Let's normalize treesdb - solution

Load the `tressdb-v02.sql.gz` file into pgAdmin

The goal of this exercise is to transform the flat one table `treesdb` database into a fully normalized database by applying 1NF, 2NF and 3NF forms.

For each entity that you feel would benefit from a stand alone table, the process is

1. create a new table with a primary key
2. inserts values from the trees column
3. add a foreign key to the trees table
4. delete the original column

The entity can be composed of multiple original columns. For instance `address` and `suppl_address`

Column grouping

Here is the solution we will implement

Keep the following columns in the trees table

column	table
id	trees
id_location	trees
idbase	trees
remarkable	trees
anomaly	trees

Location related columns all go into a new `location` table

column	table
address	location
suppl_address	location
arrondissement	location
geolocation	location

`domain` has its own table

and `stage` also has its own table

column	table
domain	tree_domain
stage	tree_stage

Finally the each column related to taxonomy will have its own table but the link between the trees and the taxonomy table will be kept in an intermediate table to keep the relation between the different taxonomy elements

```
CREATE TABLE tree_taxonomy (
    id SERIAL PRIMARY KEY,
    name_id INTEGER REFERENCES tree_name(id),
    species_id INTEGER REFERENCES tree_species(id),
    variety_id INTEGER REFERENCES tree_varieties(id)
    ...
);
```

We keep the tree measurements in the trees table.

location

Let's create a table locations with

```sql
create table locations (
    id serial primary key,
    suppl_address  varchar,
    address        varchar,
    arrondissement varchar,
    geolocation    point
);
```

To insert into locations the values from trees we do

```sql
INSERT INTO locations (suppl_address, address, arrondissement, geolocat
SELECT  suppl_address, address, arrondissement, geolocation
FROM trees
```

Then we create the `location_id` foreign key

```sql
ALTER TABLE trees
ADD COLUMN location_id INTEGER;
```

normalize addresses

step 1: create the location address

```sql
create table locations (
    id serial primary key,
    suppl_address  varchar,
    address        varchar,
    arrondissement varchar,
    geolocation    varchar
);
```

Step 2: Copy data from trees table to the new locations table

```sql
INSERT INTO locations (suppl_address, address, arrondissement, geolocat
SELECT  suppl_address, address, arrondissement, geolocation
FROM trees
WHERE suppl_address IS NOT NULL
   OR address IS NOT NULL
   OR arrondissement IS NOT NULL
   OR geolocation IS NOT NULL;
```

Step 3: Add a location_id column to the trees table

```sql
ALTER TABLE trees ADD COLUMN location_id INTEGER;
```

step 4: connect location_id to location.id

- it is not sufficient to connect on equality of geolocation sine there are multiple equal geolocations (12 of them)
- so we need identification on the whole address

```sql
SELECT COUNT(*) as tree_count, geolocation::text
FROM locations
GROUP BY geolocation::text
HAVING COUNT(*) > 1
ORDER BY tree_count DESC;
```

A quick check shows that geolocation duplicates all have the same address

```sql
select * from locations where geolocation::text in (SELECT  geolocation
HAVING COUNT(*) > 1) order by geolocation::text asc;
```

so we can delete locations duplicates with

```SQL
WITH numbered_duplicates AS (
    SELECT id, geolocation,
           ROW_NUMBER() OVER (PARTITION BY geolocation::text ORDER BY i
    FROM locations
    WHERE geolocation::text IN (
        SELECT geolocation::text
        FROM locations
        GROUP BY geolocation::text
        HAVING COUNT(*) > 1
    )
)
DELETE FROM locations
WHERE id IN (
    SELECT id
    FROM numbered_duplicates
    WHERE row_num > 1
);
```

and check that the query should return 0

```SQL
SELECT COUNT(*) as tree_count, geolocation::text
FROM locations
GROUP BY geolocation::text
HAVING COUNT(*) > 1
ORDER BY tree_count DESC;
```

As expected the trees table has 12 more rows than the locations table.

Note: Another way to avoid duplicates would have been to cast geolocation as text and use `insert from select distinct` in the query above and then to recast geolocation as point

so now we can associate trees location_id with location.id based on geolocation

```SQL
UPDATE trees t
SET location_id = l.id
FROM locations l
WHERE (t.geolocation::text = l.geolocation::text );
```

verify that 12 rows in trees have duplicate location_id

```SQL
select count(*) as n, location_id from trees group by location_id havin
```

finally add foreign key constraint in the trees db

```SQL
ALTER TABLE trees
ADD CONSTRAINT fk_location
FOREIGN KEY (location_id)
REFERENCES locations(id);
```

before dropping original columns make sure that the addresses and geolocation match

this query should return 0 rows

```SQL
select t.*, l.*
from trees t
join locations l on l.id = t.location_id
where t.geolocation::text != l.geolocation::text
limit 10;
```

Finally drop location columns from trees

```SQL
alter table trees drop column address;
alter table trees drop column suppl_address;
alter table trees drop column arrondissement;
alter table trees drop column geolocation;
```

domain and stage

create the table `tree_domains`

```SQL
create table tree_domains(
    id serial primary key,
    domain varchar
);
```

fill in data from trees into tree_domains

```SQL
insert into tree_domains (domain)
select distinct domain from trees
where domain is not null;
```

add foreign key column in trees

`sql ALTER TABLE trees ADD COLUMN domain_id INTEGER;`

update tree_domains

```SQL
UPDATE trees t
SET domain_id = td.id
FROM tree_domains td
WHERE (t.domain = td.domain );
```

foreign key

```SQL
ALTER TABLE trees
ADD CONSTRAINT fk_tree_domain
FOREIGN KEY (domain_id)
REFERENCES tree_domains(id);
```

check

```SQL
select t.*
from trees t
join tree_domains td on td.id = t.domain_id
where t.domain != td.domain;
```

drop domain column in trees

```SQL
alter table trees drop column domain;
```

Simlarly for stages

```sql
create table tree_stages(
    id serial primary key,
    stage varchar
);

insert into tree_stages (stage)
select distinct stage from trees
where stage is not null;

ALTER TABLE trees ADD COLUMN stage_id INTEGER;

UPDATE trees t
SET stage_id = ts.id
FROM tree_stages ts
WHERE (t.stage = ts.stage );

ALTER TABLE trees
ADD CONSTRAINT fk_tree_stage
FOREIGN KEY (stage_id)
REFERENCES tree_stages(id);
```

check

```sql
select t.*
from trees t
join tree_stages ts on ts.id = t.stage_id
where t.stage != ts.stage;
```

drop column stage `sql alter table trees drop column stage;`

taxonomy

-- Step 1: Create the new tables CREATE TABLE tree_names (id SERIAL PRIMARY KEY, name VARCHAR(255) UNIQUE NOT NULL);

CREATE TABLE tree_genres (id SERIAL PRIMARY KEY, genre VARCHAR(255) UNIQUE NOT NULL);

CREATE TABLE tree_species (id SERIAL PRIMARY KEY, species VARCHAR(255) UNIQUE NOT NULL);

CREATE TABLE tree_varieties (id SERIAL PRIMARY KEY, variety VARCHAR(255) UNIQUE NOT NULL);

CREATE TABLE tree*taxonomy (id SERIAL PRIMARY KEY, name*id INTEGER REFERENCES tree*names(id), genre*id INTEGER REFERENCES tree*genres(id), species*id INTEGER REFERENCES tree*species(id), variety*id INTEGER REFERENCES tree*varieties(id), UNIQUE (name*id, genre*id, species*id, variety_id));

-- Step 2: Insert data into the new tables INSERT INTO tree_names (name) SELECT DISTINCT name FROM trees WHERE name IS NOT NULL;

INSERT INTO tree_genres (genre) SELECT DISTINCT genre FROM trees WHERE genre IS NOT NULL;

INSERT INTO tree_species (species) SELECT DISTINCT species FROM trees WHERE species IS NOT NULL;

INSERT INTO tree_varieties (variety) SELECT DISTINCT variety FROM trees WHERE variety IS NOT NULL;

-- Step 3: Insert data into the tree*taxonomy table INSERT INTO tree*taxonomy (name*id, genre*id, species*id, variety*id) SELECT DISTINCT n.id AS name*id, g.id AS genre*id, s.id AS species*id, v.id AS variety*id FROM trees t LEFT JOIN tree*names n ON t.name = n.name LEFT JOIN tree*genres g ON t.genre = g.genre LEFT JOIN tree*species s ON t.species = s.species LEFT JOIN tree*varieties v ON t.variety = v.variety;

-- Step 4: Add tree*taxonomy*id column to the trees table ALTER TABLE trees ADD COLUMN tree*taxonomy*id INTEGER;

-- Step 5: Update the trees table with the corresponding tree*taxonomy*id UPDATE trees t SET tree*taxonomy*id = tt.id FROM tree*taxonomy tt LEFT JOIN tree*names n ON tt.name*id = n.id LEFT JOIN tree*genres g ON tt.genre*id = g.id LEFT JOIN tree*species s ON tt.species*id = s.id LEFT JOIN tree*varieties v ON tt.variety_id = v.id WHERE t.name = n.name AND t.genre = g.genre AND t.species = s.species AND t.variety = v.variety;

-- Step 6: Add foreign key constraint to the trees table ALTER TABLE trees ADD CONSTRAINT fk*tree*taxonomy FOREIGN KEY (tree*taxonomy*id) REFERENCES tree_taxonomy(id);

-- step check

select t.* from trees t join tree*taxonomy tt on tt.id = t.tree*taxonomy*id join tree*names tn on tn.id = tt.name_id where t.name != tn.name;

select t.* from trees t join tree*taxonomy tt on tt.id = t.tree*taxonomy*id join tree*species tn on tn.id = tt.species_id where t.species != tn.species;

-- Step 7: Remove the old columns from the trees table ALTER TABLE trees DROP

COLUMN name, DROP COLUMN genre, DROP COLUMN species, DROP COLUMN variety;