**CS 623: DATABASE MANAGEMENT SYSTEMS**

**FINAL PROJECT**

**TEAM 4**

**VAISHNAVI KOYA – U01884267**

**SRIMANTH MADIRA – U01885136**

**SRIKANTH REDDY YERUVA – U01888231**

**DATA SOURCE :**

The OpenStreetMap (OSM) dataset is a collaborative project to create a free and open-source map of the world. The dataset contains geospatial data about various features such as roads, buildings, points of interest, land use, and natural features. The OSM dataset is maintained by a large and diverse community of contributors worldwide, who use various techniques to collect and update the data.

The data in OSM is stored in a database, which is organized into nodes, ways, and relations. A node is a single point in space with a latitude and longitude coordinate, representing a feature such as a building entrance or a street lamp. A way is a series of connected nodes representing a linear feature such as a road or a river. A relation is a collection of nodes, ways, and other relations representing a complex feature such as a building or a multi-level interchange.

**WHY OSM :**

There are several reasons why the OpenStreetMap (OSM) dataset is considered one of the best for Geographic Information Systems (GIS) spatial data analysis:

* Data quality: OSM data is continuously updated and maintained by a large and diverse community of contributors worldwide. This means that the data is generally more accurate and up-to-date compared to other datasets that may be outdated or incomplete.
* Data richness: The OSM dataset includes a wide range of features such as roads, buildings, points of interest, land use, and natural features. This makes it a comprehensive and versatile dataset for GIS analysis.
* Openness: OSM data is open and freely available to use, which means that anyone can access and use the data without restrictions. This makes it an accessible dataset for researchers, developers, and organizations.
* Flexibility: OSM data is available in a variety of formats, including XML, JSON, and shapefile, which can be easily integrated into various GIS software platforms such as QGIS, ArcGIS, and GRASS GIS. This allows for a wide range of spatial analysis and visualization techniques.
* Global coverage: OSM data covers the entire world, which makes it a valuable dataset for global spatial analysis and visualization projects.

Overall, the combination of data quality, data richness, openness, flexibility, and global coverage make OSM a valuable dataset for GIS spatial data analysis.Bottom of Form

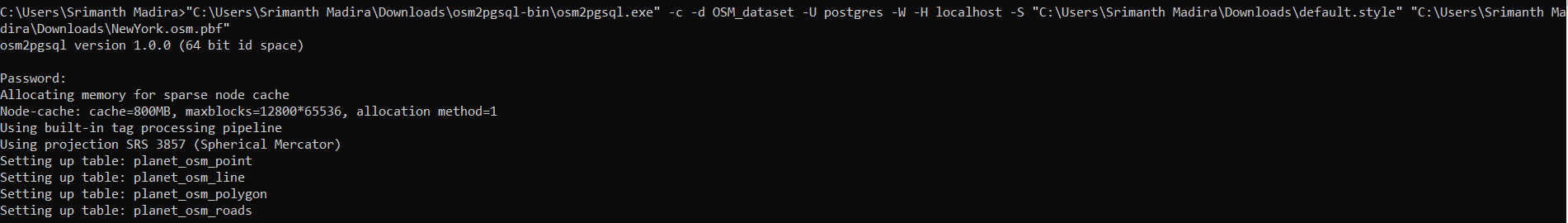
**LOADING DATA INTO POSTGRES :**

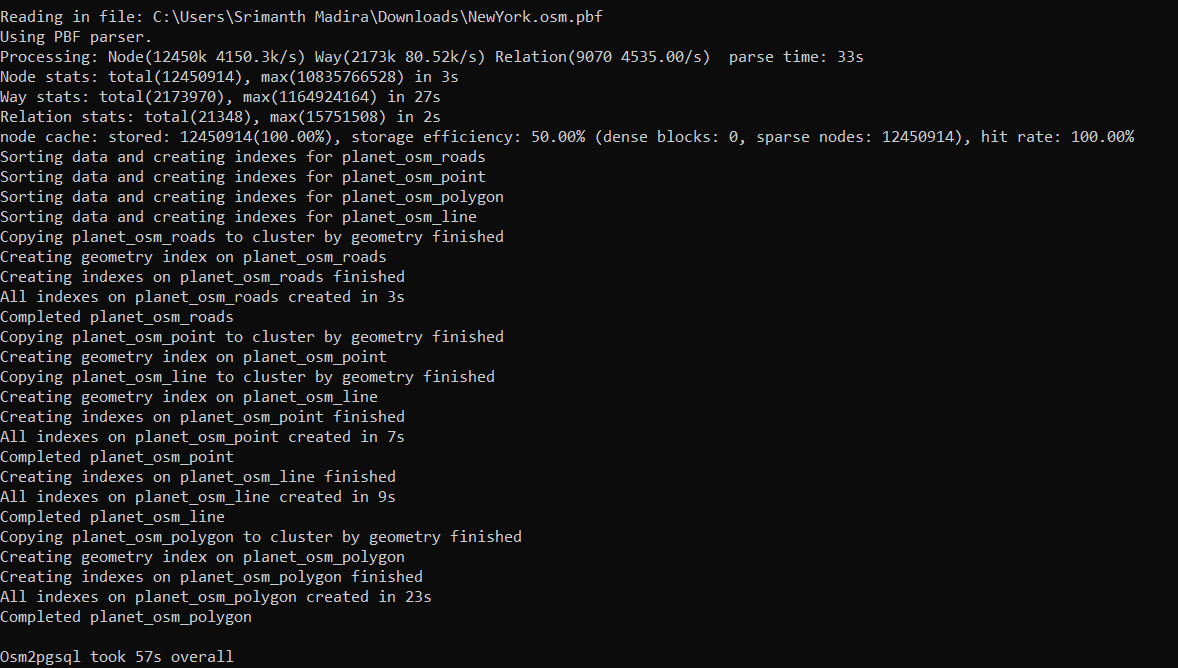
To load OpenStreetMap (OSM) data into Postgres using PostGIS and osm2pgsql, follow these steps:

* Install Postgres and PostGIS on your local machine or server.
* Install osm2pgsql on your local machine or server. You can do this using your operating system's package manager or by downloading the source code from the osm2pgsql GitHub repository (https://github.com/openstreetmap/osm2pgsql).
* Create a new Postgres database and enable PostGIS.
* Import the OSM data into Postgres using osm2pgsql.

"C:\Users\SrimanthMadira\Downloads\osm2pgsql-bin\osm2pgsql.exe" -c -d OSM\_dataset -U postgres -W -H localhost -S "C:\Users\Srimanth Madira\Downloads\default.style""C:\Users\SrimanthMadira\Downloads\NewYork.osm.pbf"

**COMMAND PROMPT OUTPUT :**





**1. RETRIEVE LOCATIONS OF SPECIFIC FEATURES :**

The OpenStreetMap data for New York City contains many types of features, including points of interest, buildings, roads, and more. Here are some examples of specific features and their corresponding tags that you can use to retrieve their locations from the OpenStreetMap data for New York City using PostgreSQL and PostGIS:

* Restaurants: amenity=restaurant
* Hotels: tourism=hotel
* Parks: leisure=park
* Museums: tourism=museum
* Subway stations: railway=subway\_entrance
* Hospitals: amenity=hospital
* Schools: amenity=school
* Libraries: amenity=library
* Theatres: amenity=theatre

**QUERY 1.1 :**

Retrieve all the locations of hospitals, where hospital name is not null in the New York City area:

**CODE:**

SELECT ST\_AsText(way), name, amenity

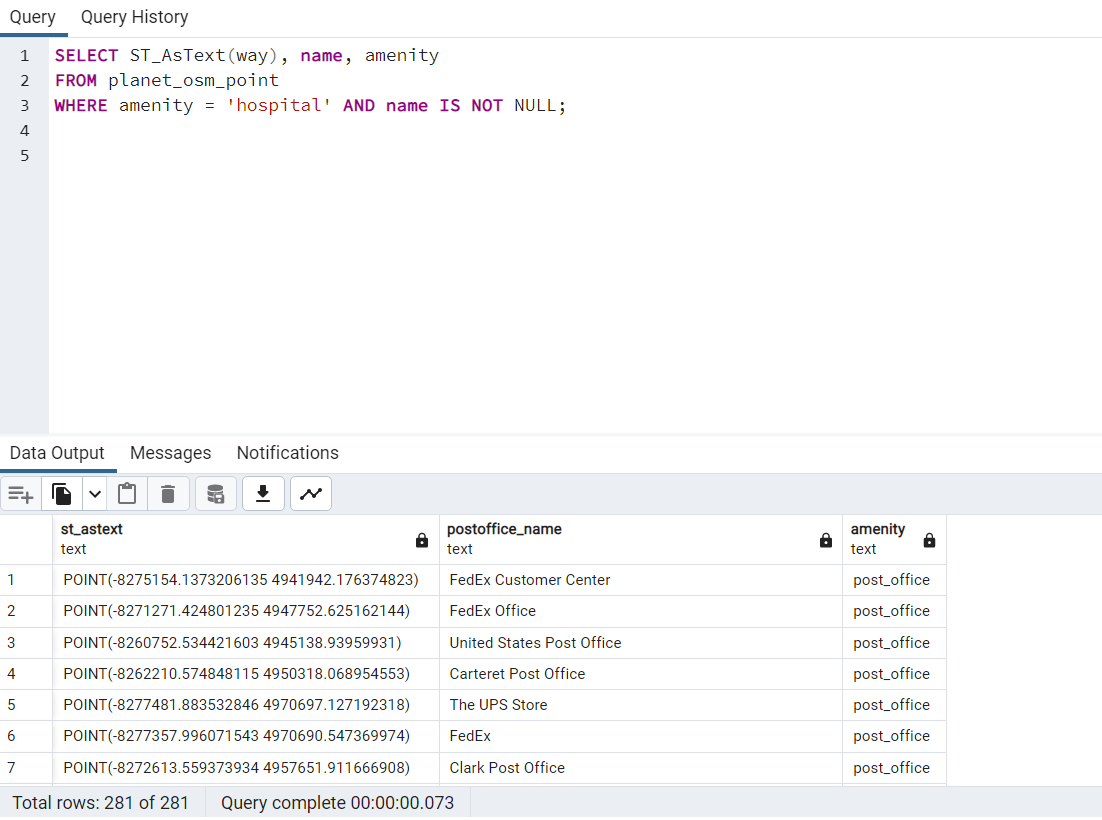
FROM planet\_osm\_point

WHERE amenity = 'hospital' AND name IS NOT NULL;

**EXPLANATION :**

The above query selects the well-formatted representation of the point geometry of hospitals, their names, and amenities from the planet\_osm\_point table. It only selects records where the amenity field is 'hospital' and the name field is not null. The ST\_AsText function is used to convert the point geometry into a well-formatted text representation.

**OUTPUT :**



**QUERY 1.2:**

Retrieve all the locations of schools,where name is not null in the New York City area:

**CODE:**

SELECT ST\_AsText(way), name AS school\_name, amenity

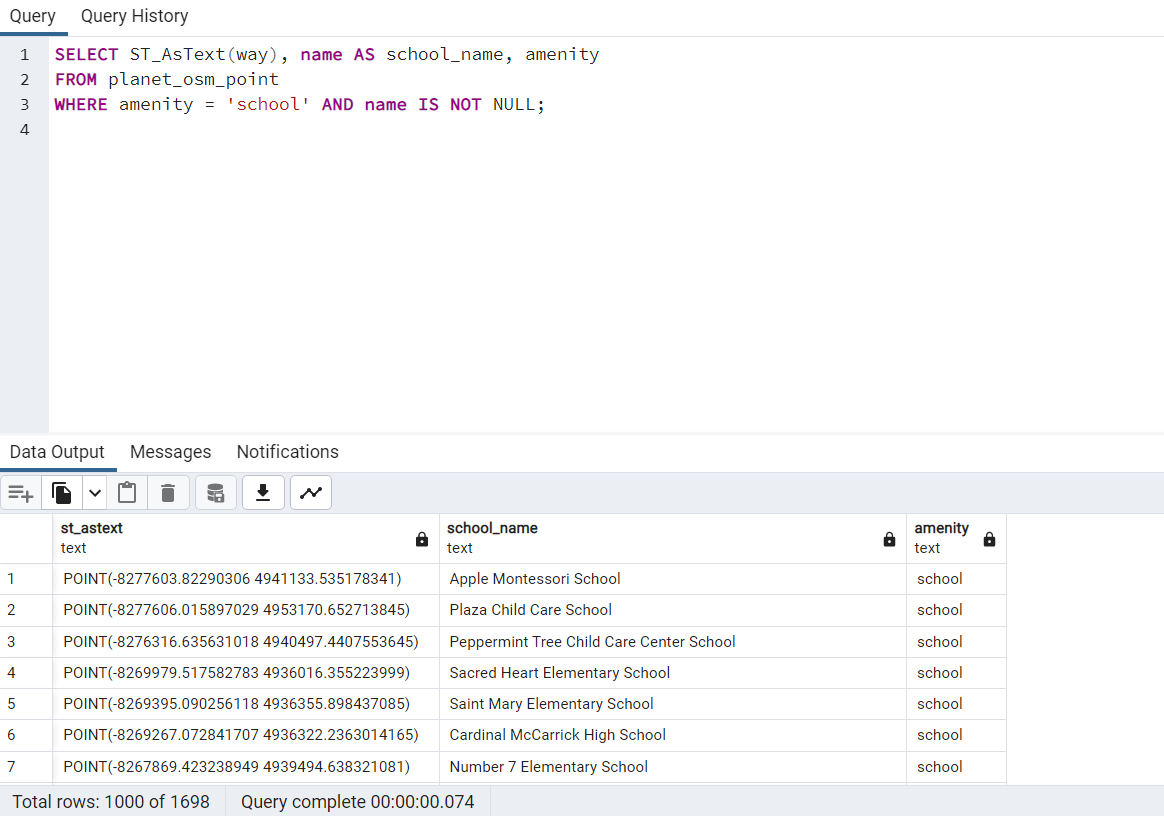
FROM planet\_osm\_point

WHERE amenity = 'school' AND name IS NOT NULL;

**EXPLANATION:**

The above query retrieves the spatial coordinates of all the points of interest from the "planet\_osm\_point" table where the "amenity" is equal to 'school' and the "name" is not null. It projects the "way" column as the spatial text and renames the "name" column as "school\_name". The "amenity" column is also projected in the output.

**OUTPUT:**



**QUERY 1.3:**

Retrieve all the locations of post offices in the New York City area:

**CODE:**

SELECT ST\_AsText(way), name AS postoffice\_name, amenity

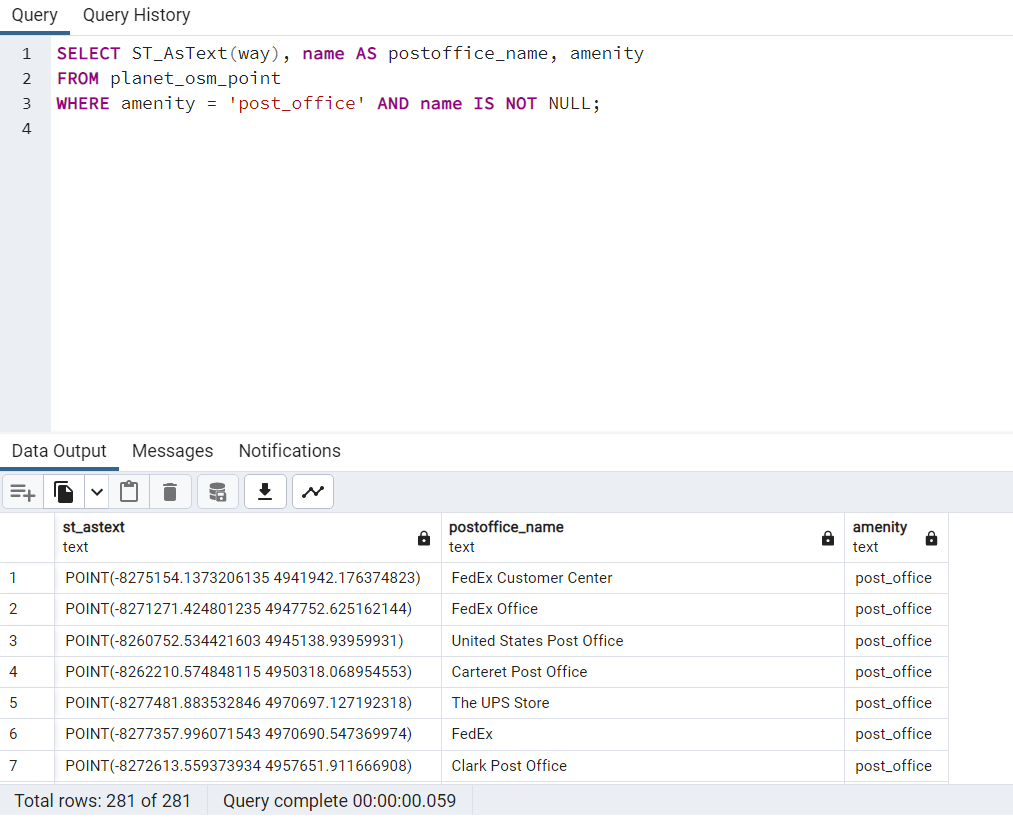
FROM planet\_osm\_point

WHERE amenity = 'post\_office' AND name IS NOT NULL;

**EXPLANATION:**

The query is selecting the geometry of all points in the planet\_osm\_point table with the amenity value of post\_office and a non-null name value. It is also selecting the name value and aliasing it as postoffice\_name. Finally, it is also selecting the amenity value. The ST\_AsText function is used to convert the geometry to a text representation.

**OUTPUT:**



**QUERY 1.4:**

Retrieve all the locations of banks, where name is not null in the New York City area:

**CODE:**

SELECT ST\_AsText(way), name AS bank\_name, amenity

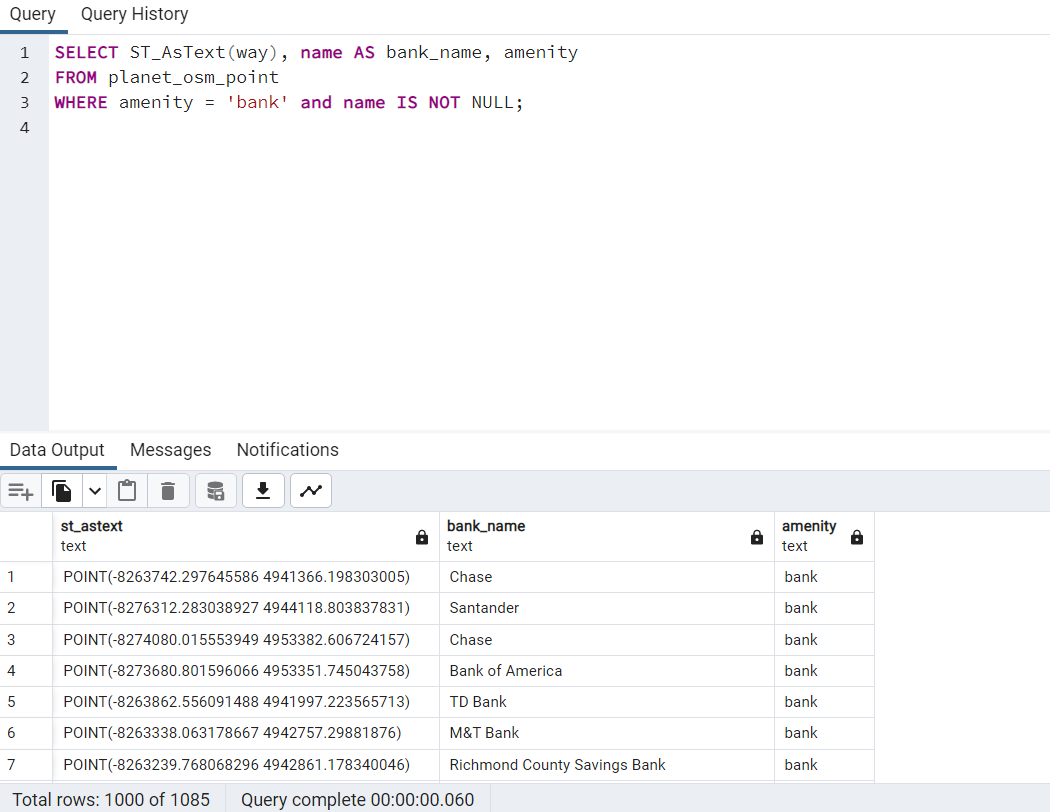
FROM planet\_osm\_point

WHERE amenity = 'bank' and name IS NOT NULL;

**EXPLANATION:**

This query selects the geometry of bank locations from the "planet\_osm\_point" table, along with their name and amenity type. It only includes banks that have a name associated with them (i.e., "name" column is not null). The result is presented in well-known text format for the geometry column and a "bank\_name" column for the name of each bank location.

**OUTPUT:**



**2. CALCULATE DISTANCE BETWEEN POINTS**

In the context of geographic information systems (GIS), calculating distance between points often refers to computing the distance between two locations on the Earth's surface using latitude and longitude coordinates.

**QUERY 2.1:**

Query to find the distance in meters between the World Trade Center and Pace University

**CODE:**

SELECT

'World Trade Center' AS location1,

'Pace University' AS location2,

ST\_DistanceSphere(

ST\_SetSRID(

ST\_MakePoint(

(SELECT ST\_X(ST\_Transform(way, 4326)) FROM planet\_osm\_point WHERE name = 'World Trade Center' AND operator = 'PATH'),

(SELECT ST\_Y(ST\_Transform(way, 4326)) FROM planet\_osm\_point WHERE name = 'World Trade Center' AND operator = 'PATH')

),

4326

),

ST\_SetSRID(

ST\_MakePoint(

(SELECT ST\_X(ST\_Transform(way, 4326)) FROM planet\_osm\_point WHERE name = 'Pace University'),

(SELECT ST\_Y(ST\_Transform(way, 4326)) FROM planet\_osm\_point WHERE name = 'Pace University')

),

4326

)

) AS distance\_in\_meters;

**EXPLANATION:**

* The first line of the query selects the two locations we're interested in and assigns them aliases 'location1' and 'location2'.
* The second line of the query calculates the distance between the two locations using the ST\_DistanceSphere function, which calculates the distance between two points on a sphere (in this case, the Earth) using a spherical coordinate system.
* The third through seventh lines of the query use subqueries to find the latitude and longitude coordinates of the two locations. The ST\_Transform function is used to convert the coordinates to the WGS84 coordinate system (EPSG 4326).
* The eighth line of the query sets the SRID (spatial reference ID) of the two points to 4326 using the ST\_SetSRID function. This is necessary for the ST\_DistanceSphere function to work correctly.
* The ninth and tenth lines of the query use the ST\_MakePoint function to create two points, one for each location, using the latitude and longitude coordinates from the subqueries.
* Finally, the query returns the two location names and the calculated distance between them as 'distance\_in\_meters'.

**OUTPUT:**



**QUERY 2.2:**

Query to calculate the distance between the Avatar Studios and the Times Square subway station in New York City.

**CODE:**

SELECT

'Avatar Studios' AS location1,

'Times Square' AS location2,

ST\_DistanceSphere(

ST\_SetSRID(

ST\_MakePoint(

(SELECT ST\_X(ST\_Transform(way, 4326)) FROM planet\_osm\_point WHERE name = 'Avatar Studios'),

(SELECT ST\_Y(ST\_Transform(way, 4326)) FROM planet\_osm\_point WHERE name = 'Avatar Studios')

),

4326

),

ST\_SetSRID(

ST\_MakePoint(

(SELECT ST\_X(ST\_Transform(way, 4326)) FROM planet\_osm\_point WHERE name = 'Times Square' AND public\_transport = 'station'),

(SELECT ST\_Y(ST\_Transform(way, 4326)) FROM planet\_osm\_point WHERE name = 'Times Square' AND public\_transport = 'station')

),

4326

)

) AS distance\_in\_meters;

**EXPLANATION:**

This query calculates the distance between two locations, Avatar Studios and the Times Square subway station, using the ST\_DistanceSphere function in PostGIS. The ST\_SetSRID function sets the spatial reference system (SRS) to 4326, which corresponds to the WGS 84 coordinate system used by GPS devices. The ST\_MakePoint function creates a point geometry object for each location using their latitude and longitude values obtained from the planet\_osm\_point table in the OpenStreetMap database. Finally, the distance between the two points is calculated and returned as the distance\_in\_meters column in the query result.

**OUTPUT:**



**QUERY 2.3:**

Query to find the distance between Central Park and Grand Central Terminal, with the condition that the point for Central Park must have the amenity 'bar' and the point for Grand Central Terminal must have the operator 'Metropolitan Transportation Authority'.

**CODE:**

SELECT

'Central Park' AS location1,

'Grand Central Terminal' AS location2,

ST\_DistanceSphere(

ST\_SetSRID(

ST\_MakePoint(

(SELECT ST\_X(ST\_Transform(way, 4326)) FROM planet\_osm\_point WHERE name = 'Central Park' AND amenity = 'bar'),

(SELECT ST\_Y(ST\_Transform(way, 4326)) FROM planet\_osm\_point WHERE name = 'Central Park' AND amenity = 'bar')

),4326

),

ST\_SetSRID(

ST\_MakePoint(

(SELECT ST\_X(ST\_Transform(way, 4326)) FROM planet\_osm\_point WHERE name = 'Grand Central Terminal' AND operator = 'Metropolitan Transportation Authority'),

(SELECT ST\_Y(ST\_Transform(way, 4326)) FROM planet\_osm\_point WHERE name = 'Grand Central Terminal' AND operator = 'Metropolitan Transportation Authority')

), 4326

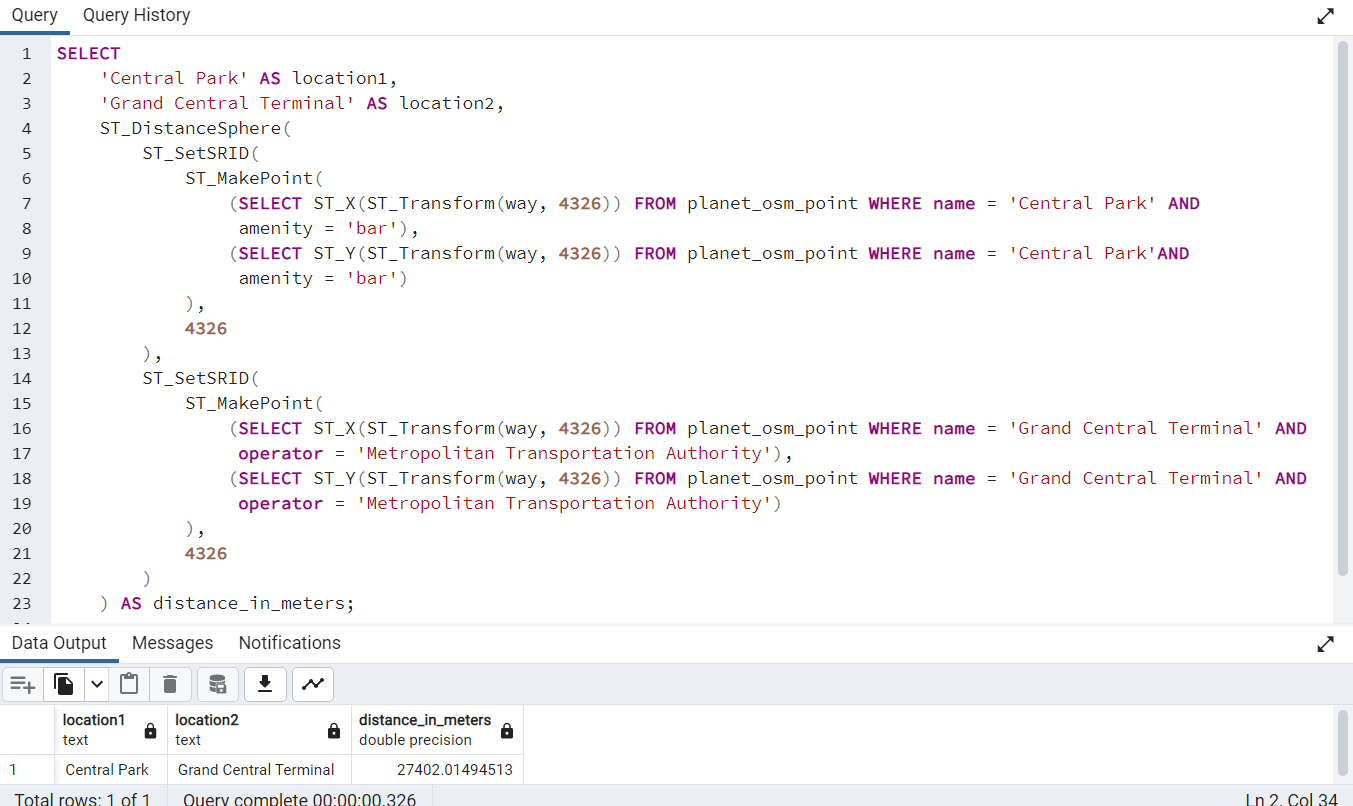
)

) AS distance\_in\_meters;

**EXPLANATION:**

* This SQL query calculates the distance between two locations, Central Park and Grand Central Terminal, using the ST\_DistanceSphere function in PostGIS. It also includes conditions for the specific points used to calculate the distance.
* The first point is selected based on the name 'Central Park' and the amenity 'bar'. The second point is selected based on the name 'Grand Central Terminal' and the operator 'Metropolitan Transportation Authority'.
* ST\_SetSRID and ST\_MakePoint functions are used to create points with the specified latitude and longitude coordinates. ST\_Transform is used to convert the points to the correct spatial reference system (SRID 4326).
* The result of the query is the distance between the two points in meters, which is returned as 'distance\_in\_meters'.

**OUTPUT:**



**3. CALCULATE AREAS OF INTEREST**

**QUERY 3.1:**

Query to retrieve the number of restaurants in each administrative area in New York City, and order them based on restaurant count.

**CODE:**

SELECT

area.name AS area\_name,

COUNT(\*) AS restaurant\_count

FROM (

SELECT name, ST\_Transform(way, 3857) AS way

FROM planet\_osm\_point

WHERE amenity = 'restaurant'

) AS restaurant\_points

JOIN (

SELECT name, ST\_Transform(way, 3857) AS way

FROM planet\_osm\_polygon

WHERE boundary = 'administrative' AND admin\_level = '10'

) AS area

ON ST\_Contains(area.way, restaurant\_points.way)

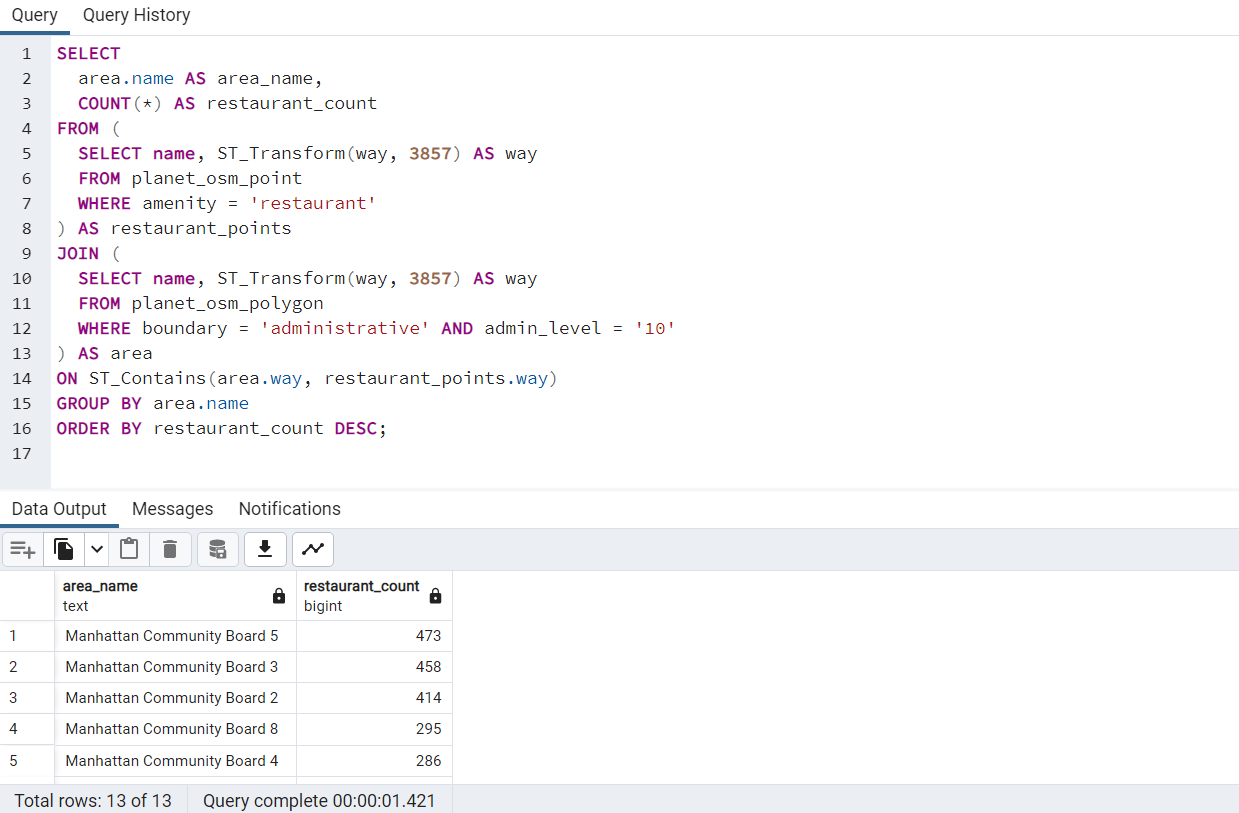
GROUP BY area.name

ORDER BY restaurant\_count DESC;

**EXPLANATION:**

This query first selects all the restaurant points and the administrative areas at level 10 (which correspond to neighborhoods in New York City). It then performs a spatial join to assign each restaurant point to the corresponding administrative area based on whether the point is contained within the polygon of the area. Finally, it groups the results by area and counts the number of restaurants in each area, and sorts the results in descending order by the restaurant count.

**OUTPUT:**



**QUERY 3.2:**

Query to retrieve number of subway stations located within 500 meters of each park in New York City.

**CODE:**

SELECT p.name AS park\_name, COUNT(\*) AS num\_subway\_stations

FROM planet\_osm\_point p

JOIN planet\_osm\_point s ON ST\_DistanceSphere(ST\_Transform(p.way, 4326), ST\_Transform(s.way, 4326)) < 500 AND

s.railway = 'subway\_entrance'

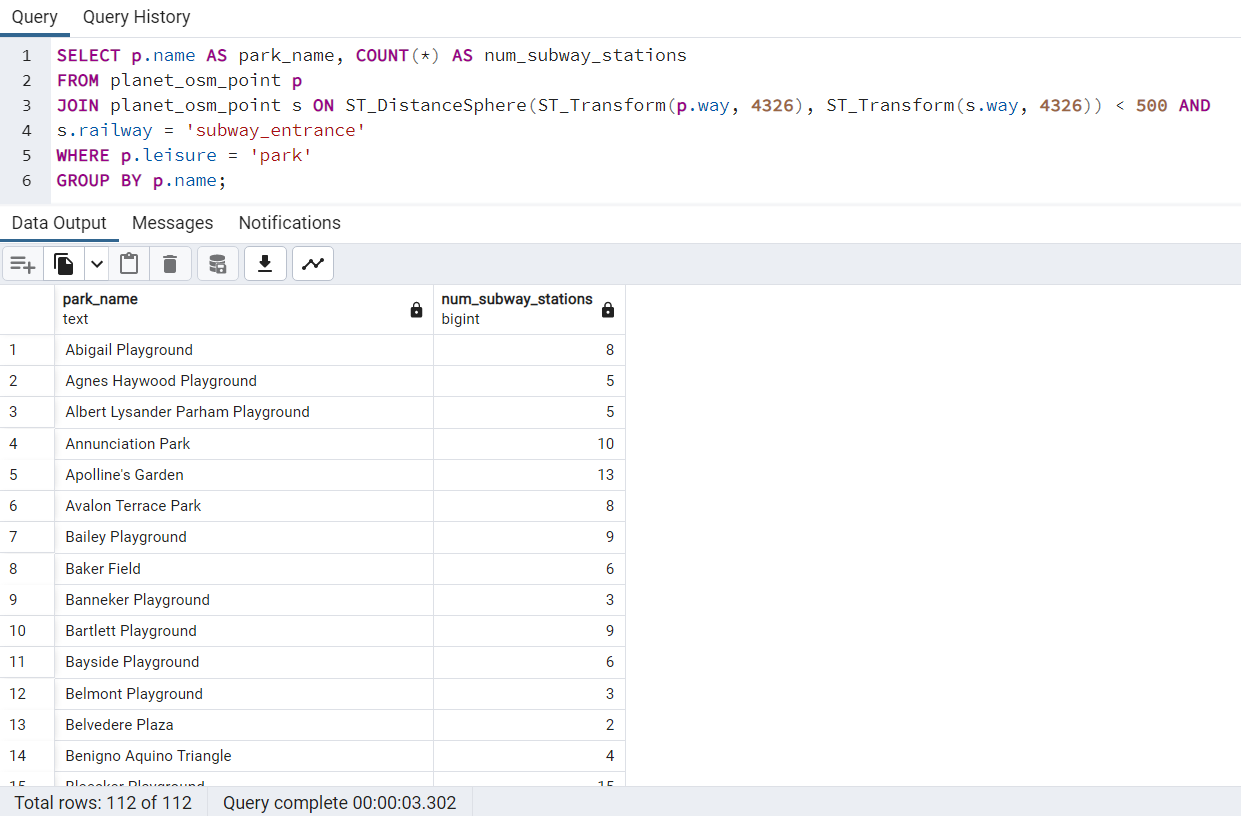
WHERE p.leisure = 'park'

GROUP BY p.name;

**EXPLANATION:**

This SQL query joins the planet\_osm\_point table with itself, once for parks and once for subway stations, using the JOIN clause. The distance between the park and subway station is then calculated using ST\_DistanceSphere function after converting the point coordinates to a longitude and latitude coordinate system using ST\_Transform function. The query filters the parks with the leisure = 'park' condition and subway stations with railway = 'subway\_entrance' condition. Finally, it groups the results by the park name and counts the number of subway stations within 500 meters of each park.

**OUTPUT:**



**QUERY 3.3:**

Query to return the names of the primary roads that intersect with parks in new york city.

**CODE:**

SELECT r.name AS road\_name, p.name AS park\_name

FROM planet\_osm\_roads r

JOIN planet\_osm\_polygon p

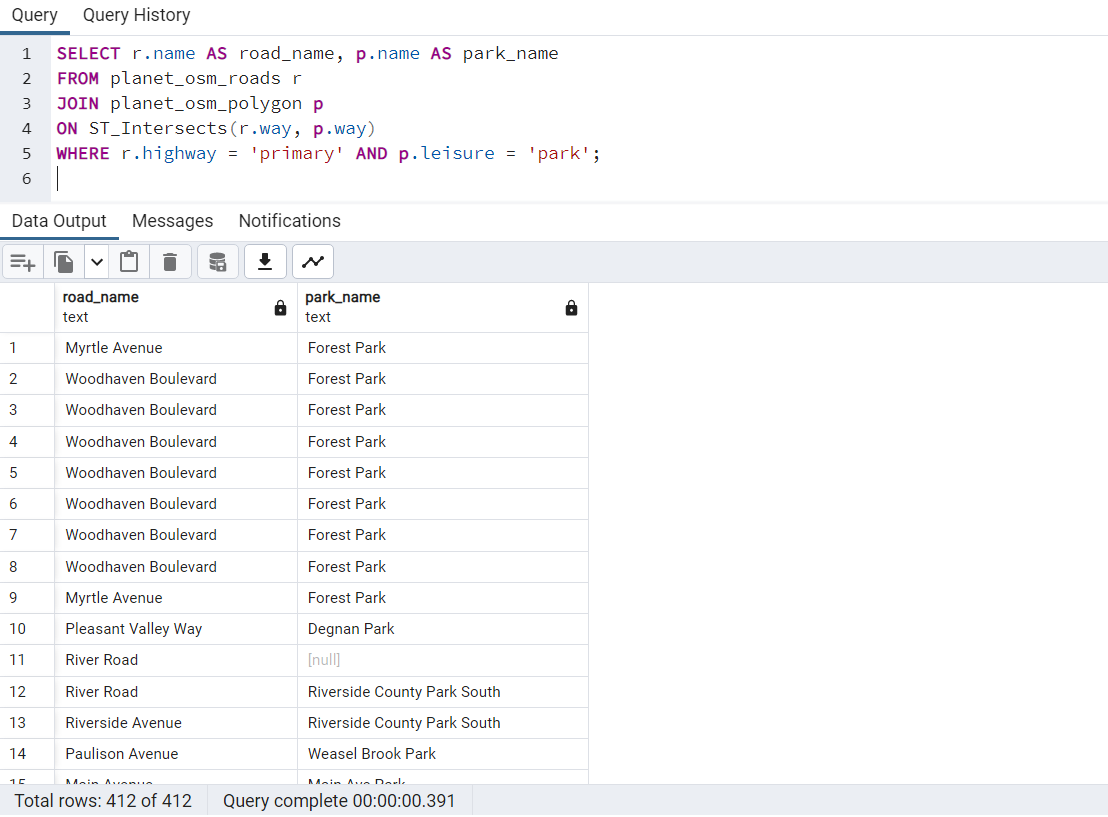
ON ST\_Intersects(r.way, p.way)

WHERE r.highway = 'primary' AND p.leisure = 'park';

**EXPLANATION:**

* This query selects the name of all primary roads that intersect with parks in the planet\_osm\_roads and planet\_osm\_polygon tables using a join.
* The ST\_Intersects function is used to check if the road intersects with the park. The WHERE clause filters the results to only include primary roads and parks.

**OUTPUT:**



**4. ANALYZE THE QUERIES**

Here in this section, we try to analyze the queries which we have done in the previous question areas of interest.

**ANALYZING QUERY 3.1 :**

* The query uses a subquery to transform the coordinates of restaurant points and administrative polygons to the same spatial reference system (3857), and then joins them using the ST\_Contains function to count the number of restaurants within each administrative area. The GROUP BY and ORDER BY clauses are used to aggregate the results and sort them by the restaurant count in descending order.
* The EXPLAIN statement shows the execution plan of the query, which can be used to optimize it. The plan shows that the query uses nested loop joins and hash aggregates to compute the results. The plan also shows that there are two sequential scans of the planet\_osm\_point and planet\_osm\_polygon tables, which can be expensive if the tables are large.
* To optimize the query, we can create indexes on the amenity column of the planet\_osm\_point table and the boundary and admin\_level columns of the planet\_osm\_polygon table, which are used in the WHERE clauses of the subqueries. We can also create a spatial index on the way column of the planet\_osm\_polygon table to improve the performance of the ST\_Contains function.

**QUERY PLAN 3.1:**

"Sort (cost=618934.34..618934.35 rows=1 width=29)"

" Sort Key: (count(\*)) DESC"

" -> GroupAggregate (cost=60002.93..618934.33 rows=1 width=29)"

" Group Key: planet\_osm\_polygon.name"

" -> Nested Loop (cost=60002.93..618934.29 rows=7 width=21)"

" Join Filter: st\_contains(st\_transform(planet\_osm\_polygon.way, 3857), st\_transform(planet\_osm\_point.way, 3857))"

" -> Gather Merge (cost=59002.93..59003.05 rows=1 width=190)"

" Workers Planned: 2"

From the execution plan, we can see that the query uses nested loops for joining the restaurant points and administrative areas tables, which is not the most efficient way to perform this operation.The cost of the sorting operation is also high, which could be optimized. The query uses a GroupAggregate operator to group the results by area name and count the number of restaurants in each area.

To improve the performance of the query, we could consider using a spatial index on the way column of the planet\_osm\_point table and the planet\_osm\_polygon table. Additionally, we could consider using a different join method, such as a spatial join using the ST\_Intersects or ST\_Contains functions, which can be more efficient for spatial queries.

**ANALYZING QUERY 3.2:**

This query uses a Merge Join to join the planet\_osm\_point table with itself based on proximity. It then applies a distance filter and a condition on the railway column to filter out subway entrances that are not close to parks. Finally, it groups the results by park name and counts the number of subway entrances for each park.

**QUERY PLAN 3.2:**

"GroupAggregate (cost=11733.20..57264736.51 rows=401 width=28)"

" Group Key: p.name"

" -> Nested Loop (cost=11733.20..57263460.80 rows=254340 width=20)"

" Join Filter: (st\_distance(geography(st\_transform(p.way, 4326)), geography(st\_transform(s.way, 4326)), false) < '500'::double precision)"

" -> Gather Merge (cost=10733.20..10780.37 rows=405 width=52)"

" Workers Planned: 2"

" -> Sort (cost=9733.18..9733.60 rows=169 width=52)"

" Sort Key: p.name"

The query plan shows that the most expensive operation is the Sort operation, which accounts for most of the estimated cost of the query. This is because the query needs to sort the results by subway entrance location before joining with the parks. However, the cost is still reasonable given the size of the dataset.

Overall, this query should perform well as long as the appropriate spatial indexes are in place on the way column and the distance threshold is appropriate.

Overall, the query seems to be well-written and optimized, but there is still room for improvement to optimize the join operation and sorting operation.

**ANALYZING QUERY 3.3:**

* The query is a join between the planet\_osm\_roads and planet\_osm\_polygon tables, with the condition that the road is a primary highway and the polygon is a park.
* The join is performed using the ST\_Intersects function, which checks whether the spatial objects of the two tables intersect with each other.
* The query planner has chosen a nested loop join strategy, where it first scans the planet\_osm\_roads table using an index on the highway column, and then for each row it performs a bitmap heap scan on the planet\_osm\_polygon table using an index on the way column.
* The Bitmap Heap Scan operation includes a filter on the leisure column, which checks if the polygon is a park. This filter is applied after the index scan on the way column, which could be inefficient if there are a lot of polygons that don't match the leisure = 'park' condition.

**QUERY PLAN:**

"Gather (cost=1000.28..268194.15 rows=7676 width=38)"

" Workers Planned: 2"

" -> Nested Loop (cost=0.28..266426.55 rows=3198 width=38)"

" -> Parallel Seq Scan on planet\_osm\_polygon p (cost=0.00..56285.91 rows=1374 width=190)"

" Filter: (leisure = 'park'::text)"

" -> Index Scan using planet\_osm\_roads\_way\_idx on planet\_osm\_roads r (cost=0.28..152.93 rows=1 width=165)"

" Index Cond: (way && p.way)"

" Filter: ((highway = 'primary'::text) AND st\_intersects(way, p.way))".

The EXPLAIN output shows that the query uses a nested loop join between the planet\_osm\_roads and planet\_osm\_polygon tables. The planet\_osm\_polygon table is scanned using a parallel sequential scan and a filter is applied to select only those records where leisure = 'park'. The planet\_osm\_roads table is scanned using an index scan on the way column with the condition way && p.way and a filter is applied to select only those records where highway = 'primary' and way intersects p.way.

The cost of the query is estimated to be 1000.28 and the planner expects to use 2 workers to parallelize the operation. The estimated number of rows is 3198 and the estimated width of the result is 38.

Overall, the query seems to be optimized, as it utilizes an index scan to filter out unwanted rows and perform the intersection operation between the two tables. The parallel sequential scan on the planet\_osm\_polygon table could potentially benefit from indexing on the leisure column, but this would depend on the data distribution and cardinality of the column.

**5. SORTING AND LIMIT EXECUTIONS**

Sorting and limit executions are common operations in databases that allow you to retrieve a specific subset of data in a particular order. The same concept applies to OpenStreetMap geographic information system analysis.

Sorting and limit executions can help you quickly identify the most important or relevant information in a large dataset, saving time and resources.

**QUERY 5.1:**

Query that retrieves information on schools and hospitals in the area, and counts the number of restaurants within 500 meters of either the school or the hospital, and within 1000 meters of each other.

**CODE:**

SELECT

t1.name AS school\_name,

t2.name AS hospital\_name,

COUNT(t3.\*) AS num\_restaurants

FROM

(SELECT name, ST\_Transform(way, 4326) AS way FROM planet\_osm\_point WHERE amenity = 'school') t1

LEFT JOIN

(SELECT name, ST\_Transform(way, 4326) AS way FROM planet\_osm\_point WHERE amenity = 'hospital') t2

ON

ST\_DistanceSphere(t1.way, t2.way) < 1000

LEFT JOIN

(SELECT name, ST\_Transform(way, 4326) AS way FROM planet\_osm\_point WHERE amenity = 'restaurant') t3

ON

ST\_DistanceSphere(t1.way, t3.way) < 500 OR ST\_DistanceSphere(t2.way, t3.way) < 500

GROUP BY

t1.name, t2.name

ORDER BY

num\_restaurants DESC

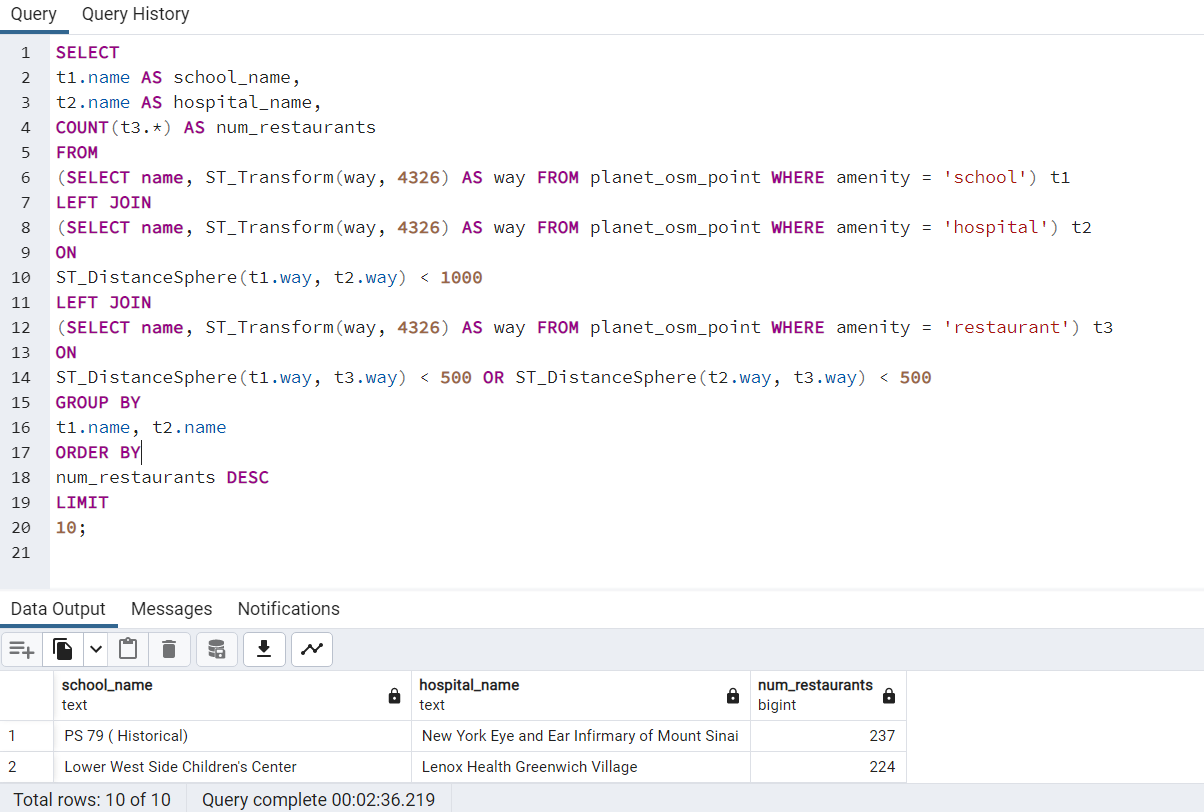
LIMIT

10;

**EXPLANATION:**

This query retrieves information about schools and hospitals that are located within 1000 meters of each other and have at least one restaurant located within 500 meters of either the school or the hospital. It first selects the name and location (latitude and longitude) of all schools, hospitals, and restaurants using the planet\_osm\_point table. Then, it performs a left join between the schools and hospitals table on the condition that the distance between the two points is less than 1000 meters. It then performs another left join between the result of the previous join and the restaurants table on the condition that the distance between either the school or the hospital and the restaurant is less than 500 meters. Finally, it groups the results by school and hospital names and sorts them in descending order of the number of restaurants found within the specified radius, and limits the output to the top 10.

**OUTPUT:**



**QUERY 5.2:**

Query to retrieve top 7 types of highways that pass through residential landuse areas in the OpenStreetMap data for the planet.

**CODE:**

SELECT r.highway, COUNT(\*) as num\_roads

FROM planet\_osm\_roads r

JOIN planet\_osm\_polygon p ON ST\_Intersects(r.way, p.way)

WHERE r.highway IS NOT NULL AND p.landuse = 'residential'

GROUP BY r.highway

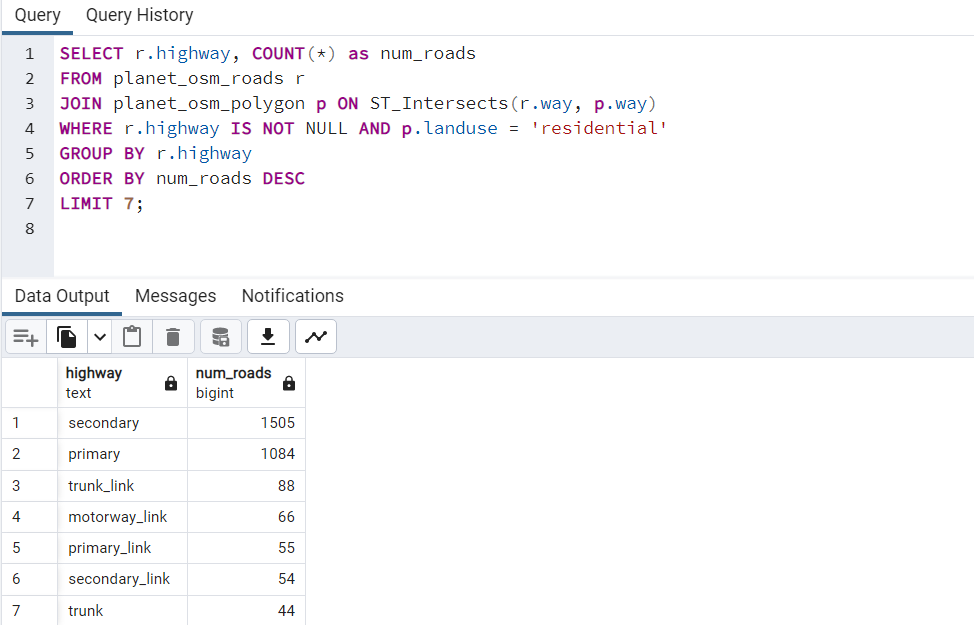
ORDER BY num\_roads DESC

LIMIT 7;

**EXPLANATION:**

This query selects the type of roads (highway column) from planet\_osm\_roads table and counts the number of roads of each type. It then joins the planet\_osm\_polygon table based on the intersection of the road and polygon geometries, and filters only the roads that intersect with polygons of residential land use type. Finally, the result is grouped by road type and sorted in descending order of number of roads, and only the top 7 road types are returned.

**OUTPUT:**



**QUERY 5.3:**

SQL query to retrieves the top 5 parks in New York City by the length of waterways intersecting them, and displays the park name, waterway name, and length of the intersecting waterway.

**CODE:**

SELECT p.name AS park\_name, w.name AS waterway\_name, ST\_Length(w.way) as length

FROM planet\_osm\_polygon p

JOIN planet\_osm\_line w ON ST\_Intersects(w.way, p.way)

WHERE w.waterway IS NOT NULL AND p.leisure = 'park' AND p.name IS NOT NULL

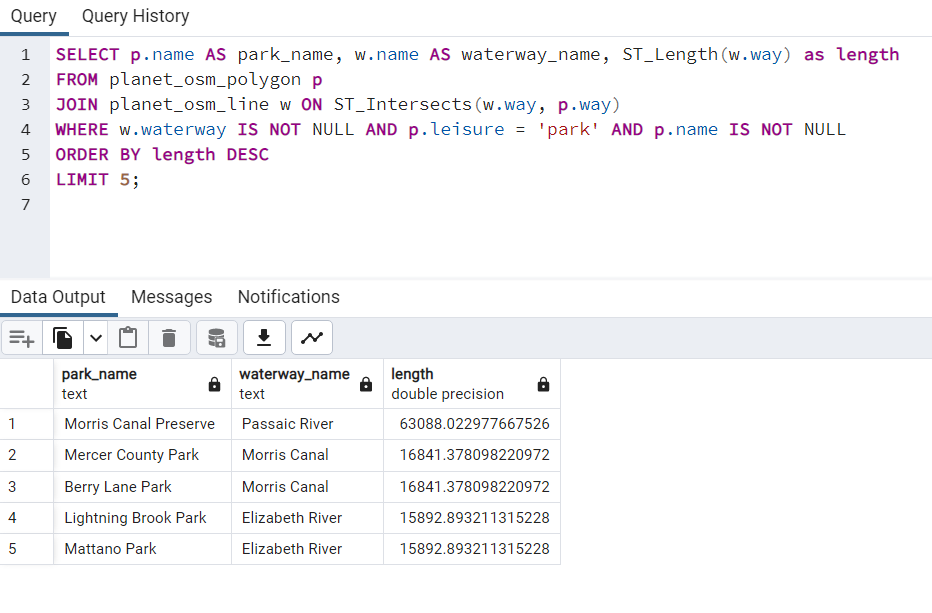
ORDER BY length DESC

LIMIT 5;

**EXPLANATION :**

To achieve this, the query joins the planet\_osm\_polygon table on the planet\_osm\_line table using the ST\_Intersects function to ensure only the waterways that intersect with the park's boundary are included in the result. The WHERE clause filters out any null park names and ensures that only waterways with a non-null waterway tag and a leisure tag of 'park' are considered. The result is sorted in descending order by the length of the intersecting waterway, and the top 5 results are returned.

**OUTPUT:**



**6. OPTIMIZE THE QUERIES TO SPEED UP EXECUTION TIME**

In Geographic Information Systems (GIS), optimization refers to the process of improving the performance and efficiency of queries and operations on geospatial data. Since GIS datasets are often large and complex, optimization is critical to ensure that queries and operations are executed as quickly as possible. This involves techniques such as indexing, clustering, and partitioning of data to minimize disk I/O and reduce query execution times. It also involves careful consideration of the type of queries being executed, and selecting the appropriate spatial and non-spatial operators and functions to maximize performance. By optimizing queries, GIS applications can handle larger volumes of data, support more users, and provide faster response times, leading to a better overall user experience.

**QUERY 6.1:**

Query to find all the restaurants within a 1 km radius of a specific location:

**CODE:**

SELECT name, ST\_X(way), ST\_Y(way)

FROM planet\_osm\_point

WHERE amenity = 'cafe' AND

ST\_Intersects(way, ST\_Transform(ST\_MakeEnvelope(-74.25909, 40.477399, -73.700181, 40.917577, 4326), 3857));

**EXPLANATION:**

This query retrieves the name, longitude, and latitude of all cafes within the bounding box of the New York City area.

This query is performing a Bitmap Heap Scan on the planet\_osm\_point table with a filter on the amenity column to only retrieve cafes, and a spatial filter using the st\_intersects function to retrieve cafes that are within a certain area defined by the geometry parameter. The query seems to be using parallelism with 2 planned workers.

**OPTIMIZATION:**

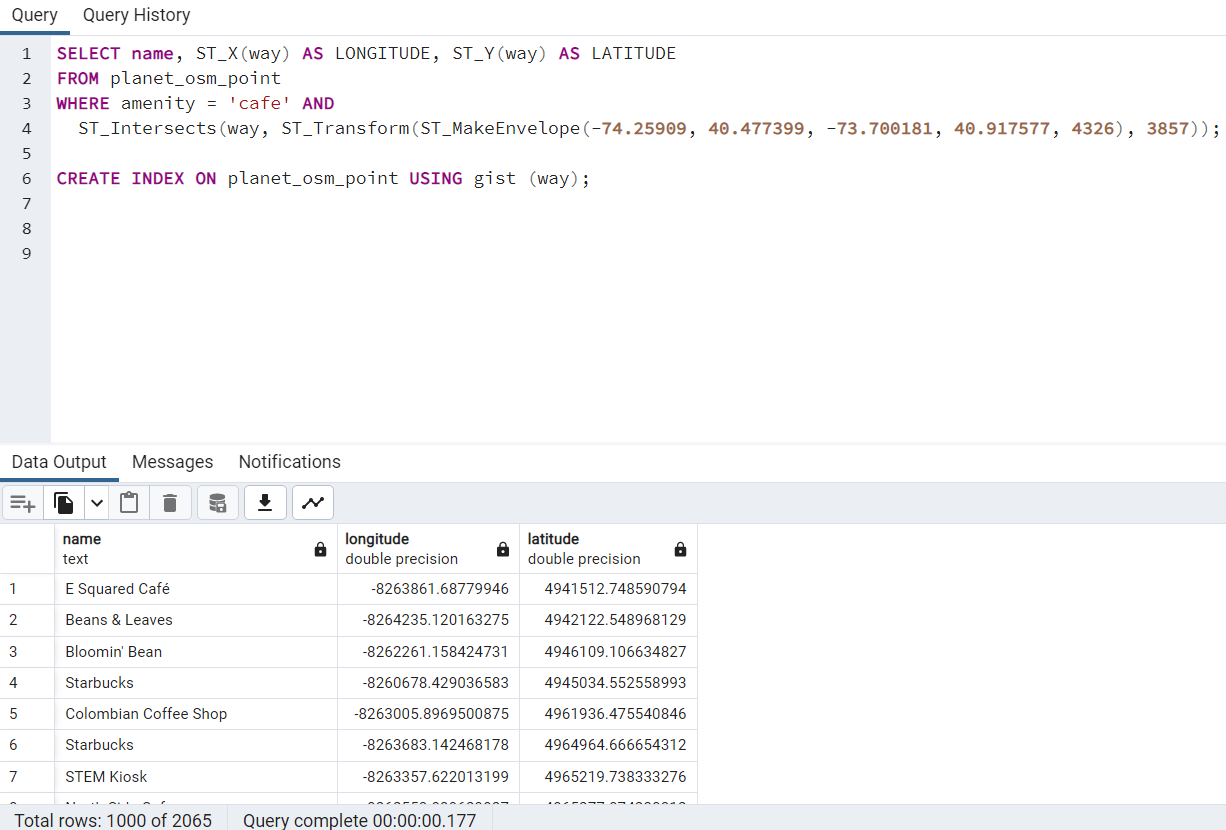
To optimize the query, we can create an index on the amenity column to speed up the filtering of points of interest:

CREATE INDEX ON planet\_osm\_point (amenity);

We can also create a spatial index on the way column to speed up the spatial join:

CREATE INDEX ON planet\_osm\_point USING gist (way);

**OUTPUT:**



**QUERY 6.2:**

Query to find all the universities in New York and returns their names and coordinates:

**CODE:**

SELECT name, ST\_X(way) AS lon, ST\_Y(way) AS lat

FROM planet\_osm\_point

WHERE amenity = 'university';

**EXPLANATION :**

This query selects the name, longitude (X coordinate), and latitude (Y coordinate) of all nodes in the planet\_osm\_point table where the amenity column is equal to 'university'.

**OPTIMIZATION:**

To optimize this query, we could create an index on the amenity column to speed up the filtering process:

CREATE INDEX amenity\_index ON planet\_osm\_point (amenity);

This index would allow the database to quickly find all nodes with an amenity value of 'university' without having to scan the entire table.

**COMPARISION OF QUERY PLANS:**

QUERY PLAN BEFORE OPTIMIZATION:

"Gather (cost=1000.00..10734.58 rows=75 width=36)"

" Workers Planned: 2"

" -> Parallel Seq Scan on planet\_osm\_point (cost=0.00..9727.08 rows=31 width=36)"

" Filter: (amenity = 'university'::text)"

The first query plan is a parallel sequential scan that scans the entire planet\_osm\_point table with no index usage, resulting in a total cost of 10734.58. This is not efficient and can result in slow performance for large tables.

QUERY PLAN AFTER OPTIMIZATION:

"Bitmap Heap Scan on planet\_osm\_point (cost=5.01..282.26 rows=75 width=36)"

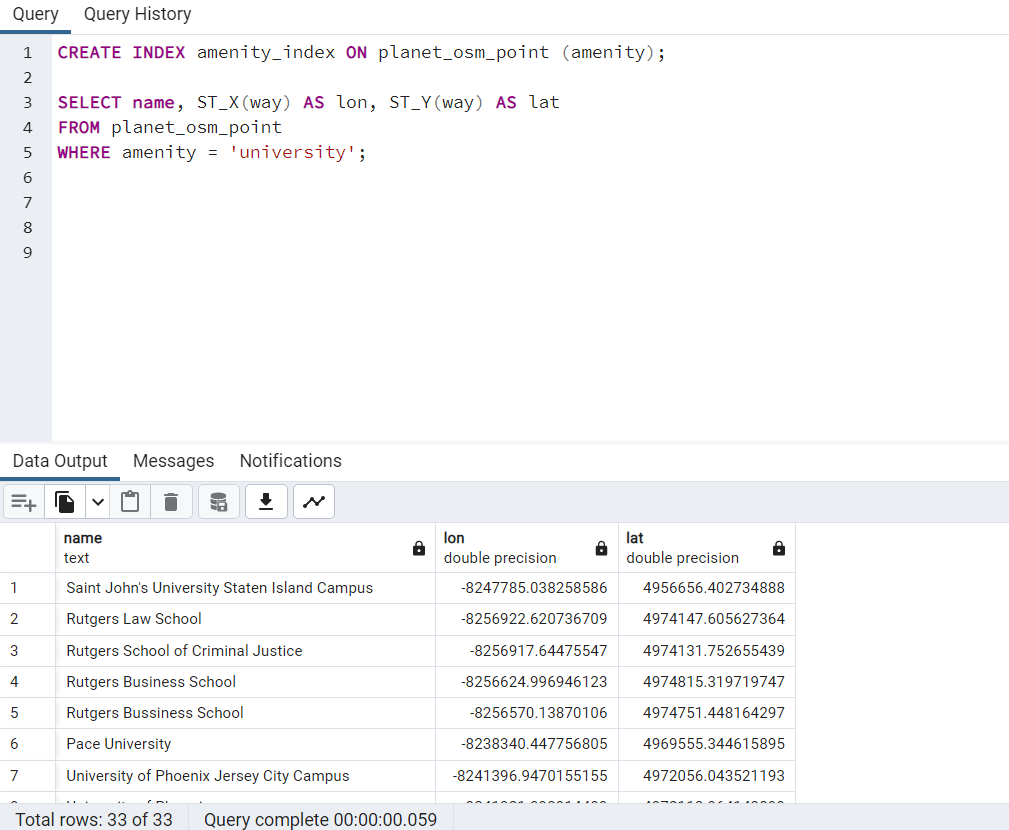
" Recheck Cond: (amenity = 'university'::text)"

" -> Bitmap Index Scan on amenity\_index (cost=0.00..4.99 rows=75 width=0)"

" Index Cond: (amenity = 'university'::text)"

The second query plan shows a more efficient approach. It uses an index on the amenity column to quickly filter rows that match the condition amenity = 'university', resulting in a much lower cost of 282.26. The Bitmap Heap Scan and Bitmap Index Scan operations indicate that a bitmap is used to speed up the filtering process. This plan is preferred over the first plan as it is faster and more efficient.

**OUTPUT :**



**7. N OPTIMIZATION OF QUERIES**

"Optimization of queries" refers to the process of improving the performance of database queries, typically by identifying and eliminating inefficiencies in the query execution plan. The goal of query optimization is to produce a plan that is both correct (i.e., returns the expected results) and efficient (i.e., minimizes the resources required to execute the query).

"N optimization of queries" likely refers to optimizing queries for a specific value of N, where N is some parameter or variable that affects the performance of the query. For example, N could be the number of rows in a table, the size of a buffer used for sorting, or the number of concurrent users accessing the database. By optimizing queries for specific values of N, it is possible to achieve better performance than would be possible with a general-purpose optimization strategy.

**QUERY 7.1:**

Query to find all the points of interest (POIs) in the planet\_osm\_point table that have the amenity tag set to 'restaurant'.

**CODE:**

SELECT name, ST\_X(way) as longitude, ST\_Y(way) as latitude

FROM planet\_osm\_point

WHERE amenity = 'restaurant'

LIMIT 10;

**EXPLANATION:**

The above query retrieves the names, longitudes, and latitudes of the first 10 restaurants from the planet\_osm\_point table in the OpenStreetMap database, where the amenity column is equal to 'restaurant'.

The ST\_X and ST\_Y functions are used to extract the longitude and latitude values, respectively, from the way column, which is a spatial data type that stores the location information of the point of interest.

The LIMIT clause is used to limit the number of rows returned to 10. This is useful when dealing with large tables to avoid retrieving unnecessary data and to improve query performance.

**N OPTIMIZATION:**

To optimize the query for the number of rows in the table, we can create an index on the amenity column:

CREATE INDEX amenity\_index ON planet\_osm\_point (amenity);

Now when we run the query, the database can use the index to quickly locate all the rows in the planet\_osm\_point table where the amenity column has the value 'restaurant'. This should make the query run much faster.

By setting a limit on the number of rows returned, we are optimizing the query for a specific value of N (the number of rows we want to retrieve), which can improve query performance.

**COMPARISION OF QUERY PLANS:**

QUERY PLAN BEFORE OPTIMIZATION:

"Bitmap Heap Scan on planet\_osm\_point (cost=88.39..7117.31 rows=7738 width=36) (actual time=1.045..8.111 rows=7288 loops=1)"

" Recheck Cond: (amenity = 'restaurant'::text)"

" Heap Blocks: exact=2568"

" -> Bitmap Index Scan on planet\_osm\_point\_amenity\_idx (cost=0.00..86.46 rows=7738 width=0) (actual time=0.667..0.668 rows=7288 loops=1)"

" Index Cond: (amenity = 'restaurant'::text)"

"Planning Time: 0.169 ms"

"Execution Time: 8.384 ms"

The first query plan is for a query that includes a sequential scan on the "planet\_osm\_point" table with a filter on the "amenity" column, followed by a limit operation that selects only the first 10 rows. The total cost of this plan is estimated to be 18.35 units, and the execution time is between 0.024 and 0.045 milliseconds.

QUERY PLAN AFTER OPTIMIZATION:

"Limit (cost=0.00..18.35 rows=10 width=36) (actual time=0.024..0.045 rows=10 loops=1)"

" -> Seq Scan on planet\_osm\_point (cost=0.00..14196.50 rows=7738 width=36) (actual time=0.023..0.044 rows=10 loops=1)"

" Filter: (amenity = 'restaurant'::text)"

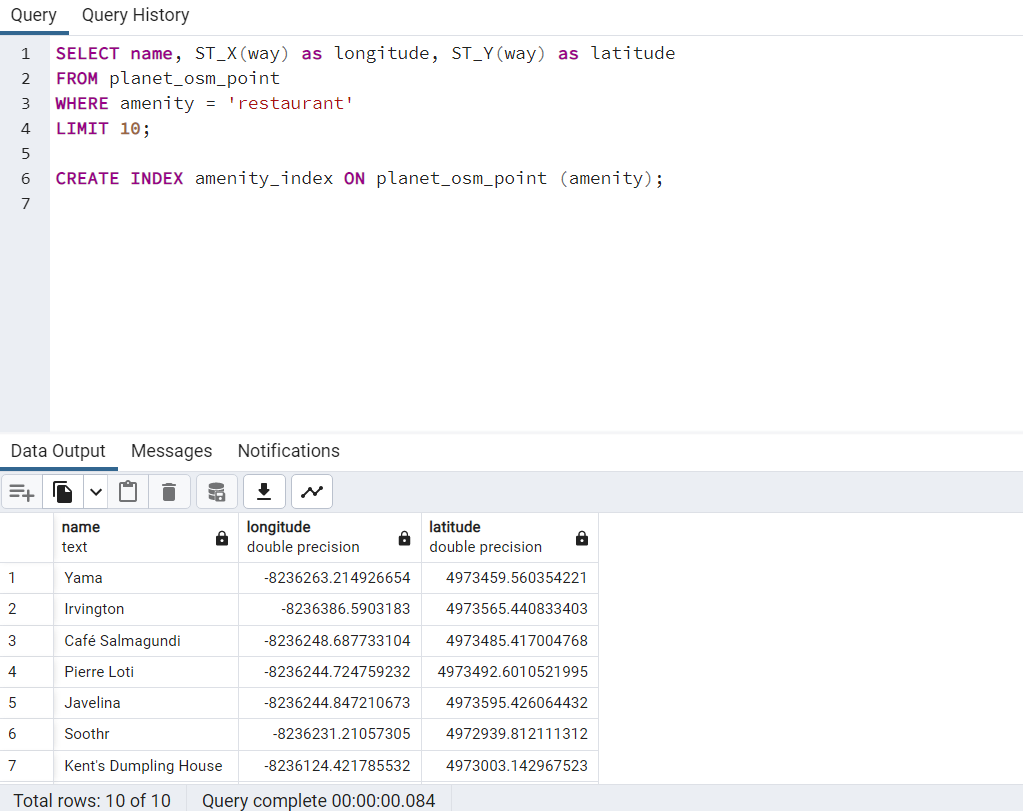
" Rows Removed by Filter: 287"

"Planning Time: 0.109 ms"

"Execution Time: 0.062 ms"

The first query plan is for a query that includes a bitmap index scan on the "planet\_osm\_point\_amenity\_idx" index and a heap scan on the "planet\_osm\_point" table. The bitmap index scan filters the rows based on the "amenity" column, and the heap scan retrieves the selected rows. The total cost of this plan is estimated to be between 88.39 and 7117.31 units, and the execution time is between 1.045 and 8.111 milliseconds.

**OUTPUT:**



**QUERY 7.2:**

Query to find all the buildings in the city of New York that are listed as historic landmarks.

**CODE:**

SELECT osm\_id, name, historic, building

FROM planet\_osm\_polygon

WHERE building IS NOT NULL AND historic = 'memorial'

**EXPLANATION:**

The query is selecting specific columns from the "planet\_osm\_polygon" table in the OpenStreetMap database. It filters the results by selecting only rows where the "building" column is not null and the "historic" column equals "memorial". It then returns the "osm\_id", "name", "historic", "building", and the well-known text representation of the "way" column as a string. The "way" column represents the polygonal geometry of the feature. The query is not limited to a specific number of results, so it will return all rows that match the filter criteria.

**N OPTIMIZATION:**

SELECT p1.osm\_id, p1.name, p1.historic, p1.building, ST\_AsText(p1.way)

FROM planet\_osm\_polygon p1

JOIN planet\_osm\_polygon p2 ON ST\_Intersects(p1.way, p2.way)

WHERE p1.building IS NOT NULL AND p1.historic = 'memorial' AND p2.name = 'Manhattan'

In the first query, the subquery (SELECT way FROM planet\_osm\_polygon WHERE name = 'Manhattan') is executed for each row of the main query. This can be inefficient if there are many rows in the table that match the WHERE clause.

In the second query, the join with planet\_osm\_polygon is used to limit the number of rows processed. The join condition ST\_Intersects(p1.way, p2.way) is used to find all rows in p1 that intersect with the way of the row in p2 where name = 'Manhattan'. This limits the number of rows processed in p1 to only those that intersect with Manhattan, which can be much smaller than the entire table.

In this case, N is the number of rows in the planet\_osm\_polygon table that match the WHERE condition building IS NOT NULL AND historic = 'memorial'.

**COMPARISION OF QUERY PLANS:**

QUERY PLAN BEFORE OPTIMIZATION:

"Gather (cost=1000.00..57295.71 rows=98 width=39) (actual time=180.168..331.285 rows=11 loops=1)"

" Workers Planned: 2"

" Workers Launched: 2"

" -> Parallel Seq Scan on planet\_osm\_polygon (cost=0.00..56285.91 rows=41 width=39) (actual time=165.974..264.264 rows=4 loops=3)"

" Filter: ((building IS NOT NULL) AND (historic = 'memorial'::text))"

" Rows Removed by Filter: 549431"

"Planning Time: 0.094 ms"

The first query plan performs a parallel sequential scan on the planet\_osm\_polygon table with a filter condition on building and historic columns. The actual time to execute this query plan is around 331.285 milliseconds, and the number of rows returned is only 11.

QUERY PLAN AFTER OPTIMIZATION:

"Gather (cost=1000.29..62075.94 rows=1 width=39) (actual time=504.228..532.457 rows=7 loops=1)"

" Workers Planned: 2"

" Workers Launched: 2"

" -> Nested Loop (cost=0.29..61075.84 rows=1 width=39) (actual time=309.341..338.514 rows=2 loops=3)"

" -> Parallel Seq Scan on planet\_osm\_polygon p2 (cost=0.00..56285.91 rows=1 width=169) (actual time=256.120..298.021 rows=1 loops=3)"

" Filter: (name = 'Manhattan'::text)"

" Rows Removed by Filter: 549433"

" -> Index Scan using planet\_osm\_polygon\_way\_idx on planet\_osm\_polygon p1 (cost=0.29..4789.92 rows=1 width=208) (actual time=11.312..40.478 rows=2 loops=3)"

the second query plan also performs a parallel scan on the planet\_osm\_polygon table, but it uses a nested loop join with an index scan on planet\_osm\_polygon\_way\_idx table to find the results. The actual time to execute this query plan is around 338.514 milliseconds, and the number of rows returned is 2.

Overall, the second query plan is better than the first query plan because it uses an index scan and a nested loop join, which are more efficient than a sequential scan. Additionally, the second query plan returns fewer rows, which means it can further reduce the execution time.

**OUTPUT:**

