables`

```
show tables
```

datacamp

PostgreSQL Basics

Learn SQL online at www.DataCamp.com

What is PostgreSQL?

PostgreSQL is an open-source relational database management system (RDBMS) known for its extensibility and standards compliance. Developed and maintained by a group of volunteers known as The PostgreSQL Global Development Group, it is popular across a wide range of organizations from enterprises to government departments. It has powerful data analysis capabilities.

> Sample Data

The dataset contains details of the world's fastest production cars by 0 to 60 mph acceleration time. Each row contains one car model, and the table is named cars.

make	model	year	propulsion_type	time_to_60_mph_s	limited_production_count
Lamborghini	Huracán Performante	2018	ICE	2.2	
Ferrari	SF90 Stradale	2021	Hybrid	2	
Tesla	Model S Plaid	2021	Electric	1.98	
Porsche	918 Spyder	2015	Hybrid	2.1	918
Rimac	Nevera	2021	Electric	1.74	150
Porsche	911 Turbo S (992)	2020	ICE	2.1	

> Querying tables

Get all the columns from a table using `SELECT *`

```
SELECT *
 FROM cars
```

Get a column from a table by name using `SELECT col`

```
SELECT model
 FROM cars
```

Get multiple columns from a table by name using `SELECT col1, col2`

```
SELECT make, model
 FROM cars
```

Override column names with `SELECT col AS new_name`

```
SELECT make, model, propulsion_type AS engine_type
 FROM cars
```

Override column names with `SELECT col AS new_name`

```
SELECT make, model, propulsion_type
 AS engine_type
 FROM cars
```

Arrange the rows in ascending order of values in a column with `ORDER BY col`

```
SELECT  make, model, time_to_60_mph_s
 FROM cars
 ORDER BY time_to_60_mph_s
```

Arrange the rows in descending order of values in a column with ORDER BY col DESC

```
SELECT make, model, model_year
 FROM cars
 ORDER BY model_year DESC
```

Limit the number of rows returned with `LIMIT n`

```
SELECT *
 FROM cars
LIMIT 2
```

Get unique values with `SELECT DISTINCT`

```
SELECT DISTINCT propulsion_type
 FROM cars
```

> Filtering Data

Filtering on numeric columns

Get rows where a number is greater than a value with `WHERE col > n`

```
SELECT make, model, time_to_60_mph_s
  FROM cars
  WHERE time_to_60_mph_s > 2.1
```

Get rows where a number is greater than or equal to a value with `WHERE col >= n`

```
SELECT make, model, time_to_60_mph_s
  FROM cars
  WHERE time_to_60_mph_s >= 2.1
```

Get rows where a number is less than a value with `WHERE col < n`

```
SELECT make, model, time_to_60_mph_s
  FROM cars
  WHERE time_to_60_mph_s < 2.1
```

Get rows where a number is less than or equal to a value with `WHERE col <= n`

```
SELECT make, model, time_to_60_mph_s
  FROM cars
  WHERE time_to_60_mph_s <= 2.1
```

Get rows where a number is equal to a value with `WHERE col = n`

```
SELECT make, model, time_to_60_mph_s
 FROM cars
 WHERE time_to_60_mph_s = 2.1
```

Get rows where a number is not equal to a value with `WHERE col <> n` **or** `WHERE col != n`

```
SELECT make, model, time_to_60_mph_s
  FROM cars
  WHERE time_to_60_mph_s <> 2.1
```

Get rows where a number is between two values (inclusive) with `WHERE col BETWEEN m AND n`

```
SELECT make, model, time_to_60_mph_s
 FROM cars
 WHERE time_to_60_mph_s BETWEEN 1.9 AND 2.1
```

Filtering on text columns

Get rows where text is equal to a value with `WHERE col = 'x'`

```
SELECT make, model, propulsion_type
 FROM cars
 WHERE propulsion_type = 'Hybrid'
```

Get rows where text is one of several values with `WHERE col IN ('x', 'y')`

```
SELECT make, model, propulsion_type
 FROM cars
 WHERE propulsion_type IN ('Electric',  'Hybrid')
```

Get rows where text contains specific letters with `WHERE col LIKE '%abc%'` (% represents any characters)

```
SELECT make, model, propulsion_type
 FROM cars
 WHERE propulsion_type LIKE '%ic%'
```

For case insensitive matching, use `WHERE col ILIKE '%abc%'`

```
SELECT make, model, propulsion_type
 FROM cars
 WHERE propulsion_type ILIKE '%ic%'
```

Filtering on multiple columns

Get the rows where one condition and another condition holds with `WHERE condn1 AND condn2`

```
SELECT make, model, propulsion_type, model_year
 FROM cars
 WHERE propulsion_type = 'Hybrid' AND model_year < 2020
```

Get the rows where one condition or another condition holds with `WHERE condn1 OR condn2`

```
SELECT make, model, propulsion_type, model_year
 FROM cars
 WHERE propulsion_type = 'Hybrid' OR model_year < 2020
```

Filtering on missing data

Get rows where values are missing with `WHERE col IS NULL`

```
SELECT make, model, limited_production_count
 FROM cars
 WHERE limited_production_count IS NULL
```

Get rows where values are not missing with `WHERE col IS NOT NULL`

```
SELECT make, model, limited_production_count
 FROM cars
 WHERE limited_production_count IS NOT NULL
```

> Aggregating Data

Simple aggregations

Get the total number of rows `SELECT COUNT(*)`

```
SELECT COUNT(*)
  FROM cars
```

Get the total value of a column with `SELECT SUM(col)`

```
SELECT SUM(limited_production_count)
  FROM cars
```

Get the mean value of a column with `SELECT AVG(col)`

```
SELECT AVG(time_to_60_mph_s)
  FROM cars
```

Get the minimum value of a column with `SELECT MIN(col)`

```
SELECT MIN(time_to_60_mph_s)
  FROM cars
```

Get the maximum value of a column with `SELECT MAX(col)`

```
SELECT MAX(time_to_60_mph_s)
 FROM cars
```

Grouping, filtering, and sorting

Get summaries grouped by values with `GROUP BY col`

```
SELECT propulsion_type, COUNT(*)
 FROM cars
 GROUP BY propulsion_type
```

Get summaries grouped by values, in order of summaries with `GROUP BY col ORDER BY smmry DESC`

```
SELECT propulsion_type, AVG(time_to_60_mph_s) AS mean_time_to_60_mph_s
 FROM cars
 GROUP BY propulsion_type
 ORDER BY mean_time_to_60_mph_s
```

Get rows where values in a group meet a criterion with `GROUP BY col HAVING condn`

```
SELECT propulsion_type, AVG(time_to_60_mph_s) AS mean_time_to_60_mph_s
 FROM cars
 GROUP BY propulsion_type
 HAVING mean_time_to_60_mph_s > 2
```

Filter before and after grouping with `WHERE condn_before GROUP BY col HAVING condn_after`

```
SELECT propulsion_type, AVG(time_to_60_mph_s) AS mean_time_to_60_mph_s
 FROM cars
 WHERE limited_production_count IS NOT NULL
 GROUP BY propulsion_type
 HAVING mean_time_to_60_mph_s > 2
```

> PostgreSQL-Specific Syntax

Not all code works in every dialect of SQL. The following examples work in PostgreSQL, but are not guaranteed to work in other dialects.

Limit the number of rows returned, offset from the top with `LIMIT m OFFSET n`

```
SELECT *
 FROM cars
 LIMIT 2 OFFSET 3
```

PostgreSQL allows text concatenation with the `||` **operator**

```
SELECT make || ' ' || model AS make_and_model
  FROM cars
```

Get the current date with `CURRENT_DATE` **and the current datetime with** `NOW()` **or** `CURRENT_TIME`

```
SELECT NOW(), CURRENT_DATE, CURRENT_TIME
```

List available tables by selecting from `pg_catalog.pg_tables`

```
SELECT *
 FROM pg_catalog.pg_tables
```