Lab on window functions and CTEs

This exercise is available as a google form https://forms.gle/Y2i6ShyzQkVcpZzr6

The goal of this worksheet is to practice using window functions and CTEs on the `treesdb` database.

You are a newly recruited analyst working in the Paris administration for parks. You are given this dataset of trees in Paris and vicinity.

Your job is to get insights on the data to facilitate tree management and improve the dataset

Load the data

At this point you should have the V02 version of the trees database (`treesdb_v02`) loaded in PostgreSQL.

If that's the case connect to it (add `-u postgres` if needed)

```Bash
psql -h localhost -d treesdb_v02
```

The database has 2 tables : `trees` and `version` . The `trees` table has an index `id` .

If you don't have the database loaded, or if in doubt, recreate the database `treesdb_v02` and restore the data. The dataset file `treesdb_v02.01.sql.backup` is available in the GitHub.

Part I: data quality and the stage column

You are not satisfied with the values in the `stage` column which are:

```sql
select count(*) as n, stage from trees group by stage order by n desc;
   n   |        stage
-------+---------------------
 79627 | Adulte
 46742 | [null]
 38915 | Jeune (arbre)
 38765 | Jeune (arbre)Adulte
  7290 | Mature
```

Too many `NULL` values and it's not clear what `Jeune (arbre)Adulte` really stands for. Is it a *jeune* (young) or an *adulte* tree?

So we want to replace the `stage` values by some new categorical column that we call `maturity`.

- For each tree, we calculate the max height for its type (genre, species).
- The ratio of height over max_height sets the `maturity`.

The maturity category for a given tree type (genre, species) and related `max_height` is given by

ratio = height / max_height	maturity
ratio < 0.25	young
0.25 <= ratio < 0.5	young adult
0.5 <= ratio < 0.75	adult
0.75 <= ratio	mature

These thresholds (0.25, 0.75) are totally arbitrary and may not reflect reality. Also we assume that tree growth is linear (although that's debatable)

1. max height per tree type

You first need to calculate the max(height) per type of tree as a standalone column.

Keep in mind that

- we want the max(height) for each genre and species
- we only consider not null values for genre and species

Write the query that returns

- for each tree, the columns: `id`, `genre`, `species`, `height` and

`max_height`

- alphabetically ordered by `genre` , `species` and by `height` and `max_height` decreasing

Hint: use `partition by genre, species`

The first rows of the result should look like

```Bash
 id    | genre   | species   | height | max_height
-------+---------+-----------+--------+------------
 43053 | Abelia  | triflora  |      6 |          6
 83127 | Abies   | alba      |     22 |         22
 97141 | Abies   | alba      |     20 |         22
 88940 | Abies   | alba      |     20 |         22
 73055 | Abies   | alba      |     20 |         22
```

> Write the query

2. Create a new `maturity` column

Create a new text column called `maturity` with data type VARCHAR(50) in the trees table.

To add a column to an existing table the query follows

```SQL
ALTER TABLE table_name ADD COLUMN column_name VARCHAR(50);
```

> Write the query

3. fill maturity with the right values : young, young adult, adult, mature

Use the query where you calculated `max_height` for each tree genre and species, as a named subquery and update the `maturity` column with the calculated maturity values.

The rule is

ratio	maturity
ratio < 0.25	young
0.25 <= ratio < 0.5	young adult
0.5 <= ratio < 0.75	adult
0.75 <= ratio	mature

where ratio = height / max_height

Hint: use `CASE WHEN` in your query to map the ratio to a maturity category.

See documentation

Hint: You can use the query structure

```SQL
WITH temp AS (
        -- some SQL query
)
UPDATE table_name
SET column_name = CASE
    WHEN (some cond 1) THEN 'label 1'
    ...
    ELSE 'other label'
END
from temp
where temp.id = table_name.id;
```

> Write the query

4. Is that maturity in accordance with the original stage values ?

Although the original stage column is showing very poor data quality we hope to keep some consistency between the new `maturity` categories and the original `stage` categories.

Let's look at diverging values for `stage = 'Jeune (arbre)'` and / or `maturity = 'young'`. Hopefully most trees in the young maturity category should also have 'Jeune (arbre)' for stage value.

The final goal in this part is to write the query that returns

- the percentage of trees that have either (non NULL values for stage only)
  - stage = 'Jeune (arbre)' AND maturity != 'young'
  - maturity = 'young' AND stage != 'Jeune (arbre)'

- for the 10 most common genre of trees (Platanus to Celtis)

The result of that query should be

```Bash
    genre     | total_trees | mismatch_trees | mismatch_percentage
--------------+-------------+----------------+---------------------
 Celtis       |        3824 |           2965 |               77.54
 Aesculus     |       22360 |          17043 |               76.22
 Platanus     |       39729 |          30107 |               75.78
 Pyrus        |        3618 |           2333 |               64.48
 Acer         |       13198 |           5508 |               41.73
 Prunus       |        4907 |           1805 |               36.78
 Quercus      |        3512 |           1034 |               29.44
 Fraxinus     |        4206 |            930 |               22.11
 Styphnolobium |        9908 |           1966 |               19.84
 Tilia        |       17543 |           2241 |               12.77
```

Let's build the query in steps

In all queries filter out null values for genre, maturity and stage.

1. 1st subquery named `genre_totals` : 10 most common genre of trees (Platanus to Celtis)
2. 2nd subquery named `genre_mismatch` : trees grouped by genre with either:
    - stage = 'Jeune (arbre)' AND maturity != 'young'
    - maturity = 'young' AND stage != 'Jeune (arbre)'
3. finally use these 2 queries to find the `mismatch_percentage` per genre
4. order by `mismatch_percentage` desc.

The structure of the final query may look like

```SQL
WITH genre_totals AS (
    --- 1st query
),
genre_mismatch AS (
    -- 2nd query
)
SELECT
    -- some columns
FROM genre_totals gt
JOIN genre_mismatch gm ON gt.genre = gm.genre
```

> Write the query

Part II: Find tall and large trees

The climate change office of the Mairie de Paris wants to find the tallest trees in Paris. Because large trees provide shelter from the heat during heat waves.

They ask you to provide the following list:

For each arrondissement, find the top 3 tallest trees. and for each tree include

- id, arrondissement, height,
- max height in arrondissement
- height rank in arrondissement
- height rank over all trees in Paris

The result of the query should look like:

Bash

```
  id   |  arrondissement   | height | max_height_in_arrdt | height_rank
-------+-------------------+--------+---------------------+------------
  5103 | BOIS DE BOULOGNE  |     45 |                  45 |
165500 | BOIS DE BOULOGNE  |     40 |                  45 |
 87670 | BOIS DE BOULOGNE  |     36 |                  45 |
 93035 | BOIS DE VINCENNES |    120 |                 120 |
100832 | BOIS DE VINCENNES |     35 |                 120 |
 15336 | BOIS DE VINCENNES |     35 |                 120 |
  8722 | BOIS DE VINCENNES |     35 |                 120 |
149302 | BOIS DE VINCENNES |     35 |                 120 |
 58508 | BOIS DE VINCENNES |     33 |                 120 |
136397 | HAUTS-DE-SEINE    |     23 |                  23 |
```

First write the query that calculates the max(height) and ranks over the height partitioned by arrondissement and over all the trees. Use `MAX()` and `DENSE_RANK()` functions.

Then use that query as a named subquery and filter its results on `height_rank_in_arrdt` to get the 3 tallest trees in each arrondissement.

> write the query

Part III find outliers

We want to find crazy values for heights

We suppose that all trees of the same genre and species should have the same height

range.

So we're going to order the trees by genre, species and height

Then using the LAG() function,

- for each tree find the height in the previous row
- if the height of the tree is more than double the previous height
- then the tree can be flagged as an outlier.

> Note: This will only flag the smallest first outlier. In a second pass we can also flag similar trees which are higher than the 1st outlier (we won't do it)

Average height

Write the query that returns the average height per tree per genre and species (Not null)

> Write the query

LAG

Now use the `LAG()` function over `height` with default value the `avg_tree` height per genre and species

- filter out null values for genre and species
- order trees by genre, species

You need to join the main query with the subquery so that you can use the columns from the subquery

The query structure follows

```SQL
WITH avg_heights AS (
    -- some sql
)
SELECT
    -- some columns
    -- LAG( ..., ah.aavg_height) OVER(...)
FROM trees t
JOIN avg_heights ah ON .... -- (genre and species)
-- filter on not null values for genre and species
-- order by
```

> Write the query

Outlier flag

Create a new column outlier as boolean default FALSE

write the query that sets the value of outlier

> if `height` > 2 * `height_lag` then `outlier` is True

Hint: re-use the previous query with the main SELECT as a named query and add an update statement

The query structure should now be like

```SQL
WITH avg_heights AS (
    -- some sql
),
lagged_heights AS (
    -- main select from previous query
)
UPDATE trees
SET outlier = CASE
    WHEN (some condition) THEN true
    ELSE false
END
FROM lagged_heights
WHERE trees.id = lagged_heights.id;
```

> Write the query