Joining Data in SQL

Cheat Sheet

Learn SQL online at www.DataCamp.com

> Definitions used throughout this cheat sheet

Primary key:
A primary key is a field in a table that uniquely identifies each record in the table. In relational databases, primary keys can be used as fields to join tables on.

Foreign key:
A foreign key is a field in a table which references the primary key of another table. In a relational database, one way to join two tables is by connecting the foreign key from one table to the primary key of another.

One-to-one relationship:
Database relationships describe the relationships between records in different tables. When a one-to-one relationship exists between two tables, a given record in one table is uniquely related to exactly one record in the other table.

One-to-many relationship:
In a one-to-many relationship, a record in one table can be related to one or more records in a second table. However, a given record in the second table will only be related to one record in the first table.

Many-to-many relationship:
In a many-to-many relationship, records in a given table 'A' can be related to one or more records in another table 'B', and records in table B can also be related to many records in table A.

> Sample Data

Artist Table

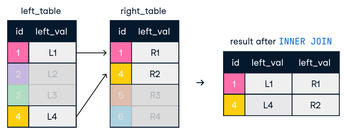
artist_id	name
1	AC/DC
2	Aerosmith
3	Alanis Morissette

Album Table

album_id	title	artist_id
1	For those who rock	1
2	Dream on	2
3	Restless and wild	2
4	Let there be rock	1
5	Rumours	6

INNER JOIN

An inner join between two tables will return only records where a joining field, such as a key, finds a match in both tables.



INNER JOIN join ON one field

```
SELECT *
FROM artist AS art
INNER JOIN album AS alb
ON art.artist_id = alb.artist_id;
```

Result after INNER JOIN:

album_id	name	artist_id
1	AC/DC	1
1	AC/DC	4
2	Aerosmith	2
2	Aerosmith	3

INNER JOIN with USING

```
SELECT *
FROM artist AS art
INNER JOIN album AS alb
USING (artist_id);
```

SELF JOIN

Self-joins are used to compare values in a table to other values of the same table by joining different parts of a table together.

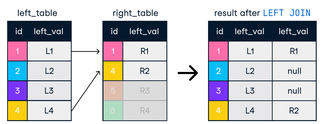
```
SELECT
  art1.artist_id,
  art1.title AS art1_title,
  art2.title AS art2_title
FROM artist as art1
INNER JOIN artist as art2
ON art1.artist_id = art2.album_id;
```

Result after Self join:

artist_id	art1_title	art2_title
1	For those who rock	For those who rock
2	Dream on	Dream on
2	Restless and wild	Dream on
1	Let there be rock	For those who rock

LEFT JOIN

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



LEFT JOIN on one field

```
SELECT *
FROM artist AS art
LEFT JOIN album AS alb
ON art.artist_id = alb.album_id;
```

Result after LEFT JOIN:

artist_id	name	album_id	title	name
1	AC/DC	1	For those who rock	1
1	AC/DC	4	Let there be rock	1
2	Aerosmith	2	Dream on	2
2	Aerosmith	3	Restless and wild	2
3	Alanis Morissette	null	null	null

RIGHT JOIN

A right join keeps all of the original records in the right table and returns missing values for any columns from the left table where the joining field did not find a match. Right joins are far less common than left joins, because right joins can always be re-written as left joins.

RIGHT JOIN on one field

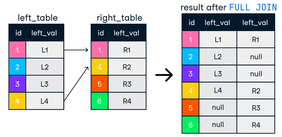
```
SELECT*
FROM artist as art
RIGHT JOIN album AS alb
ON art.artist_id = alb.album_id;
```

Result after RIGHT JOIN:

artist_id	name	album_id	title	name
1	AC/DC	1	For those who rock	1
1	Aerosmith	2	Dream on	2
2	Aerosmith	3	Restless and wild	2
2	AC/DC	4	Let there be rock	1
3	null	5	Rumours	6

FULL JOIN

A full join combines a left join and right join. A full join will return all records from a table, irrespective of whether there is a match on the joining field in the other table, returning null values accordingly.



FULL JOIN on one field

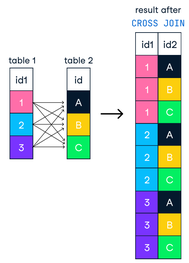
```
SELECT *
FROM artist as art
FULL OUTER JOIN album AS alb
ON art.artist_id = alb.album_id;
```

Result after FULL JOIN:

artist_id	name	album_id	title	name
1	AC/DC	1	For those who rock	1
1	AC/DC	4	Let there be rock	1
2	Aerosmith	2	Balls to the wall	2
2	Aerosmith	3	Restless and wild	2
3	Alanis Morissette	null	null	null
null	null	5	Rumours	6

CROSS JOIN

`CROSS JOIN` creates all possible combinations of two tables. `CROSS JOIN` does not require a field to join `ON`.




```
SELECT name, title
FROM artist
CROSS JOIN album;
```

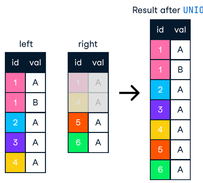
Result after CROSS JOIN:

name	title
AC/DC	For those who rock
AC/DC	Dream on
AC/DC	Restless and wild
AC/DC	Let there be rock
AC/DC	Rumours
Aerosmith	For those who rock
Aerosmith	Dream on
Aerosmith	Restless and wild
Aerosmith	Let there be rock
Aerosmith	Rumours
Alanis Morissette	For those who rock
Alanis Morissette	Dream on
Alanis Morissette	Restless and wild
Alanis Morissette	Let there be rock
Alanis Morissette	Rumours

Set Theory Operators in SQL

UNION

The `UNION` operator is used to vertically combine the results of two `SELECT` statements. For `UNION` to work without errors, all `SELECT` statements must have the same number of columns and corresponding columns must have the same data type. `UNION` does not return duplicates.



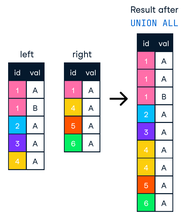

```
SELECT artist_id
FROM artist
UNION
SELECT artist_id
FROM album;
```

Result after UNION:

artist_id
1
2
3
6

UNION ALL

The `UNION ALL` operator works just like `UNION`, but it returns duplicate values. The same restrictions of `UNION` hold true for `UNION ALL`



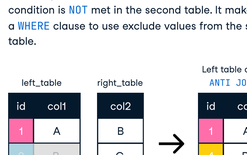

```
SELECT artist_id
FROM artist
UNION ALL
SELECT artist_id
FROM album;
```

Result after UNION ALL:

artist_id
1
2
3
1
2
2
1
6

INTERSECT

The `INTERSECT` operator returns only identical rows from two tables.




```
SELECT artist_id
FROM artist
INTERSECT
SELECT artist_id
FROM album;
```

Result after INTERSECT:

artist_id
1
2

EXCEPT

The `EXCEPT` operator returns only those rows from the left table that are not present in the right table.

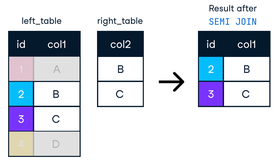

```
SELECT artist_id
FROM artist
EXCEPT
SELECT artist_id
FROM album;
```

Result after EXCEPT:

artist_id
1
2
3

SEMI JOIN

A semi join chooses records in the first table where a condition is met in the second table. A semi join makes use of a `WHERE` clause to use the second table as a filter for the first.




```
SELECT *
FROM album
WHERE artist_id IN
(SELECT artist_id
FROM artist);
```

Result after Semi join:

album_id	title	artist_id
1	For those who rock	1
2	Dream on	2
3	Restless and wild	2

ANTI JOIN

The anti join chooses records in the first table where a condition is `NOT` met in the second table. It makes use of a `WHERE` clause to use exclude values from the second table.


```
SELECT *
FROM album
WHERE artist_id NOT IN
(SELECT artist_id
FROM artist);
```

Result after Anti join:

album_id	title	artist_id
5	Rumours	6

SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

```
SELECT c1, c2 FROM t
WHERE  c1 BETWEEN low AND high;
```
Query rows between two values

```
SELECT c1, c2 FROM t
WHERE  c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

SQL CHEAT SHEET http://www.sqltutorial.org

MANAGING TABLES

```
CREATE TABLE t (
    id INT PRIMARY KEY,
    name VARCHAR NOT NULL,
    price INT DEFAULT 0
);
```
Create a new table with three columns

```
DROP TABLE t ;
```
Delete the table from the database

```
ALTER TABLE t ADD column;
```
Add a new column to the table

```
ALTER TABLE t DROP COLUMN c ;
```
Drop column c from the table

```
ALTER TABLE t ADD constraint;
```
Add a constraint

```
ALTER TABLE t DROP constraint;
```
Drop a constraint

```
ALTER TABLE t1 RENAME TO t2;
```
Rename a table from t1 to t2

```
ALTER TABLE t1 RENAME c1 TO c2 ;
```
Rename column c1 to c2

```
TRUNCATE TABLE t;
```
Remove all data in a table

USING SQL CONSTRAINTS

```
CREATE TABLE t(
   c1 INT, c2 INT, c3 VARCHAR,
   PRIMARY KEY (c1,c2)
);
```
Set c1 and c2 as a primary key

```
CREATE TABLE t1(
   c1 INT PRIMARY KEY,
   c2 INT,
   FOREIGN KEY (c2) REFERENCES t2(c2)
);
```
Set c2 column as a foreign key

```
CREATE TABLE t(
   c1 INT, c1 INT,
   UNIQUE(c2,c3)
);
```
Make the values in c1 and c2 unique

```
CREATE TABLE t(
  c1 INT, c2 INT,
  CHECK(c1> 0 AND c1 >= c2)
);
```
Ensure c1 > 0 and values in c1 >= c2

```
CREATE TABLE t(
    c1 INT PRIMARY KEY,
    c2 VARCHAR NOT NULL
);
```
Set values in c2 column not NULL

MODIFYING DATA

```
INSERT INTO t(column_list)
VALUES(value_list);
```
Insert one row into a table

```
INSERT INTO t(column_list)
VALUES (value_list),
       (value_list), ....;
```
Insert multiple rows into a table

```
INSERT INTO t1(column_list)
SELECT column_list
FROM t2;
```
Insert rows from t2 into t1

```
UPDATE t
SET c1 = new_value;
```
Update new value in the column c1 for all rows

```
UPDATE t
SET c1 = new_value,
    c2 = new_value
WHERE condition;
```
Update values in the column c1, c2 that match the condition

```
DELETE FROM t;
```
Delete all data in a table

```
DELETE FROM t
WHERE condition;
```
Delete subset of rows in a table

SQL CHEAT SHEET http://www.sqltutorial.org

MANAGING VIEWS

```
CREATE VIEW v(c1,c2)
AS
SELECT c1, c2
FROM t;
```
Create a new view that consists of c1 and c2

```
CREATE VIEW v(c1,c2)
AS
SELECT c1, c2
FROM t;
WITH [CASCADED | LOCAL] CHECK OPTION;
```
Create a new view with check option

```
CREATE RECURSIVE VIEW v
AS
select-statement -- anchor part
UNION [ALL]
select-statement; -- recursive part
```
Create a recursive view

```
CREATE TEMPORARY VIEW v
AS
SELECT c1, c2
FROM t;
```
Create a temporary view

```
DROP VIEW view_name;
```
Delete a view

MANAGING INDEXES

```
CREATE INDEX idx_name
ON t(c1,c2);
```
Create an index on c1 and c2 of the table t

```
CREATE UNIQUE INDEX idx_name
ON t(c3,c4);
```
Create a unique index on c3, c4 of the table t

```
DROP INDEX idx_name;
```
Drop an index

SQL AGGREGATE FUNCTIONS

AVG returns the average of a list

COUNT returns the number of elements of a list

SUM returns the total of a list

MAX returns the maximum value in a list

MIN returns the minimum value in a list

MANAGING TRIGGERS

```
CREATE OR MODIFY TRIGGER trigger_name
WHEN EVENT
ON table_name TRIGGER_TYPE
EXECUTE stored_procedure;
```
Create or modify a trigger

WHEN
- **BEFORE** – invoke before the event occurs
- **AFTER** – invoke after the event occurs

EVENT
- **INSERT** – invoke for INSERT
- **UPDATE** – invoke for UPDATE
- **DELETE** – invoke for DELETE

TRIGGER_TYPE
- **FOR EACH ROW**
- **FOR EACH STATEMENT**

```
CREATE TRIGGER before_insert_person
BEFORE INSERT
ON person FOR EACH ROW
EXECUTE stored_procedure;
```
Create a trigger invoked before a new row is inserted into the person table

```
DROP TRIGGER trigger_name;
```
Delete a specific trigger

SQL Basics Cheat Sheet

LearnSQL.com

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	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.

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

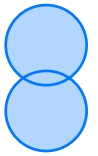
id	name	country
1	YK	DE
2	DF	DE
3	AK	PL
...	...	...

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';
```

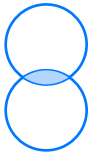


INTERSECT

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';
```

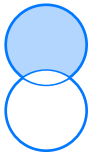


EXCEPT

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';
```

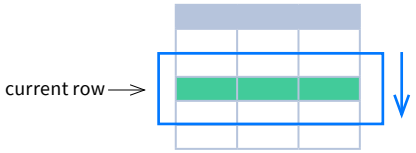


SQL Window Functions Cheat Sheet

LearnSQL.com

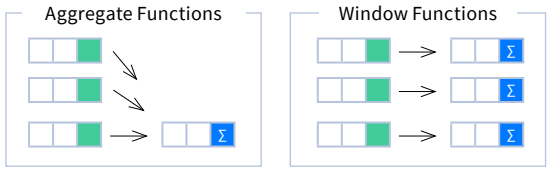
WINDOW FUNCTIONS

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

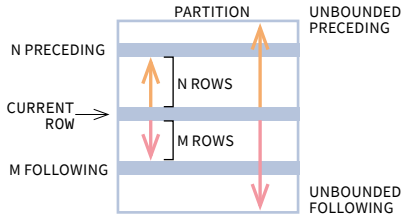
sold	city	month
300	Paris	1
500	Paris	2
200	Rome	1
300	Rome	2
400	Rome	3
100	London	1
400	London	2

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

`lag(sold) OVER(ORDER BY month)`

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

`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

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

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`).

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

Standard SQL Functions Cheat Sheet

LearnSQL.com

TEXT FUNCTIONS

CONCATENATION

Use the `||` operator to concatenate 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 the documentation for your specific database.

LIKE OPERATOR – PATTERN MATCHING

Use the `_` character to replace any single character. Use the `%` 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
WHERE name 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 `+`, `-`, `*`, `/` to do some basic math. To get the number of seconds in a week:

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

CASTING

From time to 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 (`integer`, `numeric`, `double precision`, `varchar`, 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 **greatest** integer **not greater** 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(9); -- result: 3
```

NULLs

To retrieve all rows 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:

domain	LENGTH(domain)
LearnSQL.com	12
LearnPython.com	15
NULL	*NULL*
vertabelo.com	13

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.

NULLIF(x, y)

To save yourself from *division by 0* errors:

```
SELECT
  last_month,
  this_month,
  this_month * 100.0
    / 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 basic version of `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'
  END AS 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'
  END AS 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(123 AS 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

LearnSQL.com

AGGREGATION AND GROUPING

- **COUNT(**expr**)** – the count of values for the rows within the 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;
```

Common mistake: COUNT(*) and LEFT JOIN

When you join the tables like this: `client LEFT JOIN project`, and you want to get the number of projects for every client you know, `COUNT(*)` will return **1** for each client even if you've never worked for them. This is because, they're still present in the list but with the `NULL` in the fields related to the project after the `JOIN`. To get the correct count (**0** for the clients you've never worked for), count the values in a column of the other table, e.g., `COUNT(project_name)`. Check out this exercise to see an example.

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):

```
2021-12-31 14:39:53.662522-05
└ date ┘└      time      ┘
└          timestamp          ┘
YYYY-mm-dd HH:MM:SS.ssssss±TZ
```

`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.*

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 Server.)
- CURRENT_TIMESTAMP – to get the timestamp with the two above.

Creating values

To create a `date`, `time`, or `timestamp`, simply 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 will be interpreted as 15 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: In SQL Server, intervals aren't implemented – use the `DATEADD()` and `DATEDIFF()` functions.

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

```
SELECT CAST('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`
This syntax consists of three elements: the `INTERVAL` keyword, a quoted value, and a time part keyword (in singular form.) You can use the following time parts: `YEAR`, `MONTH`, `WEEK`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`. In MySQL, omit the quotes. You can join many different `INTERVAL`s using the + 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:

```
SELECT CAST('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_DATE AND
CURRENT_DATE + INTERVAL '3' MONTH;
```

To get part of the date:

```
SELECT EXTRACT(YEAR FROM birthday)
FROM artists;
```

One of possible returned values: 1946. In SQL Server, use the `DATEPART(part, date)` function.

TIME ZONES

In the 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?"**

datacamp

SQL for Data Science

SQL Window Functions

Learn SQL online at www.DataCamp.com

> Example dataset

We will use a dataset on the sales of bicycles as a sample. This dataset includes:

The [product] table

The product table contains the types of bicycles sold, their model year, and list price.

product_id	product_name	model_year	list_price
1	Treak 820 - 2016	2016	379.99
2	Ritchey Timberwolf Frameset - 2016	2016	749.99
3	Surly Wednesday Frameset - 2016	2016	999.99
4	Trek Fuel EX 8 29 - 2016	2016	2899.99
5	Heller Shagamaw Frame - 2016	2016	1320.99

The [order] table

The order table contains the order_id and its date.

order_id	order_date
1	2016-01-01T00:00:00.000Z
2	2016-01-01T00:00:00.000Z
3	2016-01-02T00:00:00.000Z
4	2016-01-03T00:00:00.000Z
5	2016-01-03T00:00:00.000Z

The [order_items] table

The order_items table lists the orders of a bicycle store. For each order_id, there are several products sold (product_id). Each product_id has a discount value.

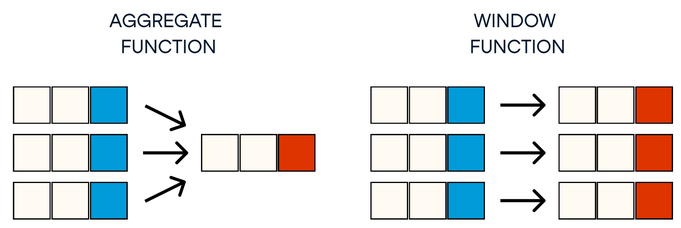
order_id	product_id	discount
1	20	0.2
1	8	0.07
1	10	0.05
1	16	0.05
1	4	0.2
2	20	0.07

What are Window Functions?

A window function makes a calculation across multiple rows that are related to the current row. For example, a window function allows you to calculate.

- Running totals (i.e. sum values from all the rows before the current row)
- 7-day moving averages (i.e. average values from 7 rows before the current row)
- Rankings

Similar to an aggregate function (GROUP BY), a window function performs the operation across multiple rows. Unlike an aggregate function, a window function does not group rows into one single row.



> Syntax

Windows can be defined in the SELECT section of the query.

```
SELECT
 window_function() OVER(
      PARTITION BY partition_expression
      ORDER BY order_expression
      window_frame_extent
 ) AS window_column_alias
 FROM table_name
```

To reuse the same window with several window functions, define a named window using the WINDOW keyword. This appears in the query after the HAVING section and before the ORDER BY section.

```
SELECT
 window_function() OVER(window_name)
 FROM table_name
 [HAVING ...]
 WINDOW window_name AS (
      PARTITION BY partition_expression
      ORDER BY order_expression
      window_frame_extent
 )
 [ORDER BY ...]
```

> Order by

ORDER BY is a subclause within the OVER clause. ORDER BY changes the basis on which the function assigns numbers to rows.

It is a must-have for window functions that assign sequences to rows, including RANK and ROW_NUMBER. For example, if we ORDER BY the expression `price` on an ascending order, then the lowest-priced item will have the lowest rank.

Let's compare the following two queries which differ only in the ORDER BY clause.

```
/* Rank price from LOW->HIGH */
SELECT
    product_name,
    list_price,
    RANK() OVER
       (ORDER BY list_price DESC) rank
FROM products
```

product_name	list_price	rank
Strider Classic 12 Balance Bike - 2018	89.99	1
Sun Bicycles Lil Kitt'n - 2017	109.99	2
Trek Boy's Kickster - 2015/2017	149.99	3

```
/* Rank price from HIGH->LOW */
SELECT
    product_name,
    list_price,
    RANK() OVER
       (ORDER BY list_price ASC) rank
FROM products
```

product_name	list_price	rank
Trek Domane SLR 9 Disc - 2018	11999.99	1
Trek Domane SLR 8 Disc - 2018	7499.99	2
Trek Domane SL Frameset - 2018	6499.99	3

> Partition by

We can use PARTITION BY together with OVER to specify the column over which the aggregation is performed.

Comparing PARTITION BY with GROUP BY, we find the following similarity and difference:

- Just like GROUP BY, the OVER subclause splits the rows into as many partitions as there are unique values in a column.
- However, while the result of a GROUP BY aggregates all rows, the result of a window function using PARTITION BY aggregates each partition independently. Without the PARTITION BY clause, the result set is one single partition.

For example, using GROUP BY, we can calculate the average price of bicycles per model year using the following query.

```
SELECT
  model_year,
  AVG(list_price) avg_price
FROM products
GROUP BY model_year
```

model_year	avg_price
2016	980.29923
2017	1279.931176
2018	1658.470441
2019	2583.323333

What if we want to compare each product's price with the average price of that year? To do that, we use the AVG() window function and PARTITION BY the model year, as such.

```
SELECT
model_year,
product_name,
list_price,
AVG(list_price) OVER
  (PARTITION BY model_year)
avg_price
FROM products
```

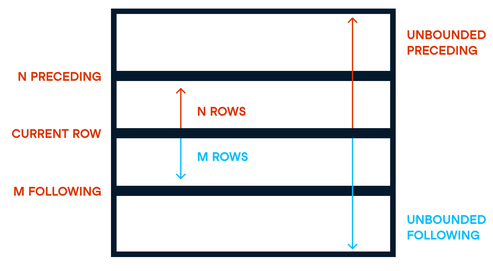
model_year	product_name	list_price	avg_price
2018	Electra Amsterdam Fashion 3i Ladies' - 2017/2018	899.99	1658.470441
2017	Electra Amsterdam Fashion 7i Ladies' - 2017	1099.99	1279.931176
2017	Electra Amsterdam Original 3i - 2015/2017	659.99	1279.931176

Notice how the avg_price of 2018 is exactly the same whether we use the PARTITION BY clause or the GROUP BY clause.

> Window frame extent

A window frame is the selected set of rows in the partition over which aggregation will occur. Put simply, they are a set of rows that are somehow related to the current row.

A window frame is defined by a lower bound and an upper bound relative to the current row. The lowest possible bound is the first row, which is known as UNBOUNDED PRECEDING. The highest possible bound is the last row, which is known as UNBOUNDED FOLLOWING. For example, if we only want to get 5 rows before the current row, then we will specify the range using 5 PRECEDING.



> Accompanying Material

You can use this https://bit.ly/3scZtOK to run any of the queries explained in this cheat sheet.

datacamp

SQL for Data Science

SQL Window Functions

Learn SQL online at www.DataCamp.com

> Ranking window functions

There are several window functions for assigning rankings to rows. Each of these functions requires an ORDER BY sub-clause within the OVER clause.

The following are the ranking window functions and their description:

Function Syntax	Function Description	Additional notes
ROW_NUMBER()	Assigns a sequential integer to each row within the partition of a result set.	Row numbers are not repeated within each partition.
RANK()	Assigns a rank number to each row in a partition.	• Tied values are given the same rank. • The next rankings are skipped.
PERCENT_RANK()	Assigns the rank number of each row in a partition as a percentage.	• Tied values are given the same rank. • Computed as the fraction of rows less than the current row, i.e., the rank of row divided by the largest rank in the partition.
NTILE(n_buckets)	Distributes the rows of a partition into a specified number of buckets.	• For example, if we perform the window function NTILE(5) on a table with 100 rows, they will be in bucket 1, rows 21 to 40 in bucket 2, rows 41 to 60 in bucket 3, et cetera.
CUME_DIST()	The cumulative distribution: the percentage of rows less than or equal to the current row.	• It returns a value larger than 0 and at most 1. • Tied values are given the same cumulative distribution value.

We can use these functions to rank the product according to their prices.

```
   /* Rank all products by price */
SELECT
   product_name,
   list_price,
   ROW_NUMBER() OVER (ORDER BY list_price) AS row_num,
   DENSE_RANK() OVER (ORDER BY list_price) AS dense_rank,
   RANK() OVER (ORDER BY list_price) AS rank,
   PERCENT_RANK() OVER (ORDER BY list_price) AS pct_rank,
   NTILE(75) OVER (ORDER BY list_price) AS ntile,
   CUME_DIST() OVER (ORDER BY list_price) AS cume_dist
FROM products
```

product_name	list_price	row_num	dense_rank	rank	pct_rank	ntile	cume_dist
Strider Classic 12 Balance Bike - 2018	89.99	1	1	1	0	1	0.0031152648
Sun Bicycles Lil Kitt'n - 2017	109.99	2	2	2	0.003125	1	0.0062305296
Trek Boy's Kickster - 2015/2017	149.99	3	3	3	0.00625	1	0.0124610592
Trek Girl's Kickster - 2017	149.99	4	3	3	0.00625	1	0.0124610592
Trek Kickster - 2018	159.99	5	4	5	0.0125	1	0.015576324
Trek Precaliber 12 Boys - 2017	189.99	6	5	6	0.015625	2	0.0218068536
Trek Precaliber 12 Girls - 2017	189.99	7	5	6	0.015625	2	0.0218068536

> Value window functions

FIRST_VALUE() and LAST_VALUE() retrieve the first and last value respectively from an ordered list of rows, where the order is defined by ORDER BY.

Value window function	Function
FIRST_VALUE(value_to_return) OVER (ORDER BY value_to_order_by)	Returns the first value in an ordered set of values
LAST_VALUE(value_to_return) OVER (ORDER BY value_to_order_by)	Returns the last value in an ordered set of values
NTH_VALUE(value_to_return, n) OVER (ORDER BY value_to_order_by)	Returns the nth value in an ordered set of values.

To compare the price of a particular bicycle model with the cheapest (or most expensive) alternative, we can use the FIRST_VALUE (or LAST_VALUE).

```
   /* Find the difference in price from
the cheapest alternative */
SELECT
  product_name,
  list_price,
  FIRST_VALUE(list_price) OVER (
     ORDER BY list_price
     ROWS BETWEEN
         UNBOUNDED PRECEDING
         AND
         UNBOUNDED FOLLOWING
     ) AS cheapest_price,
FROM products
```

product_name	list_price	cheapest_price	diff
Strider Classic 12 Balance Bike - 2018	89.99	89.99	0
Sun Bicycles Lil Kitt'n - 2017	109.99	89.99	20
Trek Boy's Kickster - 2015/2017	149.99	89.99	60
Trek Girl's Kickster - 2017	149.99	89.99	60

```
   /* Find the difference in price from
the priciest alternative */
SELECT
   product_name,
   list_price,
   LAST_VALUE(list_price) OVER (
     ORDER BY list_price
     ROWS BETWEEN
        UNBOUNDED PRECEDING
        AND
        UNBOUNDED FOLLOWING
     ) AS highest_price
FROM products
```

product_name	list_price	highest_price	diff
Strider Classic 12 Balance Bike - 2018	89.99	11999.99	11910
Sun Bicycles Lil Kitt'n - 2017	109.99	11999.99	11890
Trek Boy's Kickster - 2015/2017	149.99	11999.99	11850

> Aggregate window functions

Aggregate functions available for GROUP BY, such as COUNT(), MIN(), MAX(), SUM(), and AVG() are also available as window functions.

Function Syntax	Function Description
COUNT(expression) OVER (PARTITION BY partition_column)	Count the number of rows that have a non-null expression in the partition.
MIN(expression) OVER (PARTITION BY partition_column)	Find the minimum of the expression in the partition.
MAX(expression) OVER (PARTITION BY partition_column)	Find the maximum of the expression in the partition.
AVG(expression) OVER (PARTITION BY partition_column)	Find the mean (average) of the expression in the partition.

Suppose we want to find the average, maximum and minimum discount for each product, we can achieve it as such.

```
SELECT
   order_id,
   product_id,
   discount,
   AVG(discount) OVER (PARTITION BY product_id) AS avg_discount,
   MIN(discount) OVER (PARTITION BY product_id) AS min_discount,
   MAX(discount) OVER (PARTITION BY product_id) AS max_discount
FROM order_items
```

order_id	product_id	discount	avg_discount	min_discount	max_discount
2	16	0.05	0.113191	0.05	0.2
2	20	0.07	0.11253	0.05	0.2
3	20	0.05	0.11253	0.05	0.2
3	3	0.05	0.105581	0.05	0.2

> LEAD, LAG

The LEAD and LAG locate a row relative to the current row.

Function Syntax	Function Description
LEAD(expression [,offset[,default_value]]) OVER(ORDER BY columns)	Accesses the value stored in a row after the current row.
LAG(expression [,offset[,default_value]]) OVER(ORDER BY columns)	Accesses the value stored in a row before the current row.

Both LEAD and LAG take three arguments:

- Expression: the name of the column from which the value is retrieved
- Offset: the number of rows to skip. Defaults to 1.
- Default_value: the value to be returned if the value retrieved is null. Defaults to NULL.

With LAG and LEAD, you must specify ORDER BY in the OVER clause.

LEAD and LAG are most commonly used to find the value of a previous row or the next row. For example, they are useful for calculating the year-on-year increase of business metrics like revenue.

Here is an example of using lag to compare this year's sales to last year's.

```
   /* Find the number of orders in a year */
WITH yearly_orders AS (
   SELECT
 year(order_date) AS year,
 COUNT(DISTINCT order_id) AS num_orders
 FROM sales.orders
 GROUP BY year(order_date)
)

/* Compare this year's sales to last year's */
SELECT
   *,
   LAG(num_orders) OVER (ORDER BY year) last_year_order,
   LAG(num_orders) OVER (ORDER BY year) - num_orders diff_from_last_year
FROM yearly_orders
```

year	num_orders	last_year_order	diff_from_last_year
2016	635	null	null
2017	688	635	-53
2018	292	688	396

Similarly, we can make a comparison of each year's order with the next year's.

```
   /* Find the number of orders in a year */
WITH yearly_orders AS (
   SELECT
 year(order_date) AS year,
 COUNT(DISTINCT order_id) AS num_orders
 FROM sales.orders
 GROUP BY year(order_date)
)

/* Compare the number of years compared to next year */
SELECT *,
   LEAD(num_orders) OVER (ORDER BY year) next_year_order,
   LEAD(num_orders) OVER (ORDER BY year) - num_orders diff_from_next_year
FROM yearly_orders
```

year	num_orders	next_year_order	diff_from_next_year
2016	635	688	53
2017	688	292	-396
2018	292	null	null

The Ultimate Guide to SQL Window Functions

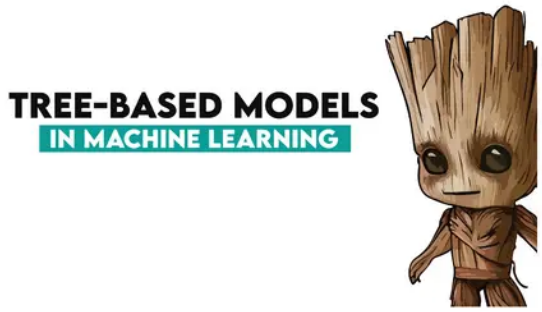
Categories SQL Interviews Guides

Written by: Nathan Rosidi

Author Bio

October 26th, 2023

Latest Posts:



Modeling Guides

Tree-Based Models in Machine Learning

Interviews

Top Data Warehouse Interview Questions with Answers

Guides

Data Cleaning 101: Avoid These 5 Traps in Your Data

Main Topics

Share

Follow

This ultimate guide is a complete overview of the types of SQL window functions, their syntax and real-life examples of how to use them in queries.

Because of how efficient SQL window functions are compared to standard SQL functions and operators, interviewers at most of the companies ask questions about them and expect the candidates to use them in their solutions. The SQL window functions are also frequently used in everyday work by data scientists and data analysts, especially when querying particularly large datasets. Their usage can not only make the SQL code faster but also clearer and easier to understand by others. However, because of the complicated syntax, arguably less intuitive than with other SQL constructions, it takes practice to master using SQL window functions in queries.

Even though SQL was first formalized in 1986, the window functions in SQL weren't added until 2003. Formally speaking, window functions use values from multiple rows to produce values for each row separately. What distinguishes window from other SQL functions, namely aggregate and scalar functions, is the keyword OVER, used to define a portion of rows the function should consider as inputs to produce an output. This portion of rows is called a 'window', hence the name of this family of functions.

This guide starts by introducing the 3 main types of SQL window functions, namely:

- Aggregate window functions;
- Ranking window functions;
- Value window functions.

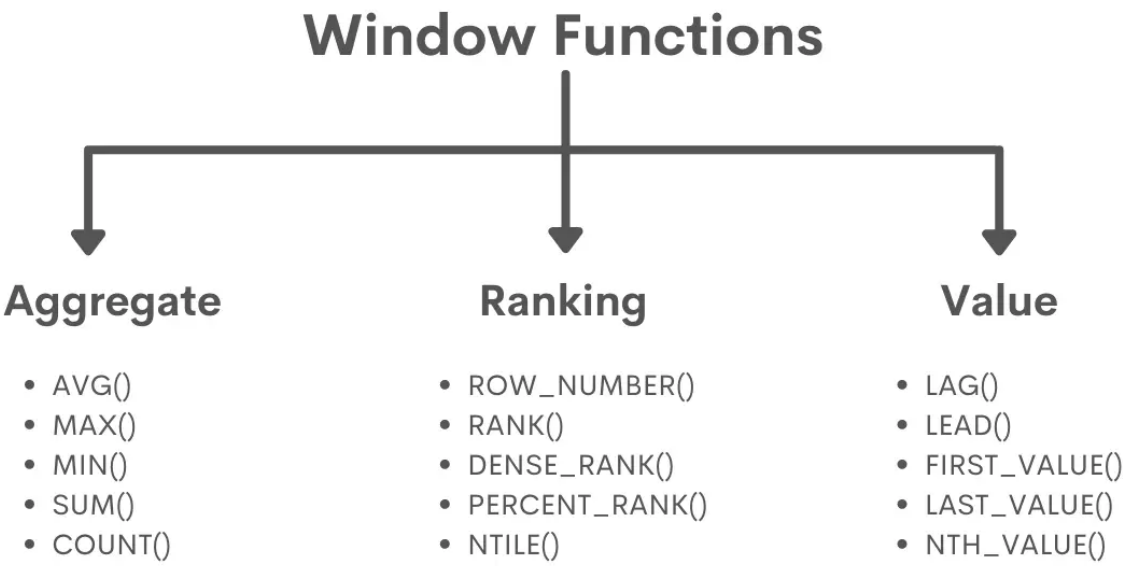
It then lists the specific functions together with examples of their usage. After that, the guide describes some more advanced syntax concepts regarding SQL window functions such as using the EXCLUDE and FILTER clauses, frame boundaries and window chaining.

List of SQL Window Functions

There is no official division of the SQL window functions into categories but nevertheless, they are frequently being divided into two or three types. The most basic classification splits the SQL window functions into aggregate and built-in functions.

The main characteristic of the aggregate window functions is that they reuse the existing simple aggregate functions (such as COUNT() or SUM()) while changing the way in which the aggregation is defined and the format of the results. Meanwhile, the built-in functions have new names and cannot be used in another context.

The built-in SQL window functions are then sometimes divided into two different types: the ranking and value functions. The ranking window functions are used to simply assign numbers to the existing rows according to some requirements. On the other hand, the value window functions are used to copy values from other rows to other rows within the defined windows.



Aggregate Window Functions in SQL

As mentioned, the aggregate window functions are exactly the same as standard aggregate functions, with the most popular being AVG(), MAX(), MIN(), SUM() and COUNT(). When used in normal circumstances, these functions either apply to the entire dataset (e.g. AVG() return the mean from all the values in a column) or go pair-in-pair with a GROUP BY statement so that the function is applied to subsets or rows, depending on another variable.

As it turns out, the GROUP BY statement can be replaced by using an aggregate window function instead and specifying the aggregation conditions in the OVER clause instead. The main difference is that while a standard aggregate function reduces the number of rows to match the number of categories to which the data are aggregated, the window function does not change the number of rows, instead assigns the correct value to each row of the dataset, even if these values are the same.

To better understand the difference and the syntax of the SQL window functions, let's consider this interview question from Twitch:

Session Type Duration

Interview Question Date: February 2021

Twitch Easy ID 2011

Data Engineer Data Scientist BI Analyst Data Analyst ML Engineer

Calculate the average session duration for each session type?

Table: twitch_sessions

Link to the question: https://platform.stratascratch.com/coding/2011-session-type-duration

It comes with a simple table containing data about streaming sessions of various users with the session_type being either 'streamer' or 'viewer'.

This question can be solved rather easily by taking the average of the differences between session end and start times, and then aggregating the results by session type which can be achieved with a GROUP BY statement.

```
SELECT session_type,
       avg(session_end - session_start) AS duration
FROM twitch_sessions
GROUP BY session_type
```

Go to the question on the platform

PostgreSQL

Tables: twitch_sessions

```
SELECT session_type,
       avg(session_end - session_start) AS duration
FROM twitch_sessions
GROUP BY session_type
```

Reset | Run Code | Check Solution

Use ⌥ + Enter to run query

This solution reduces the number of rows to match the number of distinct session types. Each row is then assigned the correct average value.

All required columns and the first 5 rows of the solution are shown

session_type	duration
streamer	411
viewer	986

But since normal aggregate functions, such as the AVG() in this case, can be replaced with window aggregate functions, let's explore the alternative solution to this interview question. To construct a window function, the whole expression avg(session_end - session_start) can be left unchanged. To this, the keyword OVER can be added that defines the window function.

```
SELECT *,
       avg(session_end -session_start) OVER () AS duration
FROM twitch_sessions
```

All required columns and the first 5 rows of the solution are shown

user_id	session_start	session_end	session_id	session_type	duration
0	2020-08-11 05:51:31	2020-08-11 05:54:45	539	streamer	698.5
2	2020-07-11 03:36:54	2020-07-11 03:37:08	840	streamer	698.5
3	2020-11-26 11:41:47	2020-11-26 11:52:01	848	streamer	698.5
1	2020-11-19 06:24:24	2020-11-19 07:24:38	515	viewer	698.5
2	2020-11-14 03:36:05	2020-11-14 03:39:19	646	viewer	698.5

Note the lack of GROUP BY clause in the query and how the window function does not reduce the number of rows, instead adds the average value to each row. The solution from above is still far from correct because the value added to each row is 698 - it is the average value between the session_end and session_start but for the entire dataset.

The next step therefore will be to define in what way the rows should be aggregated. This can be done within the OVER() clause that for now is empty. The expression will be identical as the GROUP BY statement, with the exception that another keyword, PARTITION BY, needs to be used within the OVER clause. The PARTITION BY clause is the equivalent to GROUP BY in the SQL window functions and allows to aggregate the results by another variable. Hence, the AVG() window function will look as follows:

```
SELECT *,
       avg(session_end -session_start) OVER (PARTITION BY session_type)
       AS duration
FROM twitch_sessions
```

All required columns and the first 5 rows of the solution are shown

user_id	session_start	session_end	session_id	session_type	duration
0	2020-08-11 05:51:31	2020-08-11 05:54:45	539	streamer	411
2	2020-07-11 03:36:54	2020-07-11 03:37:08	840	streamer	411
3	2020-11-26 11:41:47	2020-11-26 11:52:01	848	streamer	411
0	2020-03-11 03:01:40	2020-03-11 03:01:59	782	streamer	411
1	2020-11-20 06:59:57	2020-11-20 07:20:11	907	streamer	411

Note the lack of GROUP BY clause in the query and how the window function does not reduce the number of rows. Instead, it adds the correct average, different for ‘streamer’ sessions and ‘viewer’ sessions, to each row. However, the interview question asks to return an average value for each session type, thus this format of results isn’t the best for this purpose. To generate the correct output for this question, only two columns need to be selected in combination with the DISTINCT keyword to still reduce the number of rows.

```
SELECT DISTINCT session_type,
       avg(session_end -session_start) OVER (PARTITION BY session_type)
       AS duration
FROM twitch_sessions
```

All required columns and the first 5 rows of the solution are shown

session_type	duration
viewer	986
streamer	411

The same approach can be used with any aggregate functions. Let's now consider this interview question from Microsoft.

Finding Updated Records

Interview Question Date: November 2020

Microsoft Easy ID 10299

Data Engineer Data Scientist BI Analyst Data Analyst ML Engineer

We have a table with employees and their salaries, however, some of the records are old and contain outdated salary information. Find the current salary of each employee assuming that salaries increase each year. Output their id, first name, last name, department ID, and current salary. Order your list by employee ID in ascending order.

Table: ms_employee_salary

Link to the question: https://platform.stratascratch.com/coding/10299-finding-updated-records

In other words, the task is to find a maximum salary for each employee. The standard solution to this question looks as follows:

```
SELECT id,
       first_name,
       last_name,
       department_id,
       max(salary)
FROM ms_employee_salary
GROUP BY id,
         first_name,
         last_name,
         department_id
```

Go to the question on the platform

PostgreSQL **Tables:** ms_employee_salary

```
1  SELECT id,
2         first_name,
3         last_name,
4         department_id,
5         max(salary)
6  FROM ms_employee_salary
7  GROUP BY id,
8           first_name,
```

```
9         last_name,
10        department_id
```

Reset | Run Code | Check Solution

Use ⌥ + Enter to run query

This solution uses the MAX() function and involves aggregation by four variables. This can all still be achieved using the window function.

```
SELECT DISTINCT id,
       first_name,
       last_name,
       department_id,
       max(salary) OVER(PARTITION BY id, first_name, last_name,
       department_id)
FROM ms_employee_salary
```

However, in many cases when using the aggregate functions, the feature of the GROUP BY clause automatically reducing the number of rows and showing a simple summary of aggregated results becomes useful. So in which situation does the fact the window functions do not change the number of rows come in handy? The answer is: when the individual values need to be compared to aggregated values.

Let's consider this interview question from Salesforce on average salaries.

Average Salaries

Interview Question Date: May 2019

Glassdoor Salesforce Easy ID 9917

Data Engineer Data Scientist BI Analyst Data Analyst ML Engineer

Compare each employee's salary with the average salary of the corresponding department.
Output the department, first name, and salary of employees along with the average salary of that department.

Table: employee

Link to the question: https://platform.stratascratch.com/coding/9917-average-salaries

When using a standard approach with the GROUP BY clause, these questions would require two separate queries and wouldn't be efficient. Instead, the window function can solve it easily:

```
SELECT department,
       first_name,
       salary,
       AVG(salary) over (PARTITION BY department)
FROM employee;
```

Go to the question on the platform

PostgreSQL

Tables: employee

```
SELECT department,
       first_name,
       salary,
       AVG(salary) over (PARTITION BY department)
FROM employee;
```

Reset | Run Code | Check Solution

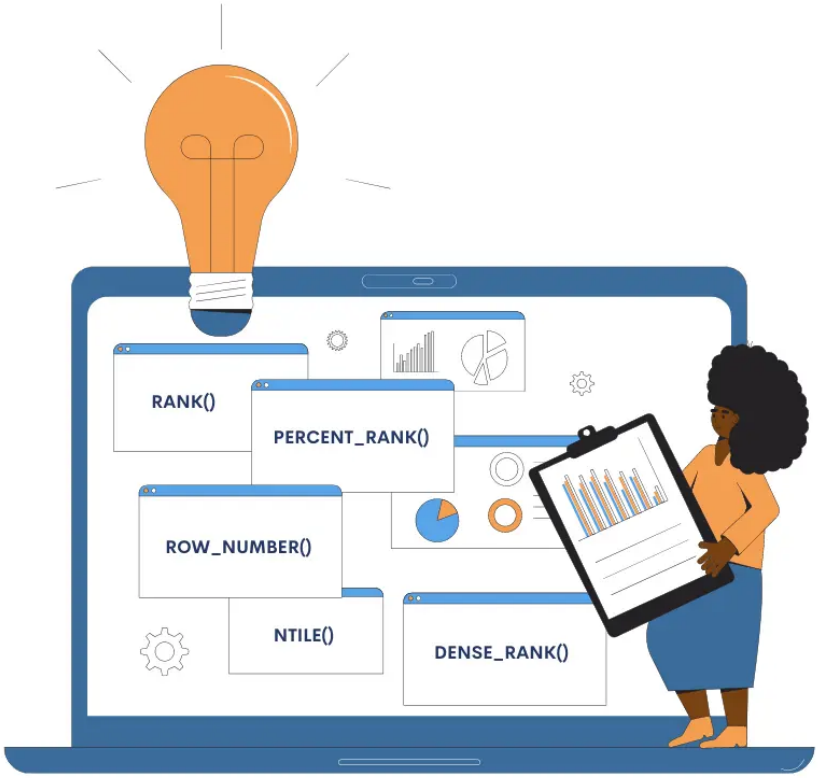
Use ⌥ + Enter to run query

All required columns and the first 5 rows of the solution are shown

department	first_name	salary	avg
Audit	Michale	700	950
Audit	Shandler	1100	950
Audit	Jason	1000	950
Audit	Celine	1000	950
Management	Allen	200000	175000

Now, in case the next task was to calculate the difference between each employee's salary and the department's average, it can easily be done, even within the same query. This is the situation when aggregate window functions prove especially useful.

Ranking Window functions in SQL



The ranking window functions are used to assign numbers to rows depending on some defined ordering. Unlike the aggregate window functions, the ranking functions have no obvious equivalents that don't use windows. What's more, attempting to solve for the same results but without using analytical functions would require multiple nested queries and would be largely inefficient. Therefore, the ranking window functions in SQL are the most commonly used out of the entire family. These include:

- ROW_NUMBER()
- RANK()
- DENSE_RANK()
- PERCENT_RANK()
- NTILE()

ROW_NUMBER()

The ROW_NUMBER() function is the simplest of the ranking window functions in SQL. It assigns consecutive numbers starting from 1 to all rows in the table. The order of the rows needs to be defined using an ORDER BY clause inside the OVER clause. This is, in fact, the necessary condition for all the ranking window functions: they don't require the PARTITION BY clause but the ORDER BY clause must always be there.

Let's consider this question from Google on SQL functions.

Activity Rank

Interview Question Date: July 2021

Google Medium ID 10351

Data Engineer Data Scientist BI Analyst Data Analyst ML Engineer

Find the email activity rank for each user. Email activity rank is defined by the total number of emails sent. The user with the highest number of emails sent will have a rank of 1, and so on. Output the user, total emails, and their activity rank. Order records by the total emails in descending order. Sort users with the same number of emails in alphabetical order. In your rankings, return a unique value (i.e., a unique rank) even if multiple users have the same number of emails. For tie breaker use alphabetical order of the user usernames.

Table: google_gmail_emails

Link to the question: https://platform.stratascratch.com/coding/10351-activity-rank

Because of this last sentence the task becomes simply assigning the consecutive numbers to users based on the number of sent emails.

The solution to this question may start like this, by counting the number of emails:

```
SELECT  from_user,
        COUNT(*) as total_emails
FROM google_gmail_emails
GROUP BY from_user
ORDER BY 2 DESC
```

All required columns and the first 5 rows of the solution are shown

from_user	total_emails
ef5fe98c6b9f313075	19
32ded68d89443e808	19
5b8754928306a18b68	18
91f59516cb9dee1e88	16
55e60cfcc9dc49c17e	16

Since the question instructs to assign different values to users with the same number of emails, the ranking can be assigned using the ROW_NUMBER() function. To use it, the ORDER BY statement from the previous query can simply be moved inside the OVER clause like this:

```
SELECT  from_user,
        COUNT(*) as total_emails,
        ROW_NUMBER() OVER ( ORDER BY count(*) desc, from_user asc)
FROM google_gmail_emails
GROUP BY from_user
```

Go to the question on the platform

PostgreSQL

Tables: google_gmail_emails

```
1  SELECT  from_user,
2          COUNT(*) as total_emails,
3          ROW_NUMBER() OVER ( ORDER BY count(*) desc, from_user asc)
```

```
4 FROM google_gmail_emails
5 GROUP BY from_user
```

Reset | Run Code | Check Solution

Use ⌥ + Enter to run query

All required columns and the first 5 rows of the solution are shown

from_user	total_emails	row_number
32ded68d89443e808	19	1
ef5fe98c6b9f313075	19	2
5b8754928306a18b68	18	3
55e60cfcc9dc49c17e	16	4
91f59516cb9dee1e88	16	5

As expected, the function simply assigned the consecutive numbers to the rows, without repeating or skipping any numbers, and it did it according to the defined order: it sorts the data by total_emails in descending order.

RANK()

The RANK() window function is more advanced than ROW_NUMBER() and is probably the most commonly used out of all SQL window functions. Its task is rather simple: to assign ranking values to the rows according to the specified ordering. Then, how is it different from the ROW_NUMBER() function?

The key difference between RANK() and ROW_NUMBER() is how the first one handles the ties, i.e. cases when multiple rows have the same value by which the dataset is sorted. The rank function will always assign the same value to rows with the same values and will skip some values to keep the ranking consistent with the number of preceding rows.

To visualize it, let's use the same interview question from Google using RANK() about the email activity rank but this time let's switch ROW_NUMBER() in the solution to RANK().

```
SELECT  from_user,
        COUNT(*) as total_emails,
        RANK() OVER (ORDER BY count(*) desc)
FROM google_gmail_emails
GROUP BY from_user
```

All required columns and the first 5 rows of the solution are shown

from_user	total_emails	rank
ef5fe98c6b9f313075	19	1
32ded68d89443e808	19	1
5b8754928306a18b68	18	3
91f59516cb9dee1e88	16	4
55e60cfcc9dc49c17e	16	4

The values in the third column are completely different than previously, even though the task remains similar: assigning consecutive values according to some ordering. In this case, the data are again sorted by total_emails in descending order. The different values are all the result of ties in the data. For instance, the first two users both have 19 total_emails. Therefore, when ranking them, it's impossible to tell which one is the first and which one the second unless another condition (e.g. ordering alphabetically by user ID) is specified. The RANK() function solves this issue by assigning the same value to both rows.

At the same time, the function still assigns the value 3 to the third row because there are 2 other rows before it and its total_emails is different from theirs. This rule stays valid even if there are more than two rows with the same value. Note how all users with 15 total_emails have the ranking 6 but the users with 14 total_emails suddenly get a ranking value of 10. That's because of how many users with 15 total_emails there are and because there are exactly 9 rows before the first user with 14 total_emails.

The RANK() function is especially useful when the task is to output the rows with the highest or the lowest values. This can easily be done by adding an outer query with a WHERE condition specifying that the rank value should be equal to 1. For example, the code below returns the users with the highest email activity rank. Note that it would be impossible with the ROW_NUMBER() function - it would only return one row which is not a correct answer.

```
SELECT from_user,
       total_emails
FROM
  (SELECT from_user,
          COUNT(*) AS total_emails,
          RANK() OVER (
                       ORDER BY count(*) DESC) rnk
   FROM google_gmail_emails
   GROUP BY from_user) a
WHERE rnk = 1
```

PostgreSQL

Tables: google_gmail_emails

```
SELECT from_user,
       total_emails
FROM
  (SELECT from_user,
          COUNT(*) AS total_emails,
          RANK() OVER (
                       ORDER BY count(*) DESC) rnk
   FROM google_gmail_emails
   GROUP BY from_user) a
WHERE rnk = 1
```

Reset | Run Code

Use ⌥ + Enter to run query

All required columns and the first 5 rows of the solution are shown

from_user	total_emails
ef5fe98c6b9f313075	19
32ded68d89443e808	19

DENSE_RANK()

The DENSE_RANK() function is very similar to the RANK() function with one key difference - if there are ties in the data and two rows are assigned the same ranking value, the DENSE_RANK() will not skip any numbers and will assign the consecutive value to the next row. To visualize it, here is the solution to the interview question from Google, using DENSE_RANK():

```
SELECT  from_user,
        COUNT(*) as total_emails,
        DENSE_RANK() OVER (ORDER BY count(*) desc)
FROM google_gmail_emails
GROUP BY from_user
```

All required columns and the first 5 rows of the solution are shown

from_user	total_emails	dense_rank
ef5fe98c6b9f313075	19	1
32ded68d89443e808	19	1
5b8754928306a18b68	18	2
91f59516cb9dee1e88	16	3
55e60cfcc9dc49c17e	16	3

Since there are two users with 19 total_emails, they are assigned the same ranking value because this is still a ranking function. However, while the RANK() function would assign a value of 3 to the third row, DENSE_RANK() assigns the ranking of 2 because it does not skip values and the number 2 hasn't been used before.

The DENSE_RANK() is most useful when solving tasks in which several highest values need to be shown. For example, if the assignment is to output the users with top 2 total_email counts, the DENSE_RANK() can be used because it ensures that the users with the second highest total_emails will be assigned the ranking value of 2.

```
SELECT from_user,
       total_emails
```

```
FROM
  (SELECT from_user,
          COUNT(*) AS total_emails,
          DENSE_RANK() OVER (
                      ORDER BY count(*) DESC) rnk
   FROM google_gmail_emails
   GROUP BY from_user) a
WHERE rnk <= 2
```

PostgreSQL

Tables: google_gmail_emails

```
SELECT from_user,
       total_emails
FROM
  (SELECT from_user,
          COUNT(*) AS total_emails,
          DENSE_RANK() OVER (
                      ORDER BY count(*) DESC) rnk
   FROM google_gmail_emails
   GROUP BY from_user) a
WHERE rnk <= 2
```

Reset **Run Code**

Use ⌥ + Enter to run query

All required columns and the first 5 rows of the solution are shown

from_user	total_emails
ef5fe98c6b9f313075	19
32ded68d89443e808	19
5b8754928306a18b68	18

The rule of thumb is to use RANK() when the task is to output rows with the highest or lowest ranking values and DENSE_RANK() when there is a need to output rows with several ranking values. But when the question is to simply add a ranking to data, it is not clear which function should be used but the more obvious choice is the RANK() function since it is considered the default one. However, if an ambiguity like that appears at an interview, it is best to ask an interviewer for a clarification. What's more, some questions use certain keywords to point to one of the functions. For instance, a sentence 'Avoid gaps in the ranking calculation', like in this question from Deloitte, suggests using a DENSE_RANK() function.

PERCENT_RANK()

The PERCENT_RANK() function is another one from the ranking functions family that in reality uses a RANK() function to calculate the final ranking. The ranking values in case of PERCENT_RANK are calculated using the following formula: *(rank - 1)/(rows - 1)*. Because of this, all the ranking values are scaled by the number of rows and stay between 0 and 1. Additionally, the rows with the first value are always assigned the ranking value of 0.

When applied to the interview question from Google, the PERCENT_RANK() results in the following ranking. Note that this table has 25 rows and while not all of them are shown, the ranking continues all the way to 1. This also explains the ranking values -

for instance, the third row is assigned RANK() of 3 and since there are 25 rows, the formula becomes (3-1)/(25-1) = 2/24 = 0.083.

```
SELECT  from_user,
        COUNT(*) as total_emails,
        PERCENT_RANK() OVER (ORDER BY count(*) desc)
FROM google_gmail_emails
GROUP BY from_user
```

PostgreSQL

Tables: google_gmail_emails

```
1  SELECT  from_user,
2          COUNT(*) as total_emails,
3          PERCENT_RANK() OVER (ORDER BY count(*) desc)
4  FROM google_gmail_emails
5  GROUP BY from_user
```

Reset | Run Code

Use ⌥ + Enter to run query

All required columns and the first 5 rows of the solution are shown

from_user	total_emails	percent_rank
ef5fe98c6b9f313075	19	0
32ded68d89443e808	19	0
5b8754928306a18b68	18	0.083
91f59516cb9dee1e88	16	0.125
55e60cfcc9dc49c17e	16	0.125

NTILE()

The NTILE() is the last major ranking window function in SQL but it is not very commonly used. In short, it works analogically to the ROW_NUMBER function but instead of assigning consecutive numbers to the next rows, it assigns consecutive numbers to the buckets of rows. The bucket is a collection of several consecutive rows and the number of buckets is set as a parameter of the NTILE() function - for example, NTILE(10) means that the dataset will be divided into 10 buckets.

When NTILE(10) is applied to the interview question from Google, it divides the dataset into buckets. The table has 25 rows, so each bucket should have 2,5 rows but since the bucket size needs to be an integer, this value gets rounded up to 3. Because of that each set of 3 rows will be given the next number from 1 to 10 (because of 10 buckets).

```
SELECT  from_user,
        COUNT(*) as total_emails,
        NTILE(10) OVER (ORDER BY count(*) desc)
FROM google_gmail_emails
GROUP BY from_user
```

PostgreSQL

Tables: google_gmail_emails

```
1  SELECT  from_user,
2          COUNT(*) as total_emails,
3          NTILE(10) OVER (ORDER BY count(*) desc)
4  FROM google_gmail_emails
5  GROUP BY from_user
```

Reset | Run Code

Use ⌥ + Enter to run query

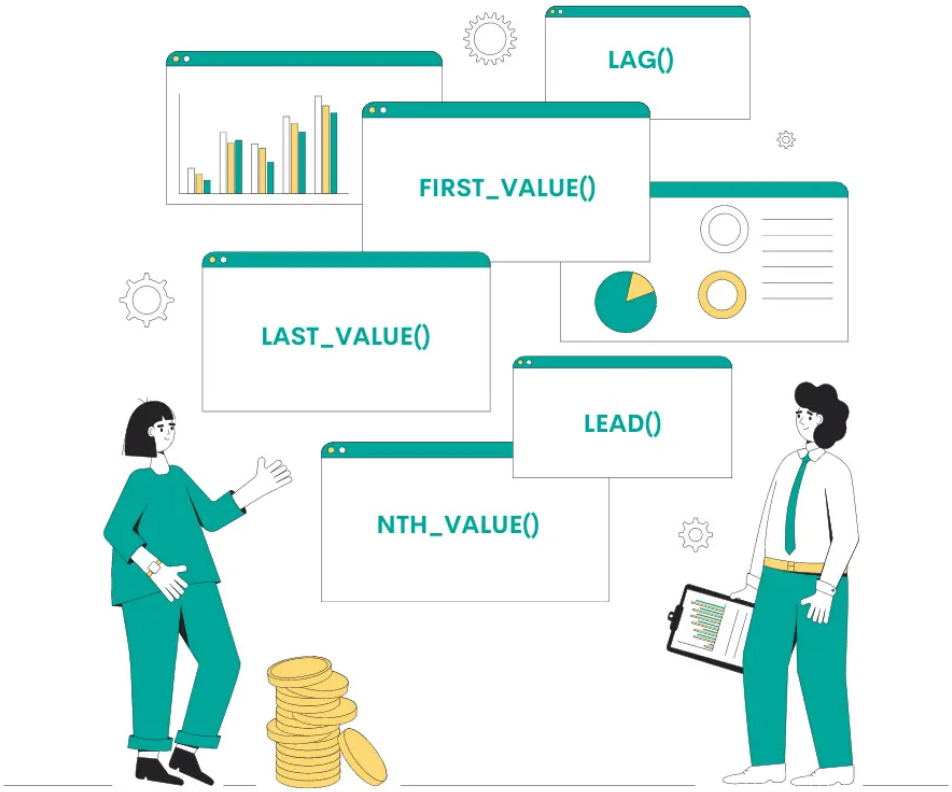
All required columns and the first 5 rows of the solution are shown

from_user	total_emails	ntile
ef5fe98c6b9f313075	19	1
32ded68d89443e808	19	1
5b8754928306a18b68	18	1
91f59516cb9dee1e88	16	2
55e60cfcc9dc49c17e	16	2

Since there are 25 rows and they are divided into buckets of size 3, the last bucket, the one assigned the value of 10, will only have 2 rows. What's more the number of buckets should always be less than the number of rows in the table, otherwise the number of rows per bucket would need to be less than 1 and since that's not possible, each bucket would have exactly 1 row and the result would be the same as with using the ROW_NUMBER() function.

The final remark regarding NTILE() is that it should not be used to calculate percentiles. That's because it only splits the results in the equal-sized buckets and it may happen, as in the example, that rows with different values will be assigned the same ranking. Because of that, when dealing with percentiles, it is advisable to use PERCENT_RANK() instead.

Value Window Functions in SQL



The value window functions in SQL are used to assign to rows values from other rows. Similarly to the ranking window functions and unlike the aggregate functions, the value functions have no obvious equivalents that don't use windows. However, it is usually possible to replicate the results of these functions using two nested queries, hence, the value window functions are not that commonly used as the ranking functions. There are following value window functions in SQL:

- LAG()
- LEAD()
- FIRST_VALUE()
- LAST_VALUE()
- NTH_VALUE()

LAG()

The LAG() function is by far the most popular out of the value window functions but at the same time is rather simple. What it does is, it assigns to each row a value that normally belongs to the previous row. In other words, it allows to shift any column by one row down and allows to perform queries using this shift of values. Naturally, the ordering of the rows matters also in this case, hence, the window function will most commonly include the ORDER BY clause within its OVER(clause.

To give an example of how the LAG() function can be used, let's consider this interview question from the City of San Francisco. It asks:

Daily Violation Counts

Interview Question Date: May 2018

City of San Francisco Medium ID 9740

Data Engineer Data Scientist BI Analyst Data Analyst ML Engineer Software Engineer

Determine the change in the number of daily violations by calculating the difference between the count of current and previous violations by inspection date.
Output the inspection date and the change in the number of daily violations. Order your results by the earliest inspection date first.

Table: sf_restaurant_health_violations

Link to the question: https://platform.stratascratch.com/coding/9740-daily-violation-counts

To answer this interview question, there are three details that need to be extracted from the original dataset: the date, the number of violations for each date, and for each date, the number of violations that were detected on the previous day. The first two are rather simple to extract with the following query:

```
SELECT inspection_date::DATE,
       COUNT(violation_id)
FROM sf_restaurant_health_violations
GROUP BY 1
```

The other task is to get the number of violations that were detected on the previous day. To achieve this, the results of the query above need to be sorted by date, so that the previous inspection date is always one row above, and the column with the COUNT() values needs to be shifted by one row down. This is what the LAG() function has been created to solve.

```
SELECT inspection_date::DATE,
       COUNT(violation_id),
       LAG(COUNT(violation_id)) OVER(ORDER BY inspection_date::DATE)
FROM sf_restaurant_health_violations
GROUP BY 1
```

All required columns and the first 5 rows of the solution are shown

inspection_date	count	lag
2015-09-08	1	
2015-09-15	0	1
2015-09-18	0	0
2015-09-23	0	0
2015-09-28	1	0

As it can be seen, the LAG() function easily shifted the values from the second column by one row. For instance, the second row means that on September 15th 2015, there were 0 violations (second column) but on the previous inspection date (September 8th), there was 1 violation (third column). Note that because of the specificity of this dataset, not all dates exist in the table because inspections do not occur every day.

Another interesting detail to realise is that when using the LAG() function, the first row in the dataset or in a partition to which the function is applied, will always be assigned the NULL value. This may cause issues in some cases and is something that the solution should account for.

To complete the solution for this example interview question, one more column needs to be added, namely, the difference between the count of current and previous violations by inspection date. This can be achieved either in an outer query or even in the same query by subtracting the value in the third column from the value in the second column.

We'll remove two columns from the previous step, as they are not required by the question.

```
SELECT inspection_date::DATE,
       COUNT(violation_id) - LAG(COUNT(violation_id)) OVER(
                                    ORDER BY inspection_date::DATE)
                                              diff
FROM sf_restaurant_health_violations
GROUP BY 1;
```

Go to the question on the platform

PostgreSQL

Tables: sf_restaurant_health_violations

```
SELECT inspection_date::DATE,
       COUNT(violation_id) - LAG(COUNT(violation_id)) OVER(
                                        ORDER BY inspection_date::DATE)
                              diff
FROM sf_restaurant_health_violations
GROUP BY 1;
```

Reset | Run Code | Check Solution

Use ⌥ + Enter to run query

All required columns and the first 5 rows of the solution are shown

inspection_date	diff
2015-09-08	
2015-09-15	-1
2015-09-18	0
2015-09-23	0
2015-09-28	1

LEAD()

The LEAD() window function is the exact opposite of the LAG() function because while LAG() returns for each row the value of the previous row, the LEAD() function will return the value of the following row. In other words, as LAG() shifts the values 1 row down, LEAD() shifts them 1 row up. Otherwise, the functions are identical in how they are called or how the order is defined.

If the interview question from the above asked to calculate the difference between the count of current and next violations by inspection date, it could be solved using the LEAD() function as follows:

```
SELECT inspection_date::DATE,
       COUNT(violation_id),
       LEAD(COUNT(violation_id)) OVER(
                                   ORDER BY inspection_date::DATE),
       COUNT(violation_id) - LEAD(COUNT(violation_id)) OVER(
                                   ORDER BY inspection_date::DATE)
                                            diff
FROM sf_restaurant_health_violations
GROUP BY 1
```

PostgreSQL

Tables: sf_restaurant_health_violations

```
1  SELECT inspection_date::DATE,
2         COUNT(violation_id),
3         LEAD(COUNT(violation_id)) OVER(
4                                     ORDER BY inspection_date::DATE),
5         COUNT(violation_id) - LEAD(COUNT(violation_id)) OVER(
6                                     ORDER BY inspection_date::DATE)
7                              diff
8  FROM sf_restaurant_health_violations
9  GROUP BY 1
```

Reset | Run Code

Use ⌥ + Enter to run query

All required columns and the first 5 rows of the solution are shown

inspection_date	count	lead	diff
2015-09-08	1	0	1
2015-09-15	0	0	0
2015-09-18	0	0	0
2015-09-23	0	1	-1
2015-09-28	1	5	-4

In this case, the first row of the table is assigned a value because there exists a row that comes right after that. But at the same time, in case of LEAD() function, the last row of the table or a partition to which the function is applied is always assigned the NULL value because there is no row afterwards.

FIRST_VALUE()

The FIRST_VALUE() function is not that commonly used but is also a rather interesting value window function in SQL. It does exactly what its name suggests - for all the rows, it assigns the first value of the table or the partition to which it is applied, according to some ordering that determines which row comes as a first one. Moreover, the variable from which the first value should be returned needs to be defined as the parameter of the function.

This function is not frequently appearing in interview questions, hence, to illustrate it with the example, let's consider a question from Microsoft which uses FIRST_VALUE() and change it slightly.

Unique Users Per Client Per Month

Interview Question Date: March 2021

Apple Dell Microsoft Easy ID 2024

Data Engineer Data Scientist BI Analyst Data Analyst ML Engineer

Write a query that returns the number of unique users per client per month
Table: fact_events

Link to the question: https://platform.stratascratch.com/coding/2024-unique-users-per-client-per-month

The original question asks to write a query that returns the number of unique users per client per month. But let's assume that there is another task to calculate the difference of unique users of each client between the current month and the first month when the data is available.

The first part of this question can easily be solved with the following query:

```
SELECT client_id,
       EXTRACT(month from time_id) as month,
       count(DISTINCT user_id) as users_num
FROM fact_events
GROUP BY 1,2
```

Go to the question on the platform

PostgreSQL

Tables: fact_events

```
SELECT client_id,
       EXTRACT(month from time_id) as month,
       count(DISTINCT user_id) as users_num
FROM fact_events
GROUP BY 1,2
```

Reset Run Code Check Solution

Use ⌥ + Enter to run query

All required columns and the first 5 rows of the solution are shown

client_id	month	users_num
desktop	2	13
desktop	3	16
desktop	4	11
mobile	2	9
mobile	3	14

There are three months of data and two clients with 3 users each. To solve the second part of the question, the task will be to subtract the value in each month from the value for February of each account. For instance, for the client 'desktop', the value for February which is 13 will be subtracted from values for February (13), March (16) and April (11). Same for the other client. This can be computed using two queries: the inner one outputting the value for the first month and the outer query performing the calculation. But it can also be solved easier using the FIRST_VALUE() function. When the function is applied to the entire dataset, it looks as follows:

The second SQL code snapshot stays the same as in the article. But use the same code to show the output below it, which is not the case currently. Here's the code to use in the widget:

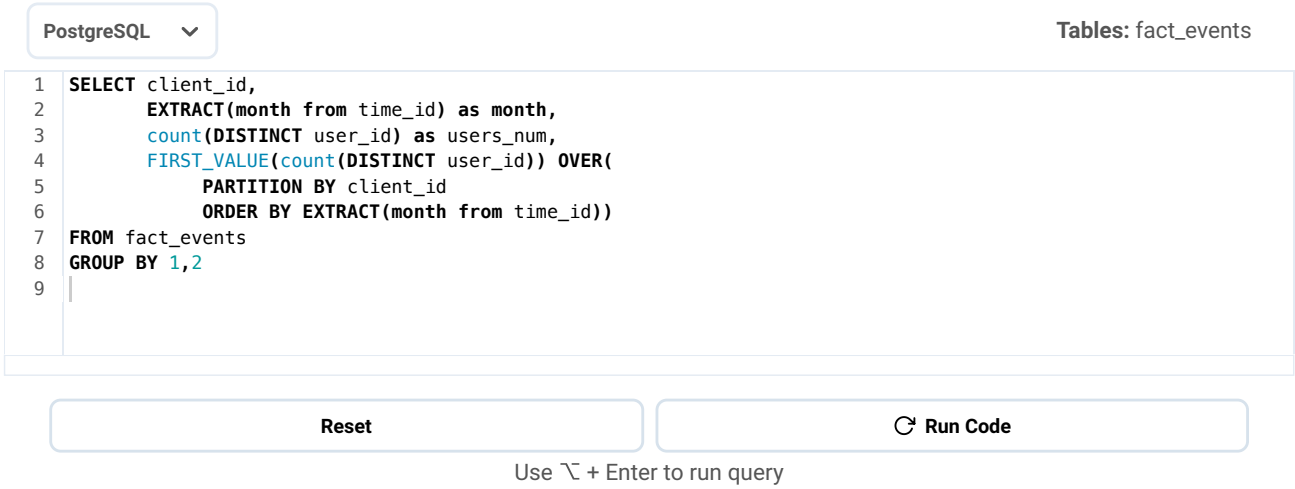
```
SELECT client_id,
       EXTRACT(month from time_id) as month,
       count(DISTINCT user_id) as users_num,
       FIRST_VALUE(count(DISTINCT user_id)) OVER(
ORDER BY client_id, EXTRACT(month from time_id))
FROM fact_events
GROUP BY 1,2
```

All required columns and the first 5 rows of the solution are shown

client_id	month	users_num	first_value
desktop	2	13	13
desktop	3	16	13
desktop	4	11	13
mobile	2	9	13
mobile	3	14	13
mobile	4	9	13

Given the defined ordering of the rows, the first value of the second column is 13, so the FIRST_VALUE() function assigns it to all the rows. Notice how the expression count(DISTINCT user_id) is also a parameter of the FIRST_ROW() function to indicate which variable should be used. However, the result above is not anywhere near the solution to this question because the results should be grouped by the client_id. This can be achieved using the PARTITION BY clause.

```
SELECT client_id,
       EXTRACT(month from time_id) as month,
       count(DISTINCT user_id) as users_num,
       FIRST_VALUE(count(DISTINCT user_id)) OVER(
            PARTITION BY client_id
            ORDER BY EXTRACT(month from time_id))
FROM fact_events
GROUP BY 1,2
```



Now, the first value that is assigned to each row is not the first value of the entire dataset but of a partition to which a particular row belongs. This can be used to answer the second part of the interview question by simply subtracting the value in the fourth column from the value in the third column.

LAST_VALUE()

The LAST_VALUE() function is the exact opposite of the FIRST_VALUE() function and, as can be deduced from the name, returns the value from the last row of the dataset or the partition to which it is applied.

To give an example, if the second task in the interview question from Microsoft was not to calculate the difference between the current month and the first month when the data is available, but between the current month and the last month when the data is available, the query could include the LAST_ROW() function:

```
SELECT client_id,
       EXTRACT(month from time_id) as month,
       count(DISTINCT user_id) as users_num,
       LAST_VALUE(count(DISTINCT user_id)) OVER(
            PARTITION BY client_id)
FROM fact_events
GROUP BY 1,2
```

PostgreSQL

Tables: fact_events

```
SELECT client_id,
       EXTRACT(month from time_id) as month,
       count(DISTINCT user_id) as users_num,
       LAST_VALUE(count(DISTINCT user_id)) OVER(
            PARTITION BY client_id)
FROM fact_events
GROUP BY 1,2
```

Reset Run Code

Use ⌥ + Enter to run query

NTH_VALUE()

Finally, the NTH_VALUE() function is very similar to both the FIRST_VALUE() and the LAST_VALUE(). The difference is that while the other functions output the value of either the first or the last row of a window, the NTH_VALUE() allows the user to define which value from the order should be assigned to other rows. This function takes an additional parameter denoting which value should be returned.

Below is the example of this function based on the interview question from Microsoft and assuming that the difference is now calculated from the number of users in the second available month of data:

```
SELECT client_id,
       EXTRACT(month from time_id) as month,
       count(DISTINCT user_id) as users_num,
       NTH_VALUE(count(DISTINCT user_id), 2) OVER(
            PARTITION BY client_id)
FROM fact_events
GROUP BY 1,2
```

PostgreSQL

Tables: fact_events

```
SELECT client_id,
       EXTRACT(month from time_id) as month,
       count(DISTINCT user_id) as users_num,
       NTH_VALUE(count(DISTINCT user_id), 2) OVER(
            PARTITION BY client_id)
FROM fact_events
GROUP BY 1,2
```

Reset Run Code

Use ⌥ + Enter to run query

Advanced Windowing Syntax in SQL

In most cases, SQL window functions use the ORDER BY clause to define the correct sorting to which a function should be applied, as well as the PARTITION BY clause when there is a need to aggregate the rows before applying the function. Both of these clauses have been covered repeatedly in this guide. However, the SQL window functions have even more potential and can be used in even more specific ways using the more advanced syntax, such as:

- Frame Specifications;
- EXCLUDE clause;
- FILTER clause;
- Window chaining.

Frame Specifications

The frame specifications determine which rows are taken as inputs by SQL window functions but while the ORDER BY and PARTITION BY clauses define the input based on the values of the dataset, the frame specifications allow defining the input from the perspective of the current row.

Before diving into the details, let's consider the most common usage of the frame specifications in this interview question from Amazon on revenue.

Revenue Over Time

Interview Question Date: December 2020

Amazon Hard ID 10314

Data Engineer Data Scientist BI Analyst Data Analyst ML Engineer Software Engineer

Find the 3-month rolling average of total revenue from purchases given a table with users, their purchase amount, and date purchased. Do not include returns which are represented by negative purchase values. Output the year-month

(YYYY-MM) and 3-month rolling average of revenue, sorted from earliest month to latest month.

A 3-month rolling average is defined by calculating the average total revenue from all user purchases for the current month and previous two months. The first two months will not be a true 3-month rolling average since we are not given data from last year. Assume each month has at least one purchase.

Table: amazon_purchases

Link to the question: https://platform.stratascratch.com/coding/10314-revenue-over-time

The task is to find the 3-month rolling average of total revenue from purchases given a table with users, their purchase amount, and date purchased. The question also specifies that a 3-month rolling average is defined by calculating the average total revenue from all user purchases for the current month and previous two months.

The first step is to compute the total revenue for each month based on the given table, which can be achieved using the following query:

```
SELECT to_char(created_at::date, 'YYYY-MM') AS MONTH,
          sum(purchase_amt) AS monthly_revenue
FROM amazon_purchases
WHERE purchase_amt>0
GROUP BY to_char(created_at::date, 'YYYY-MM')
ORDER BY to_char(created_at::date, 'YYYY-MM')
```

All required columns and the first 5 rows of the solution are shown

month	monthly_revenue
2020-01	26292
2020-02	20695
2020-03	29620
2020-04	21933
2020-05	24700

Then, the results of this query can be used to calculate the rolling average. The most popular and efficient way to do it is by using the aggregate window function AVG() as follows.

We'll also remove the monthly_revenue column because the question doesn't require it.

```
SELECT t.month,
       AVG(t.monthly_revenue) OVER(
                                   ORDER BY t.month ROWS BETWEEN 2 PRECEDING
                                   AND CURRENT ROW) AS avg_revenue
FROM
  (SELECT to_char(created_at::date, 'YYYY-MM') AS MONTH,
          sum(purchase_amt) AS monthly_revenue
   FROM amazon_purchases
```

```
    WHERE purchase_amt>0
    GROUP BY to_char(created_at::date, 'YYYY-MM')
    ORDER BY to_char(created_at::date, 'YYYY-MM')) t;
```

Go to the question on the platform

PostgreSQL

Tables: amazon_purchases

```
SELECT t.month,
       AVG(t.monthly_revenue) OVER(
                                   ORDER BY t.month ROWS BETWEEN 2 PRECEDING
                                   AND CURRENT ROW) AS avg_revenue
FROM
  (SELECT to_char(created_at::date, 'YYYY-MM') AS MONTH,
          sum(purchase_amt) AS monthly_revenue
   FROM amazon_purchases
   WHERE purchase_amt>0
   GROUP BY to_char(created_at::date, 'YYYY-MM')
   ORDER BY to_char(created_at::date, 'YYYY-MM')) t;
```

Reset | Run Code | Check Solution

Use ⌥ + Enter to run query

All required columns and the first 5 rows of the solution are shown

month	avg_revenue
2020-01	26292
2020-02	23493.5
2020-03	25535.667
2020-04	24082.667
2020-05	25417.667

Learn more about this interview question in our recent video:

In the query above, the construction within the OVER() clause of the window function 'ROWS BETWEEN 2 PRECEDING AND CURRENT ROW' is called a frame specification. This means that except for considering the ordering of the dataset, the SQL window function will compute the average only based on the 3 rows: the 2 preceding ones and the current row. This means that the input rows of the window function will be different for each row that is considered. For the 6th row, the input from which the average is calculated includes only the rows 4, 5 and 6. But for the 7th row, the input will be composed of rows 5, 6 and 7.

There are more keywords that can be used to define the frame. Firstly, the frame type may be ROWS as in the example, but it can also be GROUPS or RANGE. GROUPS refers to a set of rows that all have equivalent values for all terms of the window ORDER BY clause. RANGE is a more complicated and rarely used construction based on the ORDER BY clause that only has one term.

Then the frame boundary is defined after the BETWEEN keyword. It can use keywords such as CURRENT ROW, UNBOUNDED PRECEDING (everything before), UNBOUNDED FOLLOWING (everything after), PRECEDING and FOLLOWING. The last two keywords need to be used in combination with the number referring to how many rows, groups or ranges before or after should be considered.

EXCLUDE

The EXCLUDE clause is something that can be added to the frame specification to further change the input rows for a SQL window function. The default value is EXCLUDE NO OTHERS and does not impact the query in any way. The other possibilities are EXCLUDE CURRENT ROW, EXCLUDE GROUP or EXCLUDE TIES. These exclude from the frame respectively the current row, the whole group based on the current row, or all rows with the same value as the current row, except the current row.

To give an example, let's assume that the rolling average in the interview question from Amazon is calculated by taking the average from the previous and following month. Namely, the value for April 2020 would be calculated by taking the mean from the revenues in March 2020 and May 2020. This can be solved with the following query:

```
SELECT t.month,
       monthly_revenue,
```

```
       AVG(t.monthly_revenue) OVER(
                                  ORDER BY t.month ROWS BETWEEN 1 PRECEDING
                                  AND 1 FOLLOWING EXCLUDE CURRENT ROW)
                                  AS avg_revenue
FROM
  (SELECT to_char(created_at::date, 'YYYY-MM') AS MONTH,
          sum(purchase_amt) AS monthly_revenue
   FROM amazon_purchases
   WHERE purchase_amt>0
   GROUP BY to_char(created_at::date, 'YYYY-MM')
   ORDER BY to_char(created_at::date, 'YYYY-MM')) t
```

PostgreSQL

Tables: amazon_purchases

```
3       AVG(t.monthly_revenue) OVER(
4                                  ORDER BY t.month ROWS BETWEEN 1 PRECEDING
5                                  AND 1 FOLLOWING EXCLUDE CURRENT ROW)
6                                  AS avg_revenue
7  FROM
8    (SELECT to_char(created_at::date, 'YYYY-MM') AS MONTH,
9            sum(purchase_amt) AS monthly_revenue
10    FROM amazon_purchases
11    WHERE purchase_amt>0
12    GROUP BY to_char(created_at::date, 'YYYY-MM')
13    ORDER BY to_char(created_at::date, 'YYYY-MM')) t
```

Reset | Run Code

Use ⌥ + Enter to run query

All required columns and the first 5 rows of the solution are shown

month	monthly_revenue	avg_revenue
2020-01	26292	20695
2020-02	20695	27956
2020-03	29620	21314
2020-04	21933	27160
2020-05	24700	24810

The expression within the OVER() clause of the window functions now became 'ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING EXCLUDE CURRENT ROW' to reflect the new definition of the rolling average. See how the value for April 2020 is equal to the mean from the revenues in March 2020 and May 2020: (29620+24700)/2 = 27160.

FILTER

Another possible clause that can be used in combination with the SQL window functions is the FILTER clause. It allows defining additional conditions to exclude certain rows from the input of a function based on their values. The FILTER clause is placed between the window function and the OVER keyword and it contains a WHERE clause as its argument.

To give an example, let's assume that the rolling average in the question from Amazon is still calculated based on the revenue from the previous and following month but only for months where the revenue has been higher than 25000. This condition can be added by using the FILTER clause like that:

```
SELECT t.month,
       monthly_revenue,
       AVG(t.monthly_revenue)
FILTER(WHERE monthly_revenue > 25000)
OVER(ORDER BY t.month ROWS BETWEEN 1 PRECEDING AND 1
FOLLOWING EXCLUDE CURRENT ROW) AS avg_revenue
FROM
  (SELECT to_char(created_at::date, 'YYYY-MM') AS MONTH,
          sum(purchase_amt) AS monthly_revenue
   FROM amazon_purchases
   WHERE purchase_amt>0
   GROUP BY to_char(created_at::date, 'YYYY-MM')
   ORDER BY to_char(created_at::date, 'YYYY-MM')) t
```

PostgreSQL

Tables: amazon_purchases

```
1  SELECT t.month,
2         monthly_revenue,
3         AVG(t.monthly_revenue)
4  FILTER(WHERE monthly_revenue > 25000)
5  OVER(ORDER BY t.month ROWS BETWEEN 1 PRECEDING AND 1
6  FOLLOWING EXCLUDE CURRENT ROW) AS avg_revenue
7  FROM
8    (SELECT to_char(created_at::date, 'YYYY-MM') AS MONTH,
9            sum(purchase_amt) AS monthly_revenue
10    FROM amazon_purchases
11    WHERE purchase_amt>0
```

Reset | Run Code

Use ⌥ + Enter to run query

All required columns and the first 5 rows of the solution are shown

month	monthly_revenue	avg_revenue
2020-01	26292	
2020-02	20695	27956
2020-03	29620	
2020-04	21933	29620
2020-05	24700	27687

The query works properly because for some months the values are missing and for some the averages are only based on one month. For example, the rolling average for March 2020 cannot be computed because the revenues in both February and April are less than 25000.

Window Chaining

Window chaining is a technique that allows defining the window that can be used by window functions. Such a window needs to only be defined once but then can be reused by multiple SQL window functions. The concept is similar to the Common Table Expressions that allow defining and reusing a query. Similarly, the window can be declared separately, given an alias and reused several times by only inserting the alias in the right places.

As an example, let's consider this interview question from Amazon with window chaining.

Monthly Percentage Difference

Interview Question Date: December 2020

Amazon Hard ID 10319

Data Engineer Data Scientist BI Analyst Data Analyst ML Engineer Software Engineer

Given a table of purchases by date, calculate the month-over-month percentage change in revenue. The output should include the year-month date (YYYY-MM) and percentage change, rounded to the 2nd decimal point, and sorted from the beginning of the year to the end of the year.
The percentage change column will be populated from the 2nd month forward and can be calculated as ((this month's revenue - last month's revenue) / last month's revenue)*100.

Table: sf_transactions

Link to the question: https://platform.stratascratch.com/coding/10319-monthly-percentage-difference

The task is to, given a table of purchases by date, calculate the month-over-month percentage change in revenue. It is also said that the percentage change column will be populated from the 2nd month forward and can be calculated as ((this month's revenue - last month's revenue) / last month's revenue)*100.

This question can be solved with the following query containing two very long and unclear LAG() window functions combined in a single expression:

```
SELECT to_char(created_at::date, 'YYYY-MM') AS year_month,
  round(((sum(value) - lag(sum(value), 1) OVER (ORDER BY to_char
  (created_at::date,
  'YYYY-MM'))) / (lag(sum(value), 1) OVER (ORDER BY to_char
  (created_at::date, 'YYYY-MM')))) * 100, 2) AS revenue_diff_pct

FROM sf_transactions
GROUP BY year_month
ORDER BY year_month ASC
```

Go to the question on the platform

PostgreSQL

Tables: sf_transactions

```
1  SELECT to_char(created_at::date, 'YYYY-MM') AS year_month,
2    round(((sum(value) - lag(sum(value), 1) OVER (ORDER BY to_char
3    (created_at::date,
4    'YYYY-MM'))) / (lag(sum(value), 1) OVER (ORDER BY to_char
5    (created_at::date, 'YYYY-MM')))) * 100, 2) AS revenue_diff_pct
6
7  FROM sf_transactions
8  GROUP BY year_month
9  ORDER BY year_month ASC
```

Reset | Run Code | Check Solution

Use ⌥ + Enter to run query

All required columns and the first 5 rows of the solution are shown

year_month	revenue_diff_pct
2019-01	
2019-02	-28.56
2019-03	23.35
2019-04	-13.84
2019-05	13.49

Note how the inside of the OVER clause is the same for both functions: ORDER BY to_char(created_at::date, 'YYYY-MM'). This messy query can be made clearer and easier to read by using the window chaining. The window can be defined as follows and given an alias 'w': WINDOW w AS (ORDER BY to_char(created_at::date, 'YYYY-MM')). Then, any time a window function should be applied to this window, the OVER clause will simply contain the alias 'w'. The query from above then becomes:

```
SELECT to_char(created_at::date, 'YYYY-MM') AS year_month,
       round(((sum(value) - lag(sum(value), 1) OVER w) /
 (lag(sum(value), 1) OVER w)) * 100, 2) AS revenue_diff_pct

FROM sf_transactions
GROUP BY year_month
WINDOW w AS (ORDER BY to_char(created_at::date, 'YYYY-MM'))
ORDER BY year_month ASC
```

To practice more SQL questions, you can check out this ultimate guide to SQL interview questions. We have also covered topics like SQL JOIN Interview Questions and SQL Scenario Based Interview Questions.

Conclusion

This ultimate guide has covered all aspects of the SQL window functions, starting from listing all the functions and their categories, explaining the usage of the common ORDER BY and PARTITION BY keywords, as well as outlining the more advanced syntax. As shown using the numerous examples, the notions of the SQL window functions appear frequently in interview questions asked by top tech companies. The SQL window functions are also often used in the everyday work of data scientists and data analysts.

The SQL window functions are very powerful and efficient and their syntax, though rich and slightly different from the standard SQL, is quite logical. However, using SQL window functions comfortably requires a lot of practice, for example, solving the

interview questions from top companies.

Become a data expert. Subscribe to our newsletter.

Enter your email address Subscribe

stratascratch

Data science interview questions from your favorite companies. Prepare for a career with SQL, python, algorithms, statistics, probability, product sense, system design, and other real interview questions.

Solutions

Coding Questions

Non-Coding Questions

Login

Register

Company

Pricing

Blog

About Us

Contact Us

Support

Privacy Policy

Terms and Conditions