# SQL Notes

Giuseppe Giorgio Colabufo

November 8, 2023

***Disclaimer***

This notebook is created to provide general information and support, but there may be errors or inaccuracies in the content. If you notice any inaccuracies or issues in my notes, please report them.

It is always recommended to verify the information provided with reliable sources and consult qualified experts for specific matters. I am not responsible for any consequences arising from the use of the information contained in this notebook.

To report any errors or provide feedback, I encourage you to contact me at the specified address below:

[giuseppe.colabufo.2016@polytechnique.org]

Thank you for your understanding and cooperation in helping me improve and correct any errors in the document.

**Brief Introduction**

SQL is a widely-used database querying language, and understanding its capabilities can be essential for efficiently managing and manipulating data. In these notes, you will find information on creating efficient SQL queries, using different clauses and functions to filter and sort data, and leveraging advanced SQL features for complex analysis.

These notes consist of tricks, syntax, and recommendations for working with the SQL language that I have gathered over time and found useful in various situations. They represent a set of practical solutions I have discovered and experimented with to tackle specific challenges in my professional or personal journey.

Within these notes, you will find a variety of tips and ideas, ranging from small tricks that simplify daily tasks to advanced features that can enhance efficiency and user experience with specific tools or technologies. You will also come across code snippets that I have deemed particularly helpful in solving common problems or implementing specific functionalities.

This collection is a (more or less) organic compilation of information that I have updated and expanded over time. I invite you to explore these notes with curiosity and use the proposed solutions as inspiration for your specific needs. Remember, it is always important to adapt and personalize the solutions based on the context in which you apply them.
I hope this information proves useful to you and inspires you on your journey. Enjoy exploring and applying these insights!

**Table of Contents**

# 1 Dates

## 1.1 Compute difference between two dates

To compute the difference between two dates in SQL, you can use the DATEDIFF function. The DATEDIFF function calculates the difference between two dates in terms of a specified date part, such as days, months, or years.

Here's the basic syntax for using DATEDIFF:

```sql
SELECT DATEDIFF(datepart, start_date, end_date) AS date_difference;
```

In the above syntax, datepart specifies the unit of measurement for the difference (e.g., 'day', 'month', 'year'). start_date and end_date are the two dates between which you want to calculate the difference. The result will be returned as date_difference.

Here's an example query that calculates the difference in days between two dates:

```sql
SELECT DATEDIFF(day, '2023-06-15', '2023-06-20') AS date_difference;
```

The result of this query would be 5, as it calculates the number of days between June 15, 2023, and June 20, 2023.

You can replace the day datepart with other options like month or year depending on the desired unit of measurement for the difference.

**Note:** The availability of the DATEDIFF function depends on the specific database management system (DBMS) you are using.

I'll provide you with alternative approaches using commonly available functions in SQL for different database systems:

1. For MySQL: MySQL does have a DATEDIFF function, and you can use it as described in my previous response.
2. For SQL Server: SQL Server also has a DATEDIFF function, and you can use it as follows:

   ```sql
   SELECT DATEDIFF(day, start_date, end_date) AS date_difference;
   ```

3. For PostgreSQL: In PostgreSQL, you can use the AGE function to calculate the difference between two dates. Here's an example:

   ```sql
   SELECT (end_date - start_date) AS date_difference;
   ```

4. For Oracle: In Oracle, you can use the MONTHS_BETWEEN function to calculate the difference in months between two dates. Here's an example:

   ```sql
   SELECT MONTHS_BETWEEN(end_date, start_date) AS date_difference FROM dual;
   ```

Please note that the syntax and available functions may vary depending on the specific database management system you are using.

### 1.1.1 Dates in a string format

If your dates are in string format, you will need to convert them to the appropriate date type before performing calculations. The conversion can be done using the appropriate date conversion function provided by your database system.

Here's an example query assuming your dates are in string format and you want to calculate the difference in days:

```sql
-- Assuming the dates are in 'YYYY-MM-DD' format

-- MySQL
SELECT DATEDIFF(end_date, start_date) AS date_difference;

-- SQL Server
SELECT DATEDIFF(day, CAST(start_date AS DATE), CAST(end_date AS DATE)) AS date_difference;

-- PostgreSQL
SELECT
    (TO_DATE(end_date, 'YYYY-MM-DD') - TO_DATE(start_date, 'YYYY-MM-DD'))
    AS date_difference;

-- Oracle
SELECT
    (TO_DATE(end_date, 'YYYY-MM-DD') - TO_DATE(start_date, 'YYYY-MM-DD'))
    AS date_difference
FROM dual;
```

In the above examples, `start_date` and `end_date` are assumed to be the string representations of your dates in the 'YYYY-MM-DD' format. The conversion functions (`CAST`, `TO_DATE`) are used to convert the strings into the appropriate date type expected by the respective database systems.

Make sure to adjust the format in the conversion functions (`'YYYY-MM-DD'`) according to the actual format of your date strings.

**In SQLite:**

In SQLite, you can use the `date` function to convert strings to date format and then calculate the difference between two dates. Here's an example query:

```sql
SELECT
  julianday(end_date) - julianday(start_date) AS date_difference
FROM
  your_table;
```

In this example, `start_date` and `end_date` are assumed to be the string representations of your dates. The `julianday` function is used to convert the dates to Julian day numbers, which allows for easy calculation of the difference between two dates. The result `date_difference` will be returned in days.

Ensure that the date format in your string representations matches the format expected by SQLite.

## 1.2 Filter for timestamp greater than today

To filter the `ts` column to include timestamps greater than or equal to the current date, you can modify the query in Grafana as follows:

```sql
SELECT
  ts AS "time",
  avg(forecast_fossil_gas)
FROM r_gas_to_power_d1
WHERE
  DATE(ts) >= CURDATE()
GROUP BY
  ts
ORDER BY
  ts;
```

In the updated query, the `DATE(ts) >= CURDATE()` condition is used to filter the data based on the date portion of the `ts` column being greater than or equal to the current date.

The `CURDATE()` function retrieves the current date, and the `DATE(ts)` function extracts the date portion from the `ts` column. By comparing these two values, you can filter the data to include timestamps from today onwards.

Please note that the availability and behavior of date and time functions can vary depending on the database system you are using. The above example assumes a standard SQL database. If you are using a different database system or have specific requirements, please provide more details so that I can assist you accordingly.

If the `CURDATE()` function is not available in your database system, you can use an alternative method to get the current date. Here's an updated query that should work in most database systems:

```sql
SELECT
  ts AS "time",
  avg(forecast_fossil_gas)
FROM r_gas_to_power_d1
WHERE
  DATE(ts) >= CURRENT_DATE
GROUP BY
  ts
ORDER BY
  ts;
```

In this query, the `CURRENT_DATE` keyword is used to get the current date. The `DATE(ts)` function extracts the date portion from the `ts` column, and the comparison is made to include timestamps from today onwards.

Please note that the specific functions and syntax may vary depending on the database system you are using. If you encounter any issues or have specific requirements, please let me know the database system you are using, and I can provide a more tailored solution.

### 1.2.1 If you want the timestamp greater than tomorrow

To filter the `ts` column to include timestamps greater than or equal to tomorrow's date, you can modify the query in Grafana as follows:

```sql
SELECT
  ts AS "time",
  avg(forecast_fossil_gas)
FROM r_gas_to_power_d1
WHERE
  DATE(ts) >= (CURRENT_DATE + INTERVAL '1 day')
GROUP BY
  ts
ORDER BY
  ts;
```

In the updated query, the condition `DATE(ts) >= (CURRENT_DATE + INTERVAL '1 day')` is used to filter the data based on the date portion of the `ts` column being greater than or equal to tomorrow's date.

The `CURRENT_DATE` keyword retrieves the current date, and the `INTERVAL '1 day'` adds one day to the current date. By comparing the extracted date from the `ts` column with `(CURRENT_DATE + INTERVAL '1 day')`, you can filter the data to include timestamps from tomorrow onwards.

Please note that the specific syntax for date arithmetic and intervals may vary depending on the database system you are using. The above example assumes a standard SQL database. If you are using a different database system or have specific requirements, please provide more details so that I can assist you accordingly.

## 2 Miscellanea

### 2.1 Combine queries

To combine Query A and Query B into a single query and obtain a unique timeseries, you can use the UNION ALL operator. The UNION ALL operator allows you to combine the result sets of two separate queries into a single result set. Here's an example:

```
SELECT
  ts AS "time",
  forecast_fossil_gas as "mbi forecast"
FROM r_gas_to_power_d1
WHERE
  DATE(ts) >= CURRENT_DATE
UNION ALL
SELECT
  ts AS "time",
  forecast_fossil_gas as "mbi forecast"
FROM r_gas_to_power_d2
WHERE
  DATE(ts) >= (CURRENT_DATE + INTERVAL '1 day')
ORDER BY 1;
```

In this example, the first query retrieves data from `r_gas_to_power_d1` where the date is greater than or equal to the current date. The second query retrieves data from `r_gas_to_power_d2` where the date is greater than or equal to tomorrow's date.

The UNION ALL operator combines the results of both queries, resulting in a single result set. The column aliases ("time" and "mbi forecast") are included in each query to ensure consistent column names in the final result.

The ORDER BY clause at the end ensures that the combined result set is sorted in ascending order based on the "time" column.

Please note that the specific syntax for date arithmetic and intervals may vary depending on the database system you are using. The above example assumes a standard SQL database. If you are using a different database system or have specific requirements, please provide more details so that I can assist you accordingly.

### 2.2 Sum of quantity 'f' is this sum is positive and 0 otherwise

#### A query in sql that can give the sum of quantity 'f' is this sum is positive and 0 otherwise

In SQL, you can use a `CASE` statement with an aggregate function like `SUM` to achieve this. Here's an example query:

```
SELECT
    CASE WHEN SUM(quantity) > 0 THEN SUM(quantity)
    ELSE 0 END AS sum_positive_quantity
FROM your_table;
```

Replace `your_table` with the actual name of your table, and `quantity` with the column name that contains the quantity values you want to sum. This query will calculate the sum of the quantity values, and if the sum is greater than 0, it will return the sum. Otherwise, it will return 0.

## 2.3 Count unique couples of values

To count the number of unique couples of values from the "lat" and "long" columns, you can use the COUNT DISTINCT function along with a combination of the "lat" and "long" columns. Here's an example query:

```
SELECT COUNT(DISTINCT CONCAT(lat, ',', long)) AS unique_couples
FROM your_table;
```

In this query, the CONCAT function is used to combine the "lat" and "long" values into a single string separated by a comma (','). The DISTINCT keyword ensures that only unique combinations of "lat" and "long" are considered. Finally, the COUNT function counts the number of unique couples.

Make sure to replace "your_table" with the actual name of your table where the "lat" and "long" columns are located.

Please note that the CONCAT function used above may vary depending on the database system you are using. If you encounter any issues or have specific requirements, please provide more details about your database system so that I can assist you accordingly.

**In SQLite:**

In SQLite, you can use the `||` operator for string concatenation instead of a specific `CONCAT` function. Here's an example query to count the unique couples of values from the "lat" and "long" columns in SQLite:

```
SELECT COUNT(DISTINCT (lat || ',' || long)) AS unique_couples
FROM your_table;
```

In this query, the `||` operator is used to concatenate the "lat" and "long" values along with the comma (',') separator. The DISTINCT keyword ensures that only unique combinations of "lat" and "long" are considered. Finally, the COUNT function counts the number of unique couples.

Make sure to replace "your_table" with the actual name of your table where the "lat" and "long" columns are located.

Please note that the usage of `||` for string concatenation is specific to SQLite and may differ in other database systems. If you are using a different database system or have specific requirements, please provide more details so that I can assist you accordingly.

## 2.4 Define variables in query

### 2.4.1 PostgreSQL

In this example, we have a hypothetical table named "sales_data" with two columns: "revenue" and "expenses." We want to calculate the ratio of profitable months (where revenue exceeds expenses) to the total number of months.

Here's the SQL query to accomplish this:

```sql
-- Create a toy example table
CREATE TABLE sales_data (
  month text,
  revenue numeric,
  expenses numeric
);

-- Insert some sample data
INSERT INTO sales_data (month, revenue, expenses)
VALUES
  ('January', 5000, 3000),
  ('February', 6000, 3500),
  ('March', 5500, 4000),
  ('April', 7000, 4500);

-- Calculate the ratio of profitable months to total months
WITH summary AS (
  SELECT
    SUM(CASE WHEN revenue > expenses THEN 1 ELSE 0 END) AS profitable_months,
    COUNT(*) AS total_months
  FROM sales_data
)
SELECT
  profitable_months,
  total_months,
  CASE WHEN total_months > 0 THEN profitable_months::float / total_months::float
          ELSE 0 END AS profit_ratio
FROM summary;
```

Explanation:

1. We create a toy example table called "sales_data" with columns for "month," "revenue," and "expenses."
2. Sample data for four months (January to April) is inserted into the "sales_data" table. In this example, we assume that January, February, and April are profitable months (revenue > expenses), and March is not.
3. The CTE named "summary" calculates two values:
   - `profitable_months`: The sum of 1s for each row where revenue exceeds expenses (i.e., a profitable month).
   - `total_months`: The total count of rows in the "sales_data" table, representing the total number of months.
4. In the main query, we select `profitable_months`, `total_months`, and calculate the `profit_ratio`. The `CASE` statement ensures that we don't divide by zero (when there are no months in the table). The `::float` cast ensures that the division results in a floating-point number.
5. The result provides the count of profitable months, the total count of months, and the ratio

of profitable months to the total months.

This example demonstrates how to use a CTE to calculate the ratio of a specific condition ("profitable months" in this case) to the total count of records in a table. You can adapt this approach to your own scenarios and conditions.

```
[ ]:
```