treesdb normalization - The solution

Start

1. Restore the `treesdb_v02.01.sql.backup` file into a newly created and empty `treesdb_v03` database.
2. connect to the `treesdb_v03` database on local either
    - via psql: `psql treesdb_v03`
    - or in pgAdmin

Goal

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.

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

- And the tree measurements in the trees table.

column	table
height	trees
circumference	trees
diameter	trees

- `domain` has its own table and `stage` also each have its own table

column	table
domain	tree_domain
stage	tree_stage

- taxonomy : names, genres, species and varieties

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 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)
    ...
);
```

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

column	table
address	location
suppl_address	location
arrondissement	location
geolocation	location

Let's start with the easy ones : domain and stage

domain

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 t
where t.domain is not null
order by t.domain asc
;
```

Add foreign key column in trees

```SQL
ALTER TABLE trees ADD COLUMN domain_id INTEGER;
```

Update tree_domains with the ddomain values from trees

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

Transform `domain_id` as a foreign key in the trees table

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

Check that everything is as expected. This query should return 0 rows

```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;
```

Now, how do we now get the domain for a given tree ?

With a simple join:

```sql
select t.*, td.*
from trees t
join tree_domains td on t.domain_id = td.id
order by random()
limit 1;
```

This query returns the same results but is more elegant as the random tree selection corresponds to a dedicated sub-query.

```sql
select t.*, td.*
from (
    select *
    from trees
    order by random()
    limit 1
) t
join tree_domains td on t.domain_id = td.id;
```

Let's do the same things for stage.

Don't forget to check the mapping before deleting the stage column

stage

Simlarly for stages

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


insert into tree_stages (stage)
select distinct stage from trees t
where t.stage is not null
order by t.stage asc;

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;
```

Then drop column stage

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

How do we get the stage and domain for a given random tree ?

```sql
select t.*, td.*, ts.*
from (
    select domain_id, stage_id
    from trees
    order by random()
    limit 1
) t
join tree_domains td on t.domain_id = td.id
join tree_stages ts on t.stage_id = ts.id
;
```

Taxonomy

Step 1: Create the new tables

```sql
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
);
```

and the `taxonomy` intermediate table

```sql
CREATE TABLE 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

```sql
INSERT INTO tree_names (name)
SELECT DISTINCT name FROM trees
WHERE name IS NOT NULL
order by name asc;

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

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

INSERT INTO tree_varieties (variety)
SELECT DISTINCT variety FROM trees
WHERE variety IS NOT NULL
order by variety asc;
```

Step 3: Insert data into the taxonomy table

```SQL
INSERT INTO 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 taxonomy_id column to the trees table

```SQL
ALTER TABLE trees ADD COLUMN taxonomy_id INTEGER;
```

Step 5: Update the trees table with the corresponding taxonomy_id

```SQL
UPDATE trees t
SET taxonomy_id = tt.id
FROM 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

```SQL
ALTER TABLE trees
ADD CONSTRAINT fk_taxonomy
FOREIGN KEY (taxonomy_id)
REFERENCES taxonomy(id);
```

check

These queries should return 0 rows

```sql
select t.*
from trees t
join taxonomy tt on tt.id = t.taxonomy_id
join tree_names tn on tn.id = tt.name_id
where t.name != tn.name;
```

```sql
select t.*
from trees t
join taxonomy tt on tt.id = t.taxonomy_id
join tree_species tn on tn.id = tt.species_id
where t.species != tn.species;
```

Same for `genre` and `variety`

Step 7: Remove the old columns from the trees table

```sql
ALTER TABLE trees
DROP COLUMN name,
DROP COLUMN genre,
DROP COLUMN species,
DROP COLUMN variety;
```

Querying the new taxonomies

list the varieties ordered by their frequency in the trees table

```sql
SELECT
    tv.variety,
    COUNT(t.id) AS variety_count
FROM
    trees t
JOIN
    taxonomy tax ON t.taxonomy_id = tax.id
JOIN
    tree_varieties tv ON tax.variety_id = tv.id
GROUP BY
    tv.variety
ORDER BY
    variety_count DESC;
```

Location

Let's first create a table locations with the location related columns

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: reconcile `trees.location_id` with `location.id`

Ideally we want to reconcile on a single attribute to keep the query and logic simple.

At this point we need to deal with another anomaly in the dataset. there are multiple trees standing at the same locations.

Since there are multiple equal geolocations (12 of them)

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

So we may need to reconcile on the whole address which involves a more complex query.

However, 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;
```

We can delete locations geolocation 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
);
```

Let's check that there is no more any geolocation duplicates; this query returns 0 rows:

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

We can also check that 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 distinct when copying the values from trees to locations with `insert from select distinct` in the query above and then to recast geolocation as point.

Now we can associate `trees.location_id` with `location.id` based on `geolocation` only

```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

As usual, 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;
```

Conclusion

In this process of normalizing the original treesdb database you saw

- when to normalize an attribute or set of attributes
- how to create keys and foreign keys
- how to import data from one table to another
- how to reconcile keys between tables
- how to check the results of your import
- how to handle with duplicates

The database structure is more stable and less chaotic.

Although querying the data has become more complex as it involves multiple JOINs the database is much closer to being production ready.

In the next session we will dive deeper into querying this `treesdb_v03` database with **windows functions** and **CTEs**.