Exercise 10: Analyzing Attribute Data in PGAdmin and QGIS Part 1

Table of Contents

1. Intro-fossgis-umass

1.1. Author Attribution

Major contributors to this curriculum include (alphabetical):

Maria Fernandez

Michael Hamel

Quentin Lewis

James Peters

Charlie Schweik

Alexander Stepanov

1.2. Module Licensing Information

1.3. Reviewed by

Michael Hamel

2. Analyzing attribute data in PGAdmin and QGIS (Part I: SELECT statements)

2.1. Prerequisites

This module implies that you:

1. Have an Internet connection and
2. Have a client program pgAdmin3 [http://pgadmin3.org/] installed or you have PostGIS running on your local machine/network

if not please refer to the module(s) Installing_client_software_to_work_with_SDE [http://linuxlab.sbs.umass.edu/intro-fossgis-umass/index.php?title=Installing_client_software_to_work_with_SDE]

```
Note for the instructors: text file with SQL statements/demo can be found here
```

2.2. Introduction

As you remember, GIS data consists of two major (almost non-separatable) components describing objects in space: spatial and attriute component. General datasets, available to you (your organization), can be massive. For purpose of analysis you will need a subset of data to work with, to run your analytical tools over, etc. In this section, we will give you an overview of the tools and methods used for selecting and summarizing such manageble subsets of "general data". You can do analysis of gis data in two major ways:

1. working with spatial elements (polygons, lines, points, regions, etc) or
2. working with the attribute data.

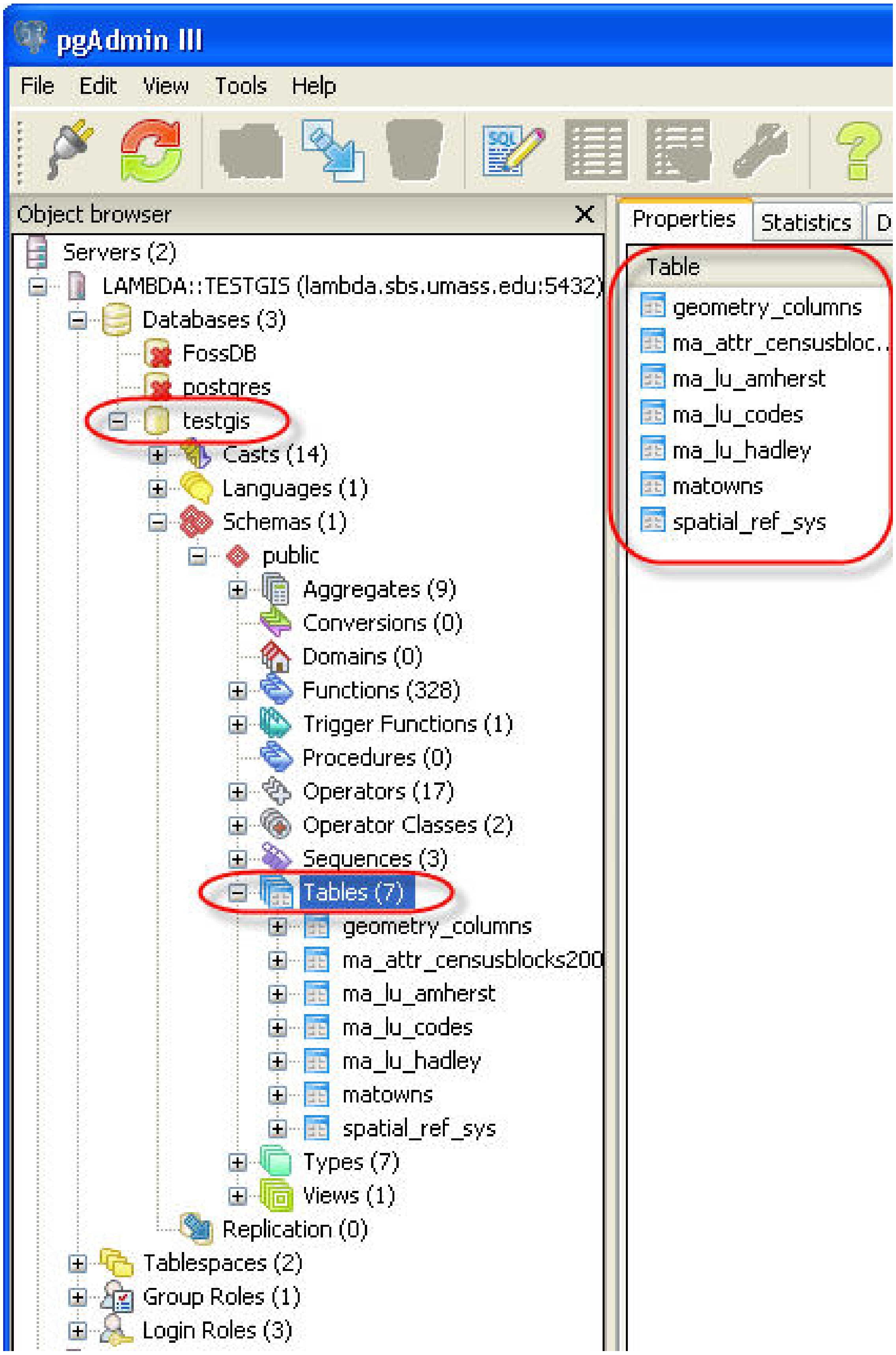
For example, doing some planning analysis, you can try to find all houses located closer than 1 mile to a railroad (spatial query) or to find all houses which were built in a specific year (attribute data query). So we will review the query components used to look at attribute data. To perform this task, we will work with attribute data using SQL language.

2.3. SQL

SQL, which stands for Structured Query Language, was developed in 1970s by IBM and was recognized as a standard language to work with relational databases. There are two standards of the SQL language: SQL92 and SQL99. One of main ideas in relational database theory, is to store information about objects in relations (tables), which comprise of columns (attributes) and rows (records). In addition, the database structure should be designed in such way that tables don't keep repetitive/excessive information and so that information from several tables can be merged. For example, one table can consist of landuse codes and assessor codes for a parcel map. Another table can consist of landuse codes and their descriptions. Such separation avoids keeping landuse code descriptions in the first table. (We will illustrate this example during this lab session). We can define two subsets of the SQL language; they are Data Definition Language (DDL) and Data Manipulation Language (DML). The first allows you to create tables, set user permissions, etc. The DML is to query data. The most common DML SQL statements are **SELECT**, **INSERT**, **UPDATE** and **DELETE**. We will review these statement in detail.

SQL: Connecting to the database

As you remember from the previous modules, we set a spatial database "TestGIS" on the FossGIS server. This spatial database, which runs with PostgreSQL and PostGIS, consists of several spatial layers: landuse layers for the Towns of Amherst and Hadley and the political boundary layer for Massachusetts. You can connect with QGIS to the PostGIS server to look at the layers. Current abilities of QGIS to work with attribute data are limited, so we will work directly with the PostGIS database using a client application/front-end pgAdminIII. Please start pgAdmin with set connection (if you didn't configure pgAdmin to connect to the FossGIS server please refer to module ....).



Servers (2)
LAMBDA::TESTGIS (lambda.sbs.umass.edu:5432)
Databases (3)
FossDB
postgres
testgis
Casts (14)
Languages (1)
Schemas (1)
public
Aggregates (9)
Conversions (0)
Domains (0)
Functions (328)
Trigger Functions (1)
Procedures (0)
Operators (17)
Operator Classes (2)
Sequences (3)
Tables (7)
geometry_columns
ma_attr_censusblocks200
ma_lu_amherst
ma_lu_codes
ma_lu_hadley
matowns
spatial_ref_sys
Types (7)
Views (1)
Replication (0)
Tablespaces (2)
Group Roles (1)
Login Roles (3)

Table
geometry_columns
ma_attr_censusbloc..
ma_lu_amherst
ma_lu_codes
ma_lu_hadley
matowns
spatial_ref_sys

Figure 1

From a list of servers on the left side of the pgAdmin, select lambda.sbs.umass.edu server. Then click on the "TestGIS" database and expand the list of database objects. Click on **Tables** (you will need to expland the objects tree from TestGIS -> Schemas -> Public) , to see a list of tables. You can click on a table to see the list of columns (attributes). The following tables:

- **ma_lu_amherst**
- **ma_lu_hadley**
- **matowns**

are the attribute tables of the spatial layers we used: landuse layers for the Towns of Amherst and Hadley, and a layer representing town boundaries for the State of Massachusetts. Other tables containing attribute data are **ma_lu_codes** and **ma_attr_censusblocks**. The table **ma_lu_codes** consists of fields for landuse codes and their description. The second table consists of Census 2000 data. We will start with simple SQL queries to pull/a?alyze some data from these tables.

SQL: Staring the Query Tool

pgAdmin has a special tool for performing SQL queries, called " the Query Tool". To start the tool, select "T**ools > Query Tool**" from the menu or click the sql query tool button on the tool panel (Figure 2).

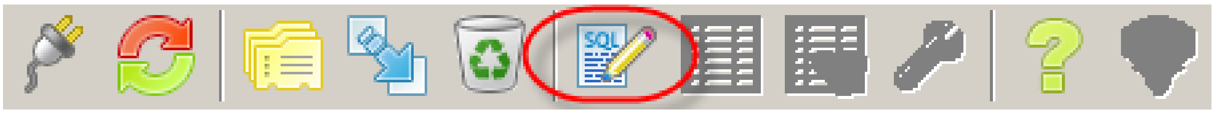


Figure 2

The new window of the Query Tool will appear (Figure 3). The tool has two major working areas: query input area and sql query output area.

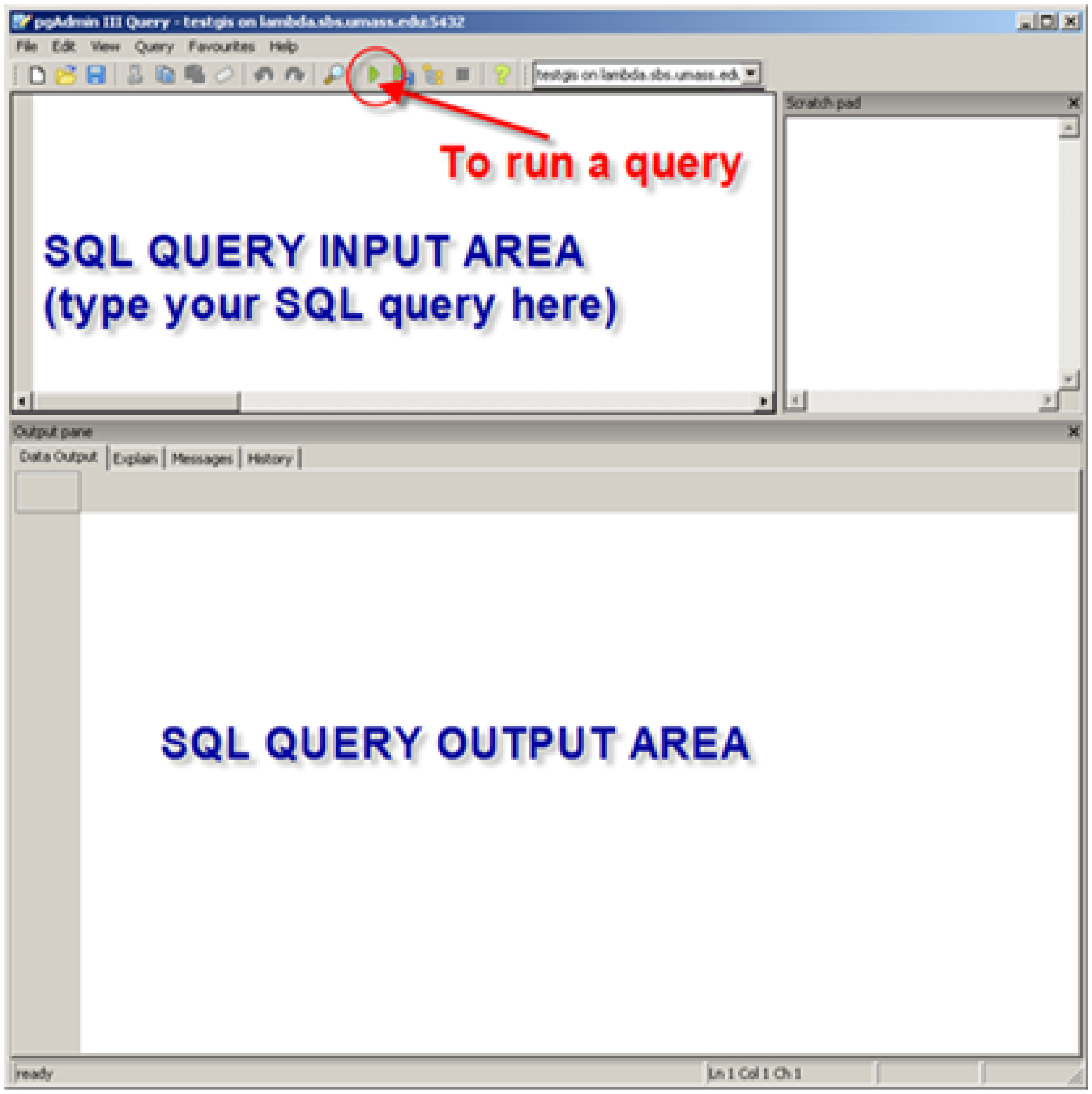

pgAdmin III Query - testgis on lambda.sbs.umass.edu:5432
File Edit View Query Favourites Help
testgis on lambda.sbs.umass.edu
Scratch pad
To run a query
SQL QUERY INPUT AREA
(type your SQL query here)
Output pane
Data Output
Explain
Messages
History
SQL QUERY OUTPUT AREA
ready
Ln 1 Col 1 Ch 1

Figure 3

The query input area is an area where you type your SQL queries. Then the query should be run or executed. To run the query, click on the "Execute Query" button, or select "Query > Execute" from the menu bar (a strike of **F5** on your keyboard will do the same - execute the query). The results of the SQL statement will be shown in the output area. Let's start with the **SELECT** SQL query.

Select Statement

Perhaps, the SELECT [http://www.w3schools.com/sql/sql_select.asp] statement is the most used among all SQL commands. The purpose of the statement is to generate/retrieve a subset of data. For example, you can select subset of one table, choosing columns/fields and records by specific conditions. Also, you can create a subset of data from several tables, merging them on specific criteria/attributes.

The SQL commands have well defined structure. The syntax for SQL command is presented below:

```
SELECT (ALL/DISTINCT)  <column(s)|*>
```

```
FROM  table(s)
WHERE  <search-condition>
GROUP BY  <group-column(s)>
HAVING  <search-condition>
ORDER BY  <sort-column(s)>
```

The basic SELECT SQL command starts with "**SELECT ... FROM**" and then you can elaborate the query adding logical conditions with the **WHERE** clause, as well as group or sort data with "**GROUP BY**" and "**ORDER BY**" clauses. We will start with the simplest form of **SELECT** statement and then will move gradually to more sophisticated statements.

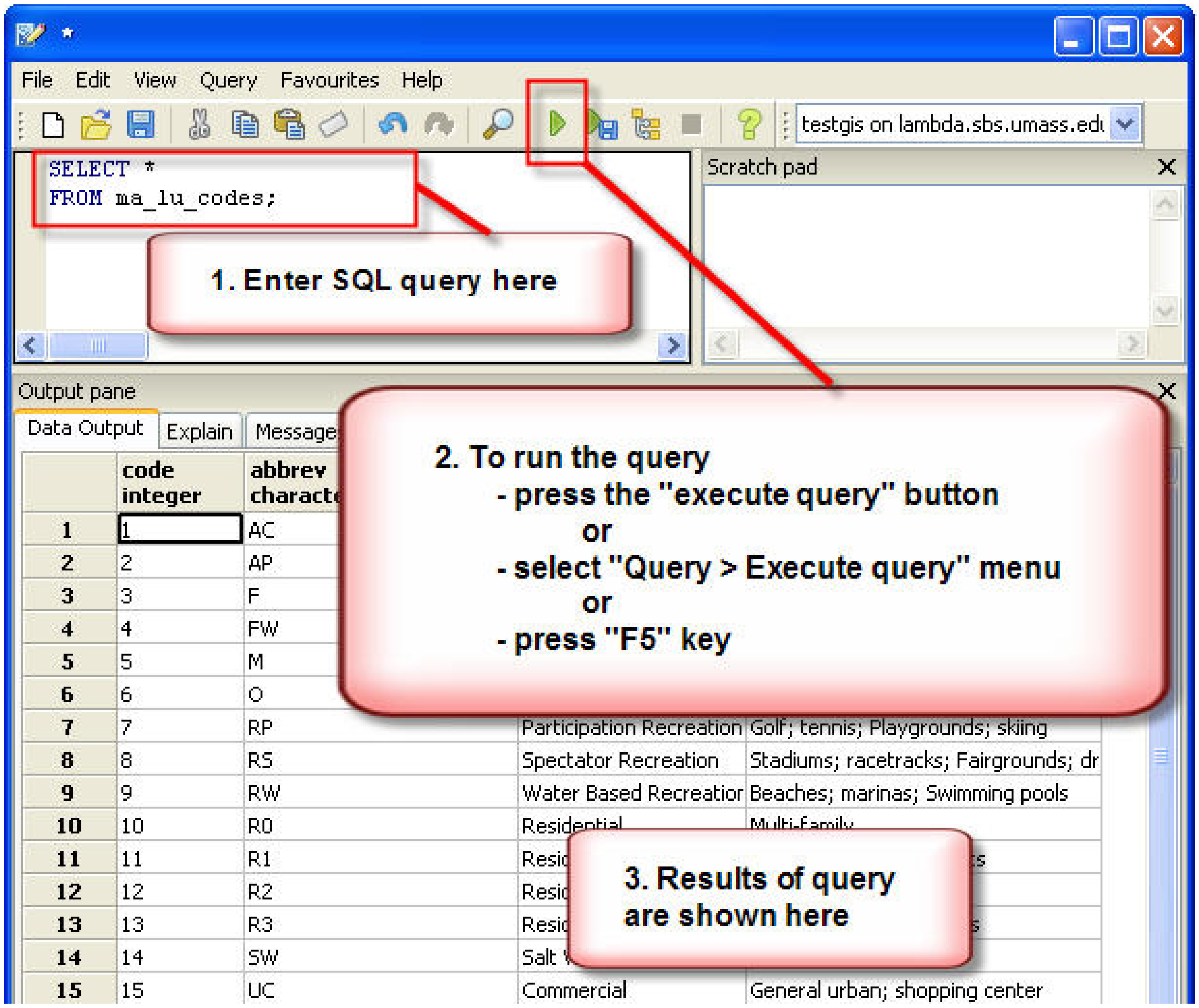
Example 1

To illustrate the basic SQL statement, let's retrieve **all** data from the table **ma_lu_codes**. We will retrieve all columns/attributes and all records from the table. To perform this operation please type the following command in the input area of the query tool:

```

SELECT *
FROM ma_lu_codes;
```

The asterisk (*) in the statement means "all columns". So this statement **selects** "all columns" (and all records) **from** the table **ma_lu_codes**. After entering the command into the input area of the query tool, execute the query (Figure 4).

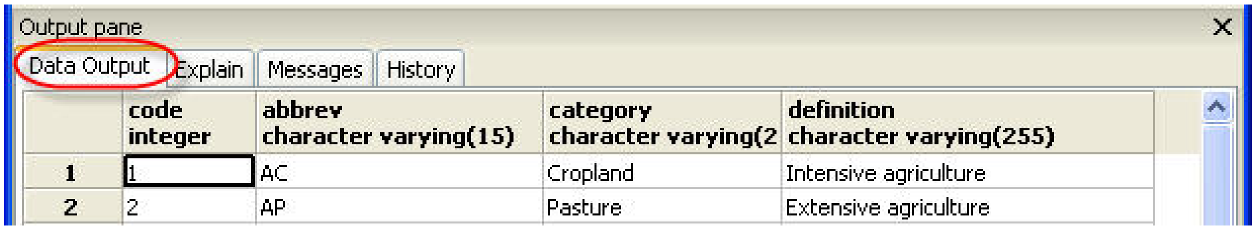

File Edit View Query Favourites Help
testgis on lambda.sbs.umass.edu
Scratch pad
1. Enter SQL query here
Output pane
Data Output
Explain
Messages
2. To run the query
- press the "execute query" button
or
- select "Query > Execute query" menu
or
- press "F5" key
3. Results of query are shown here

```
SELECT *
FROM ma_lu_codes;
```

	code integer	abbrev characte		
1	1	AC		
2	2	AP		
3	3	F		
4	4	FW		
5	5	M		
6	6	O		
7	7	RP	Participation Recreation	Golf; tennis; Playgrounds; skiing
8	8	RS	Spectator Recreation	Stadiums; racetracks; Fairgrounds; dr
9	9	RW	Water Based Recreatior	Beaches; marinas; Swimming pools
10	10	R0	Residential	Multi-family
11	11	R1	Resi	s
12	12	R2	Resi	
13	13	R3	Resi	s
14	14	SW	Salt	
15	15	UC	Commercial	General urban; shopping center

Figure 4

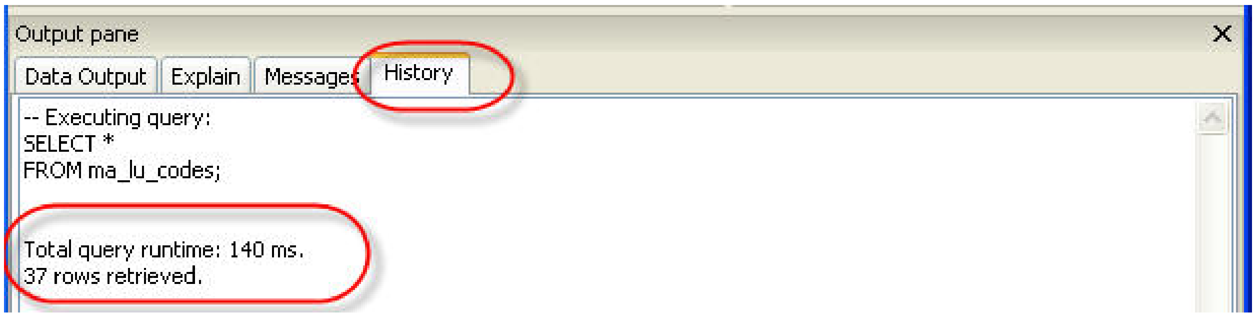
Let's analyze the output of this operation. The results of the SQL operation are presented in the "Output Pane". The output pane contains four tabs: "Date Output", "Explain", "Messages" and "History". The "Date Output" tab shows the results of the query. You may have noted that, in general, results of the SQL query is a relation/table: where a set of columns and records satisfies conditions of the SQL query (Figure 5). Take a moment and check how many columns and records were returned with the query.



	code integer	abbrev character varying(15)	category character varying(2	definition character varying(255)
1	1	AC	Cropland	Intensive agriculture
2	2	AP	Pasture	Extensive agriculture

Figure 5

The tab "History" keeps record of all your queries, which were performed during the "Query Tool" session. It shows, for example, the query, time of execution and the the number of records returned (or satisfying selection criteria). In our case, the statement/command returned *37* records. In other words, the table "ma_lu_codes" consists 37 different code definitions (Figure 6).



```
-- Executing query:
SELECT *
FROM ma_lu_codes;


Total query runtime: 140 ms.
37 rows retrieved.
```

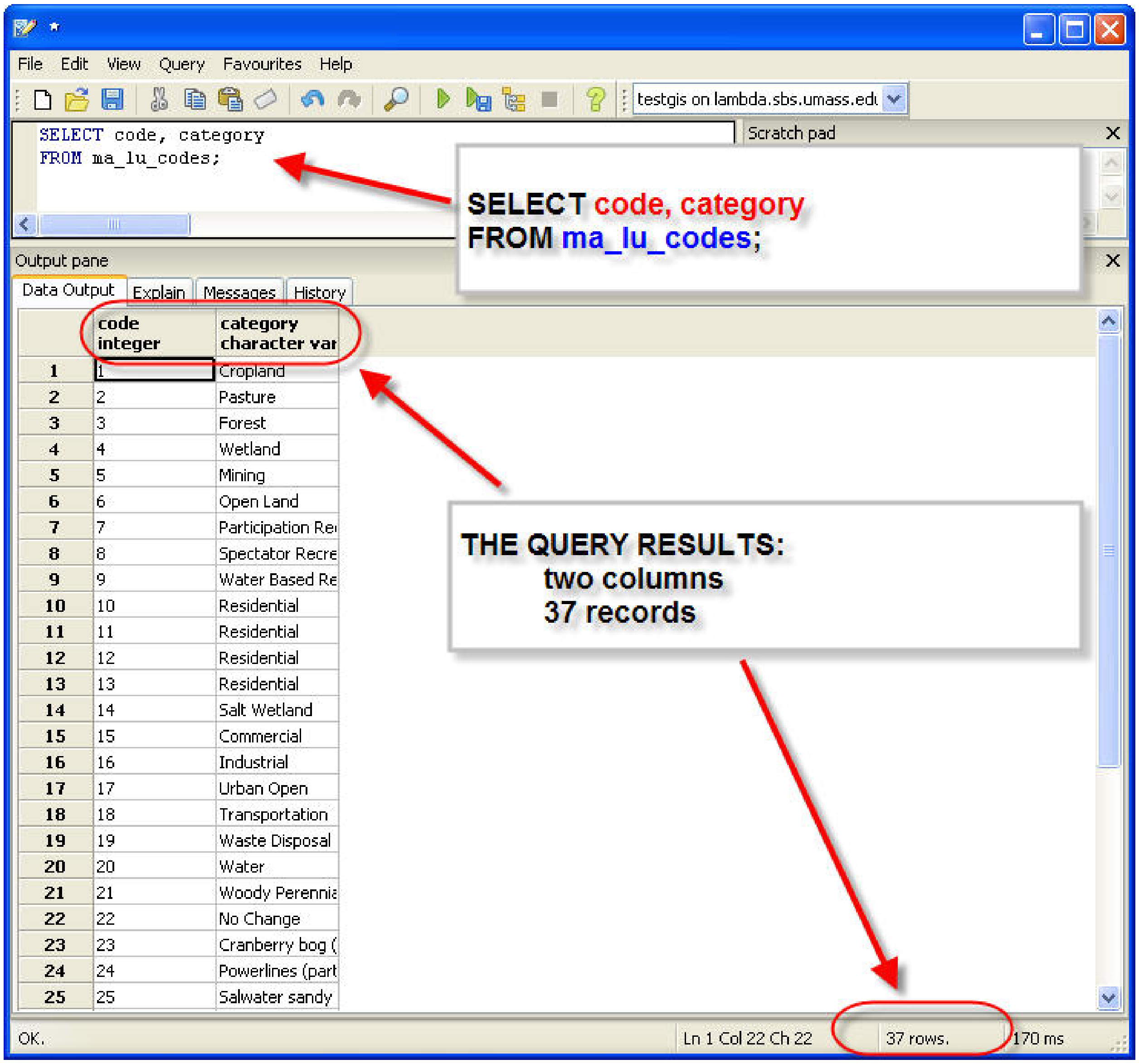
Figure 6

Example 2

In the previous example, we retrieve the whole dataset: all columns and all records. Let's limit our selection to two columns from the table "ma_lu_table". Such selection can be generated with the following SQL query:

```
SELECT code, category
FROM ma_lu_codes
```

We use a comma-separated list of columns after the "SELECT" keyword to specify what columns/fields should be retrieved from the whole dataset. Compare this statement with the statement from the previous example. Enter the statement into the input area of the Query tool and run it.



```
SELECT code, category
FROM ma_lu_codes;
```

	code integer	category character var
1	1	Cropland
2	2	Pasture
3	3	Forest
4	4	Wetland
5	5	Mining
6	6	Open Land
7	7	Participation Re
8	8	Spectator Recre
9	9	Water Based Re
10	10	Residential
11	11	Residential
12	12	Residential
13	13	Residential
14	14	Salt Wetland
15	15	Commercial
16	16	Industrial
17	17	Urban Open
18	18	Transportation
19	19	Waste Disposal
20	20	Water
21	21	Woody Perennia
22	22	No Change
23	23	Cranberry bog (
24	24	Powerlines (part
25	25	Salwater sandy

```
OK.                    Ln 1 Col 22 Ch 22     37 rows.     170 ms
```

Figure 7

Results of this query are presented in Figure 7. As you can see the SQL statement returned **2** columns out of 4 and the same number of records: **37**.

Example 3: Specifying logical conditions with the WHERE clause

The previous statements returned **all** records from the dataset/table(s). In real application, it means that information on **all** parcels from the landuse layer were retrieved or on **all** books from the library catalog, etc. How could we get only specific records which have particular interest for our research/task? Selection based on logical conditions can be performed with the **WHERE** clause. You just add WHERE keyword to a SQL statement specifying logical conditions based on one or several attributes.

```
SELECT *|(list of comma-separated columns)
```

```
FROM table(s)
WHERE column operator value ( AND,  OR additional logical conditions)
```

For example, let's retrieve information on **residential parcels only** for the Town of Amherst. This data are stored in the table **ma_lu_amherst**. In particular field 'lu21_1999' consists of land use codes (assessors codes) for parcels based on 1999 inventory. Residential parcel codes are 10, 11, 12 and 13. How do we know that? It is simple, check the results of SQL queries in the first two examples. Now let's complete our query as follows and run it:

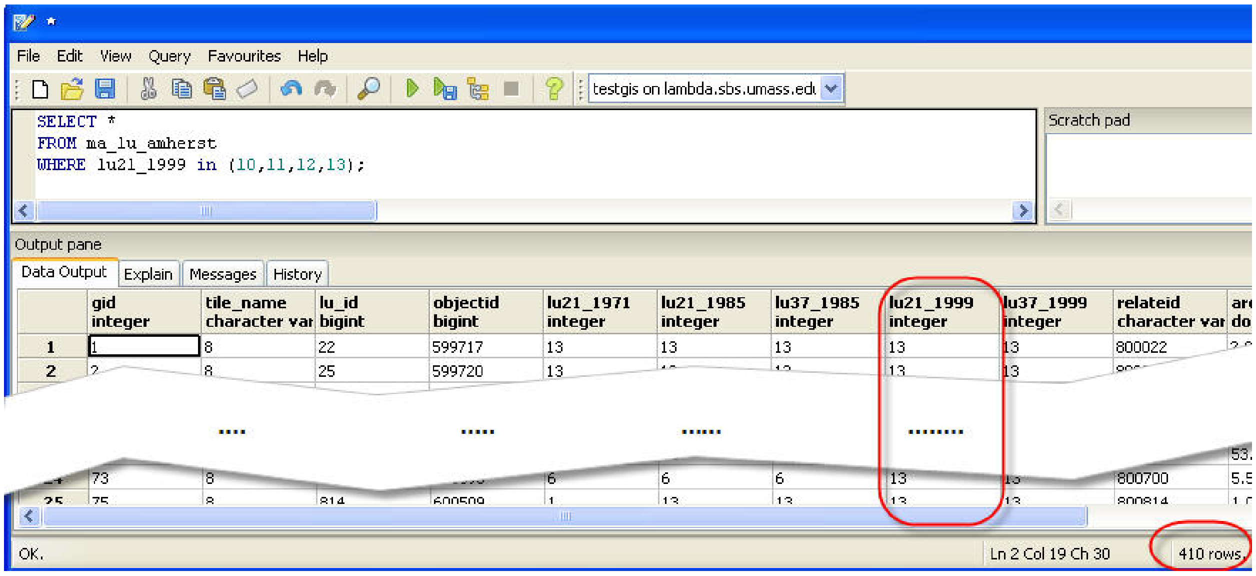
```
SELECT *
FROM ma_lu_amherst
WHERE LU21_1999 >= 10  AND LU21_1999 <=13;
```

The part **SELECT * FROM ...** should look familiar for you by now. We add the **WHERE** clause with the logical condition that selected records values in field **lu21_1999** should be larger or equal to 10 and smaller or equal to 13. This logical condition assures that land use codes are 10,11,12 **or** 13. Try to run this query and analyze the output. How many fields and records are in the resulting dataset? Try to specify different logical conditions using information from different fields and different logical operators (AND, OR, <, =, etc [http://www.w3schools.com/sql/sql_where.asp]). Don't be afraid to experiment with SQL SELECT statement, you cannot brake anything as you read (not write or change) data.

Another approach to specify such query is to use "IN" keyword within "WHERE" clause. "IN" specifies that the value of a specific attribute is IN INCLUSIVE RANGE. Please see an example below:

```
SELECT *
FROM ma_lu_amherst
WHERE lu21_1999  in (10,11,12,13);
```

Please try to execute this SQL statement and check results (Figure 8)



```
SELECT *
FROM ma_lu_amherst
WHERE lu21_1999 in (10,11,12,13);
```

	gid integer	tile_name character var	lu_id bigint	objectid bigint	lu21_1971 integer	lu21_1985 integer	lu37_1985 integer	lu21_1999 integer	lu37_1999 integer	relateid character var
1	1	8	22	599717	13	13	13	13	13	800022
2	2	8	25	599720	13			13	13	
	73	8			6	6	6	13		800700

Figure 8

In particular, please note the set of values for field "lu21_1999" in the resulting field and the number of resulting records in the recordset.

The WHERE cl?use is a very powerful tool, which allows one to put a filter on retrieving data. Later, in the next few modules, we will use the WHERE clause to merge data from several tables into one dataset.

Example 5: Aggregate functions

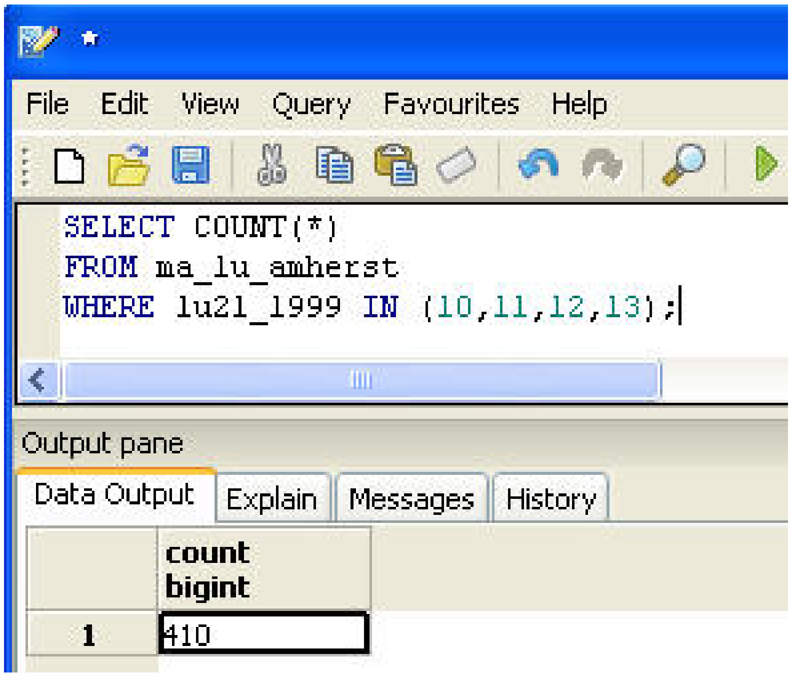
For now, before reviewing GROUP BY and SORT BY statements, we would like introduce (very briefly) aggregate functions. Aggregate functions perform calculations on column(s) and over values of all records. For example the function SUM(column) will find the sum of all values stored in column for a specific dataset. The general format is:

```
SELECT FUNC(column)
FROM table(s)
[WHERE ...]
```

This function is useful to get the cumulative/descriptive statistics on a dataset or specific attribute. Common aggregate functions are SUM(), MIN(), MAX(), COUNT() and COUNT(*). The list of functions and examples of their usage can be found here [http://www.w3schools.com/sql/sql_functions.asp]. For example, the function COUNT(*) returns the number of records in the dataset set. Combined with the WHERE clause aggregate function it is powerful! The following example allows one to find out the number of parcels zoned for residential development in the Town of Amherst.

```
SELECT COUNT(*)
FROM ma_lu_amherst
WHERE lu21_1999 IN (10,11,12,13);
```

Note that the WHERE clause specifies that only residential parcels are selected, the COUNT(*) function computes the number of records. Type in this statement and run it within the Query Builder (Figure 9).



```
SELECT COUNT(*)
FROM ma_lu_amherst
WHERE lu21_1999 IN (10,11,12,13);
```

```
  count
  bigint
1 410
```

Figure 9

As a result of this query we find that: **410** parcels (polygons) of the Amherst landuse layer are zoned for residential development.

Example 6: Selecting unique/distinct records

In many cases, it's necessary to retrieve only unique records without any duplicates. For example, some towns in the spatial layer **ma_towns** (which represents political boundaries of towns) can be 'coded'/presented as several polygons.

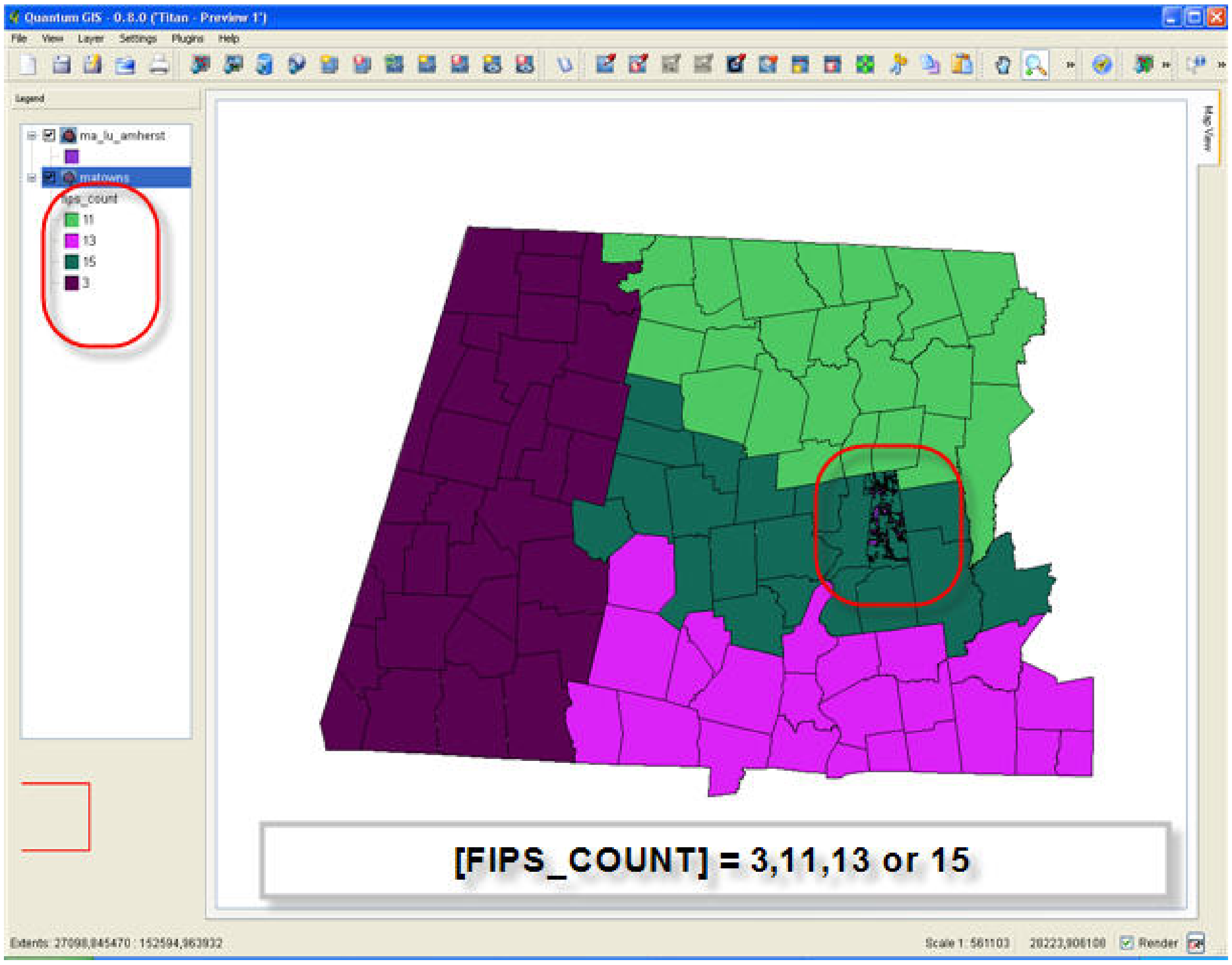

Quantum GIS - 0.8.0 ('Titan - Preview 1')
File View Layer Settings Plugins Help
Legend
ma_lu_amherst
matowns
fips_count
11
13
15
3
Map View
[FIPS_COUNT] = 3,11,13 or 15
Extents: 27098,845470 : 152594,963932
Scale 1: 561103
20223,906100
Render

Figure 10. The Western Massachusetts: town boundaries

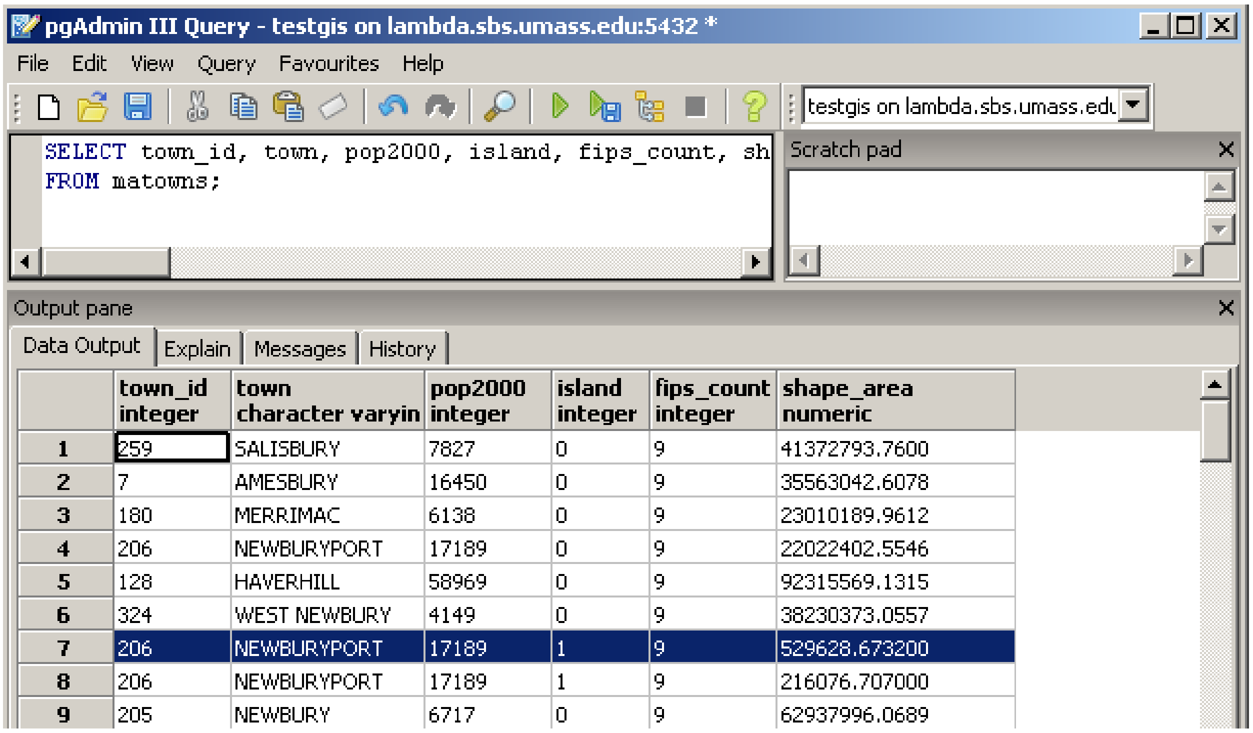
It can happen if a town is divided by a river, or has an island part, etc. How could we retrieve data on polygons/spatial features without duplication? At first let's get all attribute da?a from the spatial layer "ma_towns". Please run the following SQL query in the Query Tool:

```
SELECT *
FROM matowns;
```

The resulting recordset consists of **631** records, although only **351** towns comprise The Commonwealth of Massachusetts. (some towns are presented with several polygons). Take a moment to study columns/attributes and check the metadata for this layer [http://www.mass.gov/mgis/towns.htm]. To get better understanding of the issue, lets select town id (column town_id), town name (town), data on population (pop2000), code of the county (fips_count) and area in sq. meters (shape_area). Please run the following query:

```
SELECT town_id, town, pop2000, island, fips_count, shape_area
FROM matowns;
```

The results of the query are shown in Figure 10. You can see that, for example, there are two records for the town on Newburyport (town_id = 206). The total population of the town is 17,189 persons (each record stores this info = duplication), though area for each polygon representing Newburyport is unique.


pgAdmin III Query - testgis on lambda.sbs.umass.edu:5432 *
File
Edit
View
Query
Favourites
Help
testgis on lambda.sbs.umass.edu
Scratch pad
Output pane
Data Output
Explain
Messages
History

```
SELECT town_id, town, pop2000, island, fips_count, sh
FROM matowns;
```

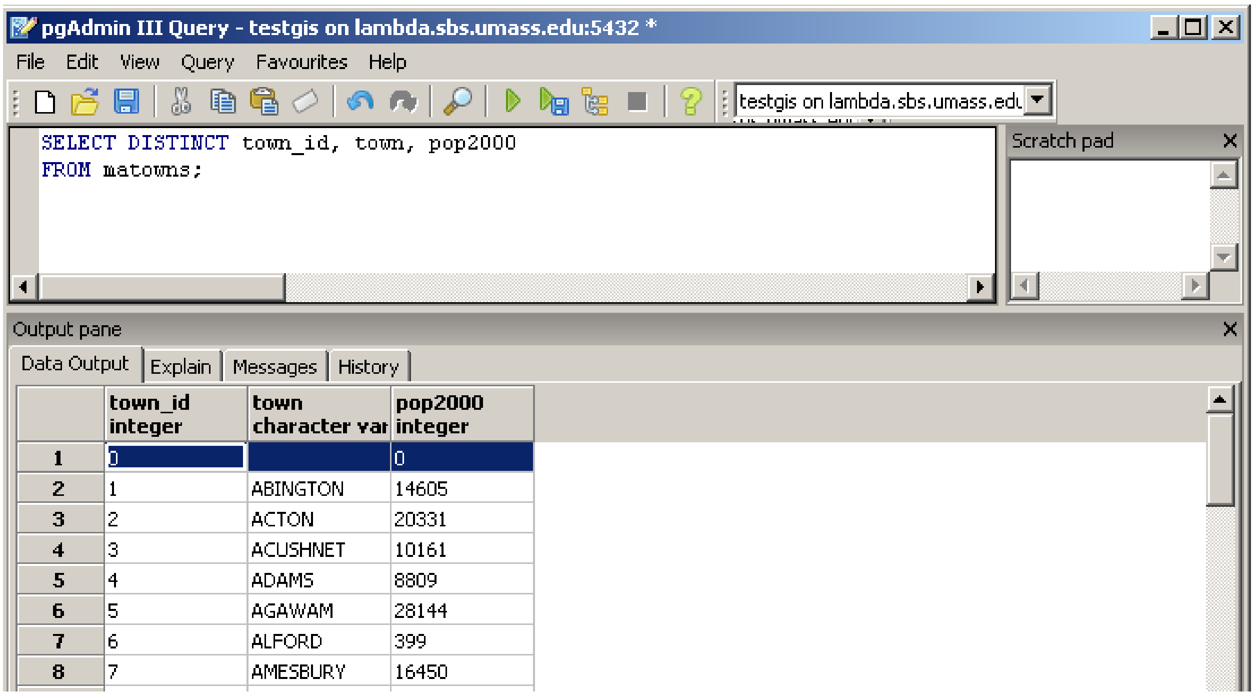
	town_id integer	town character varyin	pop2000 integer	island integer	fips_count integer	shape_area numeric
1	259	SALISBURY	7827	0	9	41372793.7600
2	7	AMESBURY	16450	0	9	35563042.6078
3	180	MERRIMAC	6138	0	9	23010189.9612
4	206	NEWBURYPORT	17189	0	9	22022402.5546
5	128	HAVERHILL	58969	0	9	92315569.1315
6	324	WEST NEWBURY	4149	0	9	38230373.0557
7	206	NEWBURYPORT	17189	1	9	529628.673200
8	206	NEWBURYPORT	17189	1	9	216076.707000
9	205	NEWBURY	6717	0	9	62937996.0689

Figure 11

To select unique records, we use the special keyword **DISTINCT**. An SQL query with the DISTINCT keyword will return *only* unique records. A record is considered unique if one or more fields/attributes differ from other records. Please try to run the following query and analyze the results (Figure 12).

```
SELECT DISTINCT town_id, town, pop2000
FROM matowns;
```

As you can see the total number of towns is **351**. What will happen if you add 'shape_area' field into this query? Could you explain the difference?


pgAdmin III Query - testgis on lambda.sbs.umass.edu:5432 *
File Edit View Query Favourites Help
testgis on lambda.sbs.umass.edu
Scratch pad
Output pane
Data Output Explain Messages History

```
SELECT DISTINCT town_id, town, pop2000
FROM matowns;
```

	town_id integer	town character var	pop2000 integer
1	0		0
2	1	ABINGTON	14605
3	2	ACTON	20331
4	3	ACUSHNET	10161
5	4	ADAMS	8809
6	5	AGAWAM	28144
7	6	ALFORD	399
8	7	AMESBURY	16450

Figure 12

Example 7: Sorting data

To sort data, there is an **ORDER BY** clause. The usage of this clause is clear from the following two examples. Please execute the statement below and compare with Figure 12.

```
SELECT DISTINCT town_id, town, pop2000
FROM matowns
ORDER BY pop2000 ;
```

Now add the DESC keyword to sort in reverse order. Please run the query and compare it with the results of the previous query and the query in Figure 12.

```
SELECT DISTINCT town_id, town, pop2000
FROM matowns
ORDER BY pop2000 DESC;
```

Now we will review usage of the GROUP BY and ORDER BY clauses.

Example 8: Sorting selected data by several attributes

Usually the results of the SQL query can be sorted, so (in general) the ORDER BY clause can be used in conjunction with other keywords, e.g. WHERE clause. Let's try to run the following query. We try to get information (data on town id, town name, population, county and area) for towns in the Western Massachusetts.

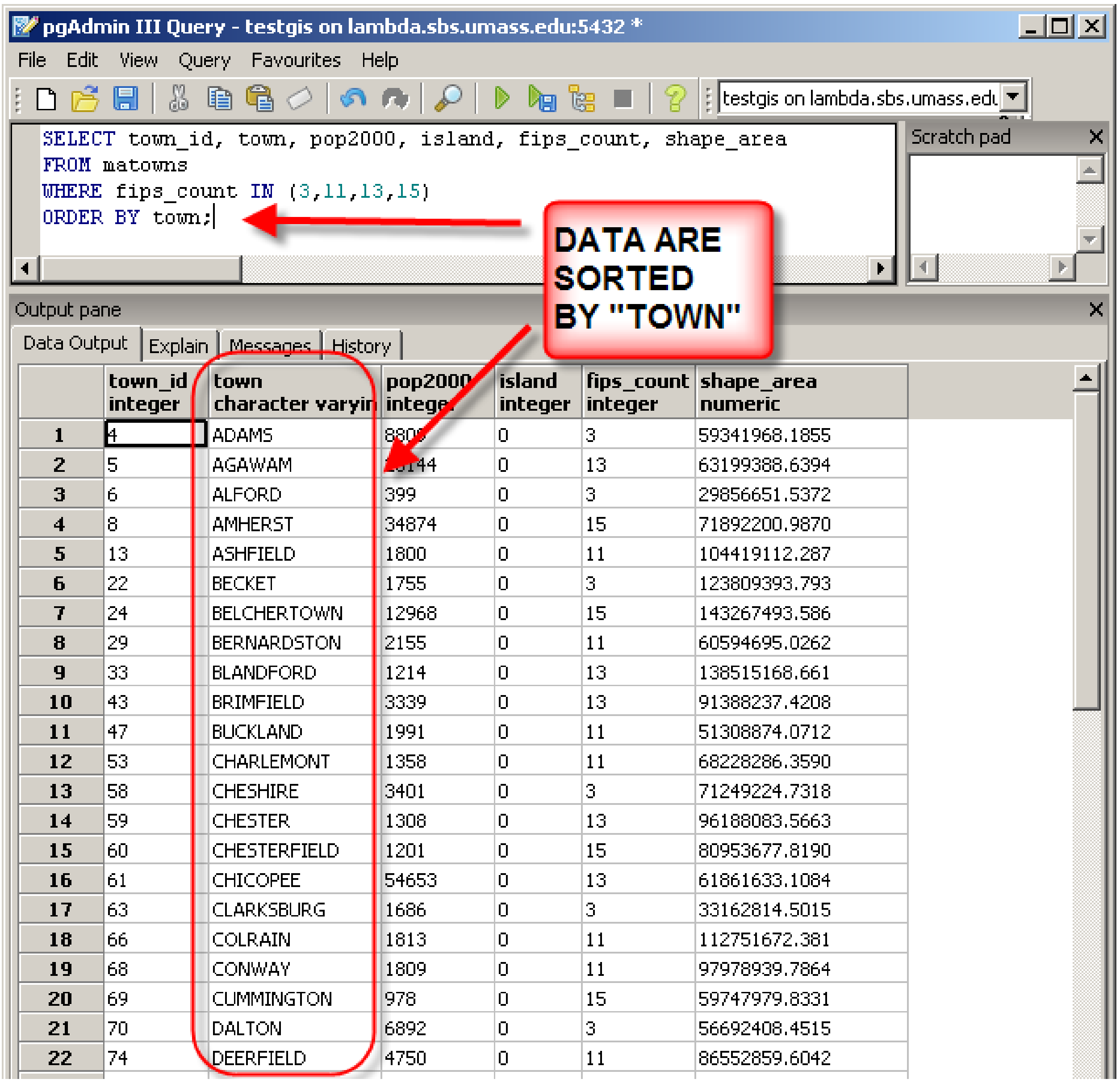
Towns in Western Massachusetts have FIPS county codes of 3,11,13 or 15 (Figure 10)

```
SELECT town_id, town, pop2000, island, fips_count, shape_area
FROM matowns
WHERE fips_count  IN (3,11,13,15);
```

The query returns **101** records. Lets modify our query to sort results by town name in ascending order:

```
SELECT town_id, town, pop2000, island, fips_count, shape_area
FROM matowns
WHERE fips_count  IN (3,11,13,15)
ORDER  BY town;
```

Bingo! Records are sorted now (Figure 13).


pgAdmin III Query - testgis on lambda.sbs.umass.edu:5432 *
File Edit View Query Favourites Help
testgis on lambda.sbs.umass.edu
Scratch pad
DATA ARE SORTED BY "TOWN"
Output pane
Data Output
Explain
Messages
History

```
SELECT town_id, town, pop2000, island, fips_count, shape_area
FROM matowns
WHERE fips_count IN (3,11,13,15)
ORDER BY town;
```

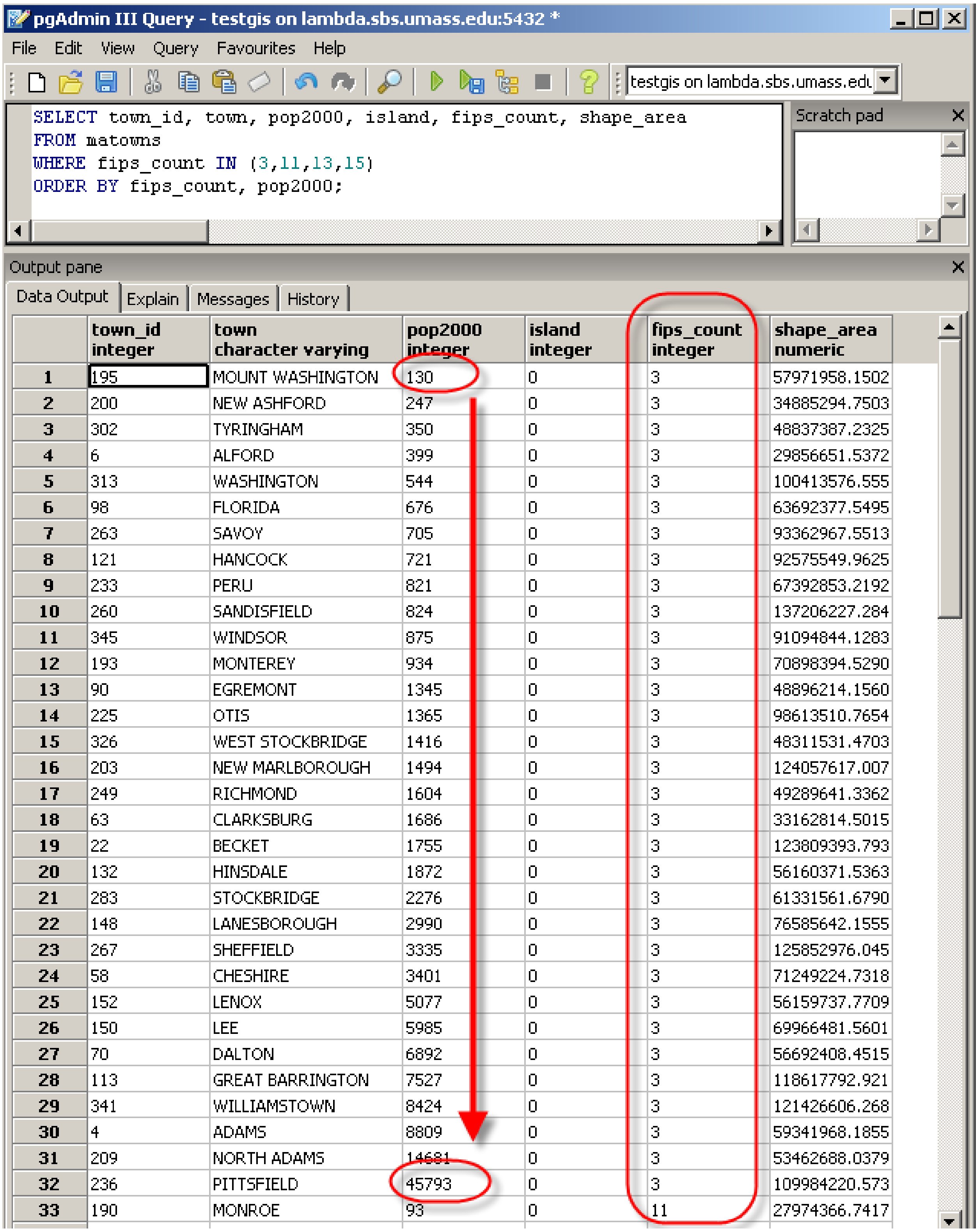
	town_id integer	town character varyin	pop2000 integer	island integer	fips_count integer	shape_area numeric
1	4	ADAMS	8809	0	3	59341968.1855
2	5	AGAWAM	28144	0	13	63199388.6394
3	6	ALFORD	399	0	3	29856651.5372
4	8	AMHERST	34874	0	15	71892200.9870
5	13	ASHFIELD	1800	0	11	104419112.287
6	22	BECKET	1755	0	3	123809393.793
7	24	BELCHERTOWN	12968	0	15	143267493.586
8	29	BERNARDSTON	2155	0	11	60594695.0262
9	33	BLANDFORD	1214	0	13	138515168.661
10	43	BRIMFIELD	3339	0	13	91388237.4208
11	47	BUCKLAND	1991	0	11	51308874.0712
12	53	CHARLEMONT	1358	0	11	68228286.3590
13	58	CHESHIRE	3401	0	3	71249224.7318
14	59	CHESTER	1308	0	13	96188083.5663
15	60	CHESTERFIELD	1201	0	15	80953677.8190
16	61	CHICOPEE	54653	0	13	61861633.1084
17	63	CLARKSBURG	1686	0	3	33162814.5015
18	66	COLRAIN	1813	0	11	112751672.381
19	68	CONWAY	1809	0	11	97978939.7864
20	69	CUMMINGTON	978	0	15	59747979.8331
21	70	DALTON	6892	0	3	56692408.4515
22	74	DEERFIELD	4750	0	11	86552859.6042

Figure 13.

The results of the query can be sorted by the *several* attributes. Please run the following query to sort data by fips county code FIRST, and THEN by population.

```
SELECT town_id, town, pop2000, island, fips_count, shape_area
FROM matowns
WHERE fips_count  IN (3,11,13,15)
ORDER BY fips_count, pop2000;
```

Results of the query are presented in Figure 13. Please note that both fips county code and population size are sorted in the same order.


pgAdmin III Query - testgis on lambda.sbs.umass.edu:5432 *
File Edit View Query Favourites Help
testgis on lambda.sbs.umass.edu
Scratch pad
Output pane
Data Output Explain Messages History

```
SELECT town_id, town, pop2000, island, fips_count, shape_area
FROM matowns
WHERE fips_count IN (3,11,13,15)
ORDER BY fips_count, pop2000;
```

	town_id integer	town character varying	pop2000 integer	island integer	fips_count integer	shape_area numeric
1	195	MOUNT WASHINGTON	130	0	3	57971958.1502
2	200	NEW ASHFORD	247	0	3	34885294.7503
3	302	TYRINGHAM	350	0	3	48837387.2325
4	6	ALFORD	399	0	3	29856651.5372
5	313	WASHINGTON	544	0	3	100413576.555
6	98	FLORIDA	676	0	3	63692377.5495
7	263	SAVOY	705	0	3	93362967.5513
8	121	HANCOCK	721	0	3	92575549.9625
9	233	PERU	821	0	3	67392853.2192
10	260	SANDISFIELD	824	0	3	137206227.284
11	345	WINDSOR	875	0	3	91094844.1283
12	193	MONTEREY	934	0	3	70898394.5290
13	90	EGREMONT	1345	0	3	48896214.1560
14	225	OTIS	1365	0	3	98613510.7654
15	326	WEST STOCKBRIDGE	1416	0	3	48311531.4703
16	203	NEW MARLBOROUGH	1494	0	3	124057617.007
17	249	RICHMOND	1604	0	3	49289641.3362
18	63	CLARKSBURG	1686	0	3	33162814.5015
19	22	BECKET	1755	0	3	123809393.793
20	132	HINSDALE	1872	0	3	56160371.5363
21	283	STOCKBRIDGE	2276	0	3	61331561.6790
22	148	LANESBOROUGH	2990	0	3	76585642.1555
23	267	SHEFFIELD	3335	0	3	125852976.045
24	58	CHESHIRE	3401	0	3	71249224.7318
25	152	LENOX	5077	0	3	56159737.7709
26	150	LEE	5985	0	3	69966481.5601
27	70	DALTON	6892	0	3	56692408.4515
28	113	GREAT BARRINGTON	7527	0	3	118617792.921
29	341	WILLIAMSTOWN	8424	0	3	121426606.268
30	4	ADAMS	8809	0	3	59341968.1855
31	209	NORTH ADAMS	14681	0	3	53462688.0379
32	236	PITTSFIELD	45793	0	3	109984220.573
33	190	MONROE	93	0	11	27974366.7417

Figure 14.

What do you think would be result of the following query:

```
SELECT town_id, town, pop2000, island, fips_count, shape_area
FROM matowns
WHERE fips_count  IN (3,11,13,15)
ORDER BY fips_count DESC, pop2000 ASC;
```

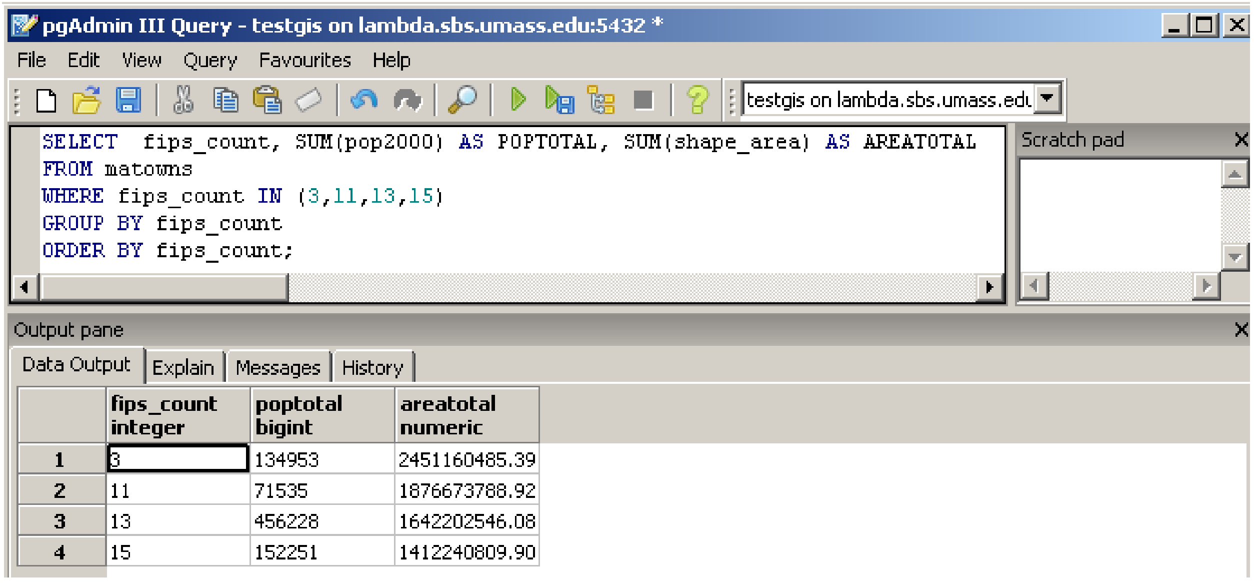
Check your quess by executing the query with Query Tools. The goal here it to have you experiment!

Example 9. Grouping data and computing subtotals

As you could see in the part "SQL Aggregative functions", functions like SUM(), AVG(), COUNT() perform operation on **all** records at once. But how could be perform such operations on a part/group of the data? For example, we need to compute total population and total area for the counties located in the Western Massachusetts. To perform such analysis, we need to introduce and use the **GROUP BY** clause. Please type and run the following query:

```
SELECT  fips_count, SUM(pop2000) AS POPTOTAL, SUM(shape_area) AS AREATOTAL
FROM matowns
WHERE fips_count  IN (3,11,13,15)
GROUP BY fips_count
ORDER BY fips_count;
```

What happens here is that **GROUP BY** groups/combines records with the same **fips_count** codes and then the SU?()function is applied to that group. In other words, we computed 'Subtotals' for each county in the Western MA. Please check and analyze the results of the query below (Figure 14):


pgAdmin III Query - testgis on lambda.sbs.umass.edu:5432 *
File Edit View Query Favourites Help
testgis on lambda.sbs.umass.edu
Scratch pad
Output pane
Data Output Explain Messages History

```
SELECT  fips_count, SUM(pop2000) AS POPTOTAL, SUM(shape_area) AS AREATOTAL
FROM matowns
WHERE fips_count IN (3,11,13,15)
GROUP BY fips_count
ORDER BY fips_count;
```

	fips_count integer	poptotal bigint	areatotal numeric
1	3	134953	2451160485.39
2	11	71535	1876673788.92
3	13	456228	1642202546.08
4	15	152251	1412240809.90

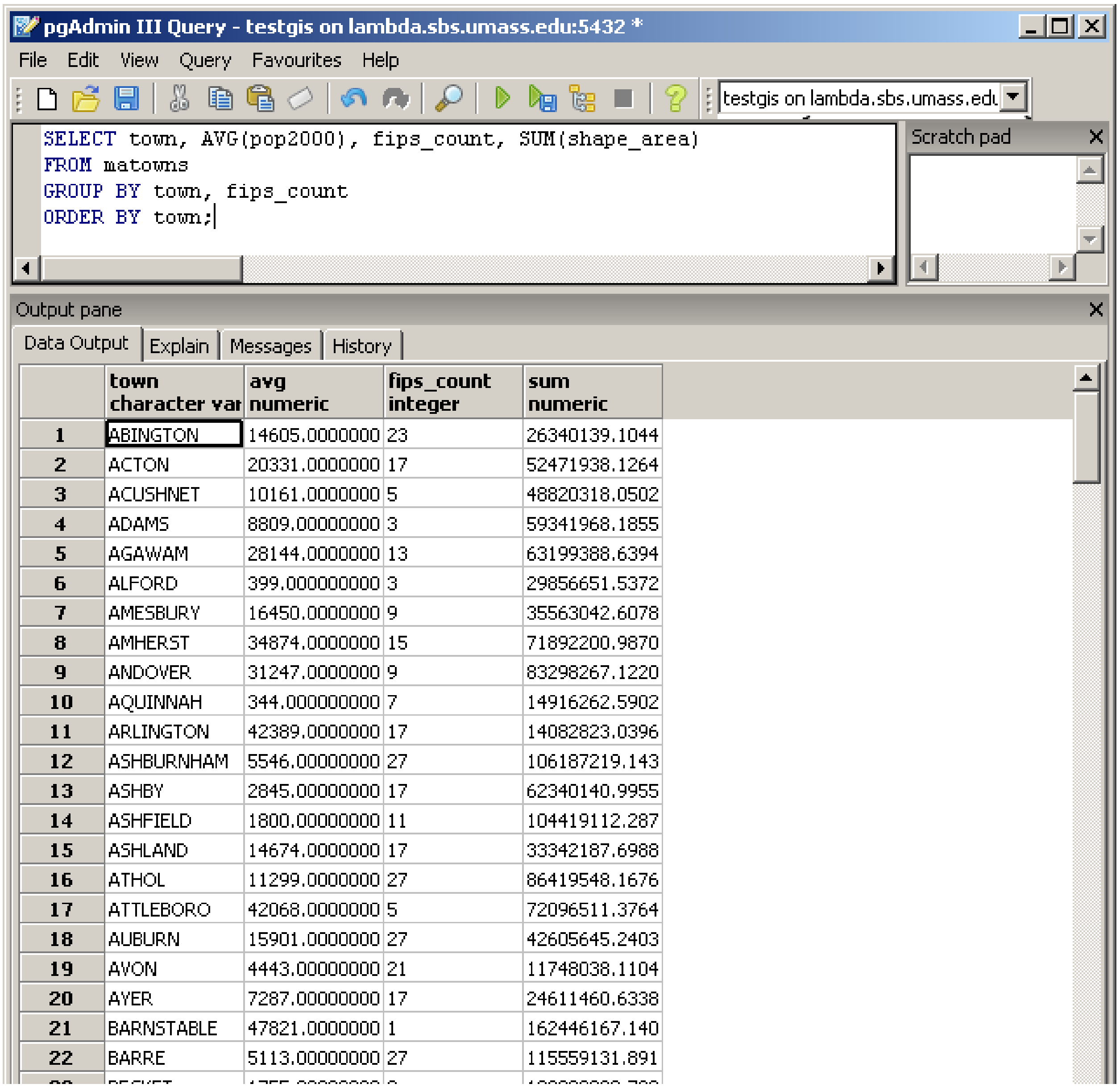
Figure 15.

Another new part in this statement is "AS POPTOTAL". With the **AS** keyword after the column name, we can specify an alias for the resulting column (please see Figure 14, where sum of population is stored in column with alias 'POPTOTAL').

You can argue that there is something strange about the way we calculated the total population. We showed before that each record for the specific town consists of the same population but different area. So usage of SUM(population) would not be correct. The correct way to go about this would be to use the AVG() function which computes average value of the dataset. You can easily see that such a query will return the correct answer. Please try to run the following query:

```
SELECT town, AVG(pop2000), fips_count, SUM(shape_area)
FROM matowns
GROUP BY town, fips_count
ORDER BY town;
```

Results are available in Figure 16.


pgAdmin III Query - testgis on lambda.sbs.umass.edu:5432 *
File Edit View Query Favourites Help
testgis on lambda.sbs.umass.edu
Scratch pad
Output pane
Data Output
Explain
Messages
History

```
SELECT town, AVG(pop2000), fips_count, SUM(shape_area)
FROM matowns
GROUP BY town, fips_count
ORDER BY town;
```

	town character var	avg numeric	fips_count integer	sum numeric
1	ABINGTON	14605.0000000	23	26340139.1044
2	ACTON	20331.0000000	17	52471938.1264
3	ACUSHNET	10161.0000000	5	48820318.0502
4	ADAMS	8809.00000000	3	59341968.1855
5	AGAWAM	28144.0000000	13	63199388.6394
6	ALFORD	399.000000000	3	29856651.5372
7	AMESBURY	16450.0000000	9	35563042.6078
8	AMHERST	34874.0000000	15	71892200.9870
9	ANDOVER	31247.0000000	9	83298267.1220
10	AQUINNAH	344.000000000	7	14916262.5902
11	ARLINGTON	42389.0000000	17	14082823.0396
12	ASHBURNHAM	5546.00000000	27	106187219.143
13	ASHBY	2845.00000000	17	62340140.9955
14	ASHFIELD	1800.00000000	11	104419112.287
15	ASHLAND	14674.0000000	17	33342187.6988
16	ATHOL	11299.0000000	27	86419548.1676
17	ATTLEBORO	42068.0000000	5	72096511.3764
18	AUBURN	15901.0000000	27	42605645.2403
19	AVON	4443.00000000	21	11748038.1104
20	AYER	7287.00000000	17	24611460.6338
21	BARNSTABLE	47821.0000000	1	162446167.140
22	BARRE	5113.00000000	27	115559131.891

Figure 16.

Conclusion

In this module we introduced SELECT SQL statements. This is the fundamental statement, and is very powerful. In the next secions we will use this statement for:

- merging data from several tables
- joining spatial and attribute data together.

Other important SQL statements are INSERT, UPDATE and DELETE. We will use the INSERT statement when we join spatial and non-spatial data.

Additional materials

1. An article on SQL on Wikipedia http://en.wikipedia.org/wiki/Sql
2. http://www.w3schools.com/sql/default.asp
3. SQL Tutorial at http://www.sql-tutorial.com/