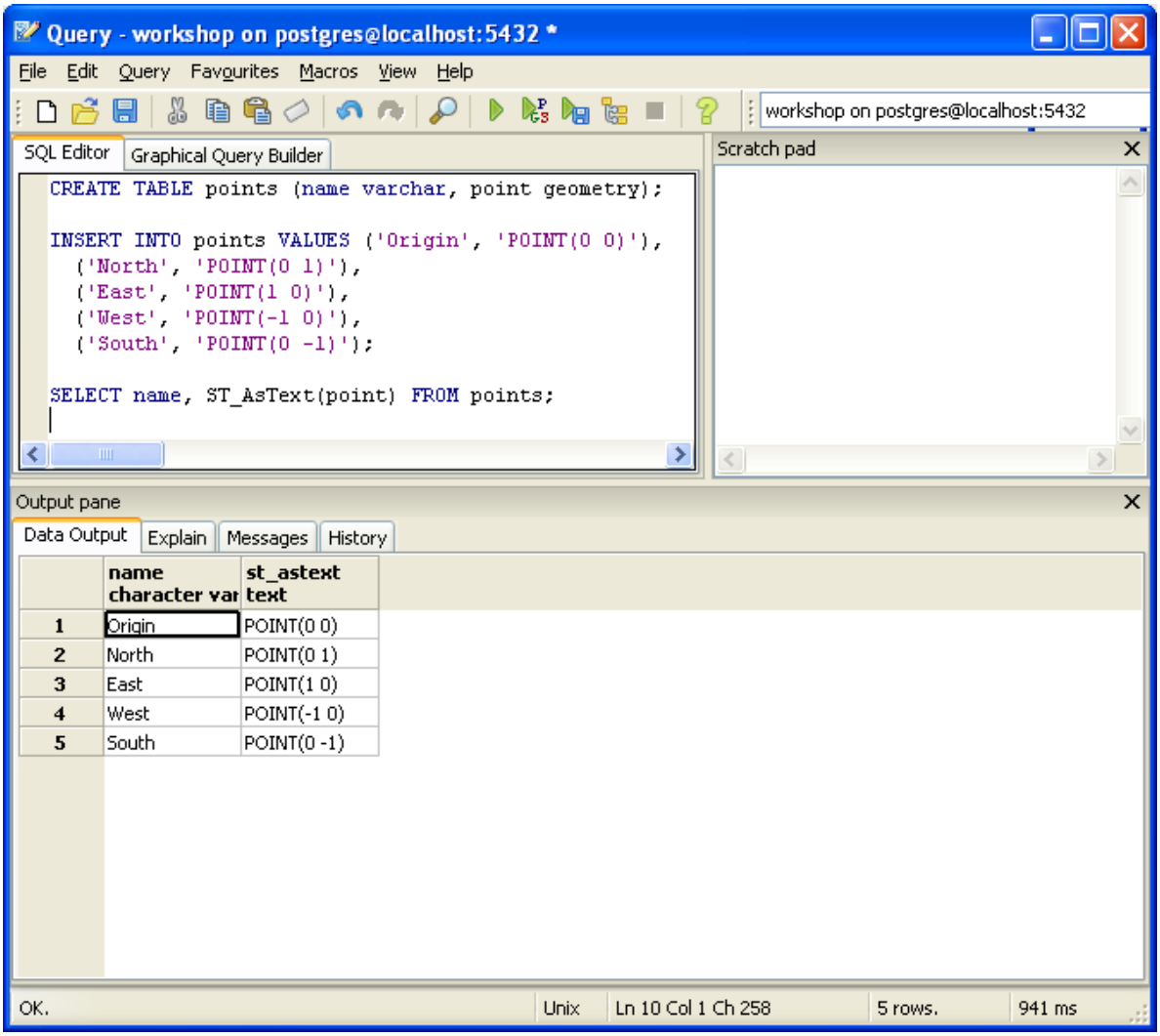
Introduction to PostGIS

Section 5: Geometries

Introduction

In the last section we loaded a variety of data, but before we start playing with that, lets have a look at some simpler examples. In pgAdmin, select the **workshop** database, then open the SQL query tool (*Tools* > *Query Tool*). Paste this example SQL code into the pgAdmin SQL Editor window (removing any text that may be there by default) and then execute (by clicking the **Play** button or pressing **F5**).

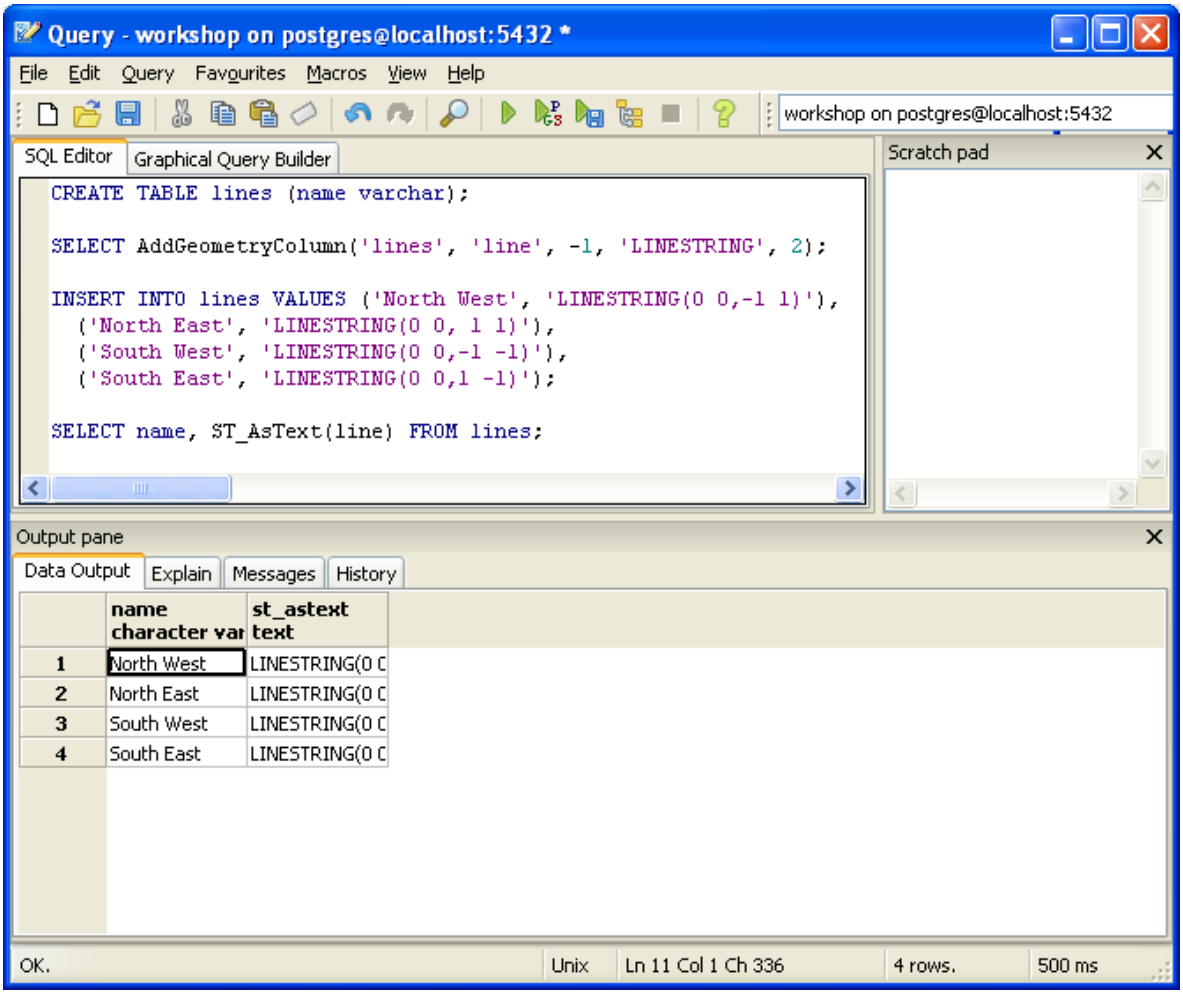
```
CREATE TABLE points (name varchar, point geometry);

INSERT INTO points VALUES ('Origin', 'POINT(0 0)'),
  ('North', 'POINT(0 1)'),
  ('East', 'POINT(1 0)'),
  ('West', 'POINT(-1 0)'),
  ('South', 'POINT(0 -1)');

SELECT name, ST_AsText(point) FROM points;
```



The above example CREATEs a table (**points**) then INSERTs five points: four points on each cardinal direction, and one at the origin. Finally, the inserted rows are SELECTed and displayed in the Output pane.

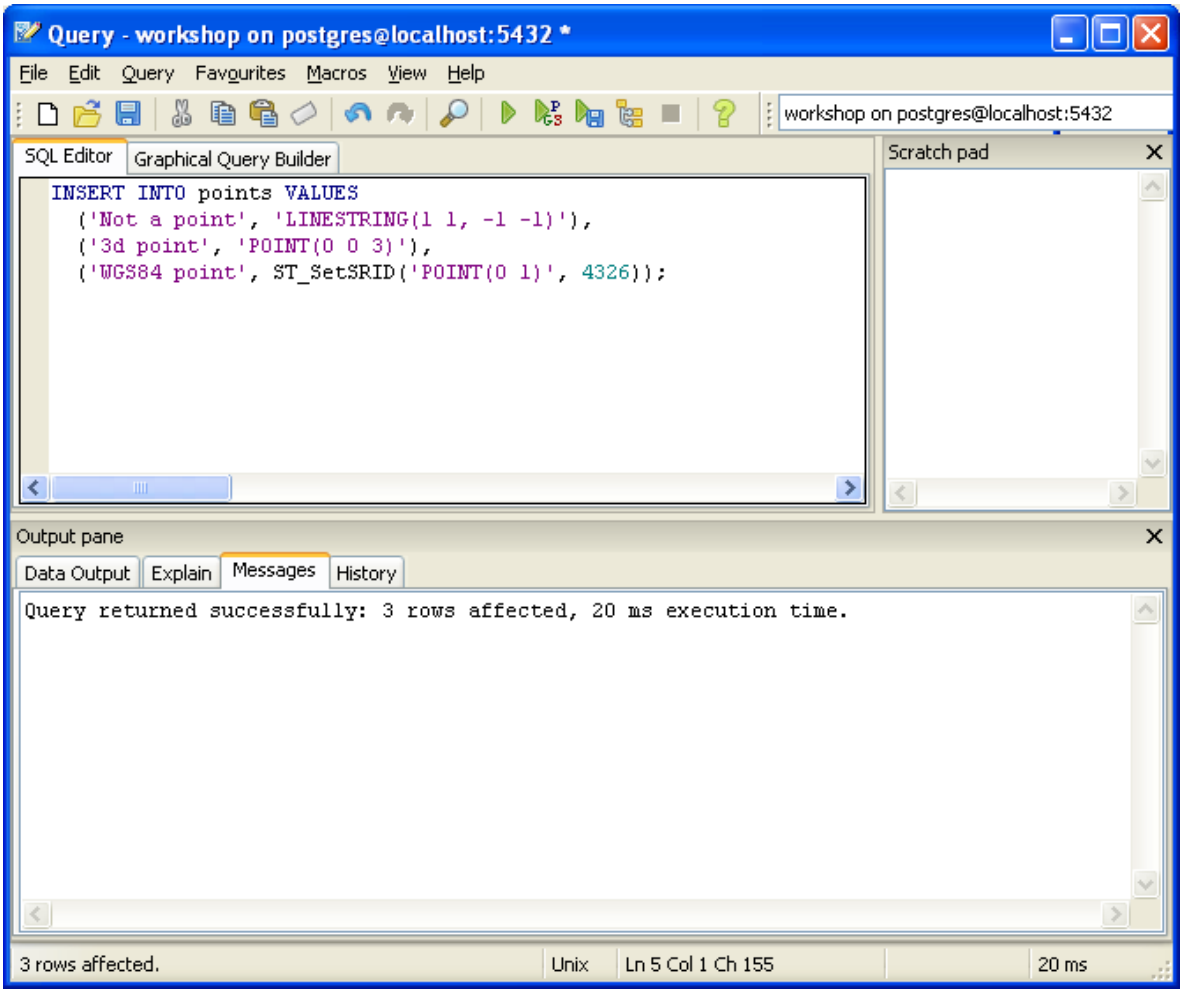
Now paste in this example (removing the previously pasted text):

```
CREATE TABLE lines (name varchar);

SELECT AddGeometryColumn('lines', 'line', -1, 'LINESTRING', 2);

INSERT INTO lines VALUES ('North West', 'LINESTRING(0 0,-1 1)'),
  ('North East', 'LINESTRING(0 0, 1 1)'),
  ('South West', 'LINESTRING(0 0,-1 -1)'),
  ('South East', 'LINESTRING(0 0,1 -1)');

SELECT name, ST_AsText(line) FROM lines;
```



This example accomplishes the much teh same thing as the previous query, but with lines. However, it takes a different approach to creating the table. To see the effects of each approach, run the following inserts:

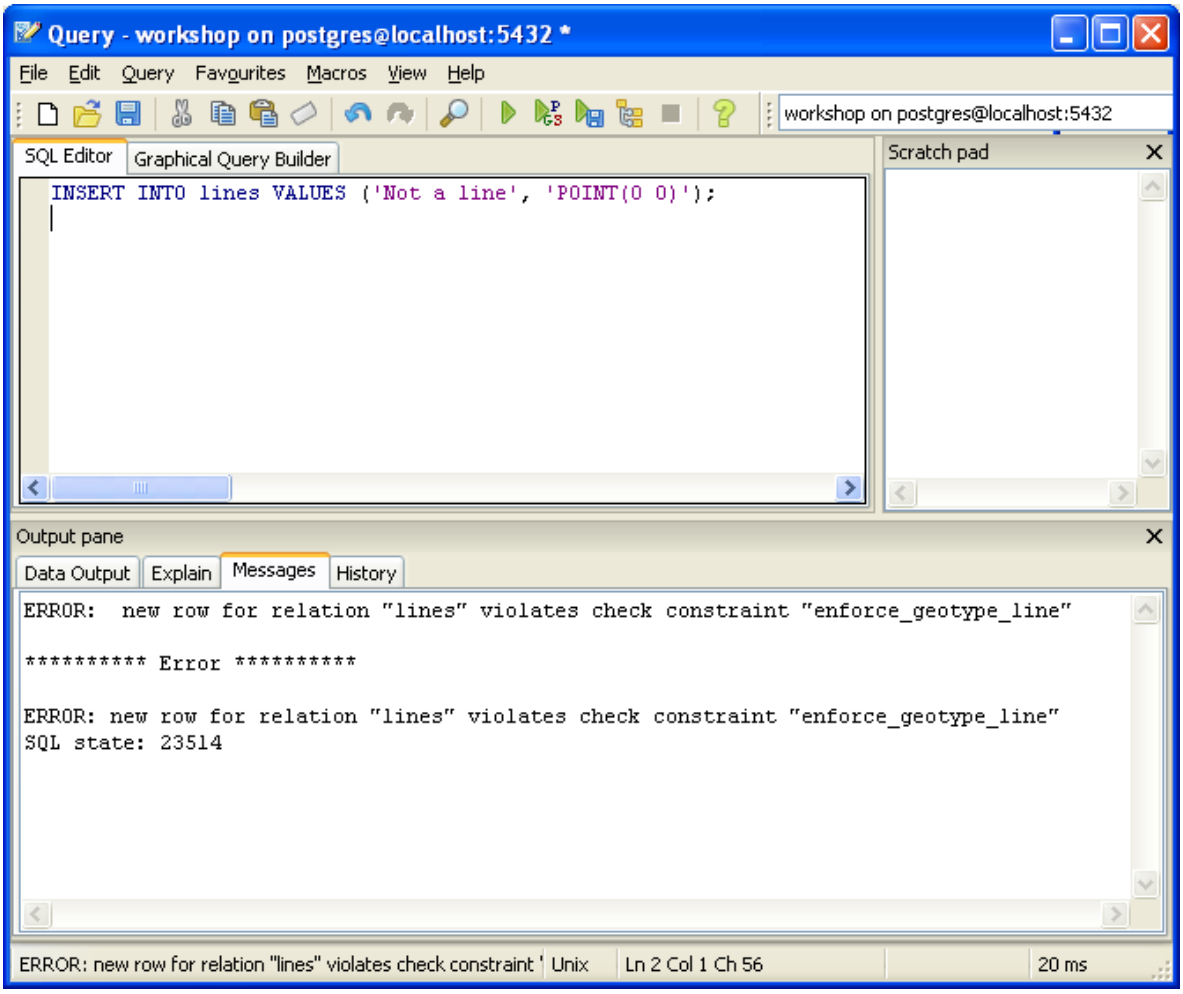
```
INSERT INTO points VALUES
  ('Not a point', 'LINESTRING(1 1, -1 -1)'),
  ('3d point', 'POINT(0 0 3)'),
  ('WGS84 point', ST_SetSRID('POINT(0 1)', 4326));
```



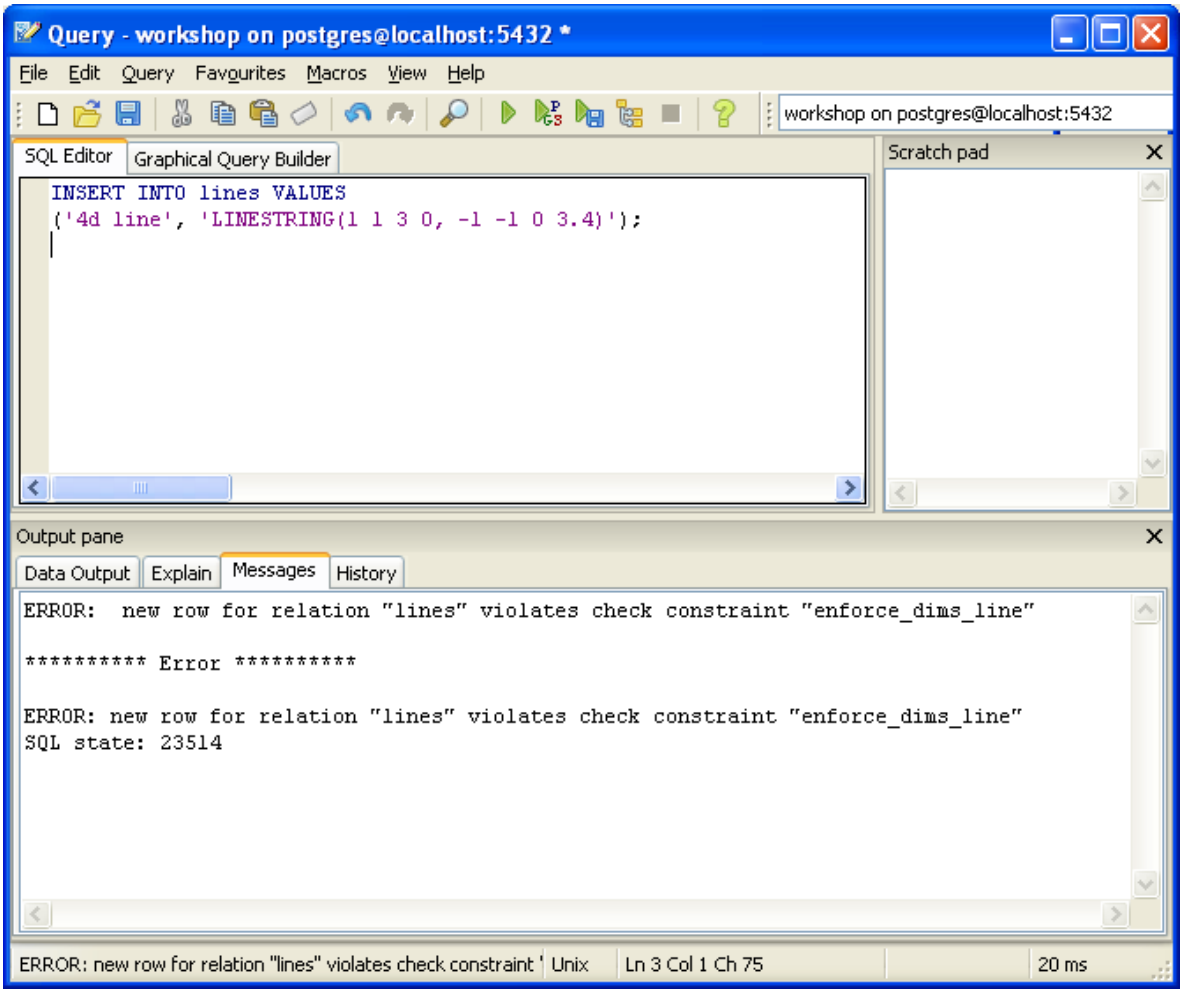
Here we're inserting some data into the **points** table that is of a different type from the original data – a line, a 3D point, and a point with a spatial reference ID. All are inserted successfully without complaint.

Now try these inserts on the **lines** table.

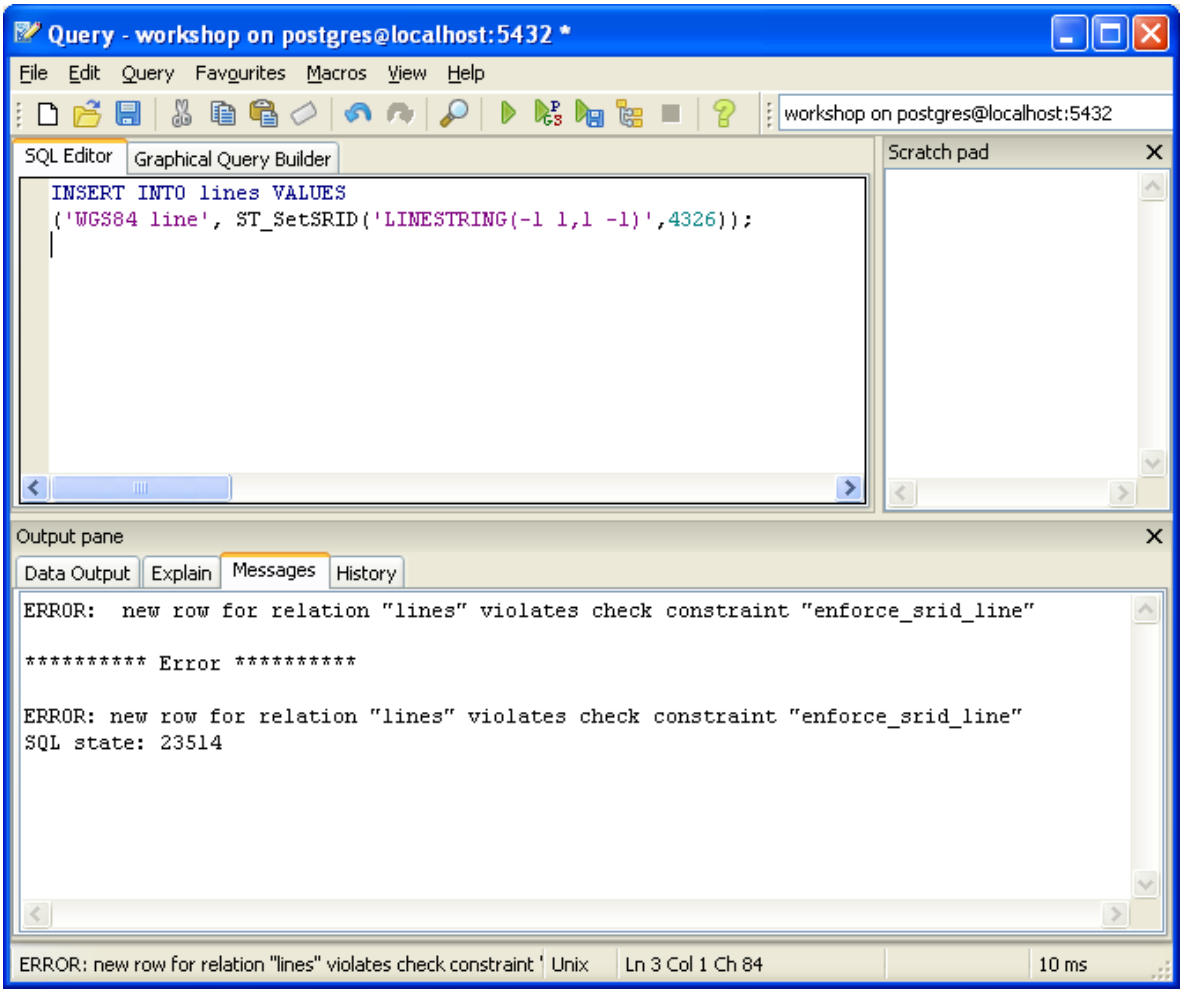
```
INSERT INTO lines VALUES ('Not a line', 'POINT(0 0)');
```




```
INSERT INTO lines VALUES
('4d line', 'LINESTRING(1 1 3 0, -1 -1 0 3.4)');
```

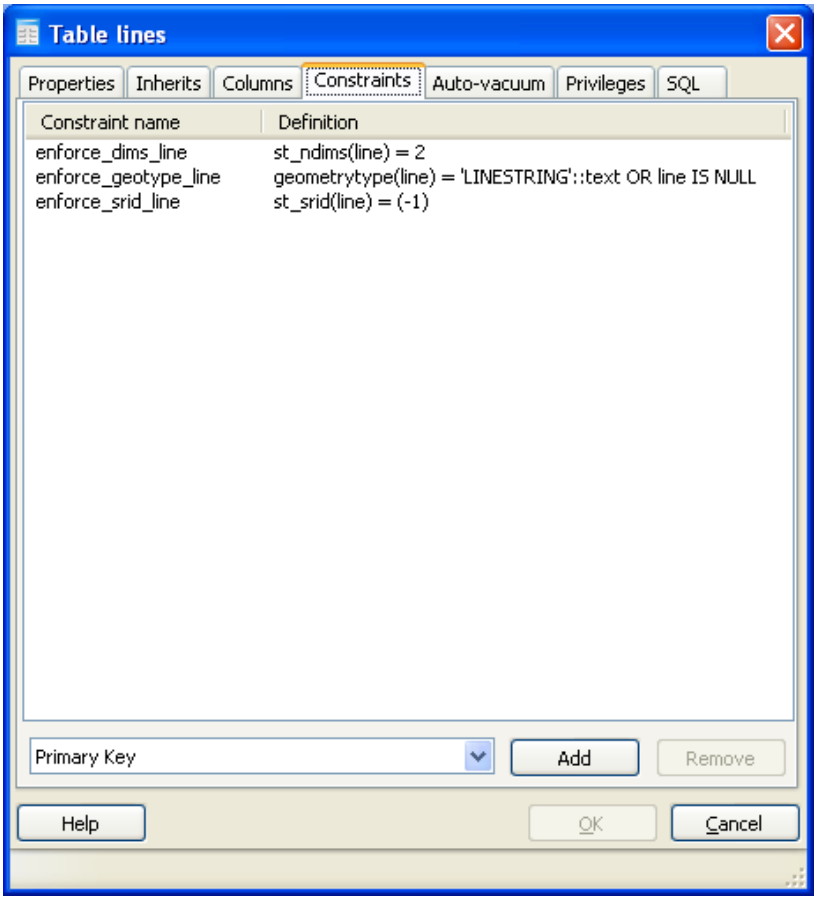



```
INSERT INTO lines VALUES
('WGS84 line', ST_SetSRID('LINESTRING(-1 1,1 -1)',4326));
```



The inserts into the **points** table were all successful, while comparable inserts into the **lines** table each failed validation. Take a look at the table definitions to understand why.

Find the **lines** table in the pgAdmin Object Browser under *Databases* > *workshop* > *Schemas* > *public* > *Tables* > *lines*. (You may need to Refresh the view.) Right-click and select the *Properties* item from the bottom of the menu and select the *Constraints* tab in the resulting dialogue. It should look like this:



Here we can see the three constraints we violated in the previous INSERT statements.

- **enforce_dims_line** declares the coordinate dimension of the geometries; it ensures that all geometries in the table are of the same dimension, in this case two-dimensional.
- **enforce_geotype_line** enforces the type of geometry, in this case **LINESTRING** (while allowing for null geometries).
- **enforce_srid_line** enforces the SRID (the identifier of the Spatial Reference system describing the geometries). In this case it is defined using the special value **-1** to indicate no Spatial Reference system is defined.

These constraints were applied to the table when the **AddGeometryColumn** function is called, to ensure consistency throughout the table. As we saw with the **points** table, simply declaring the **geometry** type in the table creation statement will not include any constraints.

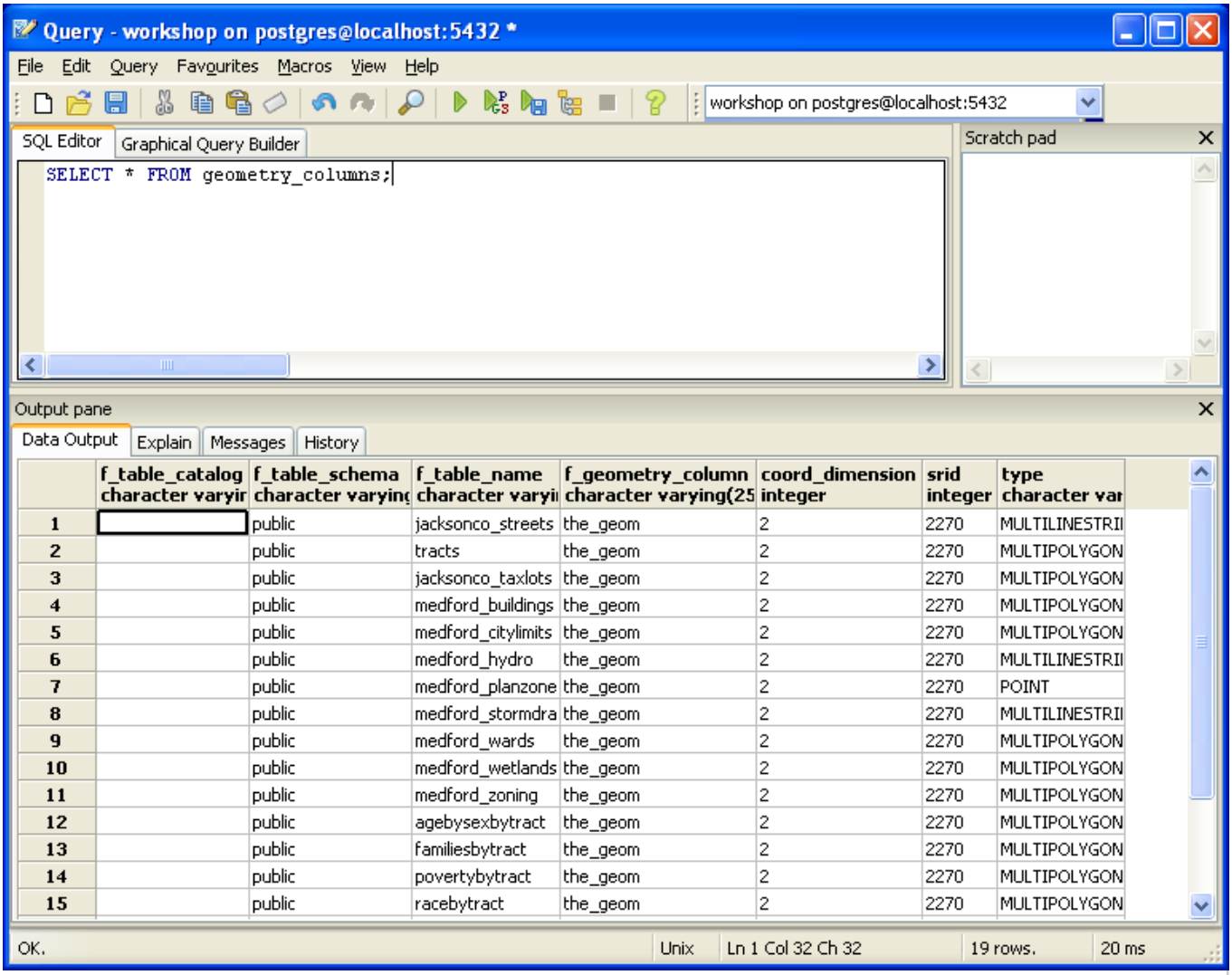
Metadata Tables

The **AddGeometryColumn** statement also tracks geometry metadata for the database. In conformance with the Simple Features for SQL (*SFSQL*) specification, PostGIS provides two tables to track and report on the geometry types available in a given database.

- The first table, **spatial_ref_sys**, defines the Spatial Reference systems known to the database and will be described in greater detail later.
- The second table, **geometry_columns**, provides a listing of all “features” (defined as an object with geometric attributes), and the basic details of those features.

Lets have a look at the **geometry_columns** table in our database. Paste this command in the Query Tool as before:

```
SELECT * FROM geometry_columns;
```



	f_table_catalog character varyir	f_table_schema character varying	f_table_name character varyi	f_geometry_column character varying(25	coord_dimension integer	srid integer	type character var
1		public	jacksonco_streets	the_geom	2	2270	MULTILINESTRI
2		public	tracts	the_geom	2	2270	MULTIPOLYGON
3		public	jacksonco_taxlots	the_geom	2	2270	MULTIPOLYGON
4		public	medford_buildings	the_geom	2	2270	MULTIPOLYGON
5		public	medford_citylimits	the_geom	2	2270	MULTIPOLYGON
6		public	medford_hydro	the_geom	2	2270	MULTILINESTRI
7		public	medford_planzone	the_geom	2	2270	POINT
8		public	medford_stormdra	the_geom	2	2270	MULTILINESTRI
9		public	medford_wards	the_geom	2	2270	MULTIPOLYGON
10		public	medford_wetlands	the_geom	2	2270	MULTIPOLYGON
11		public	medford_zoning	the_geom	2	2270	MULTIPOLYGON
12		public	agebysexbytract	the_geom	2	2270	MULTIPOLYGON
13		public	familiesbytract	the_geom	2	2270	MULTIPOLYGON
14		public	povertybytract	the_geom	2	2270	MULTIPOLYGON
15		public	racebytract	the_geom	2	2270	MULTIPOLYGON

- **f_table_catalog**, **f_table_schema** and **f_table_name** provide the fully qualified name of the feature table containing a given geometry. Because PostgreSQL doesn't make use of catalogs, **f_table_catalog** will tend to be empty.
- **f_geometry_column** is the name of the column that geometry containing column – for feature tables with multiple geometry columns, there will be one record for each.
- **coord_dimension** and **srid** define the the dimension of the geometry (2-, 3- or 4-dimensional) and the Spatial Reference system identifier that refers to the **spatial_ref_sys** table respectively.
- The **type** column defines the type of geometry as described below; we've seen Point and Linestring types so far.

By querying this table, GIS clients and libraries can determine what to expect when retrieving data and can perform any necessary projection, processing or rendering accordingly without needing to inspect each geometry as it is retrieved.

Representing Real World Objects

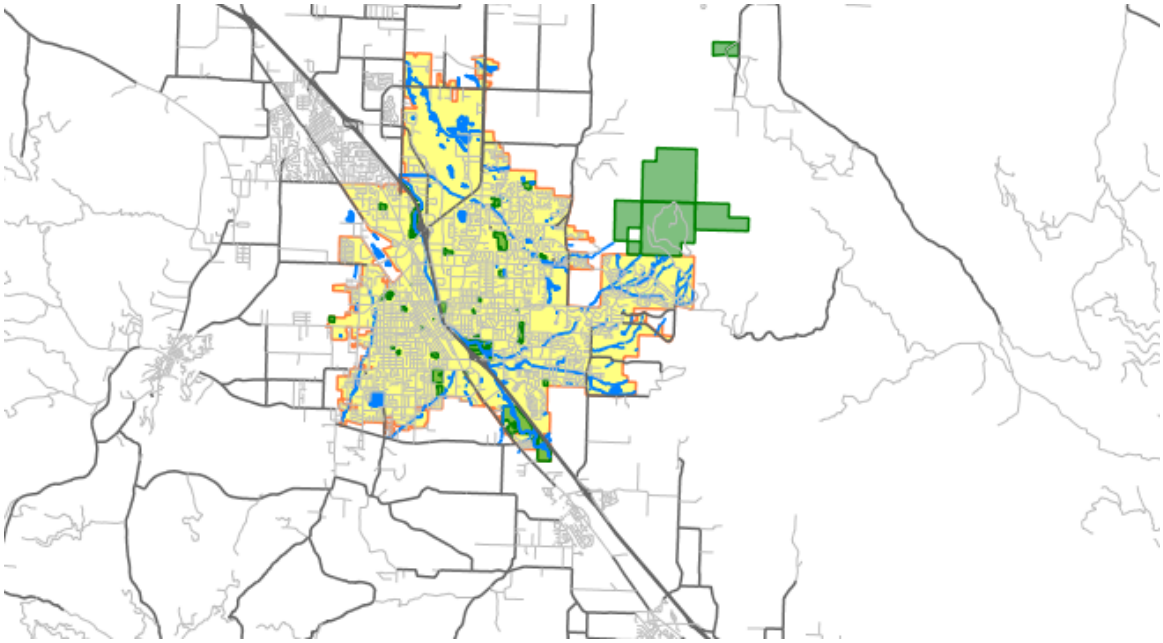
The Simple Features for SQL (*SFSQL*) specification, the original guiding standard for PostGIS development, defines how a real world object is represented. By taking a continuous shape and digitizing it at a fixed resolution we achieve a passable representation of the object. SFSQL only handled the 2-dimensional representation. PostGIS has extended that to include 3- and 4-dimensional representations; more recently the SQL-Multimedia Part 3 (*SQL/MM*) specification has officially defined their own representation. SFSQL defines three basic feature types: **POINT**, **LINESTRING** and **POLYGON**.

Points

A spatial **point** represents a single location on the Earth. This point is represented by a single coordinate (including either 2-, 3- or 4-dimensions). Points are used to represent objects when the exact details, such as shape and size,

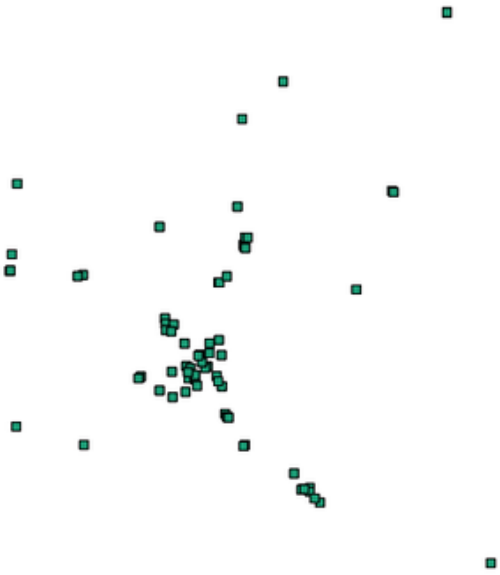
are not important at the target scale. For example, cities on a map of the world can be described as points, while a map of a single state might represent cities as polygons.

Cities represented by points



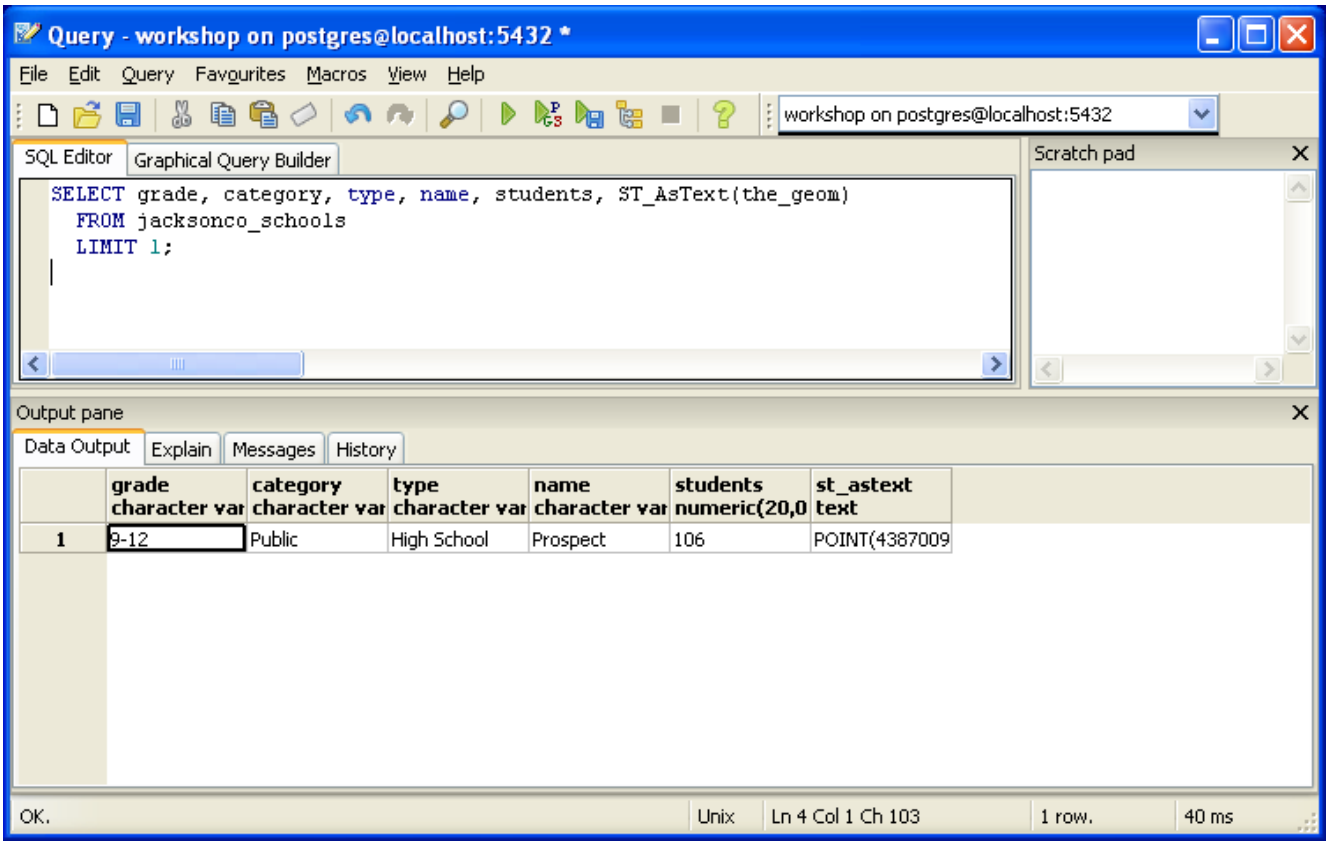
City represented by its boundary

Data sets representing schools in Jackson County (**jacksonco_schools**) were among the layers loaded earlier. See below for a visual representation of this data:



The following SQL query will return the geometry associated with one point (in the **ST_AsText** column).

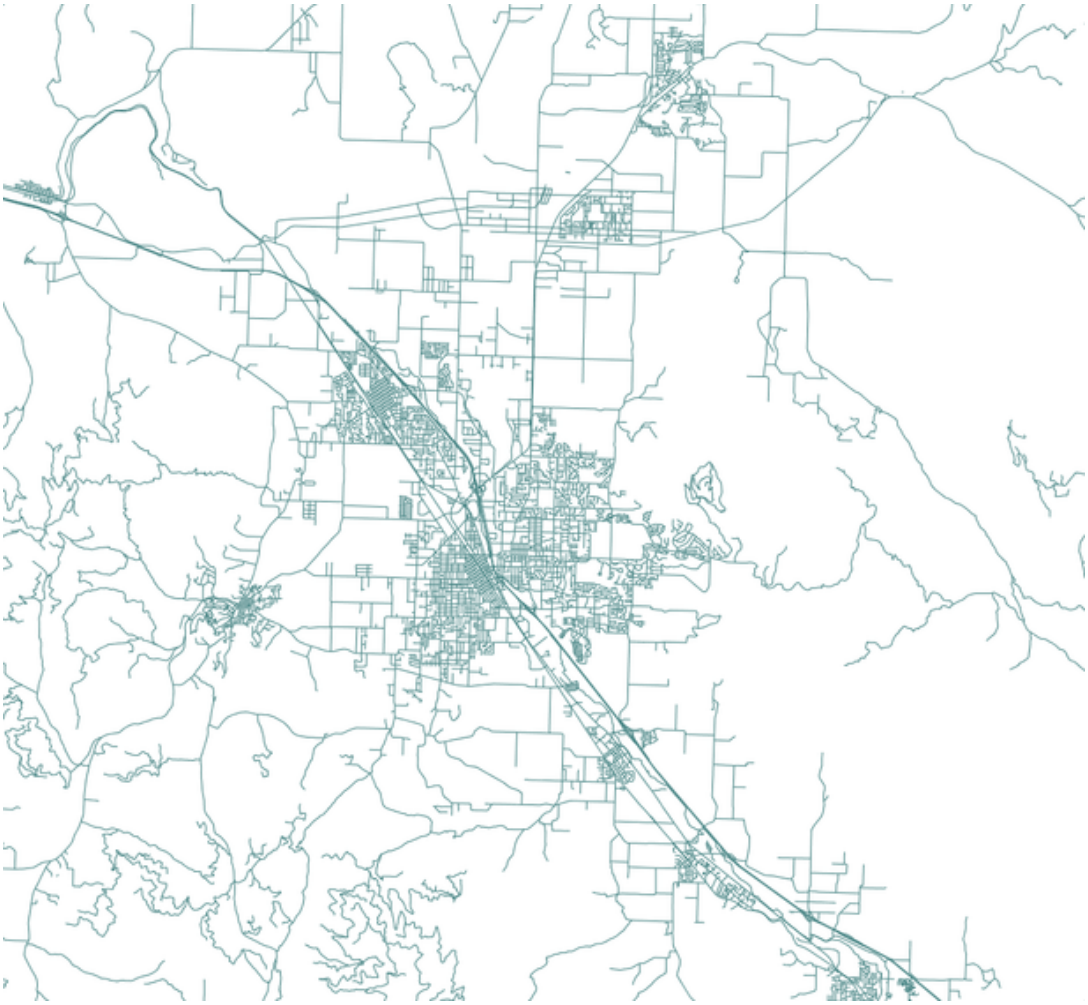
```
SELECT grade, category, type, name, students, ST_AsText(the_geom)
  FROM jacksonco_schools
  LIMIT 1;
```



Linestrings

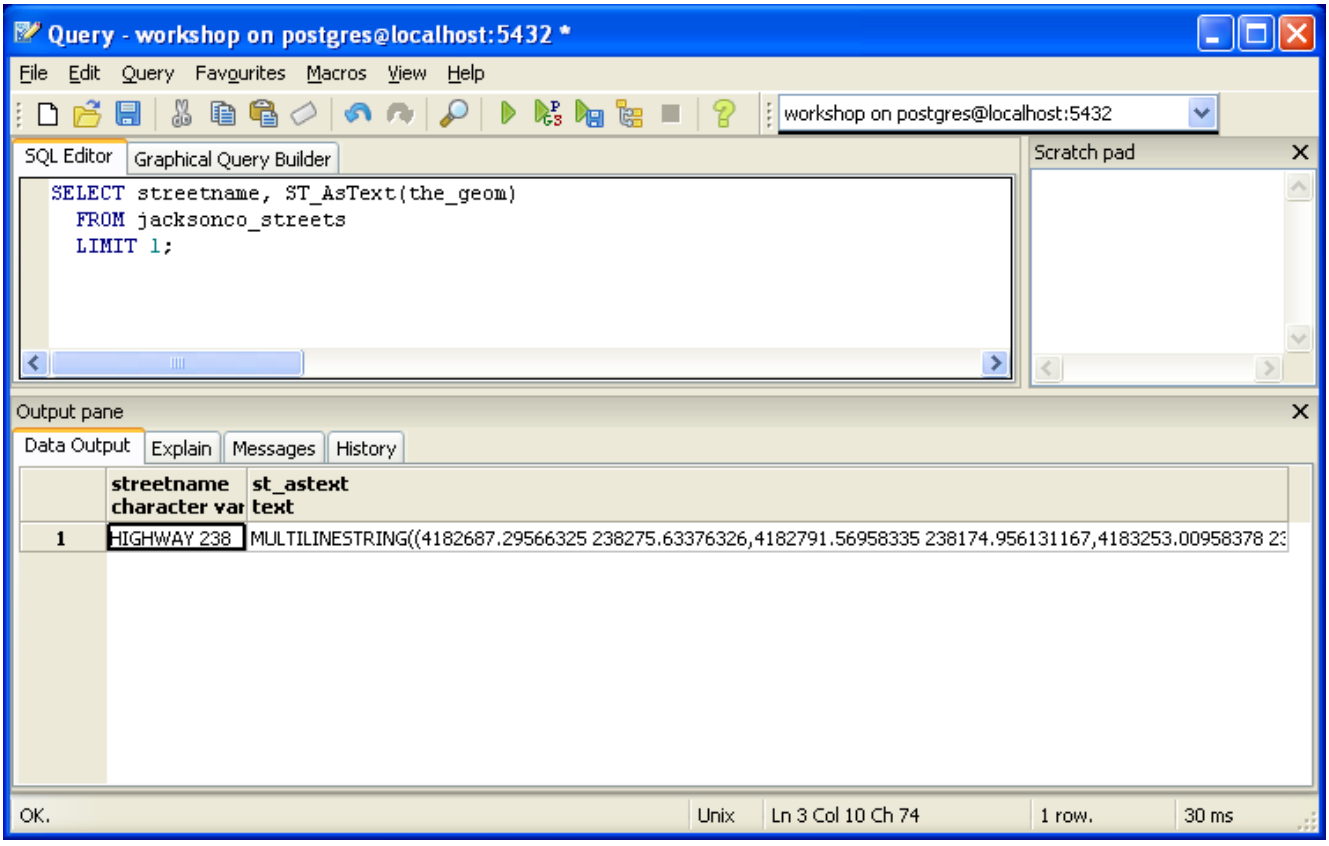
A **linestring** is a path between locations. It takes the form of an ordered series of two or more points. Roads and rivers are typically represented as linestrings, as are boundaries between political areas such as nations or cities. A linestring is said to be **closed** if it starts and ends on the same point. It is said to be **simple** if it does not cross or touch itself (except at its endpoints if it is closed). A linestring can be both **closed** and **simple**.

The street network for Jackson County (**jacksonco_streets**) was loaded earlier in the workshop. This dataset contains details such as name, surface type, and address details. A single real world street may consist of many linestrings, each representing a segment of road with different attributes.



The following SQL query will return the geometry associated with one linestring (in the **ST_AsText** column).

```
SELECT streetname, ST_AsText(the_geom)
  FROM jacksonco_streets
  LIMIT 1;
```



Polygons

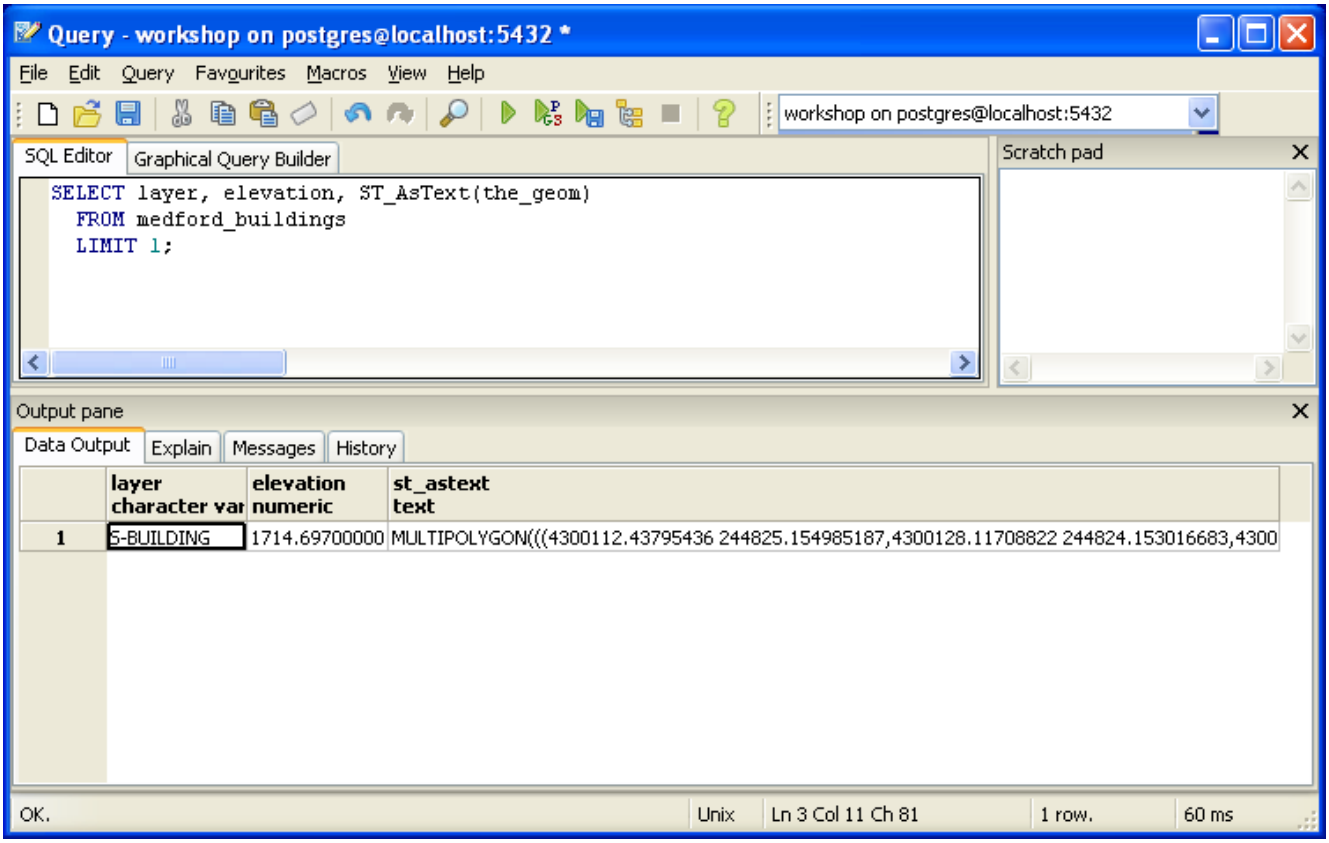
A polygon is a representation of an area. The outer boundary of the polygon is represented by a ring. This ring is a linestring that is both closed and simple as defined above. Holes within the polygon are also represented by rings.

Polygons are used to represent objects whose size and shape are important. City limits, parks, building footprints or bodies of water are all commonly represented as polygons when the scale is sufficiently high to see their area. Roads and rivers can also sometimes be represented as polygons.

A number of polygon layers were loaded earlier in the workshop, including the building footprint layer for the city of Medford, **medford_buildings**.

The following SQL query will return the geometry associated with one polygon (in the **ST_AsText** column).

```
SELECT layer, elevation, ST_AsText(the_geom)
  FROM medford_buildings
  LIMIT 1;
```

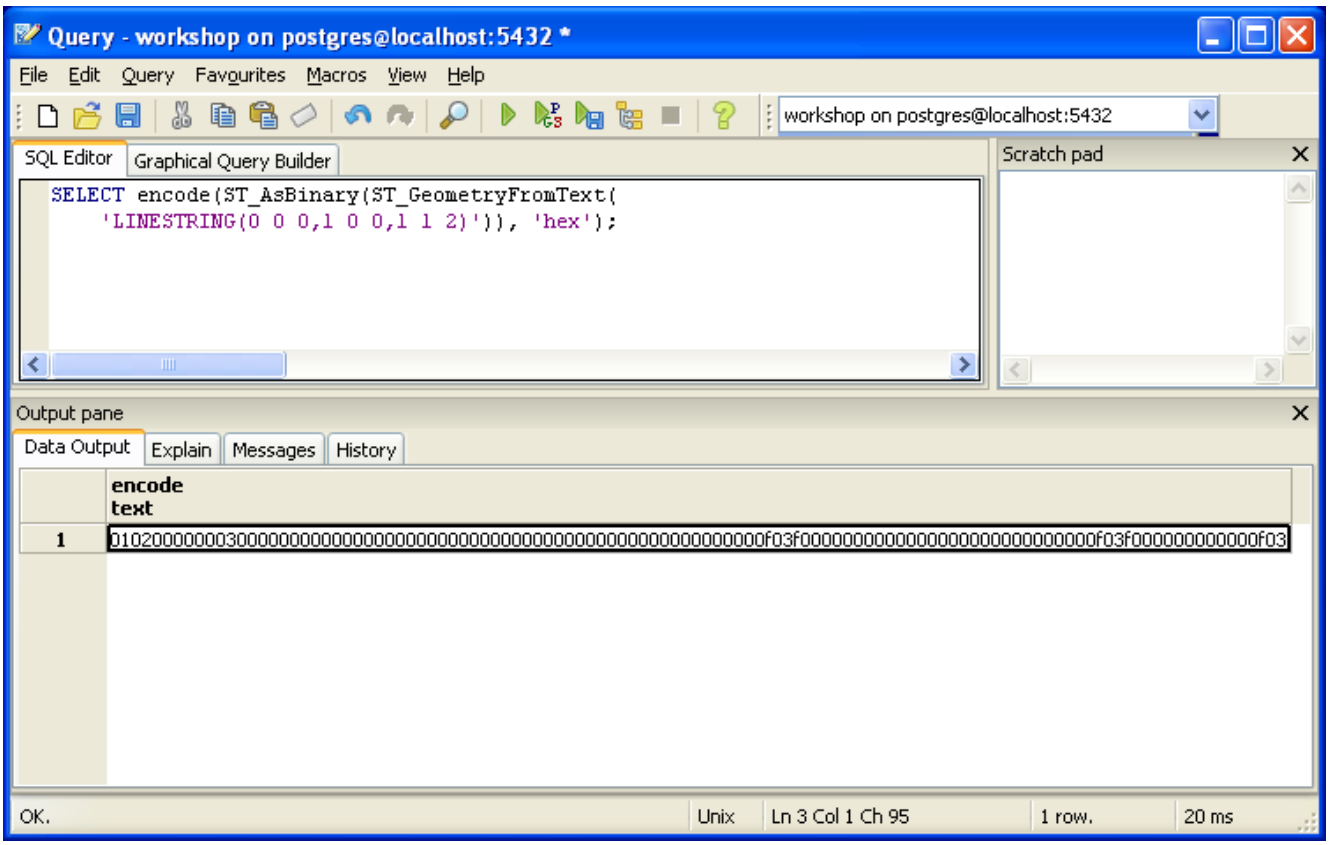


Geometry Output

You may have noticed that in all the previous queries the geometry column was wrapped in the function **ST_AsText()**. This function formats the geometry into Well Known Text (*WKT*). The WKT representation is much easier to read than the more compact Well Known Binary (WKB) version. On the other hand, WKT can cause coordinate “drift” due to conversion between decimal and binary versions of the coordinates.

The following SQL query shows an example of WKB representation:

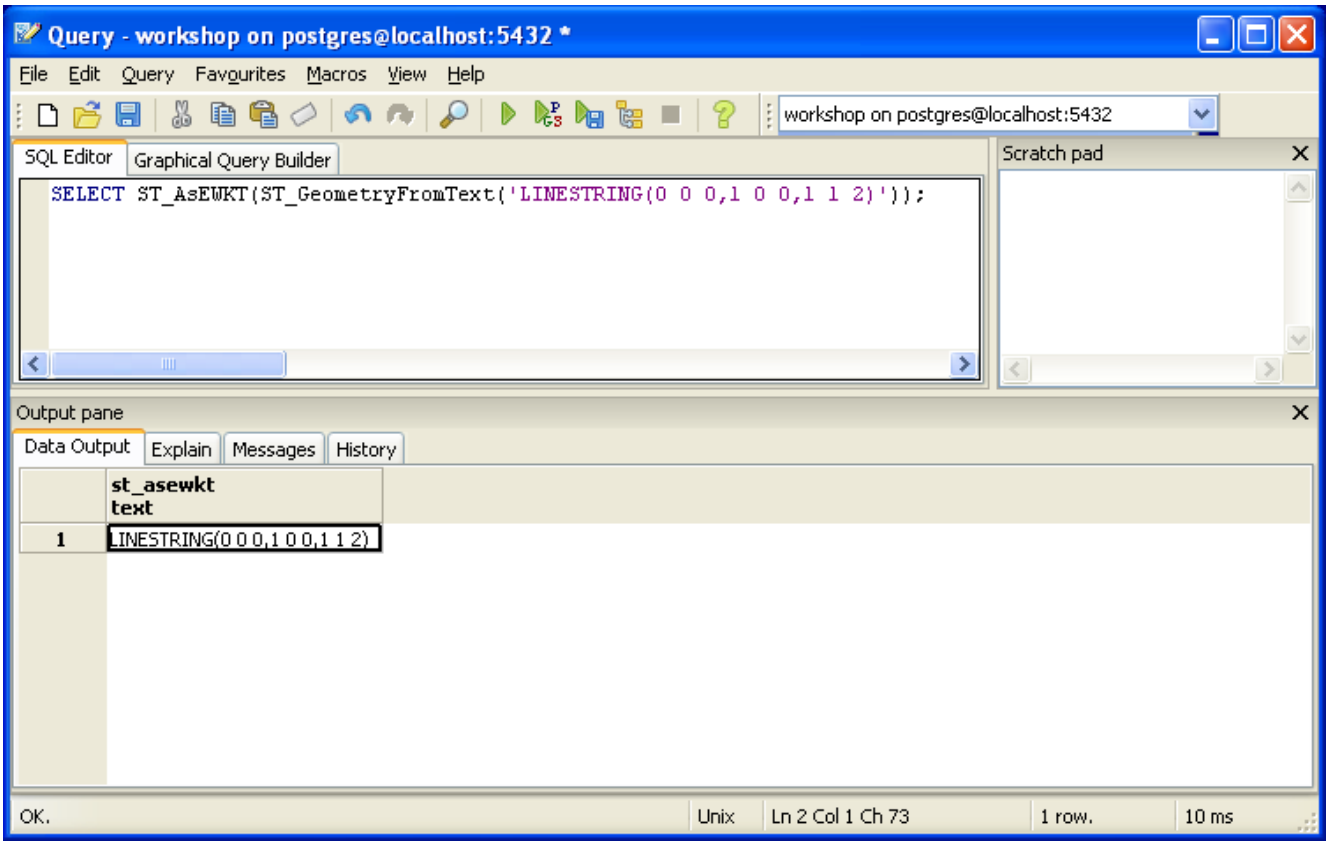
```
SELECT encode(ST_AsBinary(ST_GeometryFromText(
    'LINESTRING(0 0 0,1 0 0,1 1 2)')), 'hex');
```



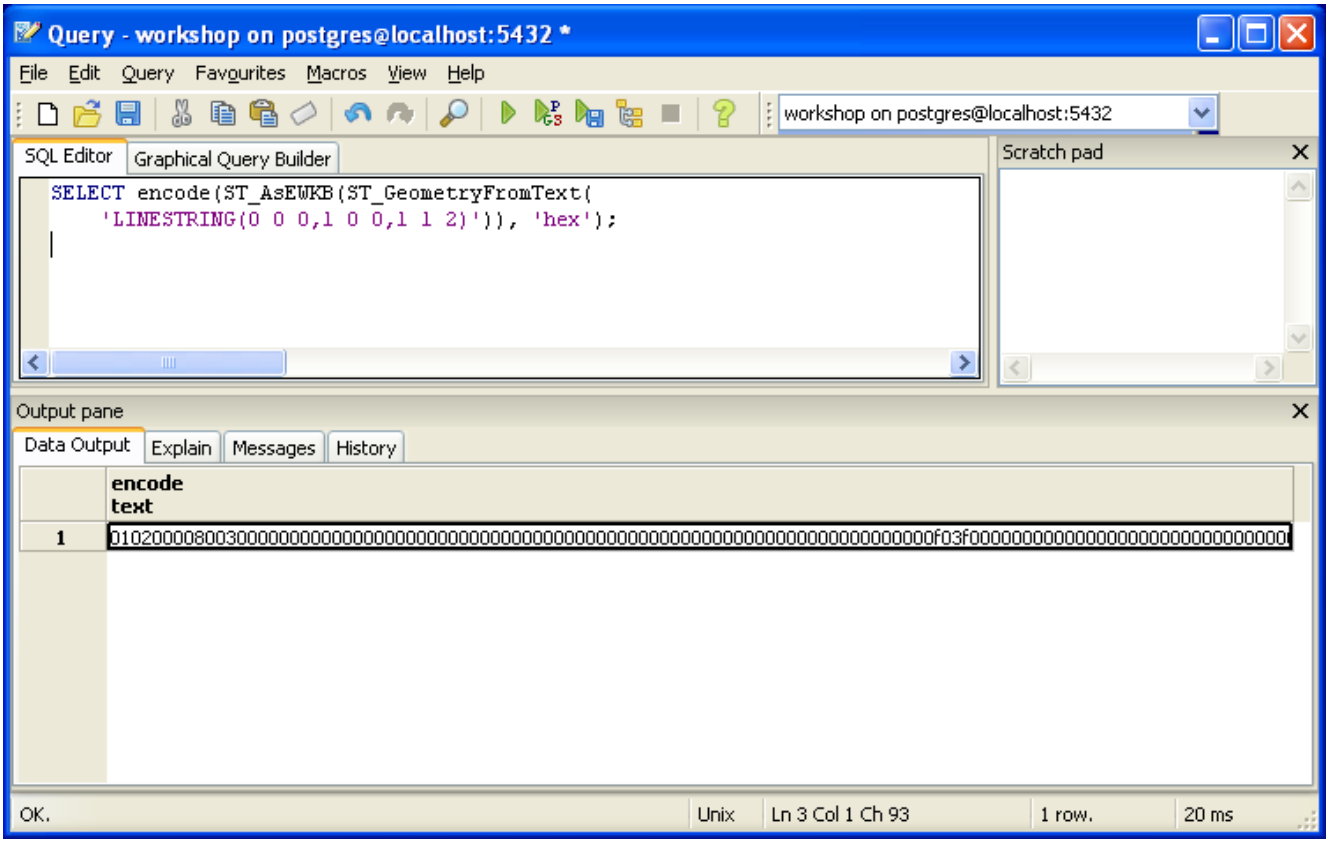
For the purposes of this workshop we will continue to use WKT in most cases to ensure you can read and understand the geometries we're viewing. However, for most actual purposes, such as viewing data in a GIS application, transferring data to a web service, or processing data remotely, WKB is the format of choice. Since WKT and WKB were defined in the *SFSQL* specification, they do not handle 3- or 4-dimensional geometries. For these cases PostGIS has defined the Extended Well Known Text (EWKT) and Extended Well Known Binary (EWKB) formats. These provide the same formatting capabilities of WKT and WKB with the added dimensionality.

Here are examples of a 3D linestring (in both WKT and WKB):

```
SELECT ST_AsEWKT(ST_GeometryFromText('LINESTRING(0 0 0,1 0 0,1 1 2)'));
```



```
SELECT encode(ST_AsEWKB(ST_GeometryFromText(
    'LINESTRING(0 0 0,1 0 0,1 1 2)')), 'hex');
```



(E)WKT and (E)WKB are not the only ways of formatting PostGIS output. There are output functions for Simple Vector Graphics (**ST_AsSVG()**), Geographic Markup Language (**ST_AsGML()**), Keyhole Markup Language (**ST_AsKML()**) and GeoJSON (**ST_AsGeoJSON()**).