***SGD LAB EXP – 5***

**Name** : Aditi Chhajed; **Reg. No.** : 221081009

**Branch** : IT ; **Course Instructor** : Prof. Vedashree Awati

Name : Aditi Chhajed

Reg.No : 22108100

***Aim:***

Executing PostGIS Geometry type queries.

***Theory:***

**Spatial Schemas and Tables Overview**

***1. Schemas:***

* In PostgreSQL, schemas act as containers that group related tables, views, and other objects, making it easier to organize and manage databases, especially when handling extensive datasets.
* In PostGIS (a spatial extension for PostgreSQL), schemas can be structured to categorize spatial data according to criteria such as geographic regions, data types, or functional purpose. For example, creating an `india` schema helps consolidate geographic data specific to India, such as administrative boundaries, roads, rivers, and landmarks. This makes it simpler to manage and query geographic datasets without cluttering other database elements.

***2. Spatial Tables:***

* Spatial tables contain geometry columns used for storing spatial data types. These tables often correspond to real-world features, such as points (e.g., individual landmarks), lines (e.g., highways), and polygons (e.g., state boundaries).
* For example:
* A `pts` table might include rows for various landmarks, each with their geographic coordinates.
* A `lines` table could hold line features like highways or rivers, defined as connected sequences of points.
* A `polys` table could contain polygons representing the boundaries of states or districts within India.

**Primary Keys and Geometry Columns**

***1. Primary Keys:***

* Each spatial table typically has a primary key (e.g., `gid`) that uniquely identifies every record. In spatial data management, this ensures each geographic feature, such as a building, a road segment, or a park, is distinctly identified.
* Example: In a table `india.roads`, each road might be assigned a unique `gid` to distinguish it from other roads.

***2. Geometry Columns:***

* Geometry columns in PostGIS store spatial data such as points, lines, and polygons. These columns are defined with specific data types, such as `POINT`, `LINESTRING`, and `POLYGON`.
* For example, in a table storing river data, a geometry column could be of type `LINESTRING` to represent the path of a river.
* The `AddGeometryColumn()` function is frequently used to create these columns while registering them with the PostGIS system catalog. This allows for the use of spatial indexes and ensures validation of the spatial data.

**Spatial Reference Systems (SRS)**

***1. SRID (Spatial Reference Identifier):***

* The SRID defines the coordinate system used for geometry data, enabling precise geographic positioning. For instance, SRID 4326 represents the WGS 84 coordinate system, which is widely used in GPS.
* Example: When storing locations of cities across India, using SRID 4326 ensures that the geographic coordinates are compatible with GPS data.
* The functions `ST\_GeomFromText()` and `ST\_SetSRID()` can assign an SRID to spatial data, guaranteeing that all geometries adhere to a consistent coordinate system for accurate spatial operations.

**Geometry Types and Related Functions**

***1. Points, Lines, and Polygons:***

* POINT: Represents a specific location in space, such as a monument or a building.
* Example: The location of the Taj Mahal could be stored as a `POINT` with its latitude and longitude.
* LINESTRING: Represents a continuous line formed by connecting a series of points, commonly used to model features like roads or rivers.
* Example: A `LINESTRING` can represent National Highway 44, defined by a series of geographic points tracing its path.
* POLYGON: Represents an enclosed area defined by a series of points that form a closed loop.
* Example: The boundaries of New Delhi can be stored as a `POLYGON` to depict its area.

***2. Key Functions:***

* ST\_GeomFromText(): Converts a textual representation of a geometry (e.g., "POINT(77.231 28.613)") into a PostGIS geometry object with a specified SRID.
* Example: `ST\_GeomFromText('POINT(77.231 28.613)', 4326)` creates a point at longitude 77.231 and latitude 28.613 using SRID 4326.
* ST\_SetSRID(): Assigns an SRID to a geometry object that may not already have one.
* ST\_MakePoint(): Creates a point geometry using specified coordinates.
* Example: `ST\_MakePoint(77.231, 28.613)` creates a point geometry for a location with specified longitude and latitude.

**Advanced Geometry Types**

***1. 3D and 4D Geometries:***

* PostGIS supports:
* 3D (XYZ) geometries: Adds a third dimension, typically for elevation.
* Example: A 3D point could represent the location of the Qutub Minar with latitude, longitude, and height above sea level.
* 3D with Measure (XYM) geometries: Includes a "measure" value, often used for representing distances or other metrics.
* Example: A route stored with an XYM line might represent distance traveled along each segment.
* 4D (XYZM) geometries: Combines both elevation and measure values for comprehensive spatial representation.
* Example: Flight paths, where each point contains latitude, longitude, altitude, and a time or distance measure.

***2. Multipart Geometries:***

* MULTIPOINT, MULTILINESTRING, and MULTIPOLYGON types represent groups of related geometries.
* Example: A `MULTIPOLYGON` can store the boundaries of all islands in the Andaman and Nicobar group as a single feature, even if they are spatially separated.

**Mixed Geometry Storage**

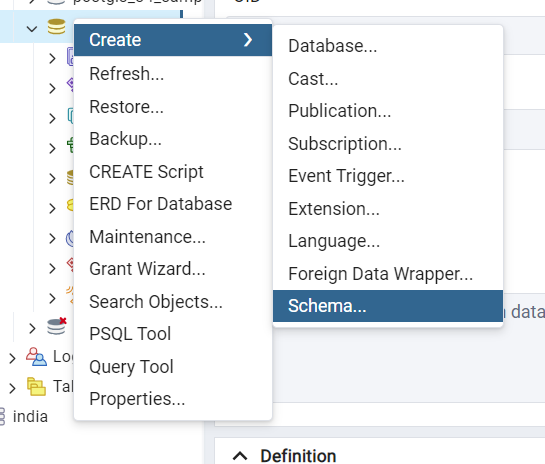
***1. Mixed Geometry Columns:***

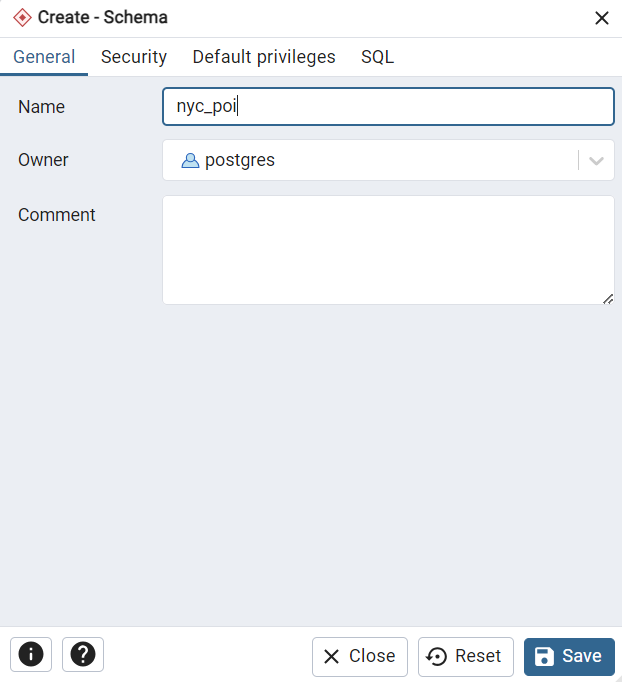
* PostGIS allows storing different geometry types within a single column, enabling a table to hold points, lines, and polygons.
* Example: In an `india.mixed` table, a single column could contain landmarks as points, rivers as lines, and parks as polygons. This flexibility simplifies data storage but may complicate certain queries or interactions with external tools.

***2. Usage Example:***

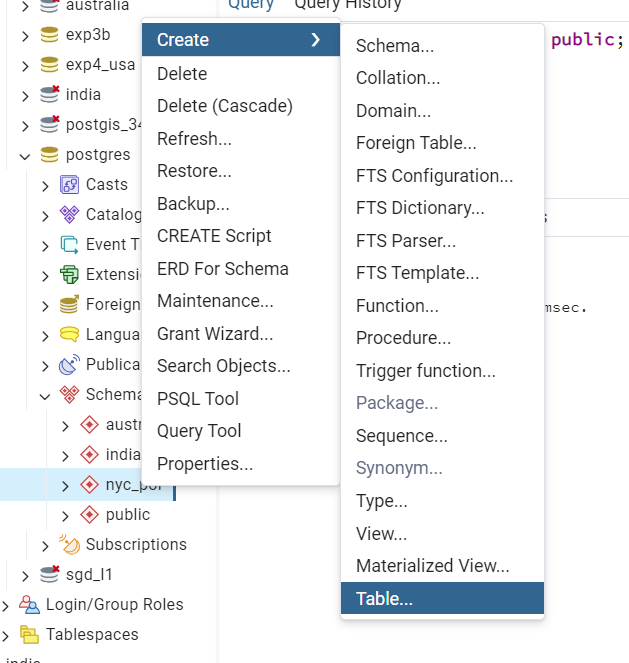
* Consider an `india.mixed` table. A row might contain a `POINT` representing the Gateway of India, another row could store a `LINESTRING` representing the Ganga River's path, and a third row could hold a `POLYGON` outlining a national park. This approach provides a convenient way to manage different geographic features within a single table.

***Implementation:***

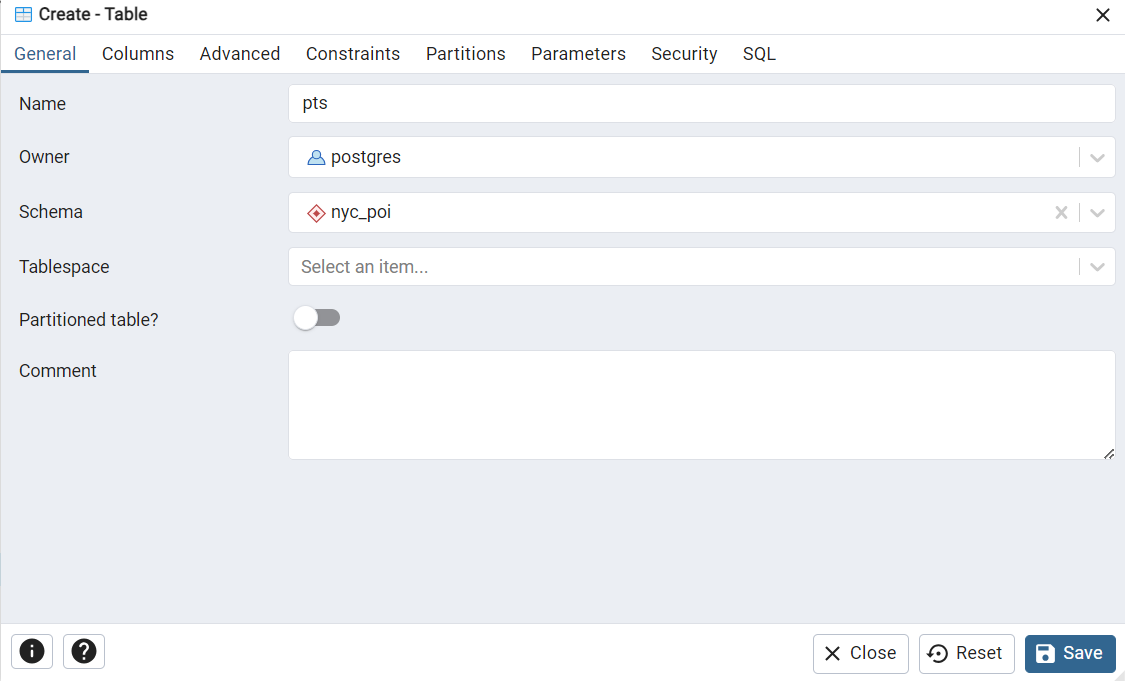
1. **Create a new empty spatial table:**
   1. In the **Browser** pane within pgAdmin, right-click on the **Schemas** node beneath the **Lesson3db** database and select **Create > Schema**. ******
   2. Set the **Name** of the schema to **nyc\_poi** and its **Owner** to **postgres,** then click **Save** to create the schema.



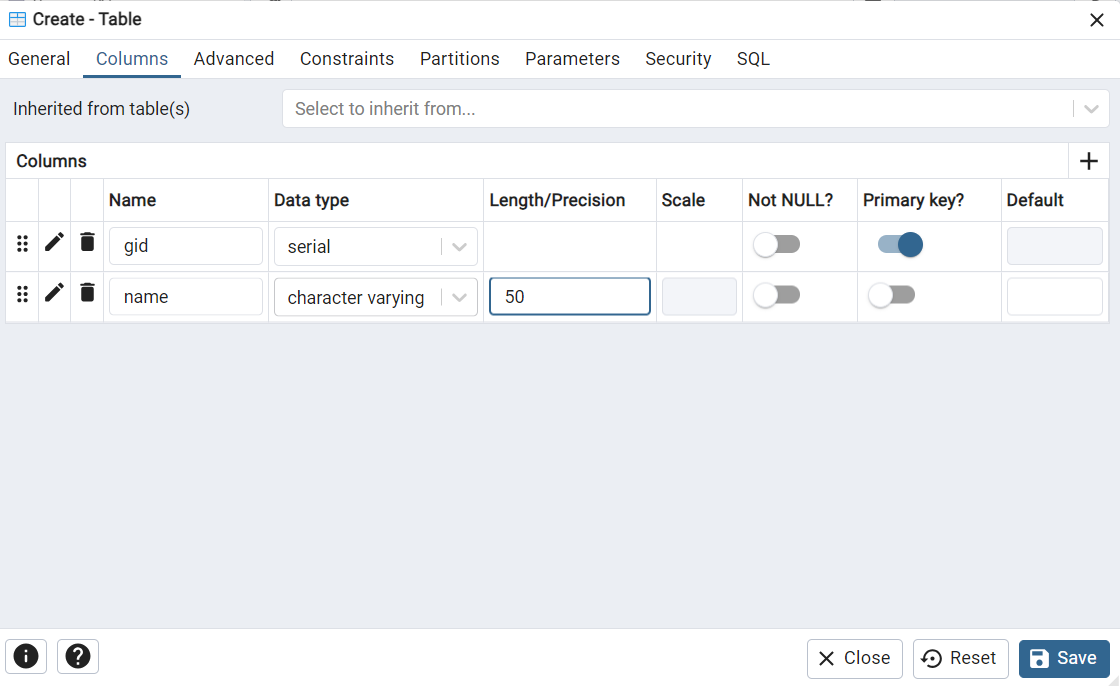
* 1. Expand the object listing associated with the **nyc\_poi** schema.
  2. Right-click on the **Tables** node, and select**Create > Table**.



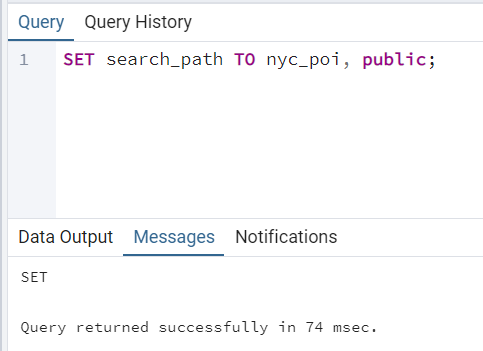
* 1. Set the table's **Name** to **pts** and its **Owner** to **postgres**.



* 1. Under the **Columns** tab, click the **+** button.
  2. For the new column, set the **Name** to **gid** (geometry ID) and the **Data type** to **serial**. This data type is roughly equivalent to the AutoNumber type we saw in Access. Define this column as the table's **Primary key**.
  3. Repeat the previous step to create a column called **name**. Set its **Data type** to **character varying** and its **Length** to **50**. This column should not be the Primary key.



* 1. Click **Save** to dismiss the dialog and create the table. Before adding the geometry column to the table, let's recall the pgAdmin search path. It's not set to include the nyc\_poi schema, so let's do that first.
  2. Reset the search path by executing the following statement in a **Query** window

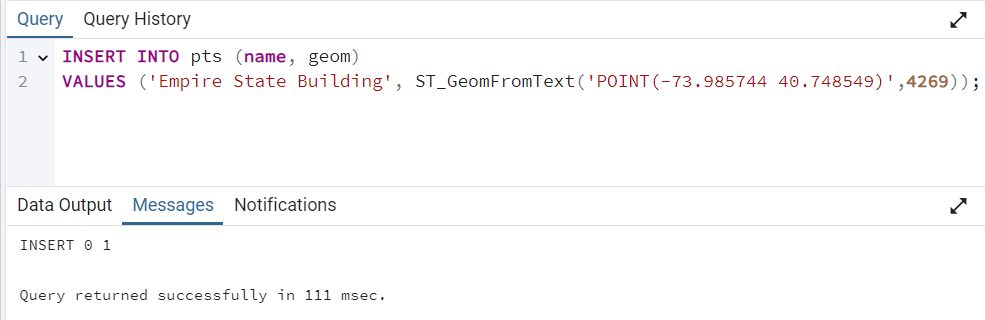


1. We first create an extension of postgis and then add the geometry column.

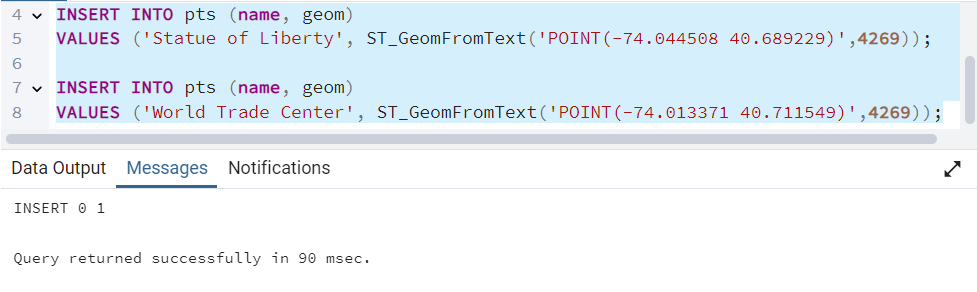
(or else if throws an error)

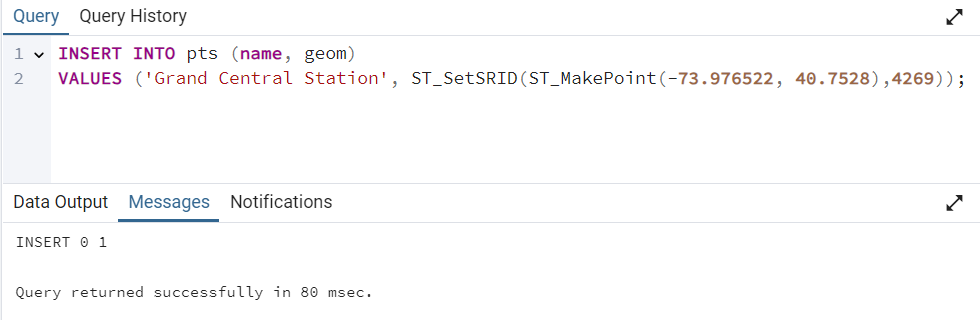


1. **Add Rows to the Spatial Table:**
2. Insert Rows into the pts Table: Open a new Query Tool window in pgAdmin. Execute the following **INSERT** statement to add the first point:

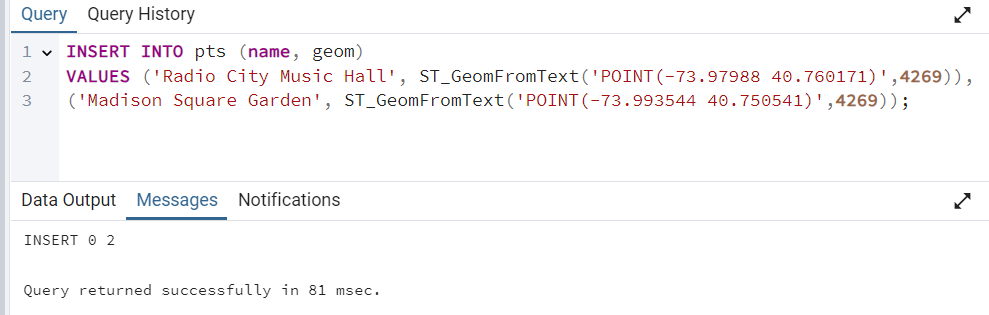


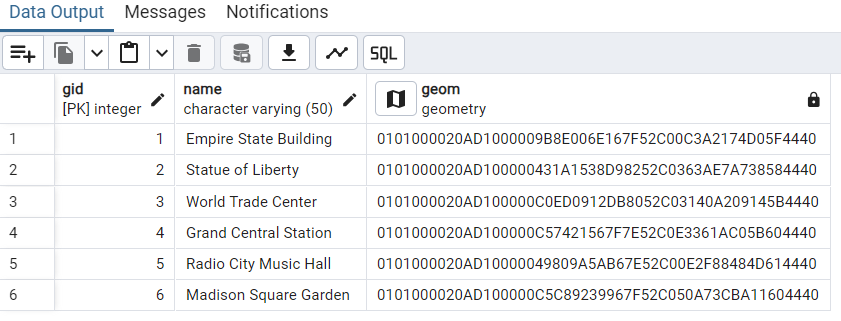
*This statement uses the ST\_GeomFromText() function to convert the specified point coordinates (longitude and latitude) into a geometry format, using SRID 4269.*

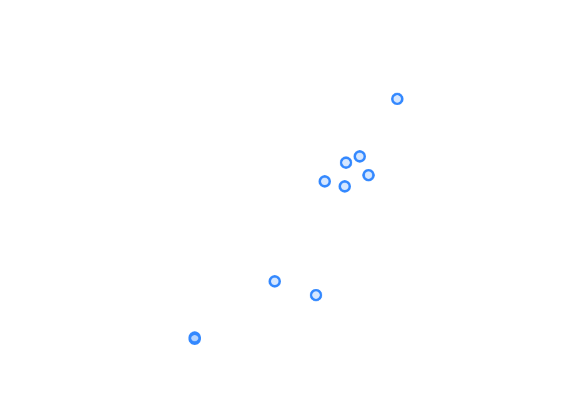
1. **Add More Rows**: Execute additional INSERT statements to add more landmarks: 
2. Use ST\_SetSRID and ST\_MakePoint:

******

1. Insert Multiple Rows in a Single Statement:

******

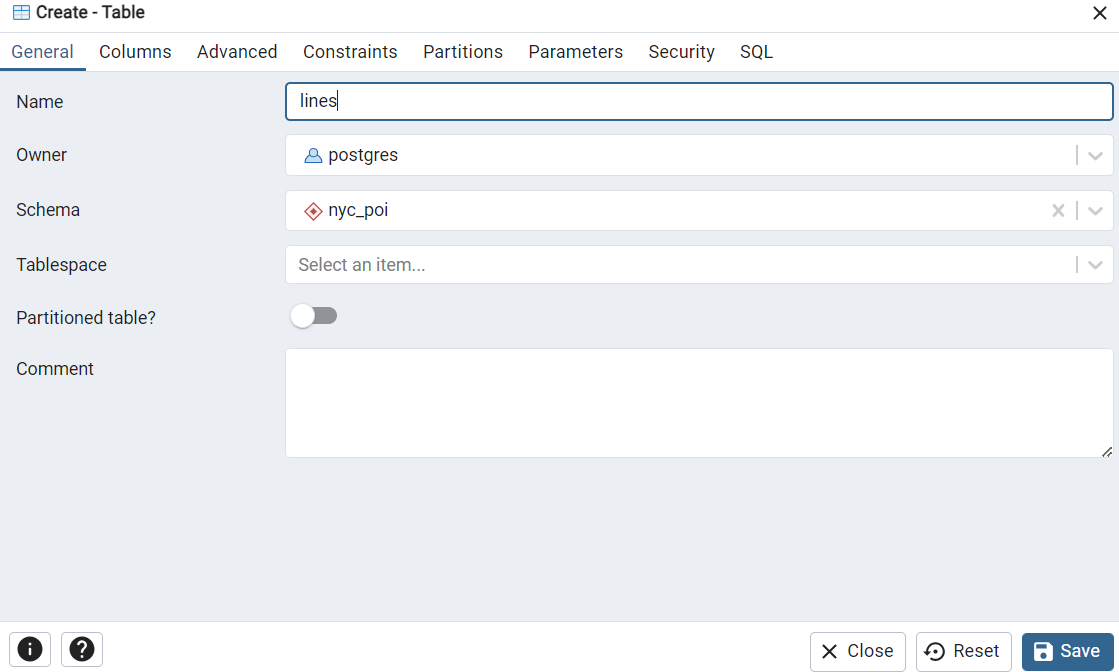
1. Confirm the Insertions: To confirm that the INSERT statements executed successfully, **right click on the pts table in the pgAdmin** window and select **View/Edit Data > All Rows**. This will allow you to view the data you've inserted into the table. 



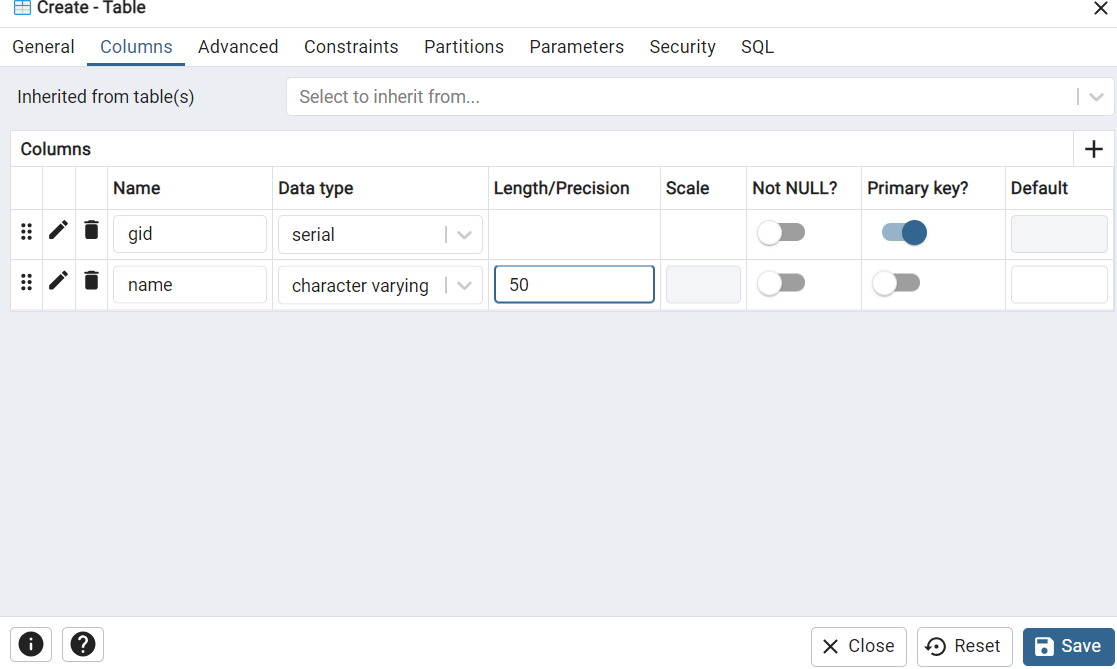
Geom output of the same

1. **Create and Populate a Table of Linestrings:**
2. Create a New Table for **Linestrings**: In the **nyc\_poi** schema, create a new table to hold line features. Follow these steps in pgAdmin:

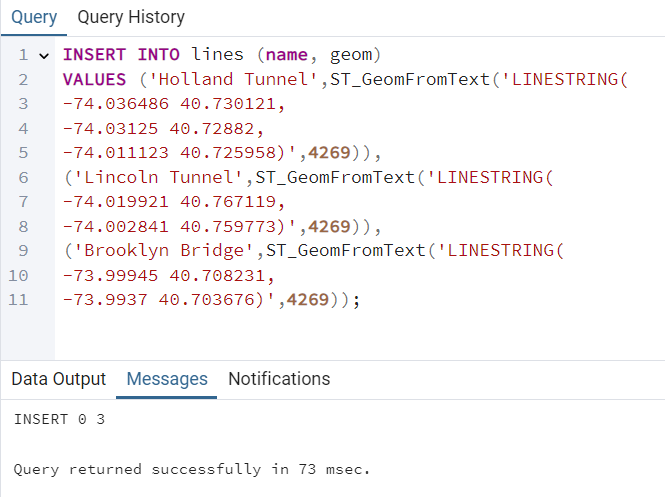
* Right-click on the Tables node within the nyc\_poi schema and select Create > Table.
* Name the table **lines** and set the Owner to **postgres**

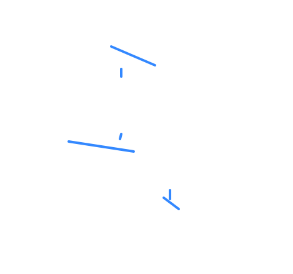


1. Under the **Columns** tab, click the **+** button.
2. For the new column, set the **Name** to **gid** (geometry ID) and the **Data type** to **serial**. This data type is roughly equivalent to the AutoNumber type we saw in Access. Define this column as the table's **Primary key**.
3. Repeat the previous step to create a column called **name**. Set its **Data type** to **character varying** and its **Length** to **50**. This column should not be the Primary key.



1. Add the Geometry Column Using **AddGeometryColumn**(): Before adding the geometry column, ensure that the search path is set correctly. 
2. **Insert Rows** into the lines Table: Add lines representing various notable road or bridge features in nyc\_poi using **INSERT** statements:

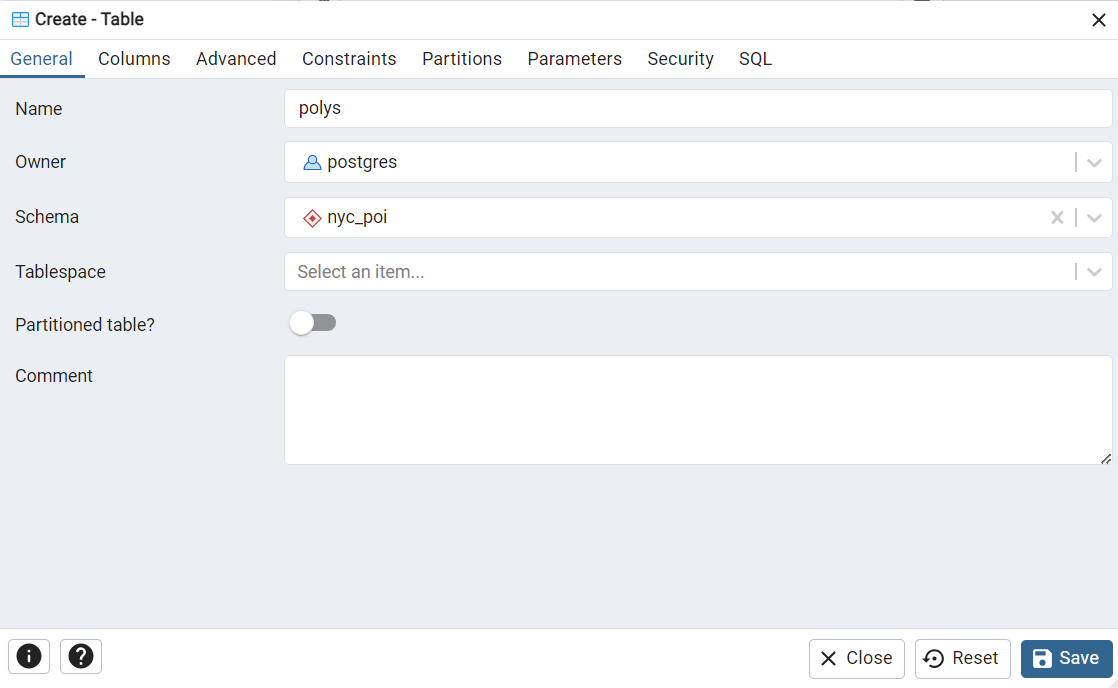
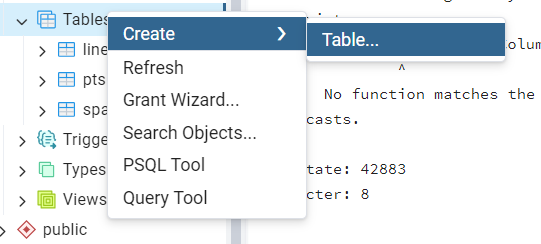




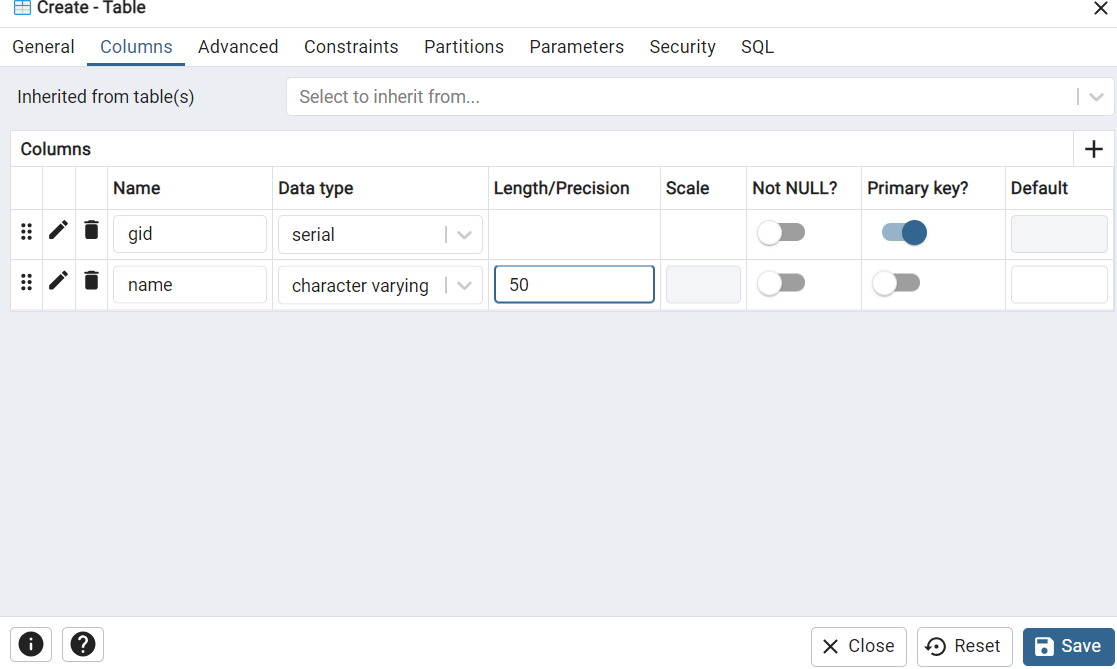
GEOM output of the lines table.

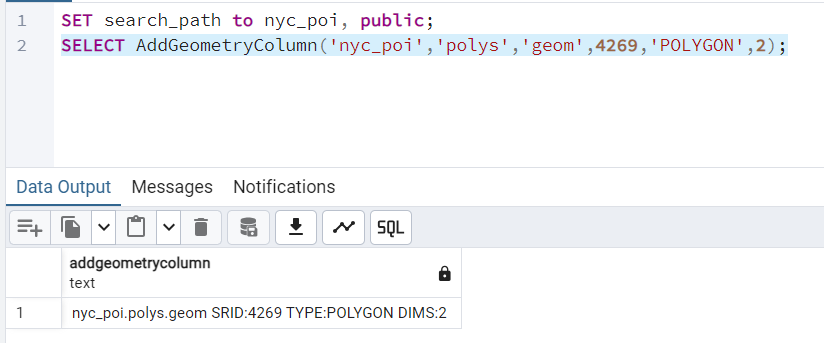
1. **Create and Populate a Table of Polygons:**
2. Create a New Table for **Polygon**: In the **nyc\_poi** schema, create a new table to hold polygon features. Follow these steps in pgAdmin:

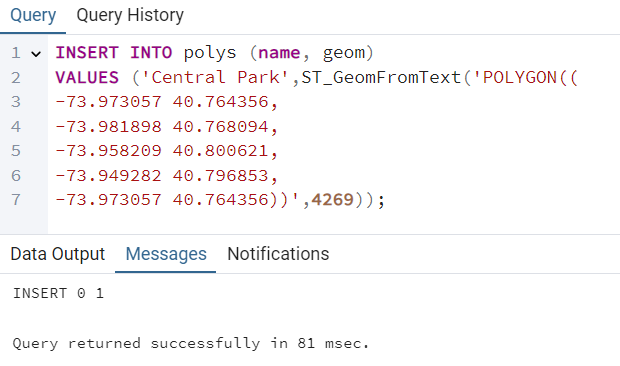
* Right-click on the Tables node within the nyc\_poi schema and select Create > Table.
* Name the table **polys** and set the Owner to **postgres**



1. Under the **Columns** tab, click the **+** button.
2. For the new column, set the **Name** to **gid** (geometry ID) and the **Data type** to **serial**. This data type is roughly equivalent to the AutoNumber type we saw in Access. Define this column as the table's **Primary key**.
3. Repeat the previous step to create a column called **name**. Set its **Data type** to **character varying** and its **Length** to **50**. This column should not be the Primary key.

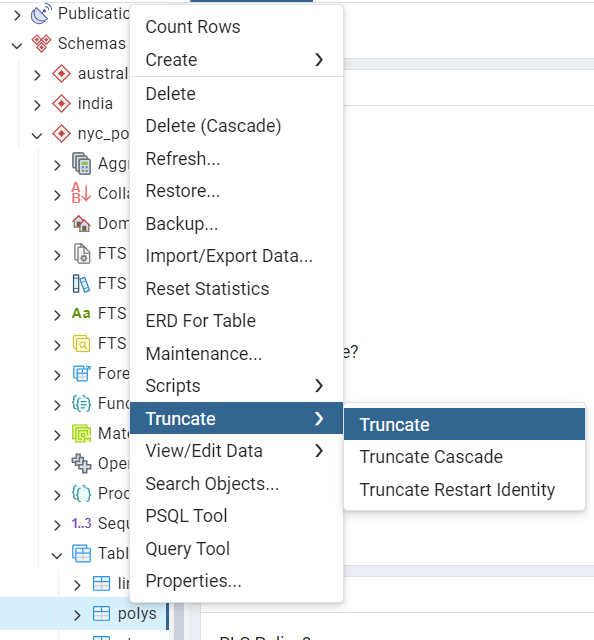


1. Add the Geometry Column Using **AddGeometryColumn**(): Before adding the geometry column, ensure that the search path is set correctly. 
2. **Insert Rows** into the lines Table: Add lines representing various notable road or bridge features in nyc\_poi using **INSERT** statements:



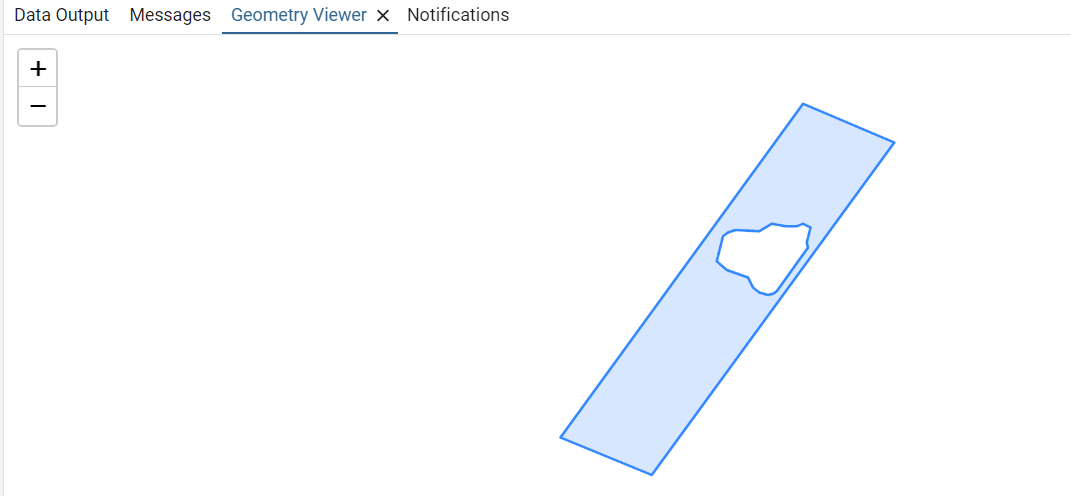


* *The first X/Y (lon/lat) pair should be the same as the last (to close the polygon).*
* Note that the coordinate list is enclosed in an additional set of parentheses. This set of parentheses is required because polygons are actually composed of potentially multiple rings. Every polygon has a ring that defines its exterior. Some polygons also have additional rings that define holes in the interior. When constructing a polygon with holes, the exterior ring is supplied first, followed by the interior rings. Each ring is enclosed in a set of parentheses, and the rings are separated by commas.  
  To see an example, let's add Central Park again, this time cutting out the large reservoir near its center.
* First, let's remove the original Central Park row. In pgAdmin, right-click on the **polys** table and select **Truncate > Truncate**. Note that this deletes all rows from the table.

**

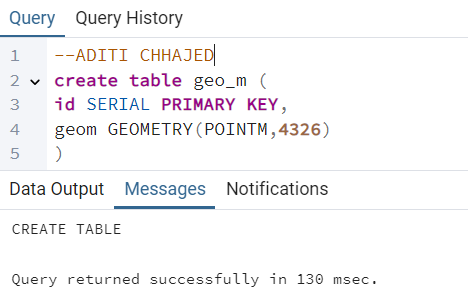
* Add Central Park (minus the reservoir) back into the table using this statement.

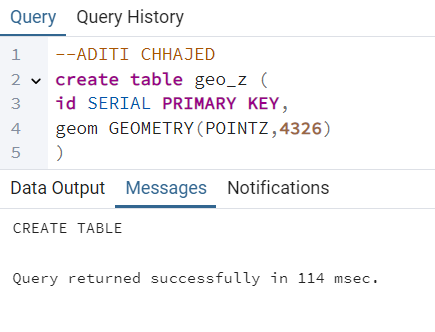
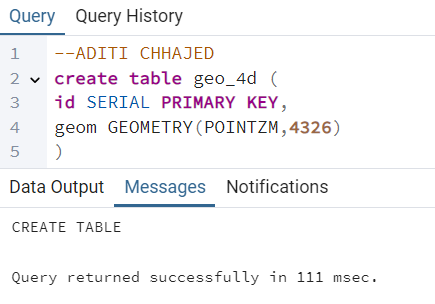
**

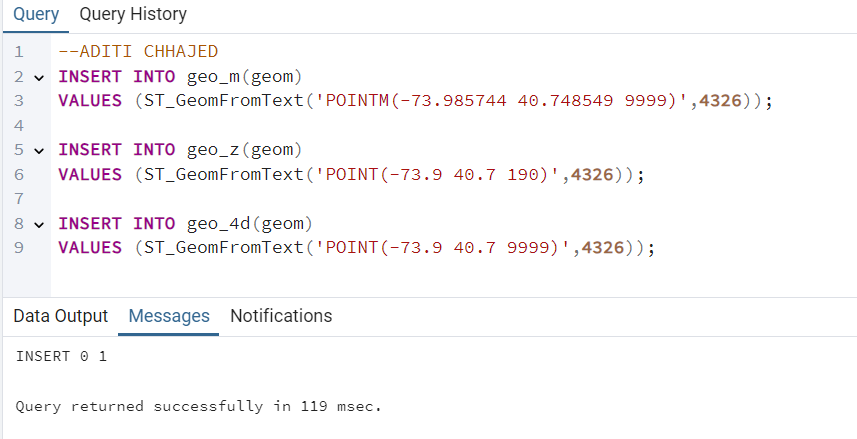
**

1. **3D and 4D Geometries:**

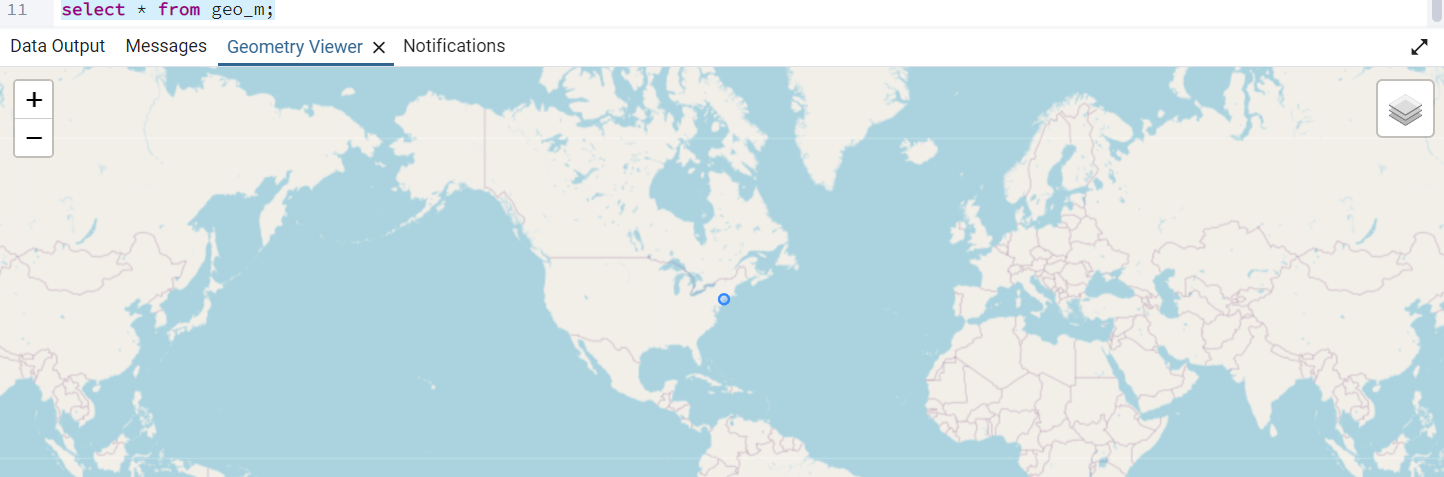
Creating tables for 3d, 4d geometries :

****

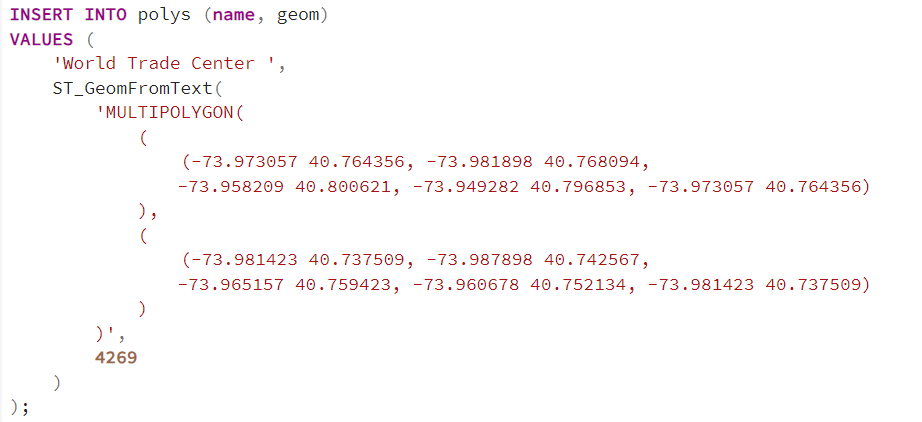
**** ****



**Geom outputs -**

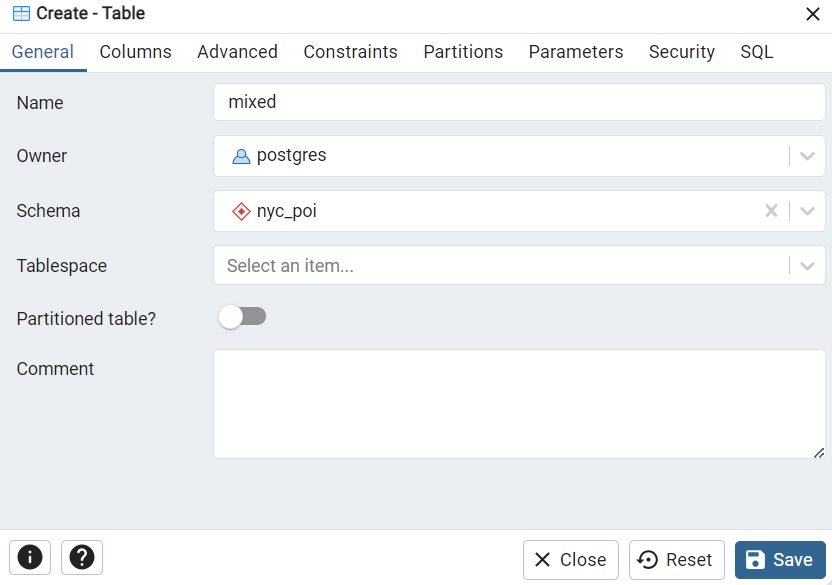
****

1. **Multipart Geometries:**
   * 1. PostGIS provides support for features with Multiple parts through the **MULTIPOINT, MULTILINESTRING, and MULTIPOLYGON** data types
     2. Let us see an example of **Multipart Polygon** (MULTIPOLYGON): A MULTIPOLYGON can represent multiple disconnected areas, such as groups of islands or regions.
     3. The footprints of the World Trade Center Towers 1 and 2 (now fountains in the 9/11 Memorial) can be represented as a single multipart polygon as follows:

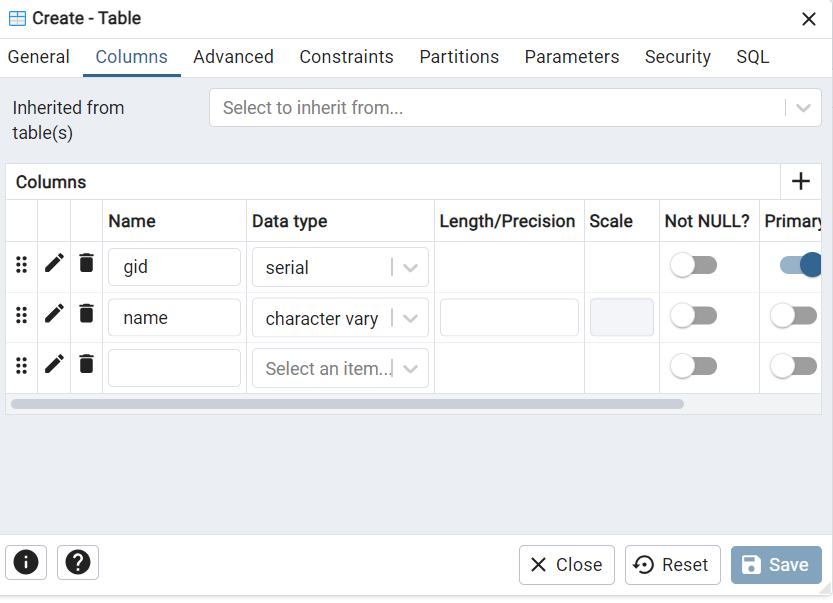


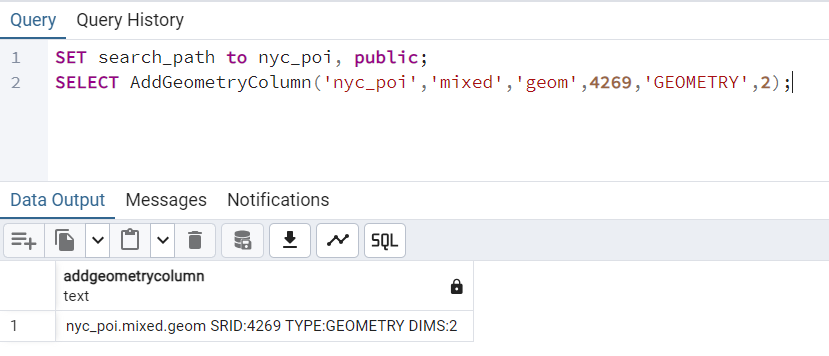
1. **Mixing Geometries:**
2. Create the **mixed Table in the nyc\_poi** Schema:

* Give the table a name of **mixed**.

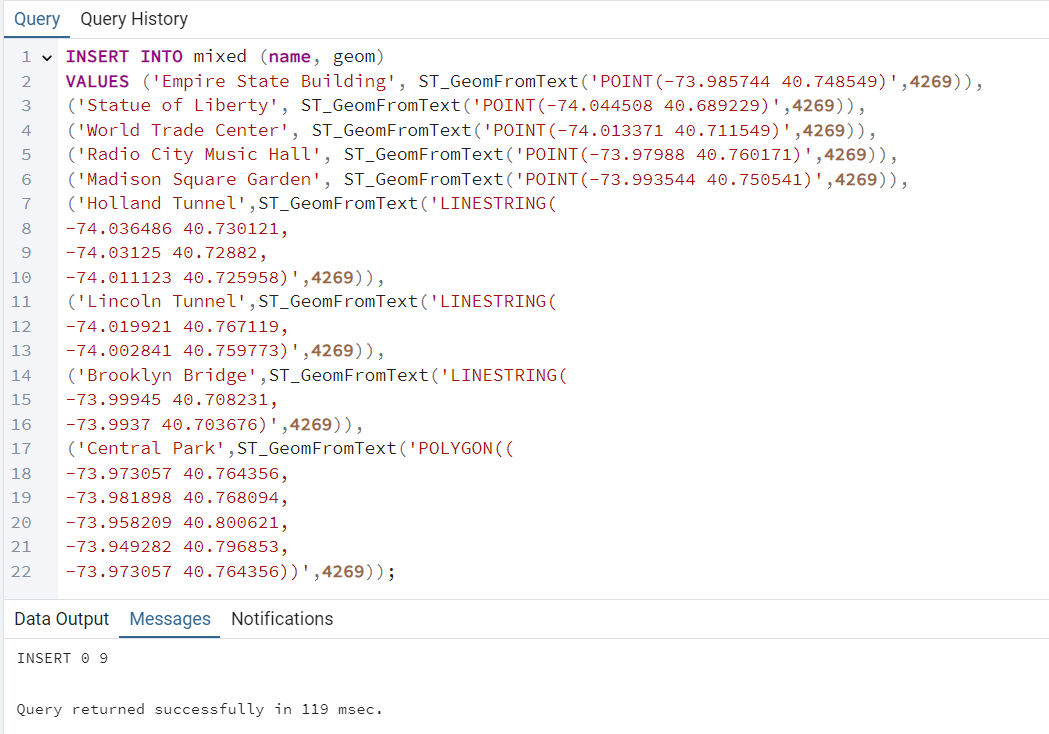


* The table should have the same column definitions, with the exception that the geometry type should be set to GEOMETRY

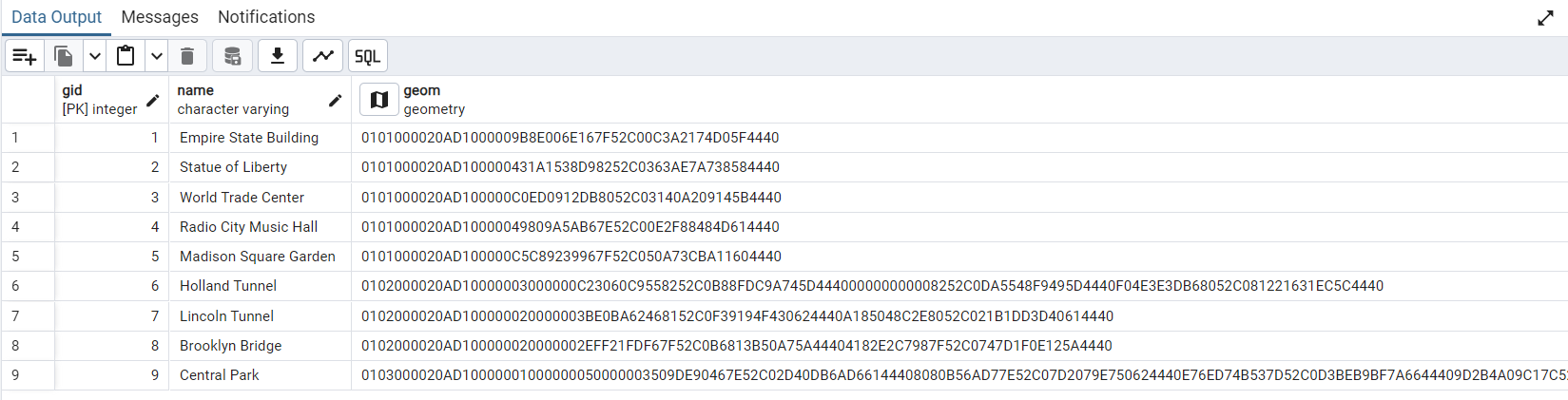




1. Add the features to this new table:



1. In the pgAdmin window right-click on the **mixed** table and select **View/Edit Data** > **All Rows** to confirm that the INSERT statement executed properly.



***Conclusion:***

*In this experiment, we* ***successfully created and managed spatial data in PostgreSQL*** *using* ***PostGIS****, exploring* ***schemas, spatial tables, and diverse geometry types*** *like* ***POINT, LINESTRING, POLYGON, 3D – (XYZ,XYM), 4D (XYZM), multipart and mixing geometries****.*

*By organizing data into different schemas and tables with geometry columns, we gained a practical understanding of managing geographic features and executing* ***spatial operations, such as inserting and querying geographic data in structured formats****.*

*Additionally, this experiment demonstrated how different geometry types can be stored and queried within a single table, providing flexibility and efficiency in handling complex spatial database management.*