# My SQL Cheat Sheet

Now, this is my personal document that I use every day whenever I need to remember something related to SQL (because note this, you do not need to know expressions by heart - like ever; you just need to know how to find it + use it). 🤝

So, this is not a doc with everything that SQL can do but it is instead a list of **the most used functions for Data Analyst.**

It is a constant WIP doc where I add expressions whenever I learn or utilize something frequently.

Whenever I forget one of these expressions, I come back here and find my solution within a minute.

Click the ▶️ to open each section!

## **INTRO ABOUT SQL**

#### Link for more information.

<https://www.notion.so/My-SQL-Cheat-Sheet-b5712a97116a4667a6625c18444054d2?pvs=4#7f83990d79d348d89c39b59b5792b887>

When data is stored in tabular form, the data is organized in tables like in a spreadsheet, which is columns and rows.

### That's a relational database.

A set of software tools for the data in the database is called a database management system or **DBMS** for short.

The terms database, database server, database system, data server, and database management systems are often used interchangeably.

For relational databases, it's called a relational database management system or **RDBMS**.

RDBMS is a set of software tools that controls data such as access, organization, and storage.

And RDBMS serves as the backbone of applications in many industries including banking, transportation, health, and so on.

**Examples of relational database** management systems are.

* my SQL,
* Oracle Database,
* DB2 Express C.

A few examples of **cloud SQL databases** are.

* IBM Db2 on Cloud,
* Compose for PostgreSQL,
* Oracle Database Cloud,
* Microsoft Azure Cloud -
* SQL Database,
* Amazon Relational Database Services,

which can run in the cloud, either in a virtual machine or as a service depending on the vendor.

Database services are logical abstractions for managing workloads in a database.

Each service represents a workload with common attributes, service level thresholds, and priorities.

The grouping is based on attributes of work that might include the application function to be used, the priority of execution for the application function,

the job class to be managed or the data range used in the application function or job class.

An **instance** of the cloud database operates as a service that handles all application requests to work with the data and any of the databases managed by that instance.

The database service instance is the target of the connection request from applications.

The application we will be using is Python.

When a connection has been completed, your Python code sends SQL statements across the connection to the instance of the database.

The database instance then resolves the SQL statements into operations against the data and objects in the database.

The **connection details** include the following:

* a host name, which is a unique name or label assigned to any device that is connected to a specific computer network.
* A port number, which is the database port.
* The database name, which is the database name.
* A user ID, which is the username you'll use to connect.
* Password is the password you'll use to connect.

For most people using a database, there are five simple statements.

* to create the table,
* insert data into the table,
* select data from the table,
* update data in the table,
* and delete data from the table.

These statements fall into **two different categories.**

* **Data definition language or DDL statements** are used to define, change, or drop data.
* **Data manipulation language or DML statements** are used to read and modify data.

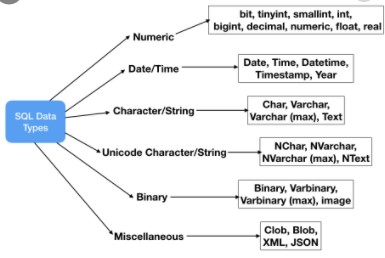
author\_id attribute is assigned as the primary key, so that no duplicate values can exist.

* T**he primary key** of a relational table uniquely identifies each tuple or row in a table.
* In a table there can be the column for
* **The foreign key** which is the primary key on another table and allows to connect the two tables.
* **Composite key** is a primary key made up of two columns (maybe because the two single columns cannot uniquely identify the row)
* **Composite key** can be made up the composition of two or more foreign keys of the table.

**Columns** are also called **attributes.**

**DATA TYPE IN SQL**

* **INT**
* **FLOAT**
* **DECIMAL (10,4):** the number in the brackets indicates the max total length of the number, and the number of decimal places that we want to store.
* **VARCHAR (100):** basically a text/string, the number in the brackets indicates the max total length of the string.
* **BLOB:** binary large object, to store large data like pictures.
* **DATE**
* **TIME**
* **TIMESTAMP**



# Different Dialects

**MySQL** has consistently been the most popular version of SQL in Stack Overflow questions. Second in line is **Microsoft SQL Server** (including **T-SQL**, the name of Microsoft’s dialect of SQL), which remains a consistently more popular tag than **PostgreSQL and SQLite**

## DIFFERENCE BETWEEN MYSQL AND POSTGRESQL

* **MySQL** uses nonstandard '**#**' to **begin a comment line**; **PostgreSQL** doesn't. Instead, **use '--'** (double dash), as this is the ANSI standard, and both databases understand it.
* **MySQL uses ' or " to quote values** (i.e. WHERE name = "John"). This is not the ANSI standard for databases. **PostgreSQL uses only single quotes for this** (i.e. WHERE name = 'John'). Double quotes are used to quote system identifiers; field names, table names, etc. (i.e. WHERE "last name" = 'Smith').
* **MySQL** uses ` (accent mark or backtick) to quote system identifiers, which is decidedly non-standard.
* **PostgreSQL is case-sensitive for string comparisons.** **The field "Smith" is not the same as the field "smith".** This is a big change for many users of MySQL and other small database systems, like Microsoft Access. In PostgreSQL, you can either:
  + Use the correct case in your query. (i.e. WHERE lname='Smith')
  + Use a conversion function, like lower () to search. (i.e. WHERE lower(lname)='smith')
  + Use a case-insensitive operator, like ILIKE or ~\*
* Database, table, field, and columns names in PostgreSQL are case-independent, unless you created them with double-quotes around their name, in which case they are case-sensitive. In MySQL, table names can be case-sensitive or not, depending on which operating system you are using.
* **PostgreSQL and MySQL seem to differ most in handling of dates, and the names of functions that handle dates.**
* MySQL uses C-language operators for logic (i.e. 'foo' || 'bar' means 'foo' OR 'bar', 'foo’ & 'bar' means 'foo' and 'bar'). This might be marginally helpful for C programmers but violates database standards and rules in a significant way. PostgreSQL, following the standard, uses || for string concatenation ('foo' || 'bar' = 'foo bar').
* There are other differences between the two, such as the names of functions for finding the current user. MySQL has a tool, Crash-Me, which can be useful for digging this out. (Ostensibly, Crash-Me is a comparison tool for databases; however, it tends to seriously downplay MySQL's deficiencies, and isn't very objective in what it lists: the entire idea of having procedural languages (a very important feature for many users!) is relegated to a single line on the bottom fifth of the document, while the fact that MySQL allows you to use || for logical-or (definitely non-standard), is listed way before this, as a feature. Be careful about its interpretations.)
* POSTGRE DateDiff

Link for more information: <https://www.sqlines.com/postgresql/how-to/datediff>.

# **THE SELECT STATEMENT**

SELECT \* >> selecting all from the table.

Operators that can be used with the **WHERE** clause:

* + AND
  + OR
  + IN, NOT IN
  + WHERE first\_name IN ('Lore', 'Sofi')
  + LIKE, NOT LIKE
  + WHERE first\_name LIKE ('Lor%') -- use % for multiple characters
  + WHERE first\_name LIKE ('Lor\_') -- use \_ for a single character (This will return for example only 'Lore')
  + BETWEEN ... AND ...
  + BETWEEN '1990-01-01' AND '2000-01-01' -- (these two dates are included)
  + NOT BETWEEN '1990-01-01' AND '2000-01-01' -- (these two dates are not included)
  + EXISTS, NOT EXISTS
  + IS NULL, IS NOT NULL
  + WHERE first\_name IS NOT NULL
  + comparison operators
    - use <> or != for not equal

Operators order: AND > OR

regardless of the orders of the AND and OR operators in your query, SQL will always start by reading the conditions around the AND operator.

WHERE lastname = 'Denis' AND gender = 'M' OR gender = 'F'

WHERE lastname = 'Denis' AND (gender = 'M' OR gender = 'F')

* + GROUP BY (after the where and before the ORDER BY)
  + HAVING (after the GROUP BY and before the ORDER BY)
    - it is like the WHERE but it applies to the GROUP BY block
    - **the condition of a WHERE cannot have aggregate functions, the condition of a HAVING can**
    - you cannot have aggregated and not aggregated conditions in the HAVING (cannot mix)
  + ORDER BY
    - at the end of the query with ASC (by default) or DESC - the field does not have to be in the select statement
  + ORDER BY first\_name DESC
  + ORDER BY first\_name, last\_name ASC -- people sharing the same name will be ordered by surname too
  + LIMIT 10 after the order by
  + OFFSET 3 after the limit, skip the first 3 rows of your result
    - you can use LIMIT and OFFSET in combination in case you want to retrieve for example only the 2 highest value excluding the top one
    - limit 1, offset 1 >> to get the second row only

### SUMMARY OF THE SELECT STATEMENT

SELECT column\_name(s)

FROM table\_name

WHERE conditions

GROUP BY column\_name(s)

HAVING conditions

ORDER BY column\_name(s)

LIMIT number.

# ALIASES AND CASE MANIPULATION

**use aliases after the aggregate function in the select statement >>** COUNT (first\_name) AS count\_names. using AS or inputting the alias without AS is the same (both for fields and tables)

SELECT LOWER ('STEPHEN') AS name FROM table;Stephen

SELECT UPPER ('stephen') AS name FROM table; STEPHEN

SELECT INITCAP ('hello stephen') AS name FROM table; Hello Stephen

# **IF VS CASE STATEMENT**

**IF**: you can have just one conditional expression

SELECT

IF (gender='M', 'Male', 'Female')

**CASE**: you can have multiple conditional expressions

SELECT

CASE

WHEN emp\_role IS NOT NULL THEN 'Manager'

WHEN .... THEN ....

ELSE 'Employees'

END AS is\_manager

# AGGREGATE FUNCTIONS

* COUNT (error if you leave a space after it - before parenthesis)
  + COUNT (first\_name) >> ignores the null
  + COUNT (\*) if you want to count the rows **INCLUDING** NULL values.
  + The \* can be used only with the count functions but not with SUM and the rest.
  + COUNT (**DISTINCT** first\_name) >> ignores the null
* SUM ()
* MIN ()
* MAX ()
* AVG () of all non-NULL values
* ROUND (#, decimal places)

ROUND (AVG (salary),2)

* IFNULL or NVL (expression1, expression2)
  + can only have two parameters and will return one of the two.

SELECT IFNULL (dept\_name, 'Department name not provided') as dept\_name

* COALESCE (expression1, expression2, expressionN)
  + it is a IFNULL with more than two parameters and will return only the first value that is not null from left to right.
  + a COALESCE with two parameters works exactly like the IFNULL.

# **FIND DUPLICATES**

**To verify the duplicates and their count**

SELECT

name, email, COUNT(\*)

FROM

users

GROUP BY

name, email

HAVING

COUNT(\*) > 1

To return the entire record for each duplicate row, we’ll need to select the entire table and join that to our duplicate rows.

SELECT a.\*

FROM users a

JOIN (SELECT username, email, COUNT(\*)

FROM users

GROUP BY username, email

HAVING count(\*) > 1 ) b

ON a.username = b.username

AND a.email = b.email

ORDER BY a.email

# DATE FUNCTIONS

Before moving to the date functions, let’s look at the popular **CAST** function, which is often used to convert a value into a specified datatype.

SELECT CAST('2017-08-25' AS datetime);

MySQL DATE is one of the five temporal data types used for managing date values. MySQL uses yyyy-mm-dd format for storing a date value.

To calculate the number of days between two date values, you use the

## DATEDIFF

function as follows:

SELECT DATEDIFF(year, '2017/08/25', '2011/08/25') AS DateDiff;

--for PostgreSQL there is no DATEDIFF but you can achieve the same with the below

DATE\_PART('year', end) - DATE\_PART('year', start)

If you want to get the day, month, quarter, and year of a date value, you can use the corresponding function

* [DAY](<https://www.mysqltutorial.org/mysql-day/>),
* [MONTH](<https://www.mysqltutorial.org/mysql-month/>),
* QUARTER, and
* [YEAR](<https://www.mysqltutorial.org/mysql-year/>) as follows:

SELECT DAY('2000-12-31') day,

MONTH('2000-12-31') month,

QUARTER('2000-12-31') quarter,

YEAR('2000-12-31') year;

## DATEPART

The DATEPART () function returns a specified part of a date.

This function returns the result as an integer value.

DATEPART (*interval*, *date*)



DATEPART(HOUR, B.TRACK\_TIME)

The interval can be:

* + year, yyyy, yy = Year
  + quarter, qq, q = Quarter
  + month, mm, m = month
  + dayofyear, dy, y = Day of the year
  + day, dd, d = Day of the month
  + week, ww, wk = Week
  + weekday, dw, w = Weekday
  + hour, hh = hour
  + minute, mi, n = Minute

## DATETRUNC

A close-up of a paper

Description automatically generated

To remove the unwanted detail of a timestamp, feed it into the DATE\_TRUNC(‘[interval]’, time\_column) function. time\_column is the database column that contains the timestamp you'd like to round, and ‘[interval]’ dictates your desired precision level.

DATE\_TRUNC(HOUR, B.TRACK\_TIME)

### DATEADD

(DATESUB is the same to subtract)

Use case example: to get the last 90 days.

DATEADD(*interval*, *number*, *date*)

* + year, yyyy, yy = Year
  + quarter, qq, q = Quarter
  + month, mm, m = month
  + dayofyear, dy, y = Day of the year
  + day, dd, d = Day
  + week, ww, wk = Week
  + weekday, dw, w = Weekday
  + hour, hh = hour
  + minute, mi, n = Minute
  + second, ss, s = Second
  + millisecond, ms = Millisecond

The number is required. The number of *intervals* to add to date. Can be **positive** (to get dates in the future) or **negative** (to get dates in the past)

DATEADD (MONTH,2,B.TRACK\_TIME) AS TWO\_MONTHS\_MORE

INTERVAL

A screenshot of a computer program

Description automatically generated

For example, you can use this to see the people whose membership expires within 7 days from today.

# OPERATIONS BETWEEN SUB-QUERIES

These type of subqueries are also called CTEs (Common Table Expressions), or WITH queries

WITH

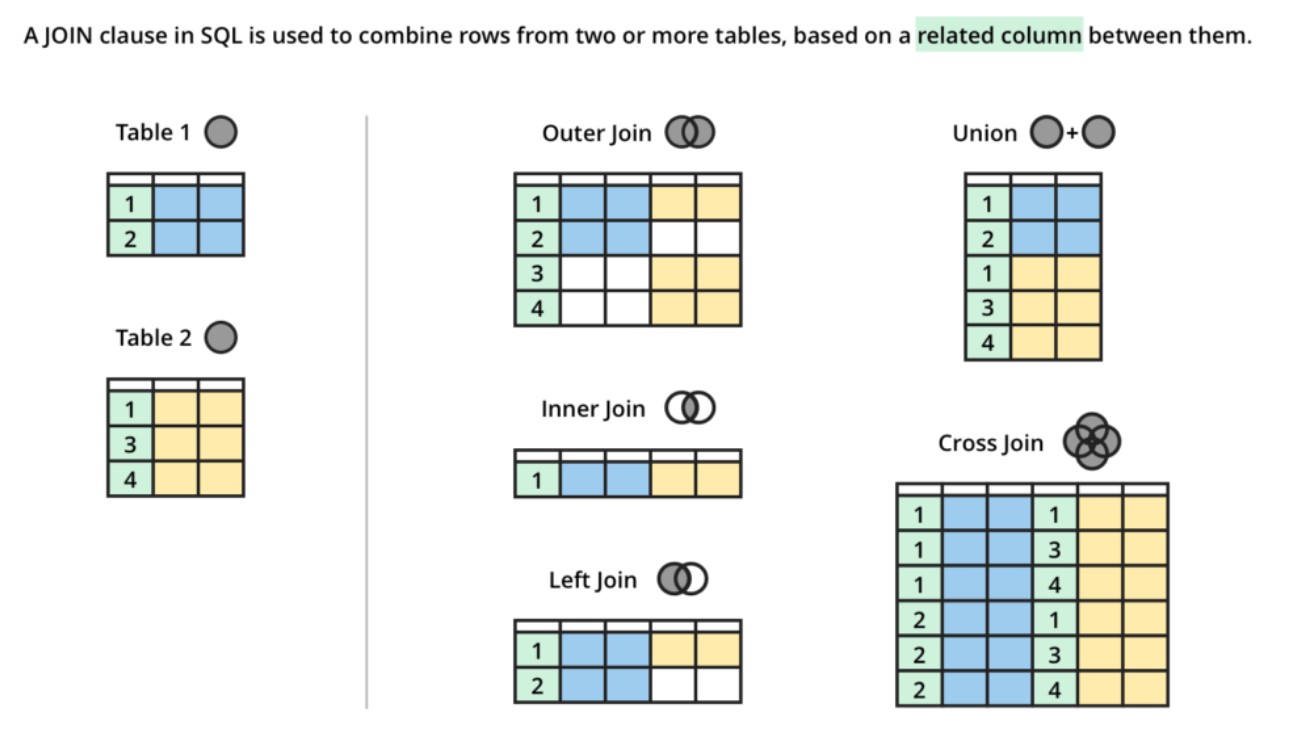
SUBQUERY1 AS (SELECT DISTINCT COUNT(xy) as NumA FROM TBL),

SUBQUERY2 AS (SELECT COUNT(DISTINCT xx) as NumB FROM TBL)

SELECT SUBQUERY1.NumA / SUBQUERY2.NumB \* 100 as FinalValue

FROM SUBQUERY1, SUBQUERY2

# **JOINS**



It is good practice to have aliases of the tables (for the joins no need to add AS but just the alias after the table name)

### INNER JOIN

A red and blue circle with black lines

Description automatically generated

Retrieves matching values of two or more tables (not matching records and null values will not appear). It will not match null values in the two tables because it wouldn't make sense.

The order of the columns after the ON does not matter.

In the query below typing INNER JOIN and JOIN is the same (will give the same result, better to use JOIN only)

SELECT

t1.column\_name, t1.other\_column\_name, t2.column\_name

FROM

table\_1 t1

INNER JOIN

table\_2 t2 ON t1.column\_name = t2.column\_name;

Be careful about duplicates, cause they will create many more duplicated records when doing the join:

A screenshot of a computer

Description automatically generated

a GROUP BY after the JOIN can be useful to deal with duplicates >> apply the GROUP BY to the field that differ the most among records (the one with less duplicates)

### LEFT JOIN

(also called left outer join)

A red and blue circles

Description automatically generated

Retrieves matching values of two tables plus the non-matching values from the left table.

SELECT

t1.column\_name, t2.column\_name

FROM

table\_1 t1

LEFT JOIN

table\_2 t2 ON t1.column\_name = t2.column\_name;

For this type of join, the order in which you join tables **matters.**

When using left joins, make sure that in the select statement you are selecting columns from the left table and not the right one.

### CROSS JOINS

Connects all the values not just those that match (cartesian product of the values of two or more sets) >> no need for the ON in the syntax.

SELECT

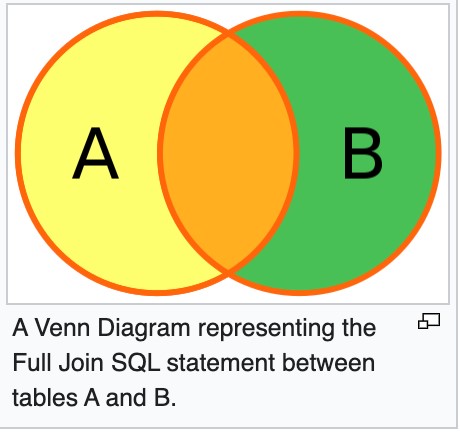
t1.column\_name, t1.column\_name, t2.column\_name

FROM

table\_1 t1

CROSS JOIN

table\_2 t2;



SELECT \* FROM

employee FULL OUTER JOIN

department ON employee.DepartmentID = department.DepartmentID;

### JOIN MULTIPLE TABLES

SELECT

...

FROM

table\_1 t1

JOIN

table\_2 t2 ON ...

JOIN

table\_3 t3 ON ...

### UNION VS UNION ALL

A screenshot of a computer

Description automatically generated

A white background with blue text

Description automatically generated

### SELF JOIN

SELECT

select\_list

FROM

T t1

[INNER | LEFT] JOIN T t2 ON

join\_predicate;

<https://www.sqlservertutorial.net/sql-server-basics/sql-server-self-join/>

Two main use cases for this:

1. Using self join to query hierarchical data
2. Using self join to compare rows within a table

* FULL OUTER JOIN

# **SUBQUERIES**

(OR INNER QUERIES/NESTED QUERIES)

SELECT

...

FROM

table\_1 t1

WHERE

column1 IN (SELECT

column2

FROM

table\_2);

## EXISTS

A screenshot of a computer

Description automatically generated

A white rectangular box with blue text

Description automatically generated with medium confidence

# **CREATE BUCKETS**

WIDTH\_BUCKET( <expr> , <min\_value> , <max\_value> , <num\_buckets> )

A screenshot of a computer code

Description automatically generated

# **WINDOW FUNCTIONS**

Window functions do not cause rows to become grouped into a single output row, the rows retain their separate identities and an aggregated value will be added to each row.

The term Window describes the set of rows in the database on which the function will operate.

We define the Window (set of rows on which functions operates) using an OVER() clause.

A close-up of a computer code

Description automatically generated

## **Arguments**

**window function** Specify the name of the window function.

**ALL:** ALL is an optional keyword. When you include ALL, it will count all values including duplicate ones. DISTINCT is not supported in window functions.

**Expression**: The target column or expression that the functions operate on. In other words, the name of the column for which we need an aggregated value. For example, a column containing order amount so that we can see total orders received.

**OVER:** Specifies the window clauses for aggregate functions.

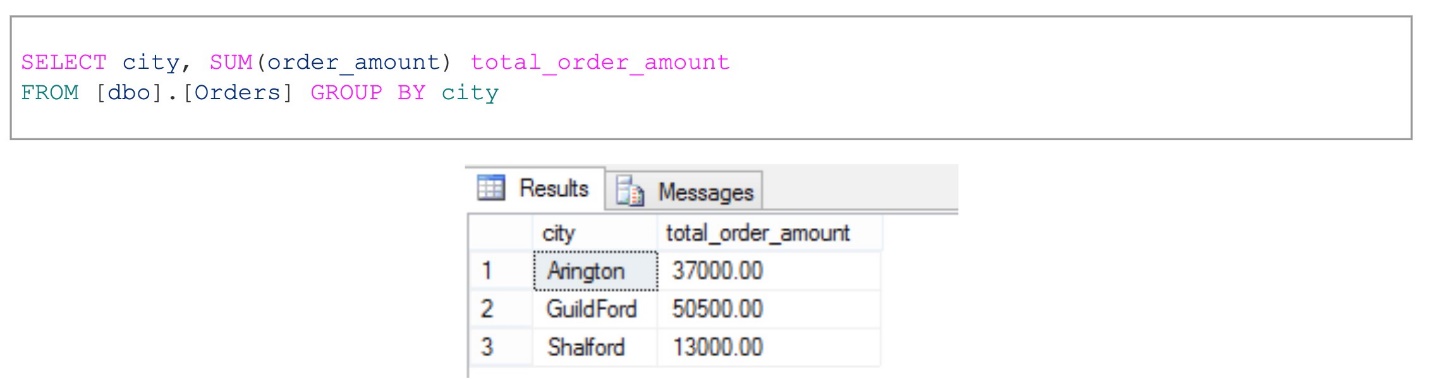
**PARTITION BY: partition list** Defines the window (set of rows on which window function operates) for window functions. We need to provide a field or list of fields for the partition after PARTITION BY clause. Multiple fields need to be separated by a comma as usual. If PARTITION BY is not specified, grouping will be done on entire table and values will be aggregated accordingly.

**ORDER BY order list** Sorts the rows within each partition. If ORDER BY is not specified, ORDER BY uses the entire table.

### **Types of Window functions**

* **Aggregate Window Functions** SUM(), MAX(), MIN(), AVG(). COUNT() >>Note that DISTINCT is not supported with window COUNT() function

### NORMAL SUM



### *WINDOW SUM*

A screenshot of a computer

Description automatically generated

### Ranking Window Functions

RANK(), DENSE\_RANK(), ROW\_NUMBER(), NTILE()

The most common use of RANKING functions is to find the top (N) records based on a certain value. For example, Top 10 highest paid employees, Top 10 ranked students, Top 50 largest orders etc.

If two records have the same value, then the **RANK ()** function will assign the same rank to both records by skipping the next rank. This means – if there are two identical values at rank 2, it will assign the same rank 2 to both records and then skip rank 3 and assign rank 4 to the next record.

*select also order amount below.*

A screenshot of a computer

Description automatically generated

### The **DENSE\_RANK ()**

function is identical to the RANK () function except that it does not skip any rank.

A screenshot of a computer

Description automatically generated

### ROW\_NUMBER ()

The name is self-explanatory. These functions assign a unique row number to each record.

The row number will be reset for each partition if PARTITION BY is specified.

A screenshot of a computer

Description automatically generated

NTILE ()is a very helpful window function. It helps you to identify what percentile (or quartile, or any other subdivision) a given row falls into.

A screenshot of a computer

Description automatically generated

## Value Window Functions

LAG(), LEAD(), FIRST\_VALUE(), LAST\_VALUE()

The **LAG** function allows to access data from the previous row in the same result set without use of any SQL joins. You can see in below example, using LAG function we found previous order date.

LEAD function allows to access data from the next row in the same result set without use of any SQL joins. You can see in below example, using LEAD function we found next order date.

A white background with black text

Description automatically generated

A screenshot of a computer screen

Description automatically generated

### **FIRST\_VALUE () and LAST\_VALUE()**

These functions help you to identify the first and last record within a partition or entire table if **PARTITION BY** is not specified.

A screenshot of a computer

Description automatically generated

# **PIVOT TABLES WITH COUNT AND CASE**

A screenshot of a computer

Description automatically generated

A screen shot of a video

Description automatically generated

OR

A computer screen shot of a program

Description automatically generated