# Chapter 3 :
# Advanced SQL

Aviral Janveja

## 1  Date

Most relational databases format dates as **YYYY-MM-DD**. This format makes a lot of sense because it sorts in chronological order even when the date field is stored as a string.

### 1.1  Date Subtraction

The following query uses date subtraction to determine how long it took companies to be acquired. Unacquired companies and those without dates entered were filtered out. Note that because the date column is stored as a string, it must be cast as a timestamp before it can be subtracted from another timestamp :

```
SELECT companies_name,
founded_date,
acquired_date,
acquired_date - founded_date::timestamp AS
time_to_acquisition
```

In the example above, the time_to_acquisition column is an interval, not another date. An interval refers to an integer that represents a period of time.

### 1.2  Interval Function

You can also introduce intervals using the interval function, for example :

```
SELECT companies_name,
founded_date::timestamp + INTERVAL '1 week' AS
plus_one_week
```

The interval is defined using plain English terms as shown above, such as **10 seconds** or **5 months**. Also note that adding or subtracting a date column and an interval column results in another date column.

### 1.3  Now Function

You can add the current time at which you run the query into your code using the now function. For example :

```
SELECT companies_name,
    founded_date,
    NOW() - founded_date::timestamp AS founded_time_ago
```

# 2 Data Types

Each column in a database table is required to have a name and a data type. The data type of a column defines what value the column can hold : integer, character, date-time, boolean, and it also identifies how SQL will interact with the stored data.

It is certainly best for data to be stored in its optimal format from the beginning, but if it is not, you can always change it in your query. It is particularly common for dates or numbers, for example, to be stored as strings. This becomes problematic when you want to sum a column and you get an error because SQL is reading numbers as strings. When this happens, you can use **CAST** or **CONVERT** to change the type of data to a numeric one that will allow you to complete the sum.

You can actually achieve this with two different types of syntax, both of which produce the same result :

```
CAST(column_name AS integer)
or
column_name::integer
```

# 3 Data Cleaning

Real world data is often messy and can lead to flawed decision-making and a waste of resources. Especially in a world of data driven decision-making, it is vital to ensure that data is clean and prepared for analysis so that we gather the most accurate insights to base our business decisions on.

Data cleaning is the process of fixing incorrect, incomplete, duplicate or otherwise erroneous data. These issues can arise from human error during data-entry or from combining different data sources that might be using different terminology. Many of these errors can lead to faulty results, skewing our understanding of the data. Let us look at the following SQL functions for data cleaning :

## 3.1 Left, Right and Length

You can use **LEFT** to pull a certain number of characters from the left side of a string and present them as a separate string. The syntax is : LEFT(string, number of characters). **RIGHT** does the same thing, but from the right side. The **LENGTH** function returns the length of a string. For example :

```
SELECT date,
    LENGTH(date),
    LEFT(date, 10) AS cleaned_date,
    RIGHT(date, 17) AS cleaned_time
```

**Note** : When using functions within other functions, the innermost function will be evaluated first, followed by the encapsulating functions.

## 3.2 Trim and From

The **TRIM** function is used to remove characters from the beginning and end of a string. It takes 3 arguments. First specify whether you want to remove characters

from the beginning : **leading**, the end : **trailing** or both : **both**. Next you must specify the characters to be trimmed. Any characters included within the single quotes will be removed. Finally, you must specify the text you want to trim using **FROM**. For example :

```
SELECT location,
    TRIM(both '()' FROM location)
```

## 3.3   Position

**POSITION** allows you to specify a sub-string and return the numerical value of the position where that sub-string first appears in the target string, counting from left. For example, the following query will return the position of the character **A** where it first appears in the descript field :

```
SELECT descript,
    POSITION('A' IN descript) AS "A_position"
```

You can also use the **STRPOS** function to achieve the same result. Just replace IN with a comma and switch the order of the string and sub-string :

```
SELECT descript,
    STRPOS(descript, 'A') AS "A_position"
```

Importantly, both the POSITION and STRPOS functions are **case-sensitive**.

## 3.4   Sub-Strings

LEFT and RIGHT both create sub-strings of a specified length, but they only do so starting from the sides of an existing string. If you want to start in the middle of a string, you can use **SUBSTR**. For example :

```
SELECT date,
    SUBSTR(date, 4, 2) AS day
```

Within SUBSTR, we first mention the field name, then the starting character position and then the number of characters to be extracted as shown above.

## 3.5   Concatenation

You can combine strings from several columns together, and add extra characters using **CONCAT**. Simply mention the fields you want to concatenate and separate them with commas. If you want to add extra characters, enclose them within single quotes. Here is an example :

```
SELECT day_of_week,
    CONCAT(day_of_week, ', ', LEFT(date, 10)) AS
    day_and_date
```

Alternatively, you can use two pipe characters (||) to perform the same concatenation operation :

```
SELECT day_of_week,
    day_of_week || ', ' || LEFT(date, 10)) AS
    day_and_date
```

## 3.6 Upper and Lower

You can use **LOWER** to force every character in a string to become lower-case. Similarly, you can use **UPPER** to make all the characters appear in upper-case :

```
SELECT address,
    UPPER(address) AS address_upper,
    LOWER(address) AS address_lower
```

## 3.7 Extract

You have learned how to construct a properly formatted date field, but what if you want to deconstruct one? You can use **EXTRACT** to pull the pieces out :

```
SELECT cleaned_date,
    EXTRACT('year' FROM cleaned_date) AS year,
    EXTRACT('month' FROM cleaned_date) AS month,
    EXTRACT('day' FROM cleaned_date) AS day
```

You can also round dates to the nearest unit of measurement. This is particularly useful if you don't care about an individual date, but the month or year that it occurred in. The **DATE_TRUNC** function rounds a date to whatever precision you specify. The value displayed is the first value in that period. So when you round-off by year, any value in that year will be listed as January 1st of that year.

```
SELECT cleaned_date,
    DATE_TRUNC('year' , cleaned_date) AS year,
    DATE_TRUNC('month' , cleaned_date) AS month,
    DATE_TRUNC('day' , cleaned_date) AS day
```

## 3.8 Current Date and Time

You can get the current date and time using the following functions. Interestingly, you can run them without a FROM clause :

```
SELECT CURRENT_DATE AS date,
    CURRENT_TIME AS time,
    CURRENT_TIMESTAMP AS timestamp,
    LOCALTIME AS localtime,
    LOCALTIMESTAMP AS localtimestamp,
    NOW() AS now
```

## 3.9 Replacing Null Values

Occasionally, you will end up with a dataset that has some null values that you prefer contained actual values instead. This happens frequently with numerical data and when performing outer-joins that result in unmatched rows. In such cases, you can use **COALESCE** to replace the null values :

```
SELECT description,
    COALESCE(description, 'No Description')
```

This will replace all the empty rows in the description column with the string 'No Description'.

# 4 Subqueries

Subqueries are queries nested inside another query. Subqueries allow you to break down complex queries into smaller, more manageable steps. The following subquery filters the given data in two steps :

```
SELECT sub.*
FROM (
  SELECT *
  FROM tutorial.sf_crime_incidents_2014_01
  WHERE day_of_week = 'Friday'
 ) sub
WHERE sub.resolution = 'NONE'
```

Let us break down the above subquery step by step. Looking at the **inner query** first :

```
SELECT *
FROM tutorial.sf_crime_incidents_2014_01
WHERE day_of_week = 'Friday'
```

The database runs the inner query first, narrowing down incidents to those that happened on a Friday. The **outer query** takes this result and then filters it further to include only unresolved incidents. That is, where resolution = 'NONE' :

```
SELECT sub.*
FROM ( ... ) sub
WHERE sub.resolution = 'NONE'
```

Subqueries are required to have names, which are added after the parentheses, the same way you would add an alias to a normal table. In the above example, we have used the name **sub**.

The outer query uses **SELECT sub.\*** to specify that it should select all columns from the result of the subquery called sub. This is equivalent to selecting each column individually, for example : sub.column1, sub.column2 and so on.

## 4.1 Using Subqueries to Aggregate

What if you wanted to know how many crime-incidents happen on average, on a Friday in January? There are two steps to this process: counting the number of incidents each day (**inner query**), then determining the monthly average (**outer query**). Let us write the inner query first :

```
SELECT day_of_week,
        date,
        COUNT(incidnt_num) AS incidents
    FROM tutorial.sf_crime_incidents_2014_01
    GROUP BY 1,2
```

Through the above inner query, we first count the number of incidents for each day of the week, Monday through Friday for all weeks included in the dataset. Next, we include the outer query which calculates the monthly average for each day of the week, such that you can compare what is the average number of crimes happening on a Friday in November versus Saturday in November or Friday in December and so on :

```
SELECT LEFT(sub.date, 2) AS cleaned_month,
    sub.day_of_week,
    AVG(sub.incidents) AS average_incidents
FROM (
    SELECT day_of_week,
           date,
           COUNT(incidnt_num) AS incidents
      FROM tutorial.sf_crime_incidents_2014_01
    GROUP BY 1,2
   ) sub
GROUP BY 1,2
ORDER BY 1,2
```

**Note** : In general, it is easier to write the inner query first, make sense of the result and then proceed to implementing the outer query.

## 4.2   Using Subqueries in Conditional Statements

You can use subqueries in conditional statements like WHERE and CASE. The following query for example, returns all of the entries from the earliest date in the dataset :

```
SELECT *
FROM tutorial.sf_crime_incidents_2014_01
WHERE Date = (SELECT MIN(date)
                FROM tutorial.sf_crime_incidents_2014_01
                )
```

**Note** that you should not include an alias when you write a subquery in a conditional statement. This is because the subquery is treated as an individual value or set of values, rather than as a table. For example, the result of the inner query above is only one cell, the minimum date value.

## 4.3   Using Subqueries with JOIN

Joining subqueries can be particularly useful when combined with aggregations. For example, the following query ranks all results according to how many incidents were reported in a given day. It does this by aggregating the total number of incidents each day in the inner query and then using those values to sort via the outer query :

```
SELECT incidents.*,
    sub.incidents AS incidents_that_day
```

```
FROM tutorial.sf_crime_incidents_2014_01 incidents
JOIN ( SELECT date,
       COUNT(incidnt_num) AS incidents
        FROM tutorial.sf_crime_incidents_2014_01
       GROUP BY 1
    ) sub
ON incidents.date = sub.date
ORDER BY sub.incidents DESC, time
```

## 4.4   Using Subqueries with UNION

It is possible that a dataset comes split into several parts, especially if it has passed through Excel at any point, since Excel can only handle about 1M rows per spreadsheet. However, we would like to perform operations on the entire dataset at once, rather than on the individual parts. You can do this by using a subquery alongside UNION :

```
SELECT COUNT(*) AS total_rows
FROM (
    SELECT *
      FROM tutorial.crunchbase_investments_part1

     UNION ALL

    SELECT *
      FROM tutorial.crunchbase_investments_part2
    ) sub
```

# 5   Window Functions