2024-11-11

DATABASES 2

LAB 7 - NOSQL DATA IN POSTGRESQL

Paulina Czarnota C21365726

INTRODUCTION

This lab focuses on using NoSQL features in PostgreSQL, specifically the ltree module for hierarchical data and PostGIS for geographical data. The following exercises covers hierarchical data queries, geographical queries, and additional tasks using the provided datasets.

EXERCISE 1: HIERARCHICAL DATA WITH LTREE

This exercise focuses on using PostgreSQL's ltree extension for hierarchical data management. The task involves creating a university dataset representing colleges, schools, and degrees as a hierarchical structure. We perform various operations such as adding, updating, querying, and deleting hierarchical data. Additionally, we compute CAO points for each degree and generate statistical insights using SQL queries.

The ltree extension simplifies working with hierarchical data by providing specialized data types (ltree) and functions for querying and manipulating tree-like structures. This exercise highlights how to efficiently model and work with hierarchical data in a relational database.

SETUP AND DATA INSERTION

Step 1: Enable the ltree Extension

To work with hierarchical data, we need to enable the ltree extension:

```
CREATE EXTENSION IF NOT EXISTS ltree;
```

Explanation:

The ltree extension introduces a new data type (ltree) and several operators to handle hierarchical data. Without enabling this extension, the functionality required for the task will not be available.

Step 2: Create the Hierarchical Table

The table tud_tree will store the hierarchical paths and CAO points for degrees:

```
CREATE TABLE tud_tree (
    path ltree,         -- Column for hierarchical paths
    cao_points INTEGER  -- Column for CAO points
);
```

Explanation:

- The path column uses the ltree data type to store the hierarchy of colleges, schools, and degrees.
- The cao_points column stores the assigned CAO points for each degree, which are used for analysis.

Step 3: Insert the Initial Data

We populate the table with the initial university structure:

```
INSERT INTO tud_tree (path)
VALUES
  ('TUD'::ltree),
  ('TUD.Science'::ltree),
  ('TUD.Science.Computer_Science'::ltree),
  ('TUD.Science.Computer_Science.Software'::ltree),
  ('TUD.Science.Computer_Science.AI'::ltree),
  ('TUD.Science.BiologicalScience'::ltree),
  ('TUD.Science.BiologicalScience.MolecularBiology'::ltree),
  ('TUD.Art'::ltree),
  ('TUD.Art.Design'::ltree),
  ('TUD.Engineering'::ltree)
ON CONFLICT DO NOTHING;
```

Explanation:

- Each row represents a node in the hierarchy.
- The ON CONFLICT DO NOTHING clause ensures that re-running the script does not insert duplicate entries.
- The hierarchy follows a tree structure, starting with the root node (TUD).

TASKS AND QUERIES

Task 1: Add a New School Named Computer_Science Under TUD.Science

```
INSERT INTO tud_tree (path)
```

```
SELECT 'TUD.Science.Computer_Science'::ltree
WHERE NOT EXISTS (
   SELECT 1 FROM tud_tree WHERE path = 'TUD.Science.Computer_Science'::ltree
);
```

Explanation:
This query checks if the Computer_Science school already exists before inserting it, preventing duplicate entries.

Task 2: Add Two Degrees Software and AI Under Computer_Science

```
INSERT INTO tud_tree (path)
VALUES
   ('TUD.Science.Computer_Science.Software'::ltree),
   ('TUD.Science.Computer_Science.AI'::ltree)
ON CONFLICT DO NOTHING;
```

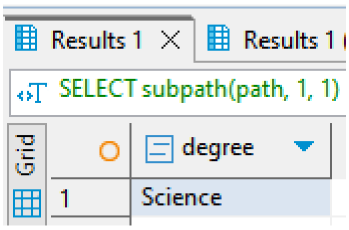
Explanation:

- Adds two degrees under the Computer_Science school.
- Ensures that re-running the script does not create duplicates.

Task 3: Find Under Which Degree MolecularBiology Falls

```
SELECT subpath(path, 1, 1) AS degree
FROM tud_tree
WHERE path ~ '*.MolecularBiology';
```

Output Table:



Explanation:
The subpath function extracts the parent node (Science) under which MolecularBiology falls. The ~ operator matches paths ending in MolecularBiology.

Task 4: Count the Total Number of Courses

```
SELECT COUNT(*) AS course_count
FROM tud_tree;
```

Output Table:

Results 1 (2) | Results 1

SELECT COUNT(*) AS course_cou

	course_count
1	154

Explanation:
This query counts all nodes in the hierarchy, representing the total number of courses, schools, and colleges.

Task 5: Find the Faculty with the Most Courses

```
SELECT subpath(path, 0, 1) AS faculty, COUNT(*) AS course_count
FROM tud_tree
GROUP BY faculty
ORDER BY course_count DESC
LIMIT 1;
```

Output Table:

Results 1 (3) | Results 1 (2) | Results 1

SELECT subpath(path, 0, 1) AS facu | Enter a SQL e

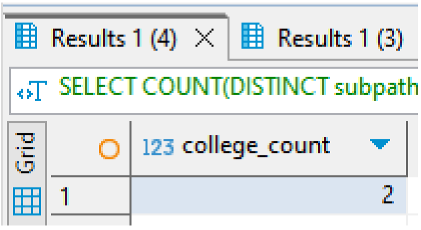
	faculty	course_count
1	TUDublin	142

Explanation:
The query groups paths by the top-level faculty, counts the number of nodes for each faculty, and identifies the one with the highest count.

Task 6: Count the Number of Colleges

```
SELECT COUNT(DISTINCT subpath(path, 0, 1)) AS college_count

FROM tud_tree;
```

Output Table:



Explanation:
This query counts distinct first-level nodes (Science, Art, Engineering) in the hierarchy.

Task 7: Rename the University from TUD to TUDublin

```
UPDATE tud_tree

SET path = regexp_replace(path::text, '^TUD', 'TUDublin')::ltree

WHERE path ~ 'TUD.*';
```

Explanation:
This query replaces TUD with TUDublin in all paths to update the university's name.

Task 8: Delete the BiologicalScience School and Its Courses

```
DELETE FROM tud_tree

WHERE path <@ 'TUDublin.Science.BiologicalScience'::ltree;
```

Explanation:
The <@ operator matches all descendants of BiologicalScience and deletes them.

Task 9: Add a Column for CAO Points

```
ALTER TABLE tud_tree ADD COLUMN cao_points INTEGER;
```

Explanation:
Adds a new column to store CAO points for each degree.

Task 10: Assign CAO Points Based on Faculty

```
UPDATE tud_tree
SET cao_points =
  CASE
    WHEN path ~ 'TUDublin.Art.*' THEN 300
    WHEN path ~ 'TUDublin.Science.*' THEN 450
    WHEN path ~ 'TUDublin.Engineering.*' THEN 400
    ELSE 350
  END;
```

Explanation:
Assigns CAO points based on the faculty type using a CASE statement.

Task 11: Assign 500 CAO Points to Computer_Science Degrees

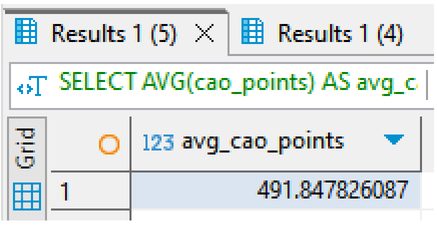
```
UPDATE tud_tree
SET cao_points = 500
WHERE path ~ 'TUDublin.Science.Computer_Science.*';
```

Explanation:
Specifically updates CAO points for all degrees under Computer_Science to 500.

Task 12: Compute the Average CAO Points for the Science College

```
SELECT AVG(cao_points) AS avg_cao_points
FROM tud_tree
WHERE path ~ 'TUDublin.Science.*';
```

Output Table:



Explanation:
Calculates the average CAO points for all degrees under the Science college.

Task 13: Verify the Final Table State

SELECT DISTINCT path, cao_points

FROM tud_tree

ORDER BY path;

Output Table:

	path	cao_points
1	TUDublin	350
2	TUDublin.Art	300
3	TUDublin.Art.Design	300
4	TUDublin.Engineering	400
5	TUDublin.Science	450
6	TUDublin.Science.Computer_Science	500
7	TUDublin.Science.Computer_Science.AI	500
8	TUDublin.Science.Computer_Science.Software	500

Explanation:
Displays the final state of the tud_tree table for validation.

CONCLUSION

This exercise highlights the powerful capabilities of PostgreSQL's ltree extension for managing hierarchical data. The tasks demonstrated how to build, query, update, and analyse a hierarchical dataset efficiently.

EXERCISE 2: GEOGRAPHICAL DATA WITH POSTGIS

This exercise demonstrates the capabilities of PostGIS, a PostgreSQL extension for managing and analysing geographical data. Using datasets representing NYC streets, stations, and neighbourhoods, the tasks cover a range of spatial queries to extract insights such as street intersections, neighbourhood statistics, and road measurements.

Step 1: Activating PostGIS

Before performing any spatial queries, the PostGIS extension must be activated in the database.

```
CREATE EXTENSION IF NOT EXISTS postgis;
```

Explanation:

- **Purpose**: This enables the use of geographical and geometrical data types (geometry, geography) and spatial functions in PostgreSQL.
- **Functionality Enabled**:
    - Spatial relationships (e.g., intersections, containment).
    - Measurement functions (e.g., length, area).
    - Advanced geometry operations.

Step 2: Queries

(a) Streets inside or crossing the East Village

```
SELECT ns.name AS street_name
FROM nycstreet ns
WHERE ST_Crosses(ns.geom, (SELECT geom FROM nycnb WHERE name = 'East Village'
LIMIT 1))
   OR ST_Contains(ns.geom, (SELECT geom FROM nycnb WHERE name = 'East Village'
LIMIT 1))
ORDER BY ns.name;
```

Explanation:

- **Objective**: Identify streets that either cross the boundary of or are fully contained within the "East Village" neighborhood.
- **Functions Used**:
    - ST_Crosses: Checks if a street's geometry intersects the neighborhood's geometry.
    - ST_Contains: Verifies if the entire street geometry is within the neighborhood.
- **Result**:
    - A list of street names in or crossing the East Village, sorted alphabetically.

- This query relies on accurate geometries in both the nycstreet and nycnb tables.

Output Table:

Street name

nycstreet 3 | Results 1

SELECT ns.name AS street_name

	street_name
1	1st Ave
2	2nd Ave
3	3rd Ave
4	Astor Pl
5	Astor Pl
6	Avenue A
7	Avenue B
8	Avenue C
9	Avenue D
10	Bleecker St
11	Bond St
12	Bowery
13	Cooper Sq
14	E 10th St
15	E 11th St
16	E 12th St
17	E 13th St
18	E 14th St
19	E 1st St
20	E 2nd St
21	E 4th St
22	E 6th St
23	E 7th St
24	E 9th St
25	FDR Dr
26	FDR Dr
27	FDR Dr
28	Great Jones St
29	Lillian Wald Dr
30	Stuyvesant Loop S
31	[NULL]
32	[NULL]

(b) Neighbourhood with the most tube stations

```
SELECT nb.name AS neighborhood,
       COUNT(st.gid) AS station_count
FROM nycstation st
JOIN nycnb nb ON ST_Contains(nb.geom, st.geom)
GROUP BY nb.name
```

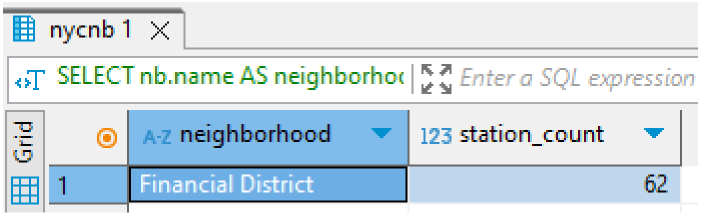
```
ORDER BY station_count DESC
LIMIT 1;
```

Explanation:

- **Objective**: Determine which neighborhood has the highest number of tube stations.
- **Process**:
  - Each station point is checked to see if it is contained within a neighborhood polygon using ST_Contains.
  - The stations are grouped by neighborhood, and the count is calculated.
- **Result**:
  - The neighborhood with the highest count is displayed along with the total number of stations.

Output Table:

Neighbourhood name and station count



	neighborhood	station_count
1	Financial District	62

(c) Longest road

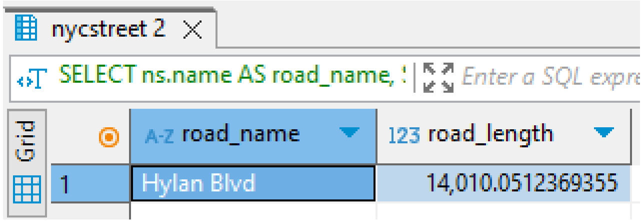
```
SELECT ns.name AS road_name,
       ST_Length(ns.geom) AS road_length
FROM nycstreet ns
ORDER BY road_length DESC
LIMIT 1;
```

Explanation:

- **Objective**: Identify the longest road in the dataset.
- **Function Used**:
  - ST_Length: Computes the length of a geometry in meters.
- **Process**:
  - The length of each street is calculated, and the streets are sorted in descending order of length.
- **Result**:
  - The name and length of the longest road are displayed.

Output Table:

Road name and its length



	road_name	road_length
1	Hylan Blvd	14,010.0512369355

(d) Smallest neighborhood

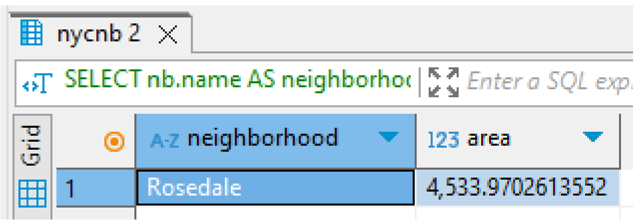
```
SELECT nb.name AS neighborhood,
       ST_Area(nb.geom) AS area
FROM nycnb nb
ORDER BY area ASC
LIMIT 1;
```

Explanation:

- **Objective**: Find the smallest neighborhood by area.
- **Function Used**:
  - ST_Area: Computes the area of a polygon in square meters.
- **Process**:
  - The areas of all neighborhoods are calculated, and the neighborhoods are sorted in ascending order of area.
- **Result**:
  - The name and area of the smallest neighborhood are displayed.

Output Table:

Neighborhood name and its area



	neighborhood	area
1	Rosedale	4,533.9702613552

(e) Total length of Broadway in Manhattan

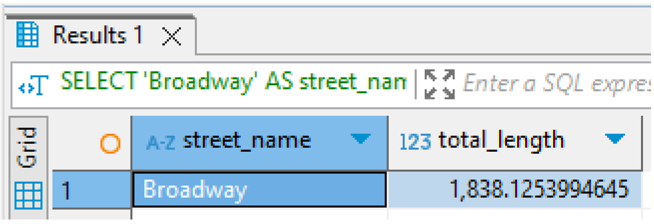
```
SELECT 'Broadway' AS street_name,
       SUM(ST_Length(ns.geom)) AS total_length
FROM nycstreet ns
JOIN nycnb nb ON ST_Within(ns.geom, nb.geom)
WHERE ns.name = 'Broadway' AND nb.boroname = 'Manhattan';
```

Explanation:

- **Objective**: Calculate the total length of all segments of Broadway within Manhattan.
- **Functions Used**:
  - ST_Length: Measures the length of each geometry.
  - ST_Within: Checks if a street segment lies entirely within Manhattan.
- **Process**:
  - Filters for segments of Broadway within Manhattan and sums their lengths.
- **Result**:
  - The total length of Broadway in Manhattan.

Output Table:

Total length of Broadway in Manhattan



	street_name	total_length
1	Broadway	1,838.1253994645

(f) Other streets named Broadway and their lengths

```
SELECT ns.name AS street_name,
       nb.boroname AS borough,
       ST_Length(ns.geom) AS road_length
FROM nycstreet ns
JOIN nycnb nb ON ST_Within(ns.geom, nb.geom)
WHERE ns.name = 'Broadway' AND nb.boroname != 'Manhattan'
ORDER BY road_length DESC;
```

Explanation:

- **Objective**: Find all other streets named Broadway outside Manhattan and list their lengths.
- **Functions Used**:
  - ST_Length: Measures the length of each geometry.
  - ST_Within: Ensures the streets are correctly assigned to a borough.
- **Process**:
  - Filters for streets named Broadway outside Manhattan and sorts them by length.
- **Result**:
  - A list of other Broadway streets with their boroughs and lengths.

Output Table:

Broadway streets outside Manhattan

nycstreet(+) 1

SELECT ns.name AS street_name,

	street_name	borough	road_length
1	Broadway	Staten Island	2,073.0486571503
2	Broadway	Staten Island	2,073.0486571503
3	Broadway	Queens	467.8246994938
4	Broadway	Queens	467.8246994938
5	Broadway	Queens	128.282009372
6	Broadway	Queens	128.282009372

VALIDATION QUERIES

To verify dataset completeness:

1. **Neighborhood Count**:

SELECT COUNT(*) AS neighborhood_count FROM nycnb;

Output Table:

	neighborhood_count
1	258

2. **Station Count**:

SELECT COUNT(*) AS station_count FROM nycstation;

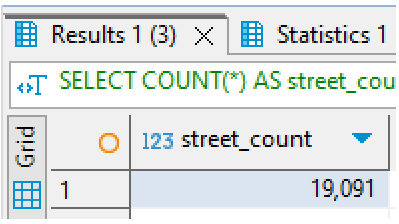
Output Table:

	station_count
1	863

3. **Street Count**:

SELECT COUNT(*) AS street_count FROM nycstreet;

Output Table:



CONCLUSION

This exercise highlights the capabilities of PostGIS for spatial data analysis, including identifying relationships between geometries, calculating measurements, and summarizing spatial features.

EXERCISE 3: EUROPEAN CAPITALS WITH POSTGIS

This exercise explores the PostgreSQL PostGIS extension's capabilities for managing and analysing geographical data. The dataset includes European capitals' names, latitudes, and longitudes. The objective is to create a table with a geography column, load data, and perform spatial queries to extract insights.

SETUP AND DATA PREPARATION

Step 1: Enable the PostGIS Extension

```
CREATE EXTENSION IF NOT EXISTS postgis;
```

Explanation:

This enables spatial data types (geometry, geography) and spatial functions for analysis.

Step 2: Drop and Create the Table

```
DROP TABLE IF EXISTS capital_cities;

CREATE TABLE capital_cities (
    id SERIAL PRIMARY KEY,
    country VARCHAR(100),
    capital VARCHAR(100),
    latitude FLOAT,
    longitude FLOAT,
    coord GEOGRAPHY(Point, 4326)
);
```

Explanation:

The coord column stores geographical points using latitude and longitude with SRID 4326 (Earth's coordinate system).

Step 3: Load Data into the Table

```
COPY capital_cities (country, capital, latitude, longitude)
FROM 'C:\\Program Files\\PostgreSQL\\17\\europe_capitals.csv'
```

```
DELIMITER ','
CSV HEADER;
```

Explanation:

Imports the dataset into PostgreSQL.

Step 4: Update the coord Column

```
UPDATE capital_cities
SET coord = ST_SetSRID(ST_MakePoint(longitude, latitude), 4326);
```

Explanation:

- Converts longitude and latitude into geographical points using ST_MakePoint.
- Sets the SRID (Spatial Reference Identifier) to 4326 for compatibility with Earth-based coordinates.

QUERIES AND OUTPUTS

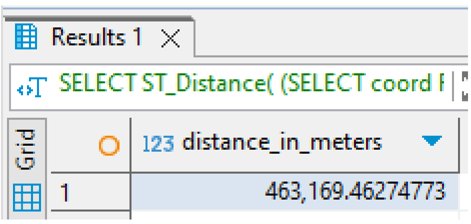
Query 1: Distance Between Dublin and London

```
SELECT
   ST_Distance(
      (SELECT coord FROM capital_cities WHERE capital = 'Dublin' LIMIT 1),
      (SELECT coord FROM capital_cities WHERE capital = 'London' LIMIT 1)
   ) AS distance_in_meters;
```

Explanation:

Computes the geographical distance between Dublin and London.

Output Table:



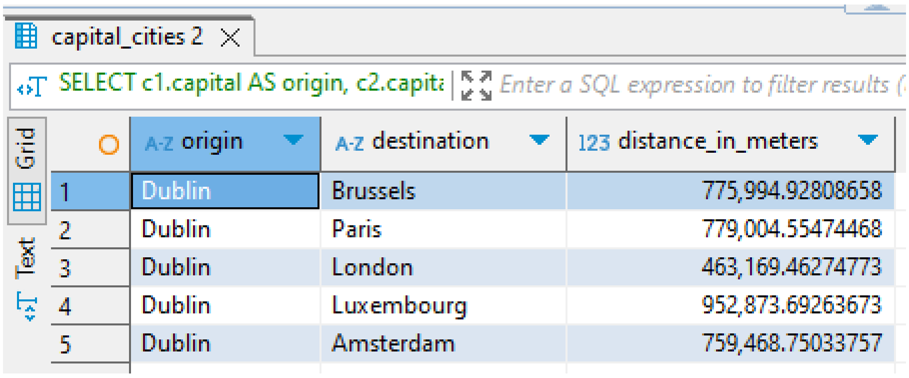
Query 2: Capitals Within 1000 km of Dublin

```
SELECT
    c1.capital AS origin,
    c2.capital AS destination,
    ST_Distance(c1.coord, c2.coord) AS distance_in_meters
FROM capital_cities c1
JOIN capital_cities c2 ON c1.capital <> c2.capital
WHERE
    c1.capital = 'Dublin' AND
    ST_Distance(c1.coord, c2.coord) < 1000000;
```

Explanation:

Identifies capitals within 1000 km of Dublin using ST_Distance.

Output Table:



	origin	destination	distance_in_meters
1	Dublin	Brussels	775,994.92808658
2	Dublin	Paris	779,004.55474468
3	Dublin	London	463,169.46274773
4	Dublin	Luxembourg	952,873.69263673
5	Dublin	Amsterdam	759,468.75033757

Query 3: Most Distant Capital From Dublin

```
SELECT
    c1.capital AS origin,
    c2.capital AS destination,
    ST_Distance(c1.coord, c2.coord) AS distance_in_meters
FROM capital_cities c1
JOIN capital_cities c2 ON c1.capital <> c2.capital
WHERE c1.capital = 'Dublin'
```

ORDER BY distance_in_meters DESC

LIMIT 1;

Explanation:

Determines the capital farthest from Dublin using ST_Distance and ORDER BY.

Output Table:

capital_cities 2

SELECT c1.capital AS origin, c2.capit… | Enter a SQL expression to filter results

	origin	destination	distance_in_meters
1	Dublin	Nicosia	3,683,001.56372973

Query 4: Capital With the Most Reachable Capitals Within 500 km

SELECT

c1.capital AS origin,

COUNT(c2.capital) AS reachable_capitals

FROM capital_cities c1

JOIN capital_cities c2 ON ST_Distance(c1.coord, c2.coord) < 500000 AND c1.capital <> c2.capital

GROUP BY c1.capital

ORDER BY reachable_capitals DESC

LIMIT 1;

Explanation:

Counts capitals reachable within 500 km for each capital using ST_Distance.

Output Table:

capital_cities 1

SELECT c1.capital AS origin, COUNT(| Enter a SQL ex

	origin	reachable_capitals
1	Belgrade	11

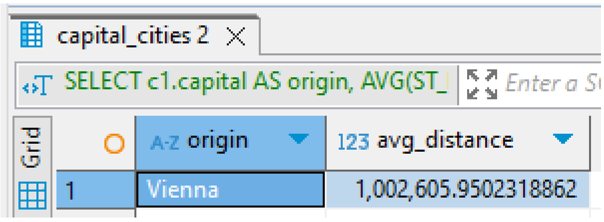
Query 5: Capital With Minimum Average Distance to All Others

```
SELECT
    c1.capital AS origin,
    AVG(ST_Distance(c1.coord, c2.coord)) AS avg_distance
FROM capital_cities c1
JOIN capital_cities c2 ON c1.capital <> c2.capital
GROUP BY c1.capital
ORDER BY avg_distance ASC
LIMIT 1;
```

Explanation:

Finds the capital with the minimum average distance to all other capitals using AVG and GROUP BY.

Output Table:



VALIDATION

Check Table Data and coord Column

```
SELECT * FROM capital_cities;
```

Output Table:

capital_cities 1

SELECT * FROM capital_cities | Enter a SQL expression to filter results (use Ctrl+Space)

	id	country	capital	latitude	longitude	coord
1	1	ad	Andorra	42.5	1.5165	POINT (1.5165 42.5)
2	2	al	Tirana	41.3275	19.8189	POINT (19.8189 41.3275)
3	3	at	Vienna	48.2	16.3666	POINT (16.3666 48.2)
4	4	ba	Sarajevo	43.85	18.383	POINT (18.383 43.85)
5	5	be	Brussels	50.8333	4.3333	POINT (4.3333 50.8333)
6	6	bg	Sofia	42.6833	23.3167	POINT (23.3167 42.6833)
7	7	by	Minsk	53.9	27.5666	POINT (27.5666 53.9)
8	8	ch	Bern	46.9167	7.467	POINT (7.467 46.9167)
9	9	cy	Nicosia	35.1667	33.3666	POINT (33.3666 35.1667)
10	10	cz	Prague	50.0833	14.466	POINT (14.466 50.0833)
11	11	de	Berlin	52.5218	13.4015	POINT (13.4015 52.5218)
12	12	dk	Copenhagen	55.6786	12.5635	POINT (12.5635 55.6786)
13	13	ee	Tallinn	59.4339	24.728	POINT (24.728 59.4339)
14	14	es	Madrid	40.4	-3.6834	POINT (-3.6834 40.4)
15	15	fi	Helsinki	60.1756	24.9341	POINT (24.9341 60.1756)
16	16	fr	Paris	48.8667	2.3333	POINT (2.3333 48.8667)
17	17	gb	London	51.5072	-0.1275	POINT (-0.1275 51.5072)
18	18	gr	Athens	37.9833	23.7333	POINT (23.7333 37.9833)
19	19	hr	Zagreb	45.8	16	POINT (16 45.8)
20	20	hu	Budapest	47.5	19.0833	POINT (19.0833 47.5)
21	21	ie	Dublin	53.3331	-6.2489	POINT (-6.2489 53.3331)
22	22	is	Reykjavik	64.15	-21.95	POINT (-21.95 64.15)
23	23	it	Rome	41.896	12.4833	POINT (12.4833 41.896)
24	24	li	Vaduz	47.1337	9.5167	POINT (9.5167 47.1337)
25	25	lt	Vilnius	54.6834	25.3166	POINT (25.3166 54.6834)
26	26	lu	Luxembourg	49.6117	6.13	POINT (6.13 49.6117)
27	27	lv	Riga	56.95	24.1	POINT (24.1 56.95)
28	28	mc	Monaco	43.7396	7.4069	POINT (7.4069 43.7396)
29	29	md	Chisinau	47.005	28.8577	POINT (28.8577 47.005)
30	30	me	Podgorica	42.466	19.2663	POINT (19.2663 42.466)
31	31	mk	Skopje	42	21.4335	POINT (21.4335 42)
32	32	mt	Valletta	35.8997	14.5147	POINT (14.5147 35.8997)
33	33	nl	Amsterdam	52.35	4.9166	POINT (4.9166 52.35)
34	34	no	Oslo	59.9167	10.75	POINT (10.75 59.9167)
35	35	pl	Warsaw	52.25	21	POINT (21 52.25)
36	36	pt	Lisbon	38.7227	-9.1449	POINT (-9.1449 38.7227)
37	37	ro	Bucharest	44.4334	26.0999	POINT (26.0999 44.4334)
38	38	rs	Belgrade	44.8186	20.468	POINT (20.468 44.8186)
39	39	ru	Moscow	55.7522	37.6155	POINT (37.6155 55.7522)
40	40	se	Stockholm	59.3508	18.0973	POINT (18.0973 59.3508)
41	41	si	Ljubljana	46.0553	14.515	POINT (14.515 46.0553)
42	42	sk	Bratislava	48.15	17.117	POINT (17.117 48.15)

CONCLUSION

This exercise highlighted the capabilities of PostgreSQL's PostGIS extension for geographical analysis. Key learnings included:

i. Managing spatial data in a database using the geography data type.
ii. Executing spatial queries to calculate distances and relationships.
iii. Extracting insights from geographical data using advanced queries.

EXERCISE 4: VISUALIZING QUERIES WITH QGIS

This exercise involves visualizing spatial queries using QGIS connected to a PostgreSQL database with PostGIS extension. The tasks include writing SQL queries, loading results as layers, and capturing visual outputs for neighborhoods, streets, and stations in NYC datasets.

QUERIES AND VISUALIZATIONS

Query A: Streets Crossing or Inside East Village

```
SELECT ns.gid, ns.name AS street_name, ns.geom

FROM nycstreet ns

WHERE ST_Crosses(ns.geom, (SELECT geom FROM nycnb WHERE name = 'East Village'
LIMIT 1))

   OR ST_Contains(ns.geom, (SELECT geom FROM nycnb WHERE name = 'East Village'
LIMIT 1));
```

Explanation:

This query identifies streets that either intersect or are fully contained within the East Village neighborhood.

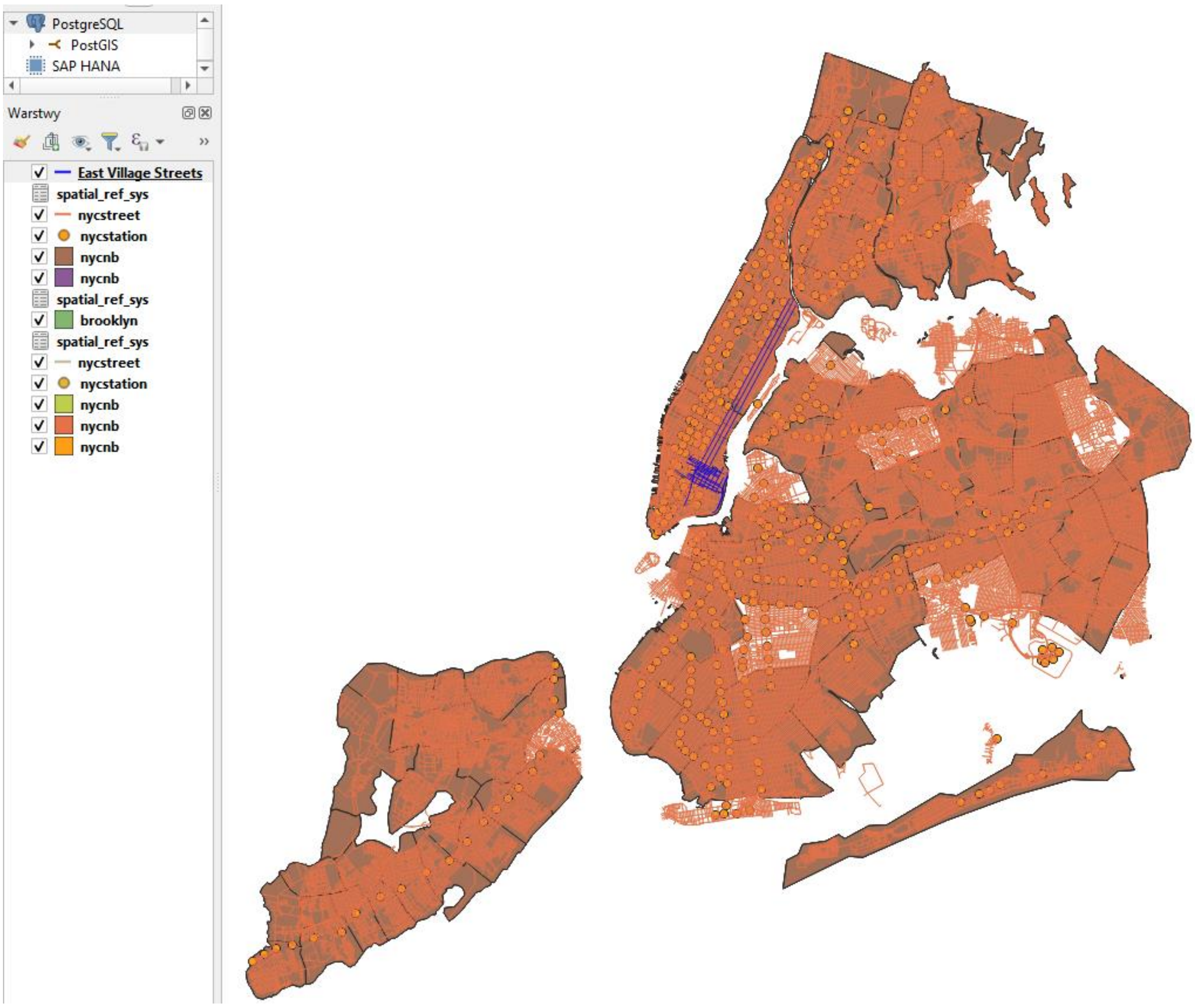
Functions Used:

- ST_Crosses: Checks for intersections between street geometries and the neighborhood boundary.
- ST_Contains: Checks if the entire street geometry is inside the neighborhood.

Outputs:

- Attributes Table showing the names of the streets.
- A map layer showing the streets inside or crossing East Village.

	gid	street_name	geom
1	107	*NULL*	0105000020E61...
2	129	*NULL*	0105000020E61...
3	17843	Bleecker St	0105000020E61...
4	18057	Bowery	0105000020E61...
5	18142	E 10th St	0105000020E61...
6	18148	Bond St	0105000020E61...
7	18179	Great Jones St	0105000020E61...
8	18187	E 12th St	0105000020E61...
9	18201	E 4th St	0105000020E61...
10	18207	E 13th St	0105000020E61...
11	18221	E 14th St	0105000020E61...
12	18284	Cooper Sq	0105000020E61...
13	18248	Astor Pl	0105000020E61...
14	18262	E 1st St	0105000020E61...
15	18292	2nd Ave	0105000020E61...
16	18299	E 6th St	0105000020E61...
17	18307	E 7th St	0105000020E61...
18	18313	Astor Pl	0105000020E61...
19	18324	E 9th St	0105000020E61...
20	18342	E 11th St	0105000020E61...
21	18356	3rd Ave	0105000020E61...
22	18422	1st Ave	0105000020E61...
23	18459	E 2nd St	0105000020E61...
24	18531	Avenue A	0105000020E61...
25	18629	Avenue B	0105000020E61...
26	18732	Avenue C	0105000020E61...
27	18772	FDR Dr	0105000020E61...
28	18779	Stuyvesant Loo...	0105000020E61...
29	18843	Avenue D	0105000020E61...
30	18889	FDR Dr	0105000020E61...
31	18923	Lillian Wald Dr	0105000020E61...
32	19040	FDR Dr	0105000020E61...



Query B: All Tube Stations in Brooklyn

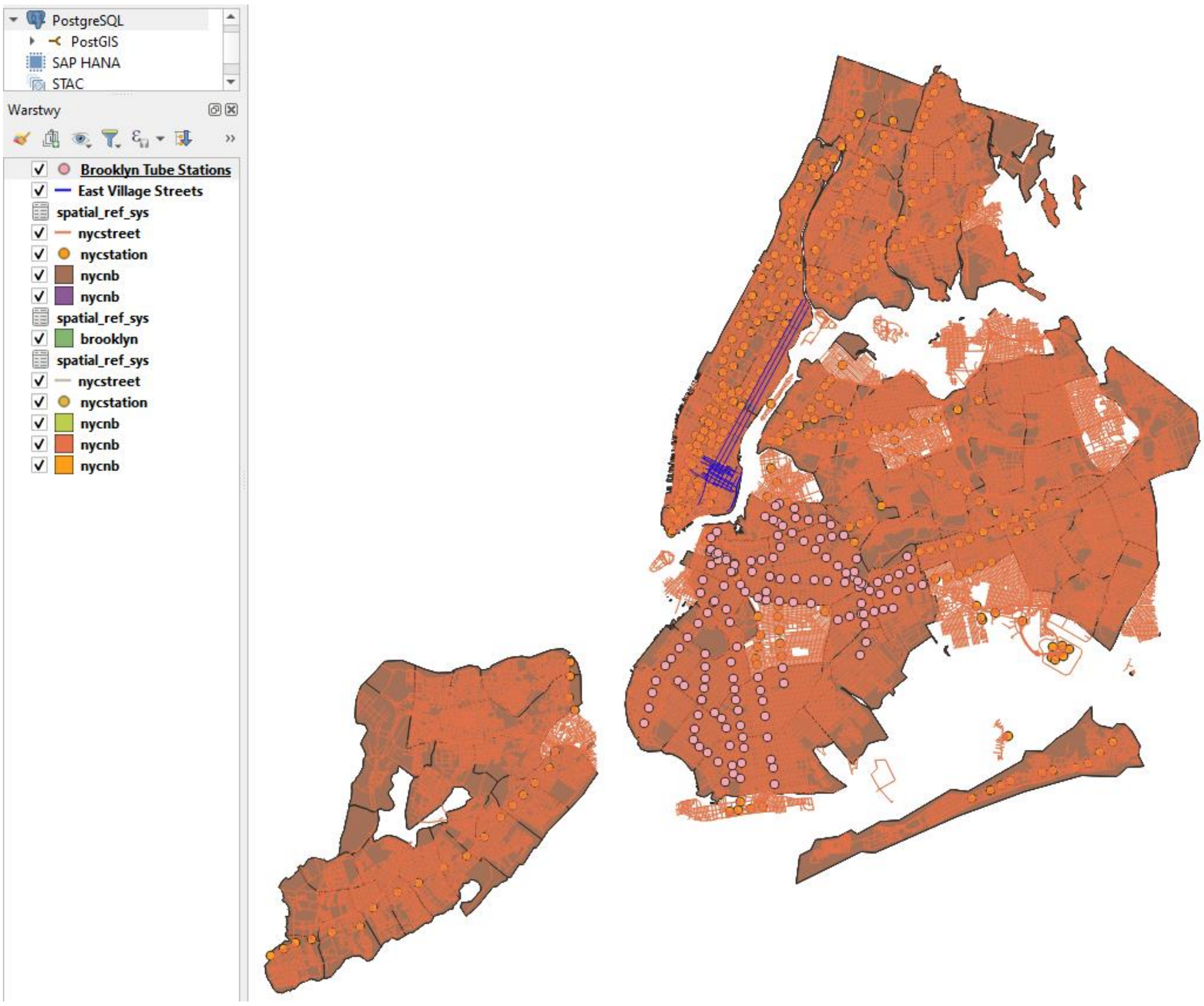
```
SELECT st.gid, st.name AS station_name, st.geom

FROM nycstation st

JOIN nycnb nb ON ST_Contains(nb.geom, st.geom)

WHERE nb.boroname = 'Brooklyn';
```

Explanation:

- Lists all tube stations in Brooklyn by checking if the station geometry is contained within the Brooklyn neighborhood polygons.
- Function Used: ST_Contains.

Outputs:

A map layer visualizing the tube stations in Brooklyn.



Query C: 5th Avenue and Broadway in Manhattan

SELECT ns.gid, ns.name AS street_name, ns.geom

FROM nycstreet ns

JOIN nycnb nb ON ST_Within(ns.geom, nb.geom)

WHERE ns.name IN ('Broadway', '5th Avenue') AND nb.boroname = 'Manhattan';

Explanation:

Filters street segments named Broadway and 5th Avenue that are completely contained within Manhattan.

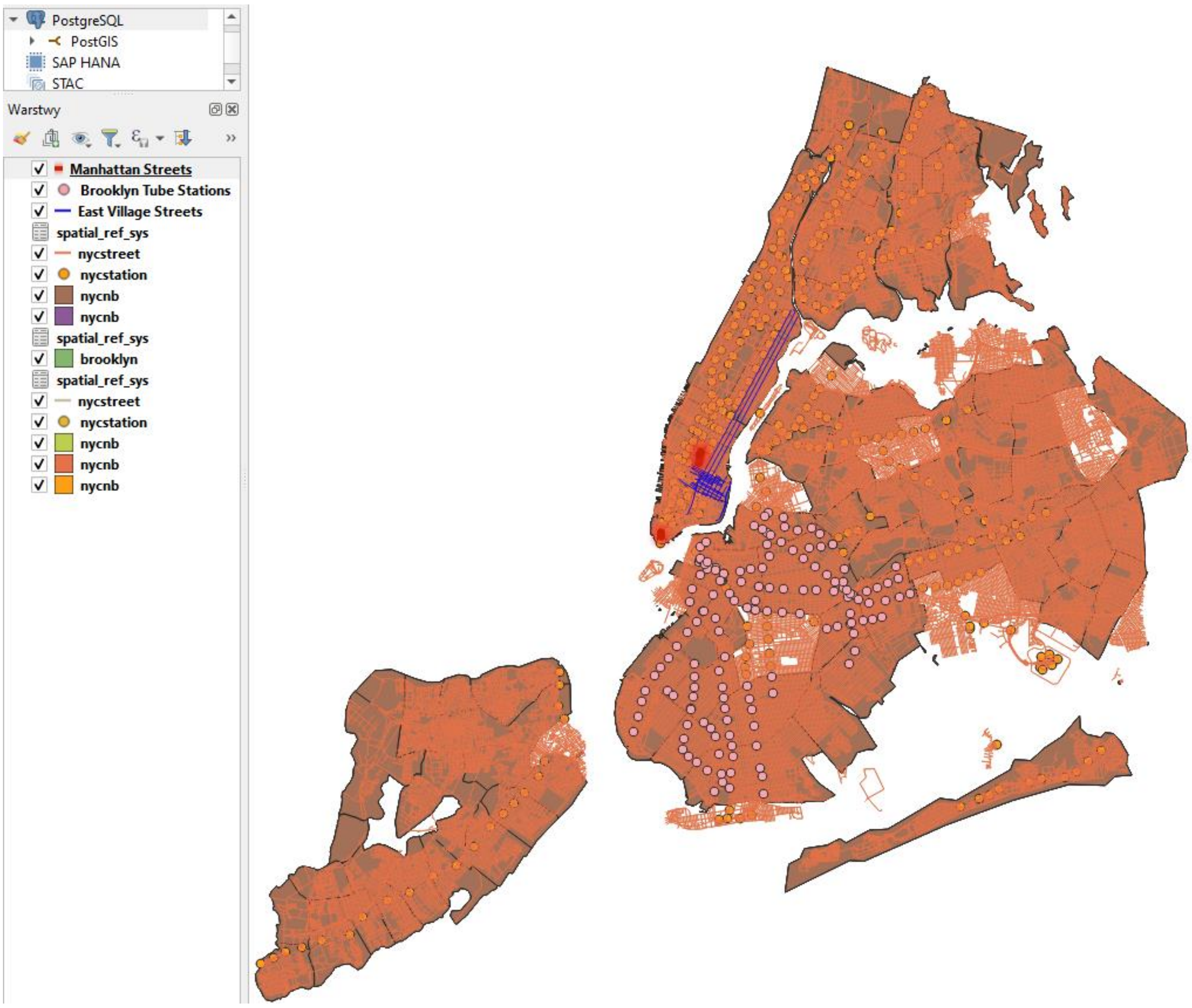
Functions Used:

ST_Within: Ensures the street geometry is inside Manhattan polygons.

Outputs:

- Attributes Table listing the names of street segments.
- A map layer highlighting the selected streets in Manhattan.

	gid	street_name	geom
1	18337	Broadway	0105000020E61...
2	18375	Broadway	0105000020E61...
3	17332	Broadway	0105000020E61...



Query D: Neighbourhoods in Manhattan Crossed by Broadway

```
SELECT
    ROW_NUMBER() OVER () AS id,
    nb.name AS neighborhood,
    nb.geom
```

```
FROM nycnb nb
JOIN nycstreet ns
  ON ST_Crosses(nb.geom, ns.geom)
WHERE ns.name = 'Broadway' AND nb.boroname = 'Manhattan';
```

Explanation:

Finds neighbourhoods in Manhattan intersected by Broadway.

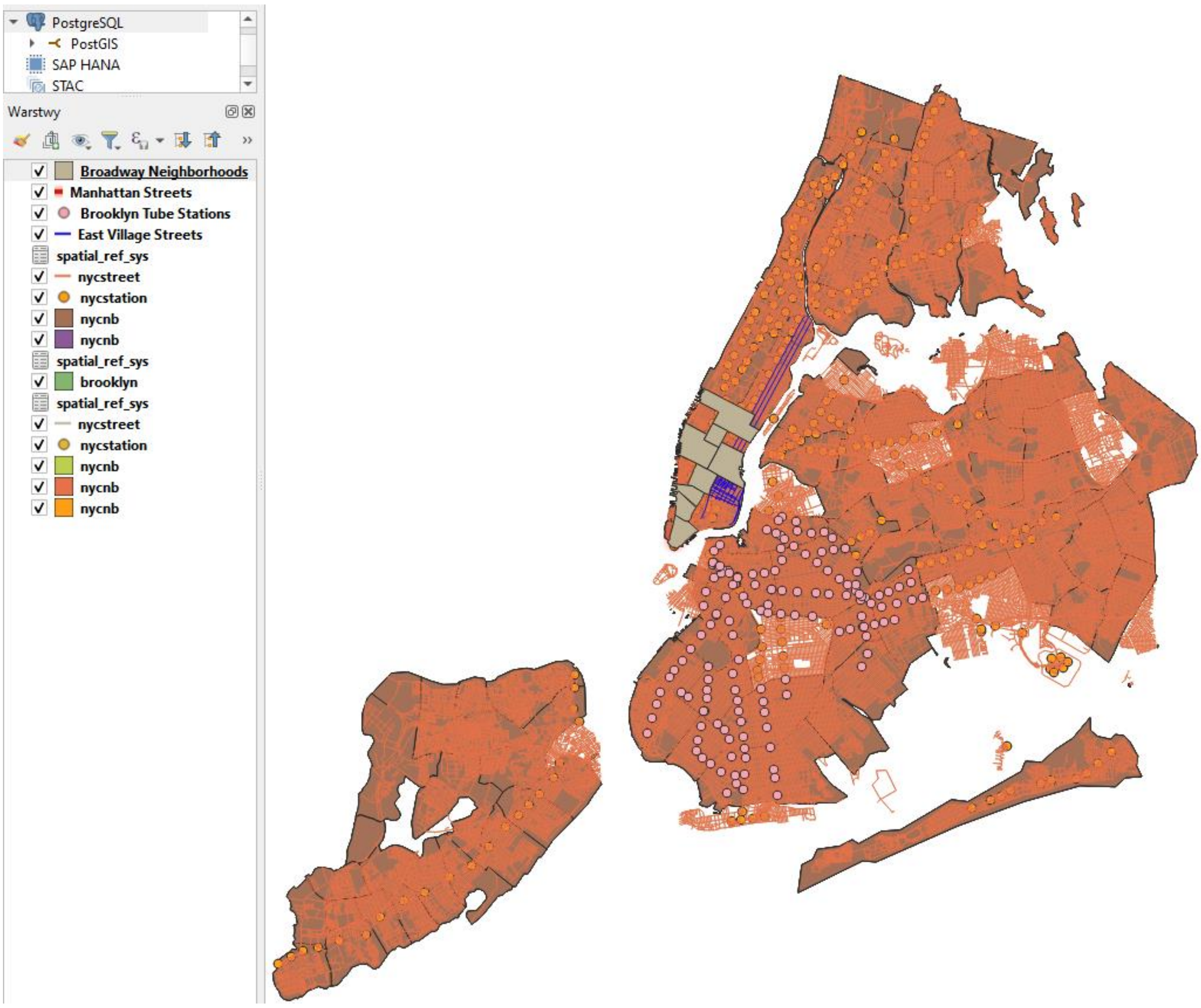
Functions Used:

ST_Crosses: Identifies geometries that intersect.

Outputs:

- Attributes Table listing neighbourhood names.
- A map layer showing the neighbourhoods crossed by Broadway.

	id	neighborhood	geom
1	1	Gramercy	0106000020E61...
2	2	Gramercy	0106000020E61...
3	3	Soho	0106000020E61...
4	4	Greenwich ...	0106000020E61...
5	5	Midtown	0106000020E61...
6	6	Tribeca	0106000020E61...
7	7	Financial District	0106000020E61...
8	8	Chelsea	0106000020E61...
9	9	Chelsea	0106000020E61...
10	10	Garment District	0106000020E61...



CONCLUSION

The PostGIS extension in PostgreSQL allows efficient spatial data management.

QGIS provides powerful visualization tools for spatial data queries.

The queries helped identify streets, tube stations, and neighbourhoods relevant to Manhattan and Brooklyn.