Improving the trees dataset

In this document we explore and improve the trees dataset treesdb_v01 that has been loaded from a sql dump.

Here are the columns of the tree table

Column	Type	Collation	Nullable	Default	Storage	Compression	Stats target	Description
idbase	integer				plain			
location_type	character varying				extended			
domain	character varying				extended			
arrondissement	character varying				extended			
suppl_address	character varying				extended			
number	character varying				extended			
address	character varying				extended			
id_location	character varying				extended			
name	character varying				extended			
genre	character varying				extended			
species	character varying				extended			
variety	character varying				extended			
circumference	integer				plain			
height	integer				plain			
stage	character varying				extended			
geo*point*2d	character varying				extended			
remarkable	boolean				plain			

Goal

The trees dataset is not perfect. there are anomalies, missing data

The table data types is not optimal for some columns and we are missing a primary key

This is a good reflection of real world datasets that are never perfect.

In this document we get a sense of the data, deal with some anomalies and transform the table with more appropriate data types.

We also leverage the postGIS extension for spatial data. (in fact that is more complex than previously expected we'll see why)

data analysis

Let's do some data analysis of the dataset.

select a random row from the trees db

solution

SQL

```sql
your query
```

Column	value
idbase	273252
location_type	Arbre
domain	Alignement
arrondissement	PARIS 18E ARRDT
suppl_address	
number	
address	RUE DE LA CHAPELLE
id_location	000602007
name	Tilleul
genre	tilia
species	cordata
variety	Greenspire
circumference	34
height	5
stage	Jeune (arbre)
geo*point*2d	48.89291084026716, 2.359807495821241
remarkable	f

We see that we have

- some categorical columns related to the location : domain, arrondissement
- categorical columns related to the nature of the tree : name, genre, species, variety,
- and also : stage,
- dimensions of the tree : height and circumference (in meters)
- columns related to the location of the tree: address, suppl_address, number, ...
- a `remarkable` flag
- and geo location : geo*point*2d with latitude and longitude of each tree

Let's query the tree table and get a feeling for the values of the different columns

- how many trees per `domain` or `arrondissement`
- how many trees per stage, genre, species ...
- how many trees are remarkable ?
- do all trees have a height and a circumference ?
- what's the average height for different domain, stage or remarkable
- what about the location_type and number columns ?

any other thing you can think of ?

Where's the primary key ?

The table loaded from a csv has no primary key although `idbase` seems like a good candidate. (after all it has *id* in it... must mean something right ?)

Could the `idbase` column be a primary key ?

What's a primary key and what is it used for ?

A **primary key** in SQL is a **unique identifier** for each record in a table.

A primary key is a **constraint** that enforces **uniqueness** and **non-nullability** for the column or *columns* it is applied to.

Having a primary key allows databases to index the table more efficiently, making searches and retrievals faster when accessing records by their primary key value.

foreign key

In relational databases, primary keys are often used as **foreign keys** in other tables. A **foreign key** is a column (or a combination of columns) that references a primary key in another table.

conditions for a primary key

A column is a good candidate as a primary key if:

- It contains unique values for each row.
- It does not contain any NULL values.

and

- It remains consistent and does not change often.

SERIAL

A primary key is usually also **SERIAL**.

> In PostgreSQL, SERIAL is a special data type used for **auto-incrementing** integer columns, commonly employed for primary keys.
>
> When you define a column with SERIAL, PostgreSQL automatically creates a **sequence** and sets it up so that each new row gets the next value from this sequence. (we'll come back to that in a moment)

Based on this definition, the `idbase` column be a good candidate for primary key ?

Also, can it be serial ?

Let's check uniqueness of the values first :

Write a query to find out if some values of idbase have more than 2 rows

solution:

```sql
your query
```

This query returns 2 rows

maybe these are exact duplicates of trees, let's check

You can select the trees that have duplicates `idbase` with

solution:

```sql
your query
```

These trees have the exact same data except for heights and circumference which are different. At this point there's no way to know which is the tree with the true values.

So to make `idbase` the primary we would have to delete the duplicates.

- Check for null values : does the idbase have null values ?
- are the `idbase` values sequential ? (check min and max and total count of trees)

Note: a SERIAL primary does not have to be sequential.

So we have 2 options

- either add a new column `id` as SERIAL primary key
- or transform the `idbase` as SERIAL primary key

The second option requires to 1) manually create a sequence, 2) delete some records.

The 1st option is the easiest one as create the sequence automatically

Always be lazy when you can

(lazy = KISS attitude not *don't do the work* attitude)

so the easiest solution is to create a new serial primary key

which we can do with

```sql
alter table trees add COLUMN id SERIAL PRIMARY KEY;
```

As expected, creating a primary key also creates a sequence ;

```
treesdb=# \d


 Schema |      Name      |   Type    | Owner
--------+--------------+----------+--------
 public | trees         | table     | alexis
 public | trees_id_seq | sequence | alexis
```

Postgres Sequence

> a sequence is a special database object designed to generate a sequence of unique, incremental numbers. It is commonly used to create auto-incrementing values, typically for columns like primary keys.

- **Unique**: Each number in the sequence is guaranteed to be unique.
- **Incremental**: The sequence can increment (or decrement) by a specified value.
- **Independent Object**: A sequence is a separate object in the database and is not directly tied to any table or column, though it is often associated with a column (like SERIAL or BIGSERIAL columns).
- **NEXTVAL Function**: To get the next value in the sequence, you call nextval('sequence_name'), which increments the sequence and returns the next number.
- **START, INCREMENT, and MAXVALUE**: You can specify the starting value, how much to increment by, and an optional maximum value for the sequence.

```sql
\d+ trees_id_seq

  Type   | Start | Minimum |  Maximum   | Increment | Cycles? | Cache
---------+-------+---------+------------+-----------+---------+-------
 integer |     1 |       1 | 2147483647 |         1 | no      |     1
Owned by: public.trees.id
```

and

```sql
\d+ trees

    Column     |        Type        | Collation | Nullable |              Default
---------------+--------------------+-----------+----------+-----------------------------------
 id             | integer            |           | not null | nextval('trees_id_seq'::regclass)
```

notice the `nextval('trees_id_seq'::regclass)` which increments the counter in the sequence each time it is called

also notice the new index

```sql
Indexes:
    "trees_pkey" PRIMARY KEY, btree (id)
```

We shall keep the `idbase` column for future references but we will use `id` as the primary key.

Improving the column types

At this point, all columns are `varchar` (synonym for `text` in postgresSQL) except for the height and circumference (integers)

that does not make sense for columns such as: `remarkable`, or `geo_point_2d` (latitude and longitude)

`remarkable` should be a boolean

What are the values taken by `remarkable` ?

How to transform the column which has 'NON', 'OUI; or '' (empty string) as boolean : t, f and null

- create a new column `remarkable_bool`
- get the proper values in the new column from the old column
- drop the old column
- rename the new column into the old column name

solution

```sql
your queries (4)
```

`geo_point_2d` **is a also varchar**

(*This is where it gets tricky with the postGIS extension*)

`geo_point_2d` holds the latitude and longitude of the trees. We could transform `geo_point_2d` as an array of floats. However there are specific data types for geo localization.

Using the proper data type will allow us to more easily carry out calculations specific to locations. For instance find the nearest trees, or calculate the distance between trees.

We have a choice to use a native **POINT** data column (available by default in PostgreSQL) or a **PostGIS** geography data type (needs the PostGIS extension)

Comparison between POINT and GEOGRAPHY

Feature	POINT	GEOGRAPHY
Coordinate System	Flat, Cartesian (x, y)	Spherical (longitude, latitude)
Earth's Curvature	Not accounted for	Accounted for
Distance Calculations	Euclidean (straight line on flat plane)	Great circle (curved line on Earth's surface)
Accuracy over Large Distances	Less accurate	Maintains global accuracy
Performance	Generally faster for basic operations	May be slower but more accurate for geographic calculations
Use Cases	Local, small-scale applications (e.g., floor plans, 2D games)	Global, large-scale geographic applications (e.g., GPS, GIS)
Additional Functionality	Limited to basic geometric operations	Extensive GIS functions available through PostGIS
Data Representation	Simple (x, y) coordinates	Complex spheroidal calculations
Spatial Reference System	Typically assumes a flat plane	Supports various geographic coordinate systems (e.g., WGS84)
Storage Size	Smaller	Larger due to additional metadata

Let's use the extension PostGIS.

We won't go into details about PostGIS

Here is the site and a list of tutorials if you need to go deeper

[PostGIS documentation]()

Install the PostGIS extension

First we need to install the PostGIS extension if it's not installed yet.

Check if PostGIS is installed or not with

```sql
SELECT * FROM pg_extension WHERE extname = 'postgis';
```

if this returns 0 rows you need to install PostGIS on the server and then activate it

install PostGIS on Windows

Install PostGIS:

1. Use the Stack Builder application that comes with PostgreSQL.
2. Launch Stack Builder and select your PostgreSQL installation.
3. In the "Spatial Extensions" section, select PostGIS.
4. Follow the prompts to download and install PostGIS.

install PostGIS on Ubuntu or Debian

first, always start with `shell sudo apt update`

Check the postgreSQL version with

```
psql --version
```

Then (replace with the version of postgreSQL you see)

```
apt search postgresql-16 | grep postgis
```

and install the version that was found

```
sudo apt install postgresql-16-postgis-3
```

install PostGIS on mac

```
brew install postgis
brew restart
```

Note

Installing postGIS with `brew install postgis` on Mac, requires postgres14. The command will even install postgres@14 if it's not on the system.

If you have already installed postgres16, you'd have to install postgres14 and things will probably become messy

An alternative is to use the [Postgres app](#) that bundles postgres164 and postgis 3.4.

I haven't tried. This will probably also require to uninstall your the postgres already installed.

So no quick and easy way to install postgis with postgres16 on Mac at this point.

If you're in that situation (existing postgres16 on Mac) just switch to using the postgres native Point data type. see below

activate the extension

In the psql console activate the extension with:

```sql
CREATE EXTENSION postgis;
```

now connect with psql and check that postGis is installed

```sql
SELECT * FROM pg_extension WHERE extname = 'postgis';
```

---

what follows assumes you have postGIS extension installed and activated which is not the case for everyone (me included).

Let's skip to the next section where we transform the geo*point*2d to a native POINT data type.

Transform the geo*point*2d from varchar to GEOGRAPHY

Add a column with the right data type

```sql
ALTER TABLE trees ADD COLUMN geo_point_geography GEOGRAPHY(POINT, 4326);
```

Update the new column with data from the existing `geo_point_2d` column
`sql UPDATE trees SET geo_point_geography = ST_SetSRID(ST_MakePoint( SPLIT_PART(geo_point_2d, ',', 2)::float, SPLIT_PART(geo_point_2`

and create an index `sql CREATE INDEX idx_trees_geography ON trees USING GIST (geo_point_geography);`

Closest trees

Now we can find the N (=10) closest trees to a given tree

```sql
WITH given_tree AS (
  SELECT id, geo_point_geography
  FROM trees
  WHERE id = 1234  -- Replace with the ID of your reference tree
)
SELECT t.id, t.geo_point_geography,
       ST_Distance(t.geo_point_geography, gt.geo_point_geography) AS distance
FROM trees t, given_tree gt
WHERE t.id != gt.id
ORDER BY t.geo_point_geography <-> gt.geo_point_geography
LIMIT 9;  -- 10-1, as we're excluding the reference tree
```

We use `ST_Distance()` function to calculate the great-circle distance between two points on the Earth's surface. The distance is returned in meters.

We also use the `<->` operator in the ORDER BY clause as a fast approximation for initial ordering, which PostGIS then refines.

Note the structure of the query.

```sql
with relation_name as (
    select statement
)
select columns from table, relation_name
```

The query is a CTE or Common Table Expressions

-> modify the query so that it takes a lat long instead of a tree_id

you can use that address to [lat long converter]

-- Query to find N nearest trees given a latitude and longitude -- Using a CTE to define the location

transform a set of lat long to a geography type with

```sql
SELECT ST_SetSRID(ST_MakePoint(-74.0060, 40.7128), 4326)::geography AS geog
```

so the query becomes

```sql
WITH location AS (
    SELECT ST_SetSRID(ST_MakePoint(-74.0060, 40.7128), 4326)::geography AS geog
)
SELECT
    t.id,
    ST_Distance(t.geo_point_geography, location.geog) AS distance_meters,
    ST_Y(t.geo_point_geography::geometry) AS tree_lat,
    ST_X(t.geo_point_geography::geometry) AS tree_long
FROM
    trees t,
    location
ORDER BY
    t.geo_point_geography <-> location.geog
LIMIT 5;
```

Transform the geo*point*2d from varchar to POINT

Point is a native data type in postgres

see https://www.postgresql.org/docs/16/datatype-geometric.html#DATATYPE-GEOMETRIC-POINTS

- We add a column `geolocation` with type POINT.
- Update the new column with POINT values : use the functiosn `TRIM(SPLIT_PART(geo_point_2d, ',', n))` to extract the latitude and longitude
- delete the original `geo_point_2d` column

solution

```sql
your 4 queries
```

Note that the lat and longitude have been swapped.

- In geo*point*2d (String representation):
    - The order is typically (latitude, longitude). This is a common human-readable format, often used in everyday applications and GPS coordinates.
- In geolocation (Point data type):
    - The order is (x, y), which for geographic coordinates translates to (longitude, latitude). This is the standard for many geographic information systems and spatial databases.

We can verify that this makes sense (use google maps)

Closest tree

Given a tree index (id = 1234), find the N closest trees

hints: * calculate the euclidian distance between 2 points of coorddinates x, y : geolocation[0], geolocation[1] * at the Paris coordinates, 1 latitude degree is equivalent to 73km and 1 longitude degree = 111km.

so to calculate the distance between A dn B with coords (a,b) and (c,d)

d in meters = SQRT(((a - c) * 73000) ^2 + ((b - d) * 111000)^2)

solution

N = 10

```sql
WITH given_tree AS (
    your query
)
SELECT
  t.id,
  t.geolocation,
  ROUND(
    SQRT(
      POW((t.geolocation[0] - gt.geolocation[0]) * 73000, 2) +
      POW((t.geolocation[1] - gt.geolocation[1]) * 111000, 2)
    )
  ) AS distance_meters
FROM trees t, given_tree gt
WHERE t.id != gt.id
ORDER BY t.geolocation <-> gt.geolocation
LIMIT 9;
```

This query directly calculates the euclidian distance to find the distance between 2 points and applies different scales to latitude and longitude (111000 and 73000 respectively).

We can then adapt the query to find the trees near a given location if we know its coordinates

Some common queries

To finish our work on the trees table, we'd like to flag trees that have anomalies

So let's create a BOOLEAN column that indicates that there's an anomaly with a tree record.

by the way: the circumference is in centimeters while the height is in meters and in the original dataset, both are recorded as ints (no decimal points are available)

solution

```sql
your query
```

Then find some weird trees

- find the trees that are too tall
- same thing for the circumference

to find anomalies in the circumference, you can convert the circumference to the diameter with

solution

```sql
your query
```

You create a new diameter column with

solution

```SQL
your query
```

and update the diameter column with **solution**

```SQL
your query
```

and find trees that have a insanely high diameter

Update the anomaly column with these trees

Also set the anomaly column to true for duplicates of `idbase`

Are there other anomalies such as duplicates addresses or zero values for height or circumference / diameter ?

Although zero values for height and circumference could simply indicate a young small tree whose measures have been rounded down a bit harshly.

To detect anomalies for a given column, you can get a good insight about a variable distribution by writing a query to find : min, max, average, median, 95 and 5 percentiles for a given float column. this mimics the `df.describe()` in pandas data frames.

You can use the function `PERCENTILE_CONT(p)` with `p` as the percentile value (0.5 for median, 0.9, 0.99 etc ...)

solution

```SQL
your query
```

It may be difficult to decide if the height or diameter of a tree is an anomaly or not.

for instance the tree 187635 has a height of 98m.

column	value
idbase	2018097
location_type	Arbre
domain	Alignement
arrondissement	PARIS 18E ARRDT
suppl_address	108V
number	
address	RUE DE LA CHAPELLE
id_location	2002004
name	Platane
genre	Platanus
species	x hispanica
variety	
circumference	68
height	**98**
stage	Jeune (arbre)
geo*point*2d	48.89815810816667, 2.3591531336170086
id	187635
remarkable	f
diameter	21.645072260497766
geolocation	(2.3591531336170086,48.89815810816667)

that's a lot but is it a valid height for a tree? Are they trees that tall in Paris ?

We can investigate in 2 ways

input the coordinates in google maps and look at the photo of the street

or check the height of nearby trees with the closest trees query

```sql
WITH given_tree AS (
  SELECT id, geolocation
  FROM trees
  WHERE id = 187635
)
SELECT
  t.id,
  t.height,
  t.geolocation,
  ROUND(
    SQRT(
      POW((t.geolocation[0] - gt.geolocation[0]) * 73000, 2) +
      POW((t.geolocation[1] - gt.geolocation[1]) * 111000, 2)
    )
  ) AS distance_meters
FROM trees t, given_tree gt

ORDER BY t.geolocation <-> gt.geolocation
LIMIT 9;
```

All surrounding trees have a height of 5 to 16 meters. So 98 meters is not a valid measurement.

Solution

the 100m threshold is arbitrary. For a real data analysis and we would have to find more relevant thresholds.

solution

```sql
your query
```

and to flag the trees with duplicate idabase

solution

```sql
your query
```

In the end we have 851 trees with anomaly measurements.

Finally let's dump the database

using pgAdmin

...

Recap

The table loaded from the csv / sql dump was lacking a primary key, proper datatypes and had many data anomalies

- We checked that the `idbase` column was not a good choice as a primary key
- Transformed remarkable as a boolean data type,
- installed and activated the postgis extension (when possible)
- which allowed us to transform the geo*point*2d into a postGIS GEOGRAPHY data type and find closest trees given a location
- we also looked at the native POINT data type
- We identified the extreme or missing values of height, circumference and diameter of some trees and flagged these trees.

The current table is much more clean and in a state more compatible with production.

Next we move away from the single table database and start building a proper relational database from that dataset.