

# Postgre/PostGIS Tutorial

---

## *Introduction to SQL*



University  
of Colorado  
Denver

Created by: Ricardo Oliveira

*ricardo.oliveira@ucdenver.edu*

July 2014

On the previous tutorial we learned how to install, how to load data, and display spatial information through the use of PostgreSQL. Now we will take a look at the basics of SQL and how we can use it to manipulate our data.

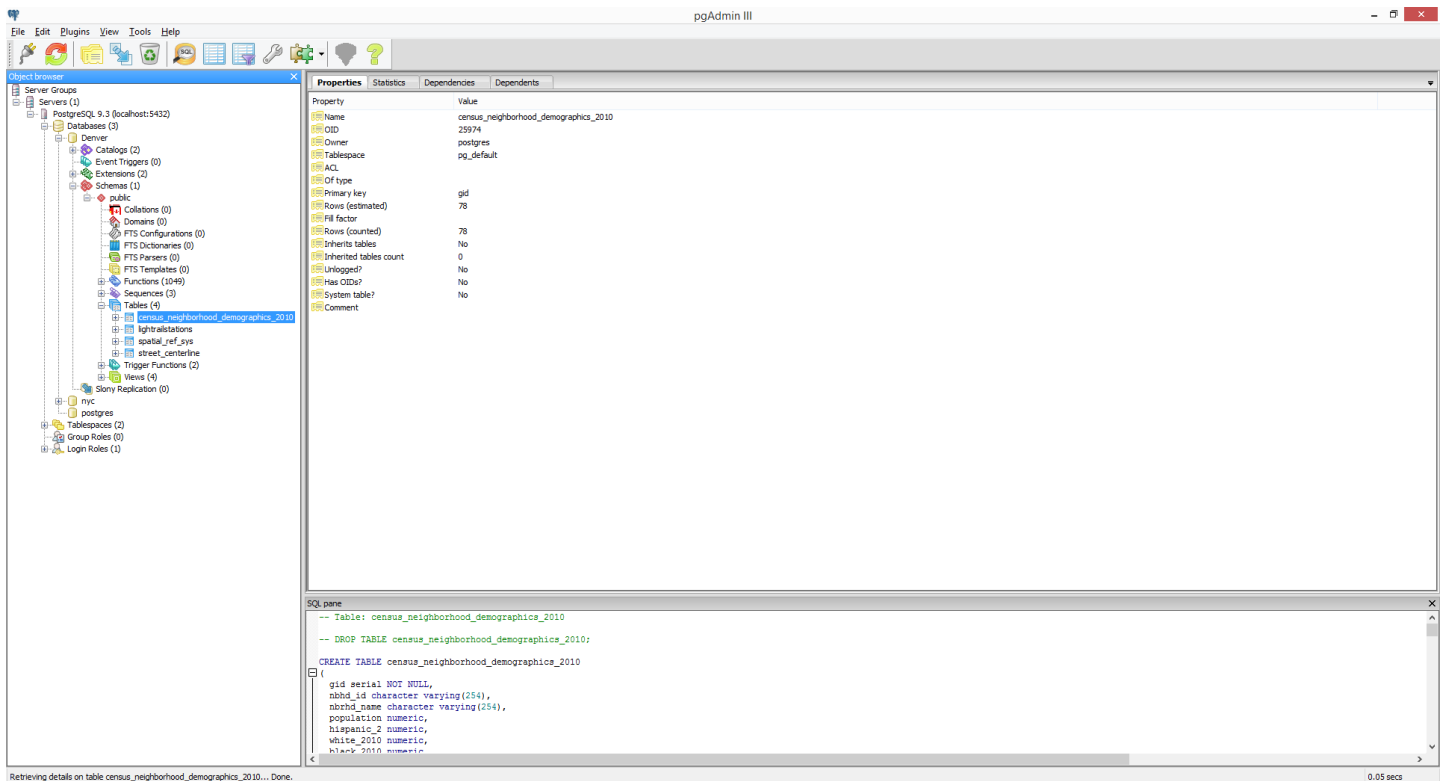
## What is SQL?

- SQL stands for Structured Query Language which is a programming language designed to manage data held inside a relational database system or RDBMS. Through the deployment of SQL we can ask questions to our data and also manipulate it.

On this tutorial you will learn the basic operators and conditions and how to construct a SQL to perform basic tasks.

We will be using the [census\\_neighborhood\\_demographics\\_2010](#) data from [Denver Open Catalog](#).

So, open pgAdmin III and let's get started.



The screenshot shows the pgAdmin III interface. The left pane displays the server tree with the table `census_neighborhood_demographics_2010` selected. The right pane shows the table's properties:

Property	Value
Name	census_neighborhood_demographics_2010
OID	25974
Owner	postgres
Tablespace	pg_default
ACL	
Of type	
Primary key	gid
Rows (estimated)	78
Fill factor	
Rows (counted)	78
Inherits tables	No
Inherited tables count	0
Unlogged?	No
Has OIDs?	No
System table?	No
Comment	

The SQL pane at the bottom shows the following SQL code:

```
-- Table: census_neighborhood_demographics_2010
-- DROP TABLE census_neighborhood_demographics_2010;
CREATE TABLE census_neighborhood_demographics_2010
(
gid serial NOT NULL,
nbhd_id character varying(254),
nbhd_name character varying(254),
population numeric,
hispanic_2 numeric,
white_2010 numeric,
black_2010 numeric
```

Retrieving details on table census\_neighborhood\_demographics\_2010... Done. 0.05 secs

Before we ask any question to our data let's investigate the contents of the table. Let's write a simple SQL what will display all the table.

The `select` query is the most basic element of any SQL sentence, though there are SQL that won't require it. `Select` will return data from a specific table based on a specific condition.

Let's try to retrieve all the data from the neighborhood table. Write the following statement of the query builder and press f5.

```
select * from census_neighborhood_demographics_2010
```

The asterisk `*` informs that we want to retrieve all the data from the source table. This should be your result.

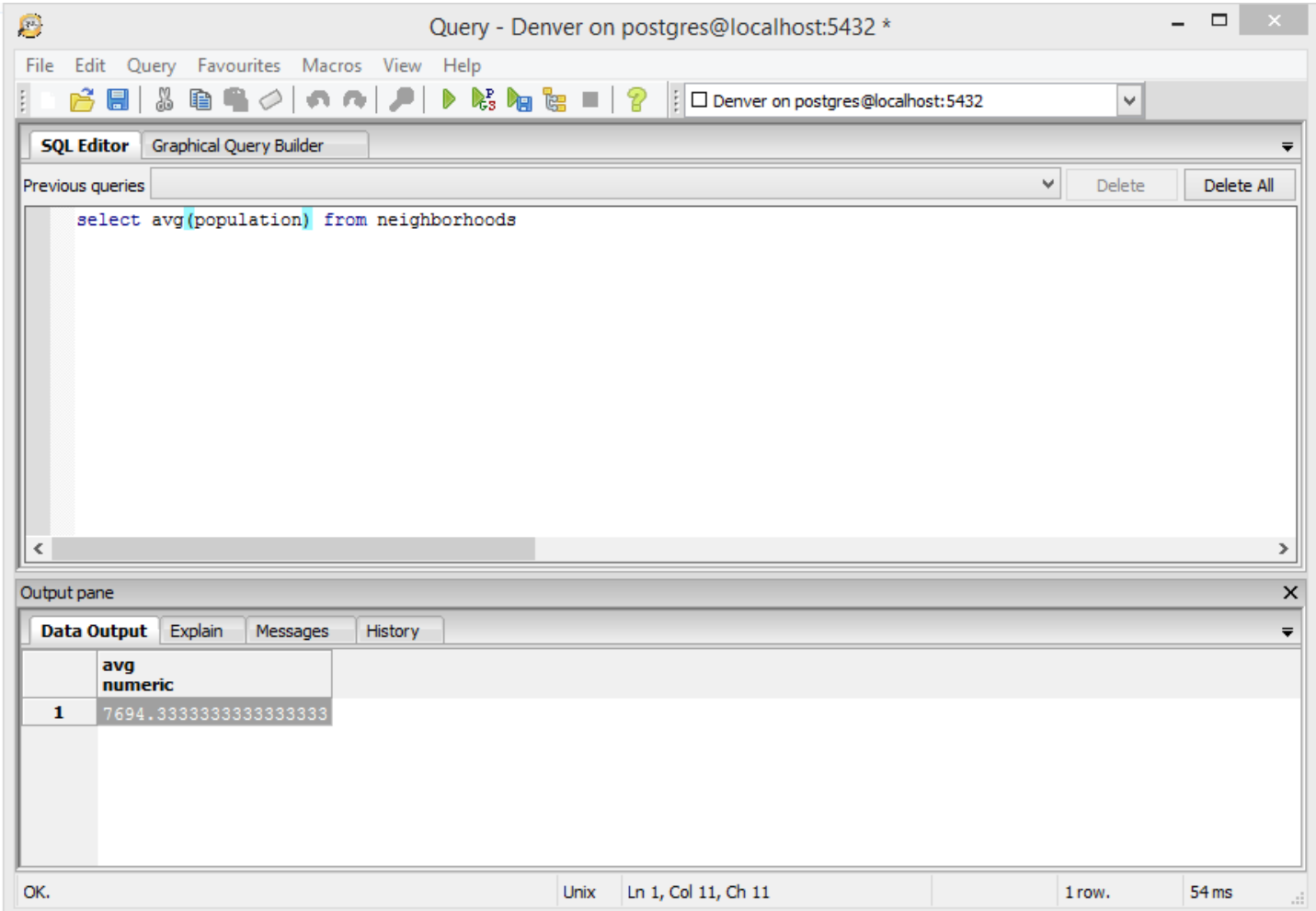
gid	nbhd_id	nbhd_name	population	hispanic_2	white_2010	black_2010	nativeam_2	asian_2010	hawpacs_2	other_2010
1	32	Hampden	17547.00000000	2505.00000000	11750.00000000	1963.00000000	64.0000000000	726.0000000000	20.0000000000	38.0000000000
2	3	Baker	4879.00000000	1664.00000000	2854.00000000	144.0000000000	43.0000000000	52.0000000000	10.0000000000	7.0000000000
3	370	Washington Park	6905.00000000	295.00000000	6356.00000000	28.0000000000	9.0000000000	119.0000000000	3.0000000000	13.0000000000
4	413	Cherry Creek	5589.00000000	332.00000000	4896.00000000	79.0000000000	17.0000000000	168.0000000000	4.0000000000	3.0000000000
5	522	Country Club	3001.00000000	94.00000000	2740.00000000	20.0000000000	8.0000000000	77.0000000000	1.0000000000	5.0000000000
6	617	Clayton	4336.00000000	2126.00000000	699.00000000	1308.00000000	32.0000000000	42.0000000000	4.0000000000	3.0000000000
7	755	Skyland	3106.00000000	573.00000000	882.00000000	1498.00000000	17.0000000000	20.0000000000	1.0000000000	12.0000000000
8	815	City Park West	4844.00000000	524.00000000	2981.00000000	995.00000000	53.0000000000	84.0000000000	1.0000000000	20.0000000000
9	99	Capitol Hill	14708.00000000	1654.00000000	11554.00000000	649.00000000	99.0000000000	333.0000000000	9.0000000000	28.0000000000
10	1047	North Capitol Hill	5823.00000000	650.00000000	4283.00000000	536.00000000	41.0000000000	148.0000000000	3.0000000000	8.0000000000
11	1116	Civic Center	1577.00000000	151.00000000	1230.00000000	81.0000000000	10.0000000000	60.0000000000	0.0000000000	0.0000000000
12	1210	CBD	3648.00000000	355.00000000	2744.00000000	254.00000000	34.0000000000	122.0000000000	4.0000000000	14.0000000000
13	1363	Union Station	4348.00000000	290.00000000	3537.00000000	94.0000000000	16.0000000000	327.0000000000	1.0000000000	2.0000000000
14	1426	Five Points	12712.00000000	2863.00000000	7240.00000000	1936.00000000	103.0000000000	219.0000000000	12.0000000000	20.0000000000
15	1560	Stapleton	13948.00000000	1812.00000000	9718.00000000	1395.00000000	80.0000000000	515.0000000000	7.0000000000	31.0000000000
16	1636	Highland	8429.00000000	3140.00000000	4840.00000000	161.0000000000	54.0000000000	115.0000000000	10.0000000000	7.0000000000
17	1729	Globeville	3687.00000000	2501.00000000	950.00000000	112.0000000000	36.0000000000	27.0000000000	5.0000000000	6.0000000000
18	1866	University Park	7491.00000000	484.00000000	6253.00000000	158.0000000000	12.0000000000	419.0000000000	4.0000000000	12.0000000000
19	1921	Cory - Merrill	3892.00000000	238.00000000	3440.00000000	38.0000000000	6.0000000000	104.0000000000	1.0000000000	9.0000000000
20	2071	Washington Park Wes	6393.00000000	444.00000000	5626.00000000	69.0000000000	28.0000000000	85.0000000000	0.0000000000	11.0000000000
21	2159	Speer	10954.00000000	1222.00000000	8971.00000000	239.0000000000	51.0000000000	242.0000000000	10.0000000000	17.0000000000
22	2220	Congress Park	10235.00000000	828.00000000	8429.00000000	452.0000000000	46.0000000000	214.0000000000	4.0000000000	25.0000000000
23	2318	Cole	4651.00000000	2847.00000000	911.00000000	752.0000000000	20.0000000000	21.0000000000	4.0000000000	17.0000000000
24	2462	Sunnyside	9726.00000000	5768.00000000	3321.00000000	286.0000000000	104.0000000000	94.0000000000	3.0000000000	29.0000000000
25	2554	Ruby Hill	9820.00000000	6797.00000000	2052.00000000	235.0000000000	88.0000000000	567.0000000000	4.0000000000	22.0000000000
26	2664	University	9375.00000000	627.00000000	7796.00000000	162.0000000000	46.0000000000	521.0000000000	4.0000000000	18.0000000000

You may have notice that the original table name is quite long, so let's change it using the `alter table` command, let's rename it to just neighborhoods.

```
alter table census_neighborhood_demographics_2010 rename to neighborhoods
```

Observe that we have a richness of demographic data on this table, let's play with this data. Let's ask what is the average population among the Denver neighborhoods, for this we will use the `avg` query.

```
select avg (population) from neighborhoods
```



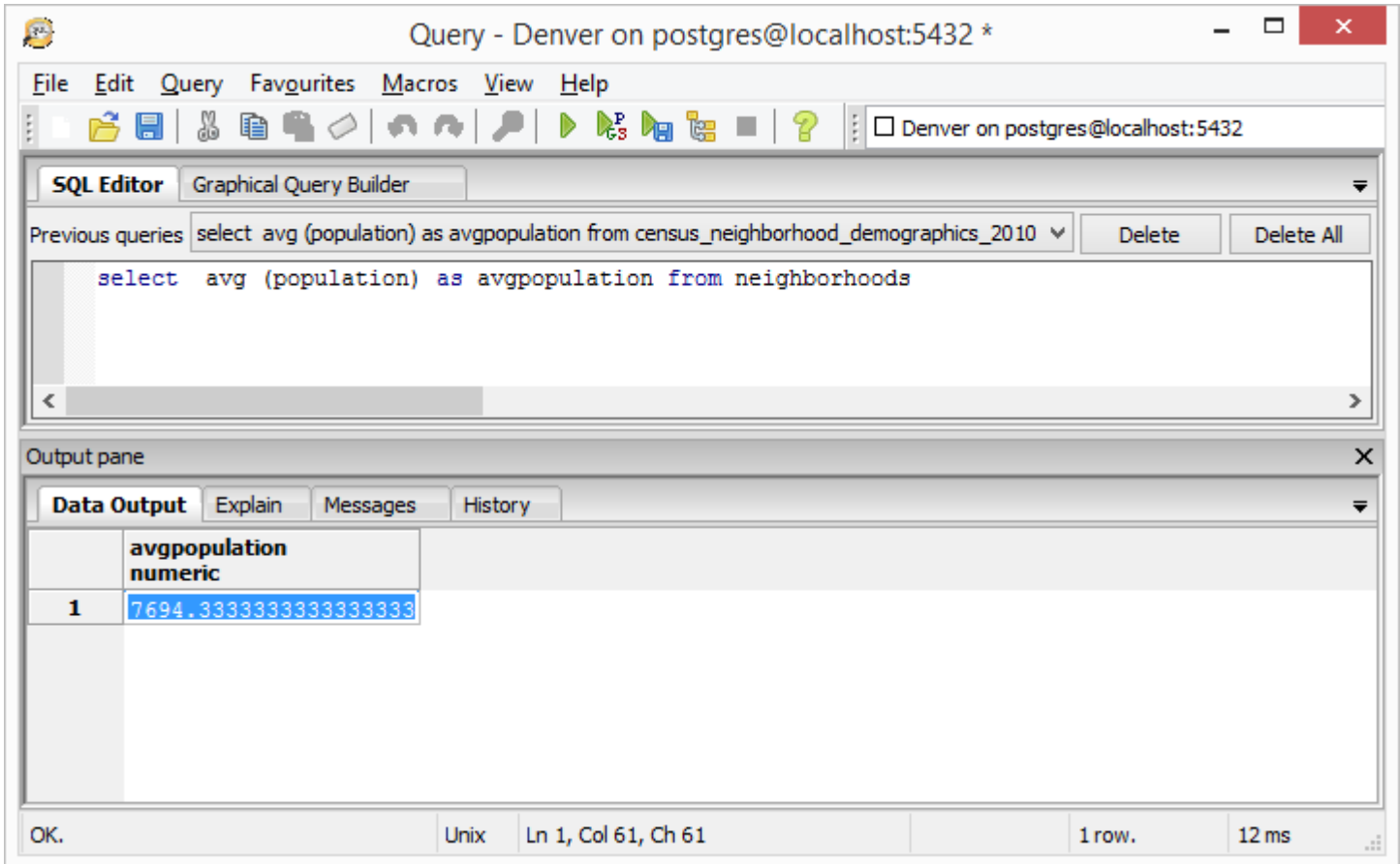
The screenshot shows a PostgreSQL SQL Editor window titled "Query - Denver on postgres@localhost:5432 \*". The window has a menu bar (File, Edit, Query, Favourites, Macros, View, Help) and a toolbar with various icons. The main area is the "SQL Editor" with a "Graphical Query Builder" tab. The query entered is `select avg(population) from neighborhoods`. Below the editor is the "Output pane" with tabs for "Data Output", "Explain", "Messages", and "History". The "Data Output" tab is active, showing a table with one row and one column. The column is labeled "avg numeric" and the row contains the value "7694.3333333333333333". The status bar at the bottom indicates "OK.", "Unix", "Ln 1, Col 11, Ch 11", "1 row.", and "54 ms".

	avg numeric
1	7694.3333333333333333

Notice that the result table display only avg as name, this is because we didn't specify any name for our result column. We can fix this by using the `as` operator, which will provide a name for our result column.

```
Select avg (population) as avgpopulation from neighborhoods
```

Now we should have a table more descriptive.



The screenshot shows a PostgreSQL query editor window titled "Query - Denver on postgres@localhost:5432 \*". The SQL Editor pane contains the query: `select avg (population) as avgpopulation from neighborhoods`. The Output pane shows the result of the query, which is a table with one row and one column. The column is named "avgpopulation" and has a data type of "numeric". The value in the row is "7694.3333333333333333333333333333".

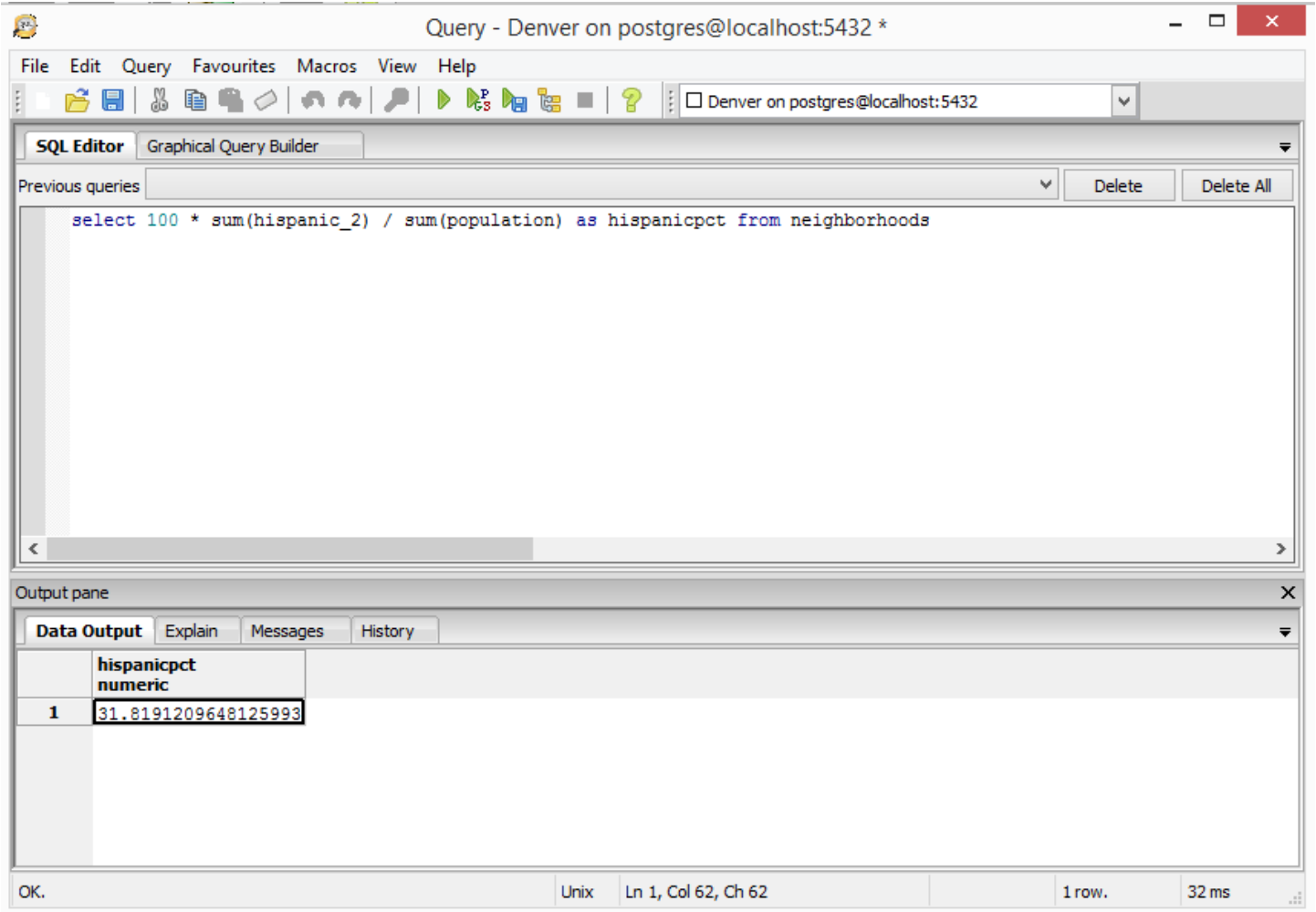
	avgpopulation numeric
1	7694.3333333333333333333333333333

OK. Unix Ln 1, Col 61, Ch 61 1 row. 12 ms

Very often in GIS we will be dealing with relationships inside the same data. In this exercise we will ask what is the percentage of Hispanic population living in Denver.

```
select 100 * sum(hispanic_2) / sum(population) as hispanicpct from neighborhoods
```

The idea is very basic, we multiply the sum of the entire Hispanic population across the city by 100 and divide by the sum of the overall population. The result is:



The screenshot shows a PostgreSQL SQL Editor window titled "Query - Denver on postgres@localhost:5432 \*". The window has a menu bar (File, Edit, Query, Favourites, Macros, View, Help) and a toolbar. The SQL Editor pane contains the following query:

```
select 100 * sum(hispanic_2) / sum(population) as hispanicpct from neighborhoods
```

The Output pane is active, showing the results of the query in a table format:

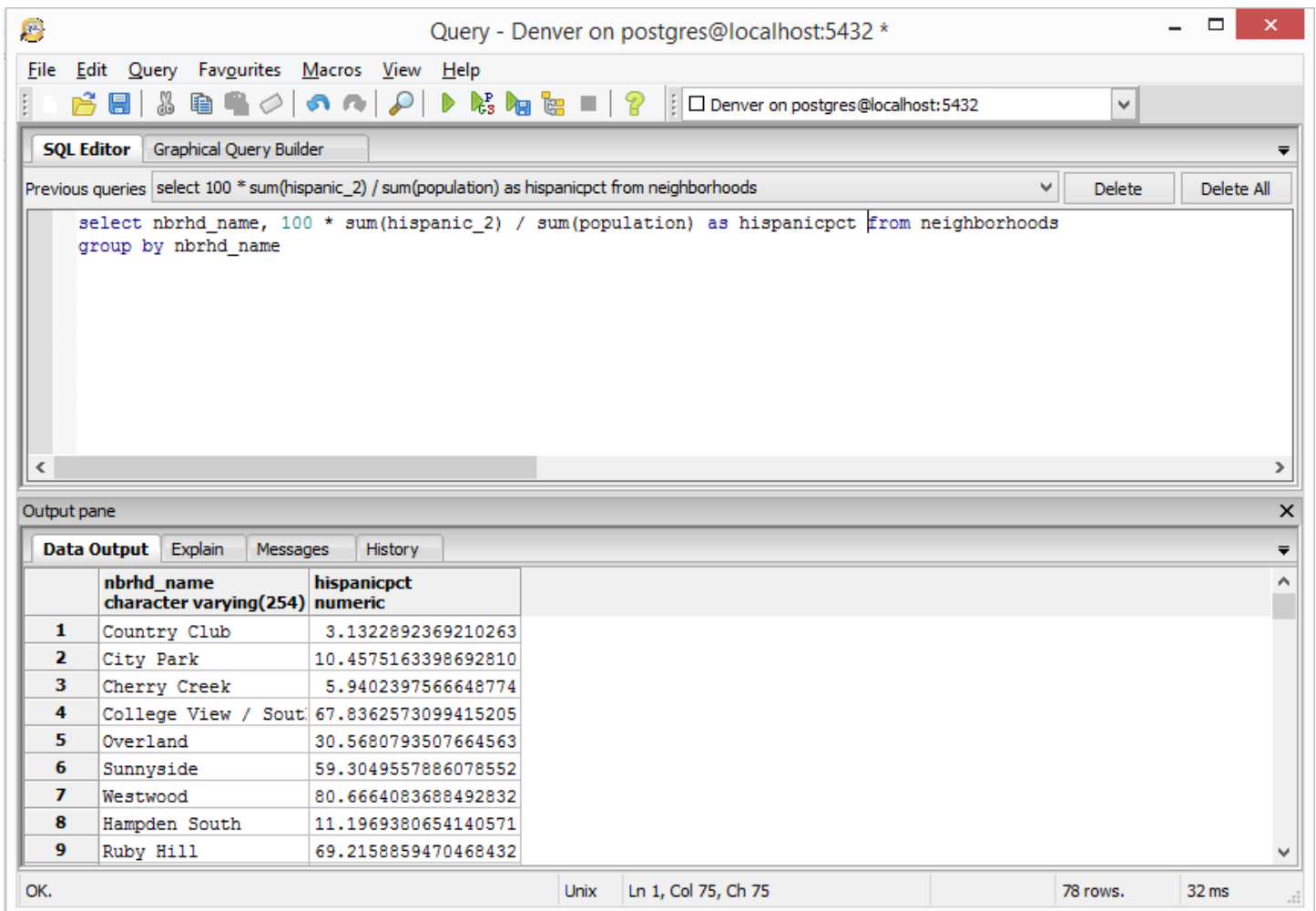
	hispanicpct numeric
1	31.8191209648125993

At the bottom of the window, the status bar displays "OK.", "Unix", "Ln 1, Col 62, Ch 62", "1 row.", and "32 ms".

Let's break the previous result into more detailed information, perhaps by neighborhood unit. For this we need to group the result by neighborhood.

```
select nbrhd_name, 100 * sum(hispanic_2) / sum(population) as hispanicpct from neighborhoods
group by nbrhd_name
```

Notice that in this case we asked the query to select and return two columns, if we don't ask for the name column the result would only the numbers. The result is a concise and read to read table that shows percentage for each neighborhood.



The screenshot shows a PostgreSQL query editor window titled "Query - Denver on postgres@localhost:5432 \*". The SQL Editor tab is active, displaying the following query:

```
select nbrhd_name, 100 * sum(hispanic_2) / sum(population) as hispanicpct from neighborhoods
group by nbrhd_name
```

The Output pane shows the results of the query in a table format:

	nbrhd_name character varying(254)	hispanicpct numeric
1	Country Club	3.1322892369210263
2	City Park	10.4575163398692810
3	Cherry Creek	5.9402397566648774
4	College View / Sout	67.8362573099415205
5	Overland	30.5680793507664563
6	Sunnyside	59.3049557886078552
7	Westwood	80.6664083688492832
8	Hampden South	11.1969380654140571
9	Ruby Hill	69.2158859470468432

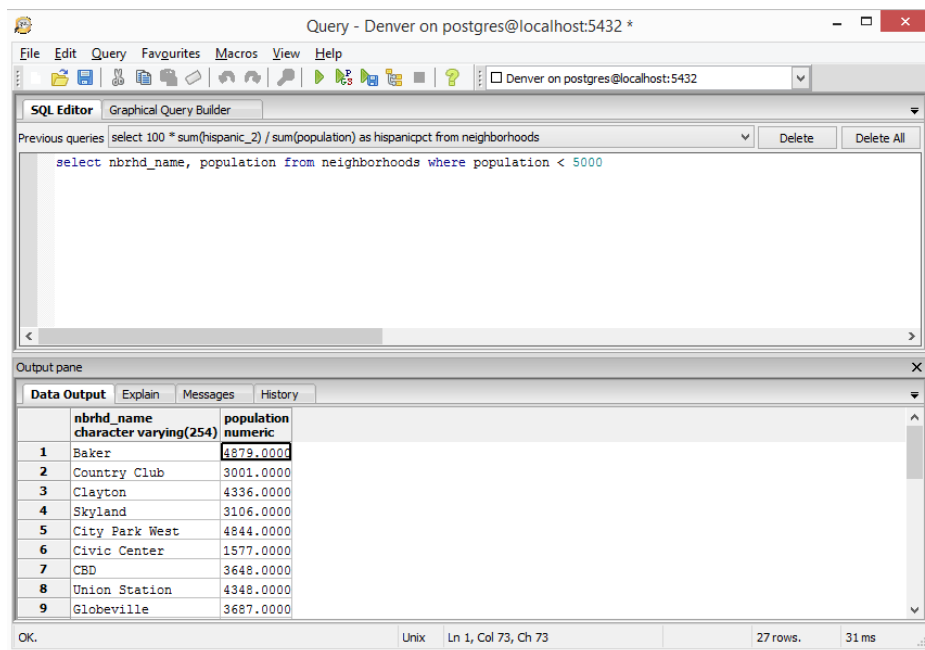
The status bar at the bottom indicates "OK.", "Unix", "Ln 1, Col 75, Ch 75", "78 rows.", and "32 ms".

What if we want to know what are the neighborhoods in Denver that have a population less than 5000? Let's write a query!

```
select nbrhd_name, population from neighborhoods where population < 5000
```

Now we introduce the `where` query. This query is probably one of the most powerful tools in SQL, it allows us to put restriction in our query and make more complex questions. The less signal `<` has the same meaning as in algebra. Other Signs are:

=	Equal to
<>	Not Equal to
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
between	Range between two value
like	Match a character pattern
as	Change the name a field name





Let's recapitulate what we used on this tutorial.

Select	Will return data based on the query
*	Return all
Alter table	Changes some aspect of the table, in our example the name
Rename to	Rename the table's name, to be used with the Alter table statement
Avg	Calculate and return the average of a given column
Group By	Group the result-set by one or more columns
Where	Compare the predicate and return only the rows that are

=	Equal to
<>	Not Equal to
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
between	Range between two value
like	Match a character pattern
as	Change the name a field name