SQL for Data Analysis Cheat Sheet

LearnSQL.com

SQL

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

SELECT

Fetch the `id` and `name` columns from the `product` table:

```
SELECT id, name
FROM product;
```

Concatenate the name and the description to fetch the full description of the products:

```
SELECT name || ' - ' || description
FROM product;
```

Fetch names of products with prices above 15:

```
SELECT name
FROM product
WHERE price > 15;
```

Fetch names of products with prices between 50 and 150:

```
SELECT name
FROM product
WHERE price BETWEEN 50 AND 150;
```

Fetch names of products that are not watches:

```
SELECT name
FROM product
WHERE name != 'watch';
```

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

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

Fetch names of products that start with any letter followed by `'rain'` (like `'train'` or `'grain'`):

```
SELECT name
FROM product
WHERE name LIKE '_rain';
```

Fetch names of products with non-null prices:

```
SELECT name
FROM product
WHERE price IS NOT NULL;
```

GROUP BY

PRODUCT	
name	**category**
Knife	Kitchen
Pot	Kitchen
Mixer	Kitchen
Jeans	Clothing
Sneakers	Clothing
Leggings	Clothing
Smart TV	Electronics
Laptop	Electronics

→

category	count
Kitchen	3
Clothing	3
Electronics	2

AGGREGATE FUNCTIONS

Count the number of products:

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

Count the number of products with non-null prices:

```
SELECT COUNT(price)
FROM product;
```

Count the number of unique `category` values:

```
SELECT COUNT(DISTINCT category)
FROM product;
```

Get the lowest and the highest product price:

```
SELECT MIN(price), MAX(price)
FROM product;
```

Find the total price of products for each category:

```
SELECT category, SUM(price)
FROM product
GROUP BY category;
```

Find the average price of products for each category whose average is above 3.0:

```
SELECT category, AVG(price)
FROM product
GROUP BY category
HAVING AVG(price) > 3.0;
```

ORDER BY

Fetch product names sorted by the `price` column in the default ASCending order:

```
SELECT name
FROM product
ORDER BY price [ASC];
```

Fetch product names sorted by the `price` column in DESCending order:

```
SELECT name
FROM product
ORDER BY price DESC;
```

COMPUTATIONS

Use `+`, `-`, `*`, `/` to do basic math. To get the number of seconds in a week:

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

ROUNDING NUMBERS

Round a number to its nearest integer:

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

Round a number to two decimal places:

```
SELECT ROUND(AVG(price), 2)
FROM product
WHERE category_id = 21;
-- result: 124.56
```

TROUBLESHOOTING

INTEGER DIVISION

In PostgreSQL and SQL Server, the `/` operator performs integer division for integer arguments. If you do not see the number of decimal places you expect, it is because you are dividing between two integers. Cast one to decimal:

```
123 / 2 -- result: 61
CAST(123 AS decimal) / 2 -- result: 61.5
```

DIVISION BY 0

To avoid this error, make sure the denominator is not `0`. You may use the `NULLIF()` function to replace `0` with a `NULL`, which results in a `NULL` for the entire expression:

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

JOIN

JOIN is used to fetch data from multiple tables. To get the names of products purchased in each order, use:

```
SELECT
 orders.order_date,
 product.name AS product,
 amount
FROM orders
JOIN product
  ON product.id = orders.product_id;
```

Learn more about JOINs in our interactive SQL JOINs course.

INSERT

To insert data into a table, use the `INSERT` command:

```
INSERT INTO category
VALUES
(1, 'Home and Kitchen'),
(2, 'Clothing and Apparel');
```

You may specify the columns to which the data is added. The remaining columns are filled with predefined default values or `NULL`s.

```
INSERT INTO category (name)
VALUES ('Electronics');
```

UPDATE

To update the data in a table, use the `UPDATE` command:

```
UPDATE category
SET
 is_active = true,
 name = 'Office'
WHERE name = 'Ofice';
```

DELETE

To delete data from a table, use the `DELETE` command:

```
DELETE FROM category
WHERE name IS NULL;
```

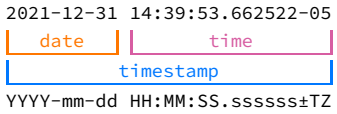
Check out our interactive course How to INSERT, UPDATE, and DELETE Data in SQL.

SQL for Data Analysis Cheat Sheet

LearnSQL.com

DATE AND TIME

There are 3 main time-related types: `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 almost 2:40 p.m. CDT (e.g., in Chicago; in UTC it'd be 7:40 p.m.). The letters 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*.

CURRENT DATE AND TIME

Find out what time it is:

```
SELECT CURRENT_TIME;
```

Get today's date:

```
SELECT CURRENT_DATE;
```

In SQL Server:

```
SELECT GETDATE();
```

Get the timestamp with the current date and time:

```
SELECT CURRENT_TIMESTAMP;
```

CREATING DATE AND TIME VALUES

To create a date, time, or timestamp, write the value as a string and cast it to the proper type.

```
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);
```

Be careful with the last example – it is interpreted as 15 minutes 31 seconds and 124769 microseconds! It is always a good idea to write 00 for hours explicitly: `'00:15:31.124769'`.

SORTING CHRONOLOGICALLY

Using `ORDER BY` on date and time columns sorts rows chronologically from the oldest to the most recent:

```
SELECT order_date, product, quantity
FROM sales
ORDER BY order_date;
```

order_date	product	quantity
2023-07-22	Laptop	2
2023-07-23	Mouse	3
2023-07-24	Sneakers	10
2023-07-24	Jeans	3
2023-07-25	Mixer	2

Use the DESCending order to sort from the most recent to the oldest:

```
SELECT order_date, product, quantity
FROM sales
ORDER BY order_date DESC;
```

COMPARING DATE AND TIME VALUES

You may use the comparison operators <, <=, >, >=, and = to compare date and time values. Earlier dates are less than later ones. For example, `2023-07-05` is "less" than `2023-08-05`.

Find sales made in July 2023:

```
SELECT order_date, product_name, quantity
FROM sales
WHERE order_date >= '2023-07-01'
  AND order_date <  '2023-08-01';
```

Find customers who registered in July 2023:

```
SELECT registration_timestamp, email
FROM customer
WHERE registration_timestamp >= '2023-07-01'
  AND registration_timestamp <  '2023-08-01';
```

Note: Pay attention to the end date in the query. The upper bound `'2023-08-01'` is not included in the range. The timestamp `'2023-08-01'` is actually the timestamp `'2023-08-01 00:00:00.0'`. The comparison operator < is used to ensure the selection is made for all timestamps less than `'2023-08-01 00:00:00.0'`, that is, all timestamps in July 2023, even those close to the midnight of August 1, 2023.

INTERVALS

An interval measures the difference between two points in time. For example, the interval between `2023-07-04` and `2023-07-06` is 2 days.

To define an interval in SQL, use this syntax:

```
INTERVAL '1' DAY
```

The syntax consists of three elements: the `INTERVAL` keyword, a quoted value, and a time part keyword. You may use the following time parts: `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

Adding intervals to date and time values

You may use + or − to add or subtract an interval to date or timestamp values.

Subtract one year from `2023-07-05`:

```
SELECT CAST('2023-07-05' AS TIMESTAMP)
       - INTERVAL '1' year;
-- result: 2022-07-05 00:00:00
```

Find customers who placed the first order within a month from the registration date:

```
SELECT id
FROM customers
WHERE first_order_date >
  registration_date + INTERVAL '1' month;
```

Filtering events to those in the last 7 days

To find the deliveries scheduled for the last 7 days, use:

```
SELECT delivery_date, address
FROM sales
WHERE delivery_date <= CURRENT_DATE
  AND delivery_date >= CURRENT_DATE
      - INTERVAL '7' DAY;
```

Note: In SQL Server, intervals are not implemented – use the `DATEADD()` and `DATEDIFF()` functions.

Filtering events to those in the last 7 days in SQL Server

To find the sales made within the last 7 days, use:

```
SELECT delivery_date, address
FROM sales
WHERE delivery_date <= GETDATE()
  AND delivery_date >=
    DATEADD(DAY, -7, GETDATE());
```

EXTRACTING PARTS OF DATES

The standard SQL syntax to get a part of a date is

```
SELECT EXTRACT(YEAR FROM order_date)
FROM sales;
```

You may extract the following fields:
`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

The standard syntax does not work In SQL Server.
Use the `DATEPART(part, date)` function instead.

```
SELECT DATEPART(YEAR, order_date)
FROM sales;
```

GROUPING BY YEAR AND MONTH

Find the count of sales by month:

```
SELECT
 EXTRACT(YEAR FROM order_date) AS year,
 EXTRACT(MONTH FROM order_date) AS month,
 COUNT(*) AS count
FROM sales
GROUP BY
 year,
 month
ORDER BY
 year
 month;
```

year	month	count
2022	8	51
2022	9	58
2022	10	62
2022	11	76
2022	12	85
2023	1	71
2023	2	69

Note that you must group by both the year and the month. `EXTRACT(MONTH FROM order_date)` only extracts the month number (1, 2, ..., 12). To distinguish between months from different years, you must also group by year.

More about working with date and time values in our interactive Standard SQL Functions course.

SQL for Data Analysis Cheat Sheet

LearnSQL.com

CASE WHEN

`CASE WHEN` lets you pass conditions (as in the `WHERE` clause), evaluates them in order, then returns the value for the first condition met.

```
SELECT
 name,
 CASE
   WHEN price > 150 THEN 'Premium'
   WHEN price > 100 THEN 'Mid-range'
   ELSE 'Standard'
 END AS price_category
FROM product;
```

Here, all products with prices above 150 get the *Premium* label, those with prices above 100 (and below 150) get the *Mid-range* label, and the rest receives the *Standard* label.

CASE WHEN and GROUP BY

You may combine `CASE WHEN` and `GROUP BY` to compute object statistics in the categories you define.

```
SELECT
 CASE
   WHEN price > 150 THEN 'Premium'
   WHEN price > 100 THEN 'Mid-range'
   ELSE 'Standard'
 END AS price_category,
 COUNT(*) AS products
FROM product
GROUP BY price_category;
```

Count the number of large orders for each customer using `CASE WHEN` and `SUM()`:

```
SELECT
  customer_id,
  SUM(
    CASE WHEN quantity > 10
    THEN 1 ELSE 0 END
  ) AS large_orders
FROM sales
GROUP BY customer_id;
```

... or using `CASE WHEN` and `COUNT()`:

```
SELECT
  customer_id,
  COUNT(
    CASE WHEN quantity > 10
    THEN order_id END
  ) AS large_orders
FROM sales
GROUP BY customer_id;
```

GROUP BY EXTENSIONS

GROUPING SETS

`GROUPING SETS` lets you specify multiple sets of columns to group by in one query.

```
SELECT region, product, COUNT(order_id)
FROM sales
GROUP BY
  GROUPING SETS ((region, product), ());
```

region	product	count	
USA	Laptop	10	GROUP BY (region, product)
USA	Mouse	5	
UK	Laptop	6	
NULL	NULL	21	GROUP BY () – all rows

CUBE

CUBE generates groupings for all possible subsets of the GROUP BY columns.

```
SELECT region, product, COUNT(order_id)
FROM sales
GROUP BY CUBE (region, product);
```

region	product	count	
USA	Laptop	10	GROUP BY region, product
USA	Mouse	5	
UK	Laptop	6	
USA	NULL	15	GROUP BY region
UK	NULL	6	
NULL	Laptop	16	GROUP BY product
NULL	Mouse	5	
NULL	NULL	21	GROUP BY () – all rows

ROLLUP

ROLLUP adds new levels of grouping for subtotals and grand totals.

```
SELECT region, product, COUNT(order_id)
FROM sales
GROUP BY ROLLUP (region, product);
```

region	product	count	
USA	Laptop	10	GROUP BY region, product
USA	Mouse	5	
UK	Laptop	6	
USA	NULL	15	GROUP BY region
UK	NULL	6	
NULL	NULL	21	GROUP BY () – all rows

COALESCE

COALESCE replaces the first NULL argument with a given value. It is often used to display labels with `GROUP BY` extensions.

```
SELECT region,
 COALESCE(product, 'All'),
 COUNT(order_id)
FROM sales
GROUP BY ROLLUP (region, product);
```

region	product	count
USA	Laptop	10
USA	Mouse	5
USA	All	15
UK	Laptop	6
UK	All	6
All	All	21

COMMON TABLE EXPRESSIONS

A common table expression (CTE) is a named temporary result set that can be referenced within a larger query. They are especially useful for complex aggregations and for breaking down large queries into more manageable parts.

```
WITH total_product_sales AS (
 SELECT product, SUM(profit) AS total_profit
 FROM sales
 GROUP BY product
)

SELECT AVG(total_profit)
FROM total_product_sales;
```

Check out our hands-on courses on Common Table Expressions and GROUP BY Extensions.

WINDOW FUNCTIONS

Window functions compute their results based on a sliding window frame, a set of rows related to the current row. Unlike aggregate functions, window functions do not collapse rows.

COMPUTING THE PERCENT OF TOTAL WITHIN A GROUP

```
SELECT product, brand, profit,
  (100.0 * profit /
    SUM(profit) OVER(PARTITION BY brand)
  ) AS perc
FROM sales;
```

product	brand	profit	perc
Knife	Culina	1000	25
Pot	Culina	3000	75
Doll	Toyze	2000	40
Car	Toyze	3000	60

RANKING

Rank products by price:

```
SELECT RANK() OVER(ORDER BY price), name
FROM product;
```

RANKING FUNCTIONS

RANK – gives the same rank for tied values, leaves gaps.
DENSE_RANK – gives the same rank for tied values without gaps.
ROW_NUMBER – gives consecutive numbers without gaps.

name	rank	dense_rank	row_number
Jeans	1	1	1
Leggings	2	2	2
Leggings	2	2	3
Sneakers	4	3	4
Sneakers	4	3	5
Sneakers	4	3	6
T-Shirt	7	4	7

RUNNING TOTAL

A running total is the cumulative sum of a given value and all preceding values in a column.

```
SELECT date, amount,
  SUM(amount) OVER(ORDER BY date)
    AS running_total
FROM sales;
```

MOVING AVERAGE

A moving average (*a.k.a.* rolling average, running average) is a technique for analyzing trends in time series data. It is the average of the current value and a specified number of preceding values.

```
SELECT date, price,
  AVG(price) OVER(
    ORDER BY date
    ROWS BETWEEN 2 PRECEDING
      AND CURRENT ROW
  ) AS moving_averge
FROM stock_prices;
```

DIFFERENCE BETWEEN TWO ROWS (DELTA)

```
SELECT year, revenue,
  LAG(revenue) OVER(ORDER BY year)
    AS revenue_prev_year,
  revenue -
    LAG(revenue) OVER(ORDER BY year)
    AS yoy_difference
FROM yearly_metrics;
```

Learn about SQL window functions in our interactive Window Functions course.

SQL JOINs Cheat Sheet

LearnSQL.com

JOINING TABLES

JOIN combines data from two tables.

TOY

toy_id	toy_name	cat_id
1	ball	3
2	spring	NULL
3	mouse	1
4	mouse	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	Kitty
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 NULLs. LEFT JOIN is also called LEFT OUTER JOIN.

```
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 NULLs. RIGHT JOIN 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	Kitty
3	mouse	1	1	Kitty
NULL	NULL	NULL	2	Hugo
1	ball	3	3	Sam
4	mouse	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. FULL JOIN is also called FULL OUTER JOIN.

```
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	Kitty
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	mouse	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

LearnSQL.com

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 Smith
2	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;
```

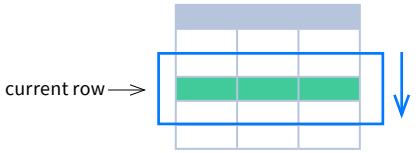
cat_name	owner_name	age	age
Kitty	John Smith	17	18
Sam	Danielle Davis	5	10

SQL Window Functions Cheat Sheet

LearnSQL.com

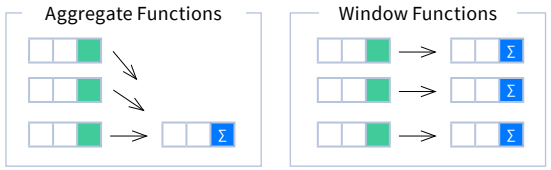
WINDOW FUNCTIONS

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. WHERE
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

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

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

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

PARTITION BY city ORDER BY month

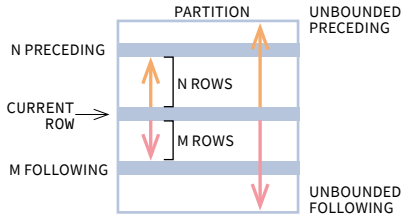
sold	city	month
300	Paris	1
500	Paris	2
200	Rome	1
300	Rome	2
400	Rome	3
100	London	1
400	London	2

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

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

LearnSQL.com

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 rows within the window frame
- **`count`***(expr)* – count of values for rows within 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 the window 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 for tied 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. Ranking functions do 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

- **`ntile`***(n)* – divide rows within a partition as equally as possible into *n* groups, and assign each row its group 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 and Window Frame: `ntile()`, `lead()`, and `lag()` require an ORDER BY. They do not accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

- **`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()`. With the default window frame for ORDER BY, RANGE UNBOUNDED PRECEDING, `last_value()` returns the value for the current row.

- **`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 (PARTITION BY city
  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`).