SQL Basics Cheat Sheet

SQL

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	6 40680
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

COMPARISONOPERATORS

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

INNERJOIN

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

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	N U L L	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

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

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

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.

SQL Basics Cheat Sheet

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	co u nt
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(DISTINCTcountry_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 quer y.

This query finds cities with a population greater than the average population in the countr y:

```
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	co u ntr y
1	YK	DE
2	ZG	DE
3	WT	PL
...	...	...

SKATING

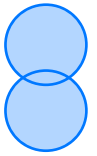
id	name	co u ntr y
1	YK	DE
2	DF	DE
3	AK	PL
...	...	...

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';
```

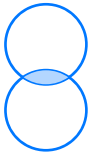


INTERSECT

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';
```

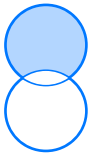


EXCEPT

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:

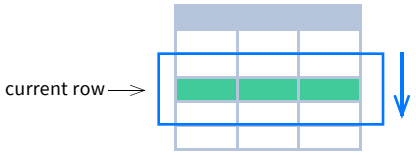
```
SELECT name
FROM cycling
WHERE country = 'DE'
EXCEPT / MINUS
SELECT name
FROM skating
WHERE country = 'DE';
```



SQL Window Functions Cheat Sheet

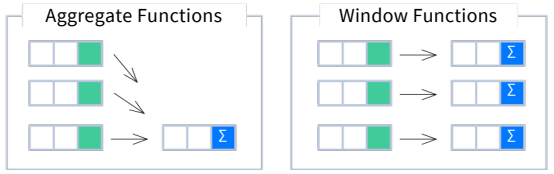
WINDOW FUNCTIONS

compute their result based on a sliding window frame, a set of rows that are somehow related to the current row.



AGGREGATE FUNCTIONS VS. WINDOW FUNCTIONS

unlike aggregate functions, window functions do not collapse rows.



SYNTAX

```
SELECT city, month,
    sum(sold) OVER (
        PARTITION BY city
        ORDER BY month
        RANGE UNBOUNDED PRECEDING) total
FROM sales;
```

```
SELECT <column_1>, <column_2>,
    <window_function>() OVER (
        PARTITION BY <...>
        ORDER BY <...>
        <window_frame>) <window_column_alias>
FROM <table_name>;
```

Named Window Definition

```
SELECT country, city,
    rank() OVER country_sold_avg
FROM sales
WHERE month BETWEEN 1 AND 6
GROUP BY country, city
HAVING sum(sold) > 10000
WINDOW country_sold_avg AS (
    PARTITION BY country
    ORDER BY avg(sold) DESC)
ORDER BY country, city;
```

```
SELECT <column_1>, <column_2>,
    <window_function>() OVER <window_name>
FROM <table_name> WHERE
<...> GROUP BY <...> HAVING
<...> WINDOW <window_name>
AS (

    PARTITION BY <...>
    ORDER BY <...>
    <window_frame>)
ORDER BY <...>;
```

PARTITION BY, ORDER BY, and window frame definition are all optional.

LOGICAL ORDER OF OPERATIONS IN SQL

1. FROM, JOIN
2. W HERE
3. GROUP BY
4. aggregate functions
5. HAVING
6. **window functions**
7. SELECT
8. DISTINCT
9. UNION/INTERSECT/EXCEPT
10. ORDER BY
11. OFFSET
12. LIMIT/FETCH/TOP

You can use window functions in SELECT and ORDER BY. However, you can't put window functions anywhere in the FROM, WHERE, GROUP BY, or HAVING clauses.

PARTITION BY

divides rows into multiple groups, called partitions, to which the window function is applied.

month	city	sold
1	Rome	200
2	Paris	500
1	London	100
1	Paris	300
2	Rome	300
2	London	400
3	Rome	400

Default Partition: with no PARTITION BY clause, the entire result set is the partition.

PARTITION BY city

month	city	sold	sum
1	Paris	300	800
2	Paris	500	800
1	Rome	200	900
2	Rome	300	900
3	Rome	400	900
1	London	100	500
2	London	400	500

ORDER BY

specifies the order of rows in each partition to which the window function is applied.

sold	city	month
200	Rome	1
500	Paris	2
100	London	1
300	Paris	1
300	Rome	2
400	London	2
400	Rome	3

Default ORDER BY: with no ORDER BY clause, the order of rows within each partition is arbitrary.

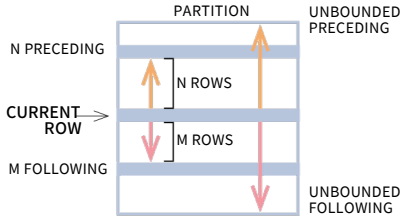
PARTITION BY city ORDER BY month

sold	city	month
300	Paris	1
500	Paris	2
200	Rome	1
300	Rome	2
400	Rome	3
100	London	1
400	London	2

WINDOW FRAME

is a set of rows that are somehow related to the current row. The window frame is evaluated separately within each partition.

ROWS | RANGE | GROUPS BETWEEN lower_bound AND upper_bound



The bounds can be any of the five options:

- UNBOUNDED PRECEDING
- n PRECEDING
- CURRENT ROW
- n FOLLOWING
- UNBOUNDED FOLLOWING

The lower_bound must be BEFORE the upper_bound

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

	city	sold	month
	Paris	300	1
	Rome	200	1
	Paris	500	2
	Rome	100	4
current row	Paris	200	4
	Paris	300	5
	Rome	200	5
	London	200	5
	London	100	6
	Rome	300	6

1 row before the current row and 1 row after the current row

RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING

	city	sold	month
	Paris	300	1
	Rome	200	1
	Paris	500	2
	Rome	100	4
current row	Paris	200	4
	Paris	300	5
	Rome	200	5
	London	200	5
	London	100	6
	Rome	300	6

values in the range between 3 and 5

ORDER BY must contain a single expression

GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING

	city	sold	month
	Paris	300	1
	Rome	200	1
	Paris	500	2
	Rome	100	4
current row	Paris	200	4
	Paris	300	5
	Rome	200	5
	London	200	5
	London	100	6
	Rome	300	6

1 group before the current row and 1 group after the current row regardless of the value

As of 2020, GROUPS is only supported in PostgreSQL 11 and up.

ABBREVIATIONS

Abbreviation	Meaning
UNBOUNDED PRECEDING	BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
n PRECEDING	BETWEEN n PRECEDING AND CURRENT ROW
CURRENT ROW	BETWEEN CURRENT ROW AND CURRENT ROW
n FOLLOWING	BETWEEN AND CURRENT ROW AND n FOLLOWING
UNBOUNDED FOLLOWING	BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

DEFAULT WINDOW FRAME

If ORDER BY is specified, then the frame is **RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.**

Without ORDER BY, the frame specification is **ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.**

SQL Window Functions Cheat Sheet

LIST OF WINDOW FUNCTIONS

Aggregate Functions

- avg()
- count()
- max()
- min()
- sum()

Ranking Functions

- row_number()
- rank()
- dense_rank()

Distribution Functions

- percent_rank()
- cume_dist()

Analytic Functions

- lead()
- lag()
- ntile()
- first_value()
- last_value()
- nth_value()

AGGREGATE FUNCTIONS

- **avg** *(expr)* – average value for rowswithin the window frame
- **count** *(expr)* – count of values forrowswithin the window frame
- **max** *(expr)* – maximum value within the window frame
- **min** *(expr)* – minimum value within the window frame
- **sum** *(expr)* – sum of values within thewindow frame

ORDER BY and Window Frame:
Aggregate functions do not require an ORDER BY. They accept window frame definition **(ROWS, RANGE, GROUPS).**

RANKING FUNCTIONS

- **row_number()** – unique number for each row within partition, with different numbers fortied values
- **rank()** –ranking within partition, with gaps and same ranking for tied values
- **dense_rank()** – ranking within partition, with no gaps and same ranking for tied values

city	price	row_number	rank	dense_rank
		over(order by price)		
Paris	7	1	1	1
Rome	7	2	1	1
London	8.5	3	3	2
Berlin	8.5	4	3	2
Moscow	9	5	5	3
Madrid	10	6	6	4
Oslo	10	7	6	4

ORDER BY and Window Frame: rank() and **dense_rank()** require ORDER BY, but row_number() does not require ORDER BY .Rankingfunctionsdo not accept window frame definition (ROWS, RANGE, GROUPS).

DISTRIBUTION FUNCTIONS

- **percent_rank()** – the percentile ranking number of a row—a value in [0, 1] interval: (rank - 1) / (total number of rows - 1)
- **cume_dist()** – the cumulative distribution of a value within a group of values, i.e., the number of rows with values less than or equal to the current row's value divided by the total number of rows; a value in (0, 1] interval

percent_rank() OVER(ORDER BY sold)

city	sold	percent_rank
Paris	100	0
Berlin	150	0.25
Rome	200	0.5
Moscow	200	0.5
London	300	1

← without this row 50% of values are less than this row's value

cume_dist() OVER(ORDER BY sold)

city	sold	cume_dist
Paris	100	0.2
Berlin	150	0.4
Rome	200	0.8
Moscow	200	0.8
London	300	1

← 80% of values are less than or equal to this one

ORDER BY and Window Frame: Distribution functions require **ORDER BY**. They do not accept window frame definition (**ROWS, RANGE, GROUPS**).

ANALYTIC FUNCTIONS

- **lead** *(expr, offset,default)* – the value for the row *offset* rows after the current; *offset* and *default* are optional; default values: *offset* = 1, *default* = NULL
- **lag** *(expr, offset, default)* – the value for the row *offset* rows before the current; *offset* and *default* are optional; default values: *offset* = 1, *default* = NULL

lead(sold) OVER(ORDER BY month)

month	sold	
1	500	300
2	300	400
3	400	100
4	100	500
5	500	NULL

order by month

lag(sold) OVER(ORDER BY month)

month	sold	
1	500	NULL
2	300	500
3	400	300
4	100	400
5	500	100

order by month

lead(sold, 2, 0) OVER(ORDER BY month)

month	sold	
1	500	400
2	300	100
3	400	500
4	100	0
5	500	0

order by month; offset=2

lag(sold, 2, 0) OVER(ORDER BY month)

month	sold	
1	500	0
2	300	0
3	400	500
4	100	300
5	500	400

order by month; offset=2

- **first_value** *(expr)* – the value for the first row within the window frame
- **last_value** *(expr)* – the value for the last row within the window frame

first_value(sold) OVER
(PARTITION BY city ORDER BY month)

city	month	sold	first_value
Paris	1	500	500
Paris	2	300	500
Paris	3	400	500
Rome	2	200	200
Rome	3	300	200
Rome	4	500	200

last_value(sold) OVER
(PARTITION BY city ORDER BY month
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)

city	month	sold	last_value
Paris	1	500	400
Paris	2	300	400
Paris	3	400	400
Rome	2	200	500
Rome	3	300	500
Rome	4	500	500

Note: You usually want to use RANGE BETWEEN **UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING with last_value()** .Withthedefault window frameforORDER BY, RANGE **UNBOUNDED PRECEDING, last_value()** returnsthevaluefor thecurrentrow.

- **ntile** *(n)* – divide rows within a partition as equally as possible into *n* groups, and assign each rowitsgroup number.

ntile(3)

city	sold	
Rome	100	1
Paris	100	1
London	200	1
Moscow	200	2
Berlin	200	2
Madrid	300	2
Oslo	300	3
Dublin	300	3

ORDER BY andWindowFrame: ntile(), **lead(), and lag()** requirean ORDER BY. Theydonotacceptwindowframe definition **(ROWS, RANGE, GROUPS).**

- **nth_value** *(expr, n)* – the value for the *n*-th row within the window frame; *n* must be an integer

nth_value(sold, 2) OVER (PARTITIONBYcity
ORDER BY month RANGE BETWEEN UNBOUNDED
PRECEDING AND UNBOUNDED FOLLOWING)

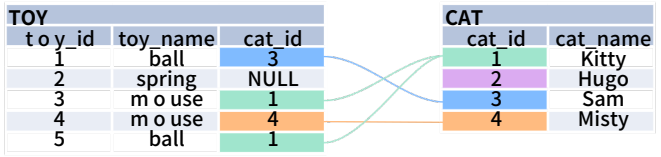
city	month	sold	nth_value
Paris	1	500	300
Paris	2	300	300
Paris	3	400	300
Rome	2	200	300
Rome	3	300	300
Rome	4	500	300
Rome	5	300	300
London	1	100	NULL

ORDER BY and Window Frame: first_value(), **last_value(), and nth_value()** do not require an ORDER BY. They accept window frame definition **(ROWS, RANGE, GROUPS).**

SQL JOINs Cheat Sheet

JOINING TABLES

JOIN combines data from two tables.



TOY

t o y_id	toy_name	cat_id
1	ball	3
2	spring	NULL
3	m o use	1
4	m o use	4
5	ball	1

CAT

cat_id	cat_name
1	Kitty
2	Hugo
3	Sam
4	Misty

JOIN typically combines rows with equal values for the specified columns. **Usually**, one table contains a **primary key**, which is a column or columns that uniquely identify rows in the table (the cat_id column in the cat table).
The other table has a column or columns that **refer to the primary key columns** in the first table (the cat_id column in the toy table). Such columns are **foreign keys**. The JOIN condition is the equality between the primary key columns in one table and columns referring to them in the other table.

JOIN

JOIN returns all rows that match the ON condition. JOIN is also called INNER JOIN.

```
SELECT *
FROM toy
JOIN cat
 ON toy.cat_id = cat.cat_id;
```

toy_id	toy_name	cat_id	cat_id	cat_name
5	ball	1	1	Kitt y
3	mouse	1	1	Kitty
1	ball	3	3	Sam
4	mouse	4	4	Misty

There is also another, older syntax, but it **isn't recommended**.
List joined tables in the FROM clause, and place the conditions in the WHERE clause.

```
SELECT *
FROM toy, cat
WHERE toy.cat_id = cat.cat_id;
```

JOIN CONDITIONS

The JOIN condition doesn't have to be an equality – it can be any condition you want. JOIN doesn't interpret the JOIN condition, it only checks if the rows satisfy the given condition.
To refer to a column in the JOIN query, you have to use the full column name: first the table name, then a dot (.) and the column name:

```
 ON cat.cat_id = toy.cat_id
```

You can omit the table name and use just the column name if the name of the column is unique within all columns in the joined tables.

NATURAL JOIN

If the tables have columns with **the same name**, you can use NATURAL JOIN instead of JOIN.

```
SELECT *
FROM toy
NATURAL JOIN cat;
```

cat_id	toy_id	toy_name	cat_name
1	5	ball	Kitty
1	3	mouse	Kitty
3	1	ball	Sam
4	4	mouse	Misty

The common column appears only once in the result table.
Note: NATURAL JOIN is rarely used in real life.

LEFT JOIN

LEFT JOIN returns all rows from the **left table** with matching rows from the right table. Rows without a match are filled with NULL s. LEFTJOIN is also called LEFTOUTERJOIN .

```
SELECT *
FROM toy
LEFT JOIN cat
 ON toy.cat_id = cat.cat_id;
```

toy_id	toy_name	cat_id	cat_id	cat_name
5	ball	1	1	Kitty
3	mouse	1	1	Kitty
1	ball	3	3	Sam
4	mouse	4	4	Misty
2	spring	NULL	NULL	NULL

whole left table

RIGHT JOIN

RIGHT JOIN returns all rows from the **right table** with matching rows from the left table. Rows without a match are filled with NULL s. RIGHTJOIN is also called RIGHT OUTER JOIN .

```
SELECT *
FROM toy
RIGHT JOIN cat
 ON toy.cat_id = cat.cat_id;
```

toy_id	toy_name	cat_id	cat_id	cat_name
5	ball	1	1	Kitt y
3	m o use	1	1	Kitty
NULL	NULL	NULL	2	Hugo
1	ball	3	3	Sam
4	m o use	4	4	Misty

whole right table

FULL JOIN

FULL JOIN returns all rows from the **left table** and all rows from the **right table**. It fills the non-matching rows with NULLs. FULLJOIN is also called FULLOUTERJOIN .

```
SELECT *
FROM toy
FULL JOIN cat
 ON toy.cat_id = cat.cat_id;
```

toy_id	toy_name	cat_id	cat_id	cat_name
5	ball	1	1	Kitt y
3	mouse	1	1	Kitty
NULL	NULL	NULL	2	Hugo
1	ball	3	3	Sam
4	mouse	4	4	Misty
2	spring	NULL	NULL	NULL

whole left table | whole right table

CROSS JOIN

CROSS JOIN returns **all possible combinations** of rows from the left and right tables.

```
SELECT *
FROM toy
CROSS JOIN cat;
```

Other syntax:

```
SELECT *
FROM toy, cat;
```

toy_id	toy_name	cat_id	cat_id	cat_name
1	ball	3	1	Kitty
2	spring	NULL	1	Kitty
3	m o use	1	1	Kitty
4	mouse	4	1	Kitty
5	ball	1	1	Kitty
1	ball	3	2	Hugo
2	spring	NULL	2	Hugo
3	mouse	1	2	Hugo
4	mouse	4	2	Hugo
5	ball	1	2	Hugo
1	ball	3	3	Sam
...	...	...	...	...

SQL JOINs Cheat Sheet

COLUMN AND TABLE ALIASES

Aliases give a temporary name to a **table** or a **column** in a table.

CAT AS c

cat_id	cat_name	mom_id	owner_id
1	Kitty	5	1
2	Hugo	1	2
3	Sam	2	2
4	Misty	1	NULL

OWNER AS o

id	name
1	John Smith
2	Danielle Davis

A **column alias** renames a column in the result. A **table alias** renames a table within the query. If you define a table alias, you must use it instead of the table name everywhere in the query. The AS keyword is optional in defining aliases.

```
SELECT
o.name AS owner_name,
c.cat_name
FROM cat      AS c
JOIN owner AS o
ON c.owner_id = o.id;
```

cat_name	owner_name
Kitty	John Smith
Sam	Danielle Davis
Hugo	Danielle Davis

SELF JOIN

You can join a table to itself, for example, to show a parent-child relationship.

CAT AS child

cat_id	cat_name	owner_id	mom_id
1	Kitty	1	5
2	Hugo	2	1
3	Sam	2	2
4	Misty	NULL	1

CAT AS mom

cat_id	cat_name	owner_id	mom_id
1	Kitty	1	5
2	Hugo	2	1
3	Sam	2	2
4	Misty	NULL	1

Each occurrence of the table must be given a **different alias**. Each column reference must be preceded with an **appropriate table alias**.

```
SELECT
child.cat_name AS child_name,
mom.cat_name      AS mom_name
FROM cat AS child
JOIN cat AS mom
ON child.mom_id = mom.cat_id;
```

child_name	mom_name
Hugo	Kitty
Sam	Hugo
Misty	Kitty

NON-EQUI SELF JOIN

You can use a **non-equality** in the ON condition, for example, to show **all different pairs** of rows.

TOY AS a

toy_id	toy_name	cat_id
3	mouse	1
5	ball	1
1	ball	3
4	mouse	4
2	spring	NULL

TOY AS b

cat_id	toy_id	toy_name
1	3	mouse
1	5	ball
3	1	ball
4	4	mouse
NULL	2	spring

```
SELECT
 a.toy_name AS toy_a,
 b.toy_name AS toy_b
FROM toy a
JOIN toy b
 ON a.cat_id < b.cat_id;
```

cat_a_id	toy_a	cat_b_id	toy_b
1	mouse	3	ball
1	ball	3	ball
1	mouse	4	mouse
1	ball	4	mouse
3	ball	4	mouse

MULTIPLE JOINS

You can join more than two tables together. First, two tables are joined, then the third table is joined to the result of the previous joining.

TOY AS t

toy_id	toy_name	cat_id
1	ball	3
2	spring	NULL
3	mouse	1
4	mouse	4
5	ball	1

CAT AS c

cat_id	cat_name	mom_id	owner_id
1	Kitty	5	1
2	Hugo	1	2
3	Sam	2	2
4	Misty	1	NULL

OWNER AS o

id	name
1	John
2	Smith Danielle Davis

JOIN & JOIN

```
SELECT t.toy_name,
c.cat_name, o.name AS
owner_name FROM toy t
JOIN cat c ON t.cat_id =
c.cat_id JOIN owner o ON
c.owner_id = o.id;
```

toy_name	cat_name	owner_name
ball	Kitty	John Smith
mouse	Kitty	John Smith
ball	Sam	Danielle Davis

JOIN & LEFT JOIN

```
SELECT t.toy_name,
c.cat_name, o.name AS
owner_name FROM toy t
JOIN cat c ON t.cat_id =
c.cat_id LEFT JOIN owner o
ON c.owner_id = o.id;
```

toy_name	cat_name	owner_name
ball	Kitty	John Smith
mouse	Kitty	John Smith
ball	Sam	Danielle Davis
mouse	Misty	NULL

LEFT JOIN & LEFT JOIN

```
SELECT
t.toy_name,
c.cat_name,
o.name AS owner_name
FROM toy t
LEFT JOIN cat c
ON t.cat_id = c.cat_id
LEFT JOIN owner o
ON c.owner_id = o.id;
```

toy_name	cat_name	owner_name
ball	Kitty	John Smith
mouse	Kitty	John Smith
ball	Sam	Danielle Davis
mouse	Misty	NULL
spring	NULL	NULL

JOIN WITH MULTIPLE CONDITIONS

You can use multiple JOIN conditions using the **ON** keyword once and the **AND** keywords as many times as you need.

CAT AS c

cat_id	cat_name	mom_id	owner_id	age
1	Kitty	5	1	17
2	Hugo	1	2	10
3	Sam	2	2	5
4	Misty	1	NULL	11

OWNER AS o

id	name	age
1	John Smith	18
2	Danielle Davis	10

```
SELECT
cat_name,
o.name AS owner_name,
c.age AS cat_age,
o.age AS owner_age
FROM cat c
JOIN owner o
ON c.owner_id = o.id
AND c.age < o.age;
```

cat_name	owner_name	age	age
Kitty	John Smith	17	18
Sam	Danielle Davis	5	10

Standard SQL Functions Cheat Sheet

TEXT FUNCTIONS

CONCATENATION

Use the || operatortoconcatenate two strings:

```
SELECT 'Hi ' || 'there!';
-- result: Hi there!
```

Remember that you can concatenate only character strings using || . Use this trick for numbers:

```
SELECT '' || 4 || 2;
-- result: 42
```

Some databases implement non-standard solutions for concatenating strings like CONCAT() or CONCAT_WS() Check thedocumentationfor your specificdatabase.

LIKE OPERATOR – PATTERN MATCHING

Usethe character toreplaceany singlecharacter.Usethe character to replace any number of characters (including 0 characters).

Fetch all names that start with any letter followed by 'atherine' :

```
SELECT name
FROM names
WHEREname LIKE '_atherine'
```

Fetch all names that end with 'a':

```
SELECT name
FROM names
WHERE name LIKE '%a'
```

USEFUL FUNCTIONS

Get the count of characters in a string:

```
SELECT LENGTH('LearnSQL.com'
-- result: 12
```

Convert all letters to lowercase:

```
SELECT LOWER('LEARNSQL.COM'
-- result: learnsql.com
```

Convert all letters to uppercase:

```
SELECT UPPER('LearnSQL.com'
-- result: LEARNSQL.COM
```

Convert all letters to lowercase and all first letters to uppercase (not implemented in MySQL and SQL Server):

```
SELECT INITCAP('edgar frank ted cODD'
-- result: Edgar Frank Ted Codd
```

Get just a part of a string:

```
SELECT SUBSTRING('LearnSQL.com', 9
-- result: .com
SELECT SUBSTRING('LearnSQL.com', 0, 6
-- result: Learn
```

Replace part of a string:

```
SELECT REPLACE('LearnSQL.com', 'SQL',
'Python'
-- result: LearnPython.com
```

NUMERIC FUNCTIONS

BASIC OPERATIONS

Use,, , / todosomebasicmath. To get the number of seconds in a week:

```
SELECT 60 * 60 * 24 * 7; -- result: 604800
```

CASTING

From timeto time, you need to change the type of a number. The CAST() function is there to help you out. It lets you change the type of value to almost anything (integernumericdouble precisionvarchar , and many more).

Get the number as an integer (without rounding):

```
SELECT CAST(1234.567 AS integer)
-- result: 1234
```

Change a column type to double precision

```
SELECT CAST(column AS double precision)
```

USEFUL FUNCTIONS

Get the remainder of a division:

```
SELECT MOD(13, 2
-- result: 1
```

Round a number to its nearest integer:

```
SELECT ROUND(1234.56789)
-- result: 1235
```

Round a number to three decimal places:

```
SELECT ROUND(1234.56789, 3
--  result:1234.568
```

PostgreSQL requires the first argument to be of the type numeric – cast the number when needed.

To round the number **up**

```
SELECT CEIL(13.1; -- result: 14
SELECT CEIL(-13.9; -- result: -13
```

The CEIL(x) function returns the **smallest** integer **not less** than x. In SQL Server, the function is called CEILING()

To round the number **down**:

```
SELECT FLOOR(13.8; -- result: 13
SELECT FLOOR(-13.2); -- result: -14
```

The FLOOR(x) function returns the **greate** than x

To round towards 0 irrespective of the sign of a number:

```
SELECT TRUNC(13.5; -- result: 13
SELECT TRUNC(-13.5); -- result: -13
```

TRUNC(x) works the same way as CAST(x AS integer) . In MySQL, the function is called TRUNCATE().

To get the absolute value of a number:

```
SELECT ABS(-12 ; -- result: 12
```

To get the square root of a number:

```
SELECT SQRT(); -- result: 3
```

NULLs

Toretrieveallrows with a missing value in the price column:

```
WHERE price IS NULL
```

To retrieve all rows with the weight column populated:

```
WHERE weight IS NOT NULL
```

Why shouldn't you use price = NULL or weight != NULL? Because databases don't know if those expressions are true or false – they are evaluated as NULLs.

Moreover, if you use a function or concatenation on a column that is NULL in some rows, then it will get propagated. Take a look:

```
LENGTH(domain) 12 LearnPython.com 15
```

domain
LearnSQL.com
NULL
vertabelo.com

USEFUL FUNCTIONS

COALESCE(x,y, ...)

To replace NULL in a query with something meaningful:

```
SELECT
domain,
  COALESCE(domain, 'domain missing'
FROM contacts;
```

domain	coalesce
LearnSQL.com	LearnSQL.com
NULL	domain missing

The COALESCE() function takes any number of arguments and returns the value of the first argument that isn't NULL

om *division by 0*
th, this_month,

/ NULLIF(last_month, 0
AS better_by_percent
FROM video_views;

last_month	this_month	better_by_percent
723786	1085679	150.0
0	178123	NULL

The NULLIF(x, y) function will return NULL if x is the same as y, else it will return the x value.

CASE WHEN

The basicversionof CASE WHEN checks if the values are equal (e.g., if fee is equal to 50, then 'normal' is returned). If there isn't a matching value in the CASE WHEN then the ELSE value will be returned (e.g., if fee is equal to 49, then 'not available' will show up.

```
SELECT
  CASE fee
    WHEN 50 THEN 'normal'
    WHEN 10 THEN 'reduced'
    WHEN  0 THEN 'free'
    ELSE 'not available'
  ENDAS tariff
FROM ticket_types;
```

The most popular type is the **searched CASE WHEN** – it lets you pass conditions (as you'd write them in the WHERE clause), evaluates them in order, then returns the value for the first condition met.

```
SELECT CASE

    WHEN score >= 90 THEN 'A'
  WHEN score >  60 THEN 'B'
  ELSE 'F'
  ENDAS grade
FROM test_results;
```

Here, all students who scored at least 90 will get an A, those with the score above 60 (and below 90) will get a B, and the rest will receive an F.

TROUBLESHOOTING

Integer division

When you don't see the decimal places you expect, it means that you are dividing between two integers. Cast one to decimal:

```
CAST(123AS decimal  / 2
```

Division by 0

To avoid this error, make sure that the denominator is not equal to 0. You can use the NULLIF() function to replace 0 with a NULL, which will result in a NULL for the whole expression:

```
count / NULLIF(count_all, 0
```

Inexact calculations If you do calculations using real (floating point) numbers, you'll end up with some inaccuracies. This is because this type is meant for scientific calculations such as calculating the velocity. Whenever you need accuracy (such as dealing with monetary values), use the decimal / numeric type (or money if available).

Errors when rounding with a specified precision

Most databases won't complain, but do check the documentation if they do. For example, if you want to specify the rounding precision in PostgreSQL, the value must be of the numeric type.

Standard SQL Functions Cheat Sheet

AGGREGATION AND GROUPING

- **COUNT(**expr – thecount of valuesfor the rowswithinthe group
- **SUM(**expr – the sum of values within the group
- **AVG(**expr – the average value for the rows within the group
- **MIN(**expr – the minimum value within the group
- **MAX(**expr – the maximum value within the group

To get the number of rows in the table:

```
SELECT COUNT(
FROM city;
```

To get the number of non-NULL values in a column:

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

To get the count of unique values in a column:

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

GROUP BY

CITY	
name	country_id
Paris	1
Marseille	1
Lyon	1
Berlin	2
Hamburg	2
Munich	2
Warsaw	4
Cracow	4

→

CITY	
country_id	count
1	3
2	3
4	2

The example above – the count of cities in each country:

```
SELECT name, COUNT(country_id
FROM city
GROUP BY name;
```

The average rating for the city:

```
SELECT city_id, AVG(rating
FROM ratings
GROUP BY city_id;
```

Commonmistake: COUNT(*) and LEFT JOIN

When youjoin the tableslikethis: client LEFT JOIN project ,and you wanttogetthenumberofprojects for every client youknow, COUNT(*) willreturn**1**foreachclient even if you've never workedforthem.Thisisbecause,they're still present inthe list butwiththe NULLinthefieldsrelated to the project after the JOIN.Togetthecorrectcount(**0**forthe clients you've never workedfor),countthevaluesinacolumn of the other table, e.g., COUNT(project_name) .Checkout this exercise to see an example.

DATE AND TIME

There are3 maintime-relatedtypes: date, time, and timestamp. Time is expressed using a 24-hour clock, and it can be as vague as just hour and minutes (e.g., 15:30 – 3:30 p.m.) or as precise as microseconds and time zone (as shown below):

14:39:53.662522-05 is almost2:40p.m. CDT (e.g., in Chicago; inUTCit'dbe7:40 p.m.).Theletters in the above example represent:

In the date part:

- YYYY – the 4-digit year.
- mm – the zero-padded month (01—January through 12 December).
- dd – the zero-padded day.

In the time part:

- HH – the zero-padded hour in a 24-hour clock.
- MM – the minutes.
- SS – the seconds. *Omissible*.
- ssssss – the smaller parts of a second – they can be expressed using 1 to 6 digits. *Omissible*.
- ±TZ – the timezone. It must start with either or , and use two digits relative to UTC. *Omissible*.

What time is it?

To answer that question in SQL, you can use:

- CURRENT_TIME – to find what time it is.
- CURRENT_DATE – to get today's date. (GETDATE() in SQL Ser ver.)
- CURRENT_TIMESTAMP – to get the timestamp with the two above.

Creating values

Tocreate a date time ,or timestamp ,simplywritethe value asastring andcastittothepropertype.

```
SELECT  CAST('2021-12-31' AS date
SELECT  CAST('15:31' AS time
SELECT  CAST('2021-12-31 23:59:29+02' AS
timestamp
SELECT  CAST('15:31.124769' AS time
```

Becareful with the lastexample –it willbeinterpretedas15 minutes 31 seconds and 124769 microseconds! It is always a good idea to write 00 explicitly for hours: '00:15:31.124769'.

You might skip casting in simple conditions – the database will know what you mean.

```
SELECT airline, flight_number, departure_time
FROM airport_schedule
WHERE departure_time < '12:00';
```

INTERVALs

Note: InSQL Server,intervals aren't implemented – use the DATEADD() and DATEDIFF() functions.

To get the simplest interval, subtract one time value from another:

```
SELECTCAST('2021-12-31 23:59:59' AS
timestamp - CAST('2021-06-01 12:00:00' AS
timestamp
-- result: 213 days 11:59:59
```

To define an interval: INTERVAL '1' DAY Thissyntaxconsistsofthreeelements:the INTERVAL keyword, a quoted value, and a time part keyword (in singular form.) You can use the followingtime parts: YEAR,MONTH, WEEK DAY, HOUR MINUTE,and SECOND.InMySQL,omitthe quotes.You canjoin many different INTERVALs usingthe or operator:

```
INTERVAL '1' YEAR + INTERVAL '3' MONTH
```

In some databases, there's an easier way to get the above value. And it accepts plural forms! INTERVAL '1 year 3 months'

There are two more syntaxes in the Standard SQL:

Syntax	What it does
INTERVAL 'x-y' YEAR TO MONTH	INTERVAL 'x year y month'
INTERVAL 'x-y' DAY TO SECOND	INTERVAL 'x day y second'

In MySQL, write year_month instead of YEAR TO MONTH and day_second instead of DAY TO SECOND.

To get the last day of a month, add one month and subtract one day:

```
SELECTCAST('2021-02-01' AS date
        + INTERVAL '1' MONTH
        - INTERVAL '1' DAY
```

To get all events for next three months from today:

```
SELECT event_date, event_name
FROM calendar
WHERE event_date BETWEEN  CURRENT_DATEAND
CURRENT_DATE + INTERVAL '3' MONTH
```

To get part of the date:

```
SELECTEXTRACT(YEAR  FROM birthday
FROM artists;
```

Oneofpossiblereturnedvalues:1946.InSQLServer, use the DATEPART(part, date) function.

TIME ZONES

Inthe SQL Standard, the date type can't have an associated time zone, but the time and timestamp types can. In the real world, time zones have little meaning without the date, as the offset can vary through the year because of **daylight saving time**. So, it's best to work with the timestamp values.

When working with the type timestamp with time zone (abbr. timestamptz), you can type in the value in your local time zone, and it'll get converted to the UTC time zone as it is inserted into the table. Later when you select from the table it gets converted back to your local time zone. This is immune to time zone changes.

AT TIME ZONE To operate between different time zones, use the AT TIME ZONE keyword.

If you use this format: {**timestamp without time zone**} AT TIME ZONE {time zone}, then the database will read the time stamp in the specified time zone and convert it to the time zone local to the display. It returns the time in the format **timestamp with time zone**.

If you use this format: {**timestamp with time zone**} AT TIME ZONE {time zone}, then the database will convert the time in one time zone to the target time zone specified by AT TIME ZONE. It returns the time in the format timestamp without time zone, in the target time zone.

You can define the time zone with popular shortcuts like UTC, MST, or GMT, or by continent/city such as: **America/New_York**, **Europe/London**, and **Asia/Tokyo**.

Examples

We set the local time zone to **'America/New_York'**.

```
SELECT  TIMESTAMP '2021-07-16 21:00:00' AT
TIME ZONE 'America/Los_Angeles';
-- result: 2021-07-17 00:00:00-04
```

Here, the database takes a timestamp without a time zone and it's told it's in Los Angeles time, which is then converted to the local time – New York for displaying. This answers the question **"At what time should I turn on the TV if the show starts at 9 PM in Los Angeles?"**

```
SELECT  TIMESTAMP WITH TIME ZONE '2021-06-20
19:30:00' AT TIME ZONE 'Australia/Sydney';
-- result: 2021-06-21 09:30:00
```

Here, the database gets a timestamp specified in the local time zone and converts it to the time in Sydney (note that it didn't return a time zone.) This answers the question **"What time is it in Sydney if it's 7:30 PM here?"**

Advanced Topics

INDEXES & PERFORMANCE TUNING

What is an index?
An index speeds up data retrieval by providing a fast lookup path for rows, like a book's index.

Why use index?
Without indexes, databases scan every row (full table scan), which is slow for large datasets.

Types of Indexes:
- Primary Index — automatically created on primary keys
- Unique Index — prevents duplicate values
- Composite Index — on multiple columns
- Clustered vs. Non-Clustered (SQL Server):
  - Clustered: physically reorders table data
  - Non-Clustered: separate from data

Example:
```
EXPLAIN ANALYZE
SELECT name
FROM city
WHERE name = 'Paris';
```

This shows how the database uses an index if available.

Check your query performance:
```
EXPLAIN SELECT
FROM city WHERE name = 'Paris'
```

Stored Procedures & Functions

What?
Reusable blocks of code stored in the database that can accept parameters.
Why?
They centralize logic and improve performance by reducing network traffic.

Example:
Procedure (Postgres example):
```
CREATE OR REPLACE
FUNCTION update_rating(city_id INT, new_rating INT)
RETURNS VOID AS $$
BEGIN
 UPDATE city
 SET rating = new_rating
 WHERE id = city_id;
END;
$$ LANGUAGE plpgsql;
```

Call it:
```
SELECT update_rating(5, 4);
```
This calls a stored function to update a city's rating.

Transactions & ACID

What?
Transactions group multiple statements into a single all-or-nothing unit.
Why?
They guarantee data consistency and allow rolling back if there's an error.

Example:
```
BEGIN;

UPDATE city
SET population = population + 1000
WHERE name = 'Rome';

SELECT *
FROM city
WHERE name = 'Rome';

ROLLBACK; -- undo changes
```

Transaction block:
```
BEGIN;
-- your queries here
COMMIT;
```

If error:
```
ROLLBACK;
```

ACID guarantees:
- Atomicity: all-or-nothing
- Consistency: data remains valid
- Isolation: transactions don't step on each other
- Durability: committed changes survive failures

Isolation levels:
- READ UNCOMMITTED
- READ COMMITTED (default)
- REPEATABLE READ
- SERIALIZABLE

Constraints

What?
Constraints enforce data rules (e.g. no duplicates, valid relationships).
Why?
They protect data integrity by automatically preventing invalid data.

Primary Key:
```
CREATE TABLE country (
 id SERIAL PRIMARY KEY,
 name VARCHAR(50)
);
```

Foreign Key:
```
ALTER TABLE city
ADD CONSTRAINT fk_country
FOREIGN KEY (country_id)
REFERENCES country(id);
```

Unique:
```
ALTER TABLE city
ADD CONSTRAINT unique_name UNIQUE (name);
```

Check:
```
ALTER TABLE city
ADD CHECK (population > 0);
```

Triggers

What?
Modern databases let you store and query JSON directly inside columns.
Why?
It supports semi-structured data without needing separate tables.

Example:
```
SELECT
 json_data->>'population'
FROM
 city_json
WHERE
 json_data->>'name' = 'Paris';
```

Views & CTEs

What?
- View: saved query as a virtual table
- CTE: named temporary result set within a query

Why?
They simplify complex queries and improve readability.

View:
```
CREATE VIEW popular_cities AS
SELECT name, population
FROM city
WHERE rating > 4;
```

CTE (Common Table Expression):
```
WITH top_countries AS (
 SELECT country_id, COUNT(*) AS num_cities
 FROM city
 GROUP BY country_id
)
SELECT *
FROM top_countries
WHERE num_cities > 5;
```

JSON Functions

What?
A trigger runs automatically when data in a table is modified.
Why?
It enforces business rules or automates actions, like logging changes.
Example:
```
-- See triggers on a table
SELECT tgname
FROM pg_trigger
WHERE tgrelid = 'city'::regclass;
```

THANK YOU

for using this SQL Basics Cheat Sheet. We hope it helps you master essential SQL skills for your data projects, interviews, and everyday work.

Keep exploring, keep learning, and keep querying!