Handy Data Cleansing Techniques Using SQL

By - Sushant(Data Analyst): https://www.linkedin.com/in/ksushant/

Real-world data is almost always messy. As a data scientist or a data analyst or a data engineer or even a developer, if you need to discover facts about data, it's vital to ensure that data is tidy enough for doing that.
In this tutorial, we will be practising some of the most common data cleaning techniques in SQL. We will create our own dummy dataset, but the techniques can be applied to the real-world data (of the tabular form) as well. The contents of this tutorial are as follows:

- Different data types & their messy values
- Problems that can raise from messy numbers
- Cleaning numeric values
- Messy strings
- Cleaning string values
- Messy date values & cleaning them
- Duplications & removing them

Different Data Types, Messy Values, & Remedies

Messy Numbers

Numbers can be in messy forms in a number of ways. Here, we will be introduced to the most common ones:

- **Undesired type/Type mismatch**: Consider there is a column named `age` in a dataset we are working with. We see the values that are present in that column are of `float` type - the sample values are like 23.0, 45.0, 34.0 & so on. In this case, you don't need the `age` column to be of `float` type.

- **Null values**: While this is particularly common with all of the data types mentioned above, null values here merely means that the values are not available/blank. However, null values can be present in other forms as well.

Let's now study the problems that can get raised from these issues & how to deal with them.

Problems With Messy Numbers & Dealing With Them

Let's now take a look at the most common problems that we may face if we don't clean the messy data

1.) Data Aggregation

Suppose we have null entries for a numeric column & we are calculating summary statistics (like mean, maximum, minimum values) on that column. The results will not get conveyed accurately in this case. There are several ways on how to address this problem:

```
- Removing the entries containing missing/null values (not recommended)
- Imputing the null entries with a numeric value (typically with mean or median of the respectiv
```

Let's now get hands-on with these problems & the second option for combating null values.

Consider the following PostgreSQL table named entries:

name text	weight_in_lbs double precision	age_in_years smallint
Christina	80.6	
Matthews		19
Gilbert	100.3	21

We can see two null entries in the above table. Suppose we want to get the average weight value from this table & we executed the following query:

```
select avg(weight_in_lbs) as average_weight_in_lbs from entries;
```

We got 90.45 as the output. Is this correct? So, what can be done? Let's fill the null entry with this average value with the help of the COALESCE() function.

Let's fill the missing values first with COALESCE() (remember that COALESCE() does not change the values in the original table, it just returns a temporary view of the table with the values changed):

```
select *, COALESCE(weight_in_lbs, 90.45) as corrected_weights from entries;
```

We should get an output like:

name text	weight_in_lbs double precision	age_in_years smallint	corrected_weights double precision
Christina	80.6		80.6
Matthews		19	90.45
Gilbert	100.3	21	100.3

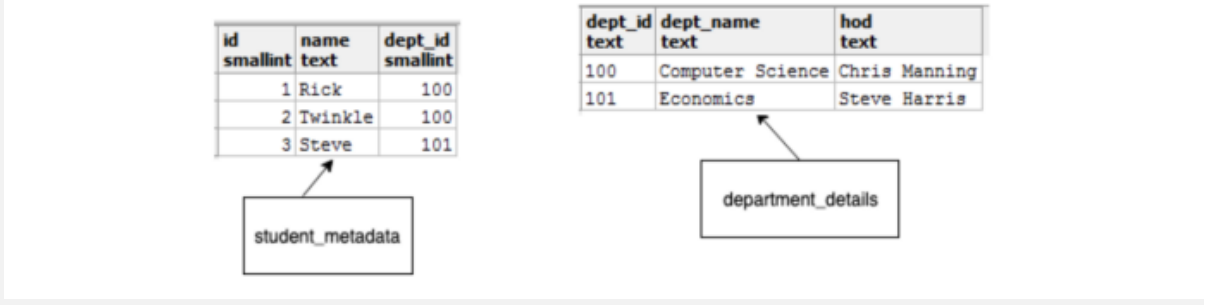
Now we can apply the AVG() again:

```
select avg(corrected_weights) from
(select *, COALESCE(weight_in_lbs, 90.45) as corrected_weights from entries) as subquery;
```

This is a much more accurate result than the earlier one. Let's now study another problem that can take place if we have mismatches in the column data types.

2.) Table Joins

Consider we are working with the following tables student_metadata & department_details:



We can see in the student_mtadata table, dept_id is of integer type, & in the department_details table, it is present in text type. Now, suppose, we want to join these two tables & want to produce a report which will contain the following columns:

- id
- name
- dept_name

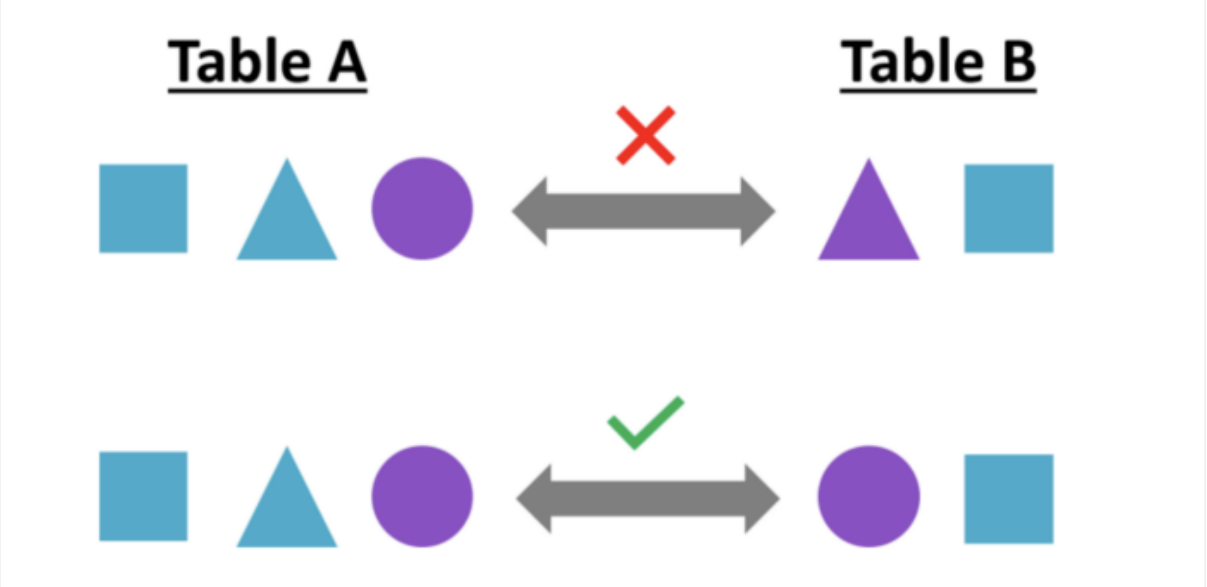
To do this, we run this query:

```
select id, name, dept_name from
student_metadata s join department_details d
on s.dept_id = d.dept_id;
```

We will encounter this error, then:

"*ERROR: operator does not exist: smallint = text*"

Here's an amazing infographic that depicts this problem:



This is happening because the data types are not getting matched while joining the two tables. Here, we can CAST the dept_id column in the department_details table to integer while joining the tables. Here's how to do that:

```
select id, name, dept_name from
student_metadata s join department_details d
on s.dept_id = cast(d.dept_id as smallint);
```

And you get your desired report:

id smallint	name text	dept_name text
1	Rick	Computer Science
2	Twinkle	Computer Science
3	Steve	Economics

Let's now discuss how strings can be present in messy forms, their problems & ways to deal with them.

Messy Strings & Cleaning Theme

String values are also very common. Let's start this section by looking at the values of a column dept_name (denoting department names) taken from a table named student_details:

id smallint	name text	dept_name text
1	Sayak	I.T
2	Hugo	Information Technology
2	Stephen	i.t
4	David	C.S.E
5	Andrew	C.S.E
6	Emily	E.C.E

String values like the above can cause a lot of unexpected problems. I.T, Information Technology & i.t all mean the same department, i.e. *Information Technology* & suppose the specification document requires the values to be present as I.T only. Now, say, you want to count the number of students belonging to the department of I.T. & you run this query:

```
select dept_name, count(dept_name) as student_count
from student_details
group by dept_name;
```

And we get:

dept_name text	student_count bigint
Information Technology	1
i.t	1
C.S.E	2
I.T	1
E.C.E	1

Is this an accurate report? — No! So, how can we address this problem?

Let's first identify the problem in a more detailed way:

- We have `Information Technology` as a value which should be converted to `I.T`
- We have `i.t` as another value which should be converted to `I.T`.

In the first case, we can `REPLACE` the value `Information Technology` to `I.T`, & in the second case, you convert the character to `UPPERcase`. We can accomplish this in a single query though it is advised to address this kind of problems in a step-by-step fashion. Here's the query to address the problem:

```
select upper(replace(dept_name, 'Information Technology', 'I.T')) as dept_cleaned,
count(dept_name) as student_count
from student_details
group by dept_cleaned;
```

And the report:

dept_cleaned text	student_count bigint
C.S.E	2
I.T	3
E.C.E	1

Let’s now discuss some examples where date values can be messy and what we can do to clean them.

Messy Dates & Cleaning Them

Consider we are working with a table named employees which contains a column called birthdate but not in an appropriate date type. Now, we want to execute queries with dedicated date functions like DATE_PART(). We will not be able to do that until & unless we CAST the birthdate column to date type. Let's see this in action.

Consider the birthdates to be in the YYYY-MM-DD format.

Here’s what the employees table looks like:

employee_id smallint	employee_name text	dept_name text	birthdate text
10	Ramesh	Retail	1990-11-08
15	Ricky	Data & Analyticss	1995-01-23
15	Richard	Customer Acquisition	1987-05-21

Now, we run the following query to extract the months from the birthdates:

```
select date_part('month', birthdate) from employees;
```

And we instantly get this error:

"*ERROR: function date_part(unknown, text) does not exist*"
Along with the error, we get a very good hint also:

"*HINT: No function matches the given name and argument types. You might need to add explicit type casts.*"

Let's follow the hint and `CAST` the birthdate to the appropriate `date` type & then apply DATE_PART():

```
select date_part('month', CAST(birthdate AS date)) as birthday_months from employees;
```

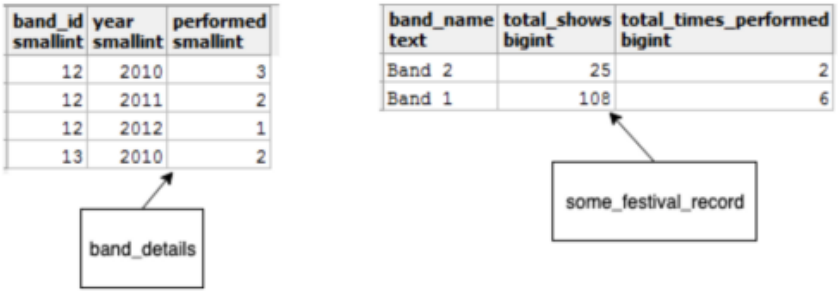
We should get the result:

birthday_months double precision
11
1
5

Let's now proceed to the pre-concluding section of this tutorial where we will study the effects of data duplications & how we can tackle them.

Data Duplications: Causes, Effects & Solutions

In this section, we will be studying some of the most common causes which lead to data duplications. We will also see their effects & some of the ways using which prevent them. Consider the following two tables `band_details` & `some_festival_record`:



The table band_details conveys information about musical bands, it contains their identifiers, names & the total shows they have delivered. On the other hand, the table some_festival_record portrays a hypothetical music festival & contains records about the bands performed there.

Now, suppose we want to produce a report which should contain band names, their show counts, & the total number of times they have performed at the festival. INNER joining is needed here. We run the following query:

```
select band_name, sum(total_show_count) as total_shows, sum(performed) as total_times_performed
from band_details b join some_festival_record s
on b.id = s.band_id
group by band_name;
```

And the query produces:

band_name text	total_shows bigint	total_times_performed bigint
Band 2	25	2
Band 1	108	6

Don't we think the total_shows values are erroneous here? Because from the band_details table, we know that Band_1 has delivered a total of 36 shows. Then what went wrong here? **Duplicates**!

While joining the two tables, we mistakenly aggregated the total_show_count column which caused a data duplication in the intermediate join results. If we remove the aggregation & modify the query accordingly we should get the desired results:

```
select band_name, total_show_count, sum(performed) as total_times_performed
from band_details b join some_festival_record s
on b.id = s.band_id
group by band_name, total_show_count;
```

We get our expected results now:

band_name text	total_show_count smallint	total_times_performed bigint
Band 2	25	2
Band 1	36	6

There is another way to prevent data duplication, i.e. add another field in our JOIN clause so that the tables get joined on stricter conditions.

Let's say we have the following employee table:

id	first_name	last_name	title	age	wage	hire_date
1	Amy	Jordan	Ms	24	15	2019-04-27
2	Bill	Tibb	Mr	61	28	2012-05-02
3	Bill	Sadat		18	12	2019-11-08
4	Christine	Riveles	Mrs	36	20	2018-03-30
5	David	Guerin	Honorable	28	20	2016-11-02

Our sample table, `employees`

Some other useful techniques are:

1. **CASE WHEN**

   CASE WHEN is a function that allows a query to map various values in a column to other values. The general format of a CASE WHEN statement is:

```
CASE
  WHEN condition1 THEN value1
  WHEN condition2 THEN value2
  ...
  WHEN condition3 THEN valueX
  ELSE else_value
END
```

As an example, let's say you wanted to return all rows for employees from the employees table. Additionally, you would like to add a column that labels an employee as being a New employee if they were hired after 2019-01-01. Otherwise, it will mark the employee as a Standard

employee. This column will be called employee_type. We can create this table by using a CASE WHEN statement as follows:

```
SELECT  *,
  CASE
    WHEN hire_date >= '2019-01-01' THEN 'New'
    ELSE 'Standard'
  END AS employee_type
FROM
  Employees;
```

2. COALESCE

Another useful technique is to replace NULL values with a standard value. This can be accomplished easily by means of the COALESCE function. COALESCE allows you to list any number of columns and scalar values, and, if the first value in the list is NULL, it will try to fill it in with the second value. The COALESCE function will keep continuing down the list of values until it hits a non-NULL value. If all values in the COALESCE function are NULL, then the function returns NULL. To illustrate a simple usage of the COALESCE function, let's say we want a list of the names and titles of our employees. However, for those with no title, we want to instead write the value 'NO TITLE'. We can accomplish this request with COALESCE:

```
SELECT
  first_name,
  last_name,
  COALESCE(title, 'NO TITLE') AS title
FROM
  employees;
```

3. **NULLIF**

NULLIF is, in a sense, the opposite of COALESCE. NULLIF is a two-value function and will return NULL if the first value equals the second value. As an example, imagine that we want a list of the names and titles of our employees. However, this time, we want to replace the title 'Honorable' with NULL. This could be done with the following query:

```
SELECT
```

```
  first_name,
  last_name,
  NULLIF(title, 'Honorable') AS title
FROM
  employees;
```

This will blot out all mentions of 'Honorable' from the title column and give the following result

first_name	last_name	title
Amy	Jordan	Ms
Bill	Tibb	Mr
Bill	Sadat	
Christine	Riveles	Mrs
David	Guerin	

4.LEAST / GREATEST

Two functions often come in handy for data preparation are the LEAST and GREATEST functions. Each function takes any number of values and returns the least or the greatest of the values, respectively. Simple use of this variable would be to replace the value if it's too high or low. For example, say the minimum wage increased to $15/hour and we need to change the wages of any employee earning less than that. We can create this using the following query:

```
SELECT
  id,
  first_name,
  last_name,
  title,
  age,
  GREATEST(15, wage) as wage,
  hire_date
FROM
  employees;
```

This query will give the following output:

id	first_name	last_name	title	age	wage	hire_date
1	Amy	Jordan	Ms	24	15	2019-04-27
2	Bill	Tibb	Mr	61	28	2012-05-02
3	Bill	Sadat		18	15	2019-11-08
4	Christine	Riveles	Mrs	36	20	2018-03-30
5	David	Guerin	Honorable	28	20	2016-11-02

Output from query using `GREATEST`

As you can see, Bill Sadat's wage has increased from $12 to $15.

5. Casting

Another useful data transformation is to change the data type of a column within a query. This is usually done to use a function only available to one data type, such as text, while working with a column that is in a different data type, such as a numeric. To change the data type of a column, you simply need to use the `column::datatype` format, where the `column` is the column name, and `datatype` is the data type you want to change the column to. For example, to change the age in the `employees` table to a text column in a query, use the following query:

```
SELECT
  first_name,
  last_name,
  age::TEXT
FROM
  Employees;
```

This will convert the `age` column from an integer to text. You can now apply text functions to this transformed column. There is one final catch; not every data type can be cast to a specific data type. For instance, `DateTime` cannot be cast to float types. Your SQL client will throw an error if you ever make an unexpected strange conversion. We can also use CAST() and CONVERT() functions like this:-

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

OR Convert an expression from one data type to another (datetime):

```
SELECT CONVERT(datetime, '2017-08-25');
```

6. **DISTINCT**

Often, when looking through a dataset, you may be interested in determining the unique values in a column or group of columns. This is the primary use case of the `DISTINCT` keyword. For example, if you wanted to know all the unique first names in the `employees` table, you could use the following query:

```
SELECT
    DISTINCT first_name
FROM
    employees;
```

This gives the following result:

first_name
Amy
Bill
Christine
David

You can also use DISTINCT with multiple columns to get all distinct column combinations present.

Cleaning strings

Most of the functions presented in this lesson are specific to certain data types. However, using a particular function will, in many cases, change the data to the appropriate type. `LEFT`, `RIGHT`, and `TRIM` are all used to select only certain elements of strings, but using them to select elements of a number or date will treat them as strings for the purpose of the function.

LEFT, RIGHT, and LENGTH

Let's start with `LEFT`. 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)`.

As a practical example, we can see that the `date` field in this dataset begins with a 10-digit date, and includes the timestamp to the right of it. The following query pulls out only the image: "/images/og-images/sql-facebook.png" date:

```
SELECT incidnt_num,
       date,
       LEFT(date, 10) AS cleaned_date
  FROM tutorial.sf_crime_incidents_2014_01
```

`RIGHT` does the same thing, but from the right side:

```
SELECT incidnt_num,
       date,
       LEFT(date, 10) AS cleaned_date,
       RIGHT(date, 17) AS cleaned_time
  FROM tutorial.sf_crime_incidents_2014_01
```

`RIGHT` works well in this case because we know that the number of characters will be consistent across the entire `date` field. If it wasn't consistent, it's still possible to pull a string from the right side in a way that makes sense. The `LENGTH` function returns the length of a string. So `LENGTH(date)` will always return `28` in this dataset. Since we know that the first 10 characters will be the date, and they will be followed by a space (total 11 characters), we could represent the `RIGHT` function like this:

```
SELECT incidnt_num,
       date,
       LEFT(date, 10) AS cleaned_date,
       RIGHT(date, LENGTH(date) - 11) AS cleaned_time
  FROM tutorial.sf_crime_incidents_2014_01
```

When using functions within other functions, it's important to remember that the innermost functions will be evaluated first, followed by the functions that encapsulate them.

TRIM

The `TRIM` function is used to remove characters from the beginning and end of a string. Here's an example:

```
SELECT location,
       TRIM(both '()' FROM location)
  FROM tutorial.sf_crime_incidents_2014_01
```

The `TRIM` function takes 3 arguments. First, you have to specify whether you want to remove characters from the beginning ('leading'), the end ('trailing'), or both ('both', as used above). Next you must specify all characters to be trimmed. Any characters included in the single quotes will be removed from both beginning, end, or both sides of the string. Finally, you must specify the text you want to trim using `FROM`.

POSITION and STRPOS

`POSITION` allows you to specify a substring, then returns a numerical value equal to the character number (counting from left) where that substring first appears in the target string. For example, the following query will return the position of the character 'A' (case-sensitive) where it first appears in the `descript` field:

```
SELECT incidnt_num,
       descript,
       POSITION('A' IN descript) AS a_position
  FROM tutorial.sf_crime_incidents_2014_01
```

You can also use the `STRPOS` function to achieve the same results—just replace `IN` with a comma and switch the order of the string and substring:

```
SELECT incidnt_num,
```

```
       descript,
       STRPOS(descript, 'A') AS a_position
  FROM tutorial.sf_crime_incidents_2014_01
```

Importantly, both the `POSITION` and `STRPOS` functions are case-sensitive. If you want to look for a character regardless of its case, you can make your entire string a single by using the `UPPER` or `LOWER` functions described below.

SUBSTR

`LEFT` and `RIGHT` both create substrings 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`. The syntax is `SUBSTR(*string*, *starting character position*, *# of characters*)`:

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

  FROM tutorial.sf_crime_incidents_2014_01
```

CONCAT

You can combine strings from several columns together (and with hard-coded values) using `CONCAT`. Simply order the values you want to concatenate and separate them with commas. If you want to hard-code values, enclose them in single quotes. Here's an example:

```
SELECT incidnt_num,

       day_of_week,

       LEFT(date, 10) AS cleaned_date,

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

  FROM tutorial.sf_crime_incidents_2014_01
```

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

```
SELECT incidnt_num,

       day_of_week,

       LEFT(date, 10) AS cleaned_date,

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

  FROM tutorial.sf_crime_incidents_2014_01
```

Changing case with UPPER and LOWER

Sometimes, you just don't want your data to look like it's screaming at you. You can use `LOWER` to force every character in a string to become lower-case. Similarly, you can use `UPPER` to make all the letters appear in upper-case:

```
SELECT incidnt_num,

       address,

       UPPER(address) AS address_upper,

       LOWER(address) AS address_lower

  FROM tutorial.sf_crime_incidents_2014_01
```

Turning strings into dates

Dates are some of the most commonly screwed-up formats in SQL. This can be the result of a few things:

- The data was manipulated in Excel at some point, and the dates were changed to MM/DD/YYYY format or another format that is not compliant with SQL's strict standards.
- The data was manually entered by someone who use whatever formatting convention he/she was most familiar with.
- The date uses text (Jan, Feb, etc.) instead of numbers to record months.

In order to take advantage of all of the great date functionality (`INTERVAL`, as well as some others you will learn in the next section), you need to have your date field formatted appropriately. This often involves some text manipulation, followed by a `CAST`. Let's revisit the answer to one of the practice problems above:

```
SELECT incidnt_num,

       date,

       (SUBSTR(date, 7, 4) || '-' || LEFT(date, 2) ||

        '-' || SUBSTR(date, 4, 2))::date AS cleaned_date

  FROM tutorial.sf_crime_incidents_2014_01
```

This example is a little different from the answer above in that we've wrapped the entire set of concatenated substrings in parentheses and cast the result in the `date` format. We could also cast it as a timestamp, which includes additional precision (hours, minutes, seconds). In this case, we're not pulling the hours out of the original field, so we'll just stick to `date`.

Turning dates into more useful dates

Once you've got a well-formatted date field, you can manipulate it in all sorts of interesting ways. To make the lesson a little cleaner, we'll use a different version of the crime incidents dataset that already has a nicely-formatted date field:

```
SELECT *

  FROM tutorial.sf_crime_incidents_cleandate
```

You've learned how to construct a date field, but what if you want to deconstruct one? You can use `EXTRACT` to pull the pieces apart one-by-one:

```
SELECT cleaned_date,

       EXTRACT('year'   FROM cleaned_date) AS year,

       EXTRACT('month'  FROM cleaned_date) AS month,

       EXTRACT('day'    FROM cleaned_date) AS day,

       EXTRACT('hour'   FROM cleaned_date) AS hour,

       EXTRACT('minute' FROM cleaned_date) AS minute,

       EXTRACT('second' FROM cleaned_date) AS second,

       EXTRACT('decade' FROM cleaned_date) AS decade,

       EXTRACT('dow'    FROM cleaned_date) AS day_of_week

  FROM tutorial.sf_crime_incidents_cleandate
```

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 do care about the week (or month, or quarter) 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 `DATE_TRUNC` 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('week'   , cleaned_date) AS week,

       DATE_TRUNC('day'    , cleaned_date) AS day,

       DATE_TRUNC('hour'   , cleaned_date) AS hour,

       DATE_TRUNC('minute' , cleaned_date) AS minute,

       DATE_TRUNC('second' , cleaned_date) AS second,

       DATE_TRUNC('decade' , cleaned_date) AS decade

  FROM tutorial.sf_crime_incidents_cleandate
```

Thanks for going through this content. Hope you liked it and helped you in some ways with the above data cleaning techniques.