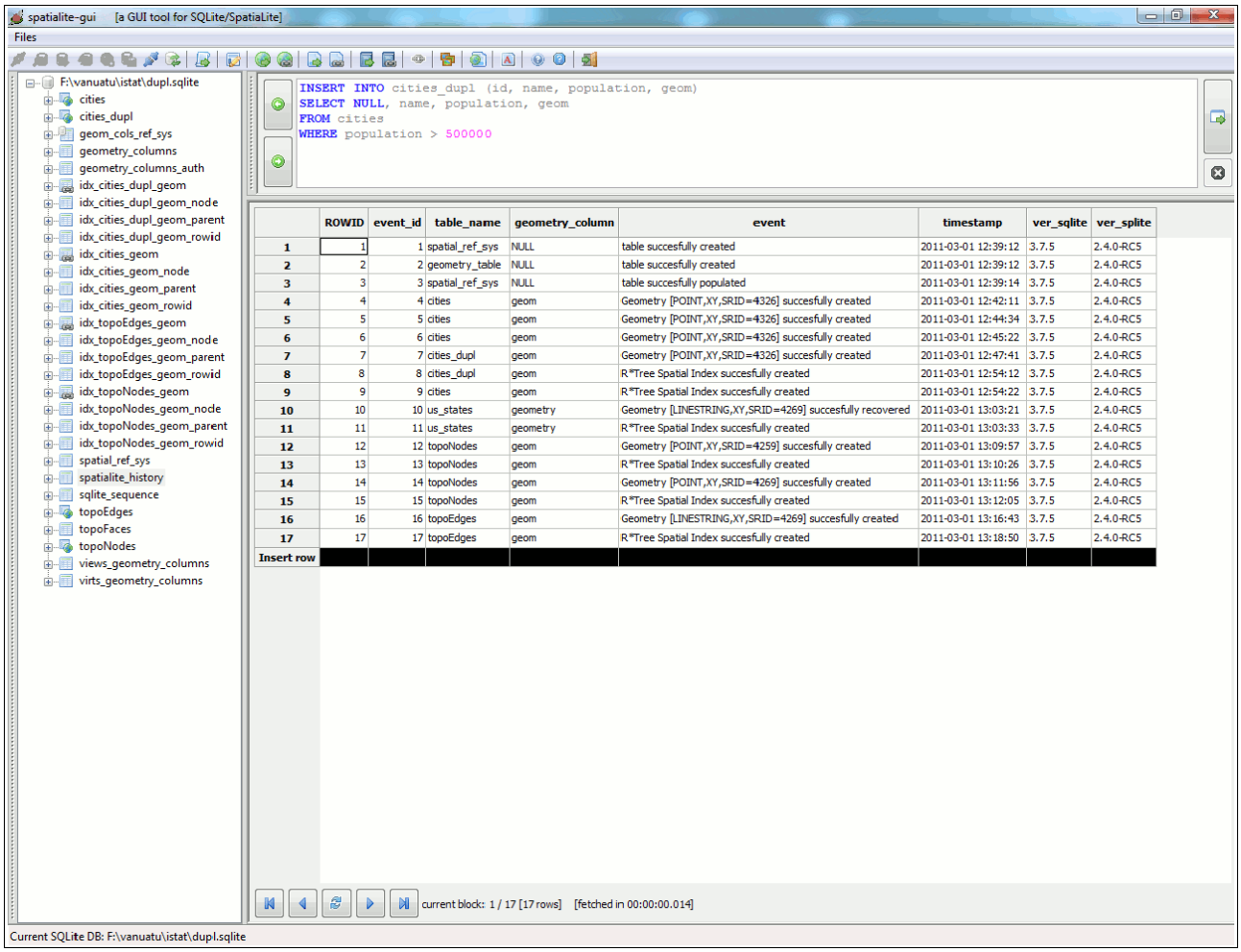
spatialite_gui-1.4.1
libspatialite v.2.4.0-RC5 Experimental

#1 The new **spatialite_history** metadata table



	ROWID	event_id	table_name	geometry_column	event	timestamp	ver_sqlite	ver_splite
1	1	1	spatial_ref_sys	NULL	table succesfully created	2011-03-01 12:39:12	3.7.5	2.4.0-RC5
2	2	2	geometry_table	NULL	table succesfully created	2011-03-01 12:39:12	3.7.5	2.4.0-RC5
3	3	3	spatial_ref_sys	NULL	table succesfully populated	2011-03-01 12:39:14	3.7.5	2.4.0-RC5
4	4	4	cities	geom	Geometry [POINT,XY,SRID=4326] succesfully created	2011-03-01 12:42:11	3.7.5	2.4.0-RC5
5	5	5	cities	geom	Geometry [POINT,XY,SRID=4326] succesfully created	2011-03-01 12:44:34	3.7.5	2.4.0-RC5
6	6	6	cities	geom	Geometry [POINT,XY,SRID=4326] succesfully created	2011-03-01 12:45:22	3.7.5	2.4.0-RC5
7	7	7	cities_dupl	geom	Geometry [POINT,XY,SRID=4326] succesfully created	2011-03-01 12:47:41	3.7.5	2.4.0-RC5
8	8	8	cities_dupl	geom	R*Tree Spatial Index succesfully created	2011-03-01 12:54:12	3.7.5	2.4.0-RC5
9	9	9	cities	geom	R*Tree Spatial Index succesfully created	2011-03-01 12:54:22	3.7.5	2.4.0-RC5
10	10	10	us_states	geometry	Geometry [LINESTRING,XY,SRID=4269] succesfully recovered	2011-03-01 13:03:21	3.7.5	2.4.0-RC5
11	11	11	us_states	geometry	R*Tree Spatial Index succesfully created	2011-03-01 13:03:33	3.7.5	2.4.0-RC5
12	12	12	topoNodes	geom	Geometry [POINT,XY,SRID=4259] succesfully created	2011-03-01 13:09:57	3.7.5	2.4.0-RC5
13	13	13	topoNodes	geom	R*Tree Spatial Index succesfully created	2011-03-01 13:10:26	3.7.5	2.4.0-RC5
14	14	14	topoNodes	geom	Geometry [POINT,XY,SRID=4269] succesfully created	2011-03-01 13:11:56	3.7.5	2.4.0-RC5
15	15	15	topoNodes	geom	R*Tree Spatial Index succesfully created	2011-03-01 13:12:05	3.7.5	2.4.0-RC5
16	16	16	topoEdges	geom	Geometry [LINESTRING,XY,SRID=4269] succesfully created	2011-03-01 13:16:43	3.7.5	2.4.0-RC5
17	17	17	topoEdges	geom	R*Tree Spatial Index succesfully created	2011-03-01 13:18:50	3.7.5	2.4.0-RC5
Insert row								

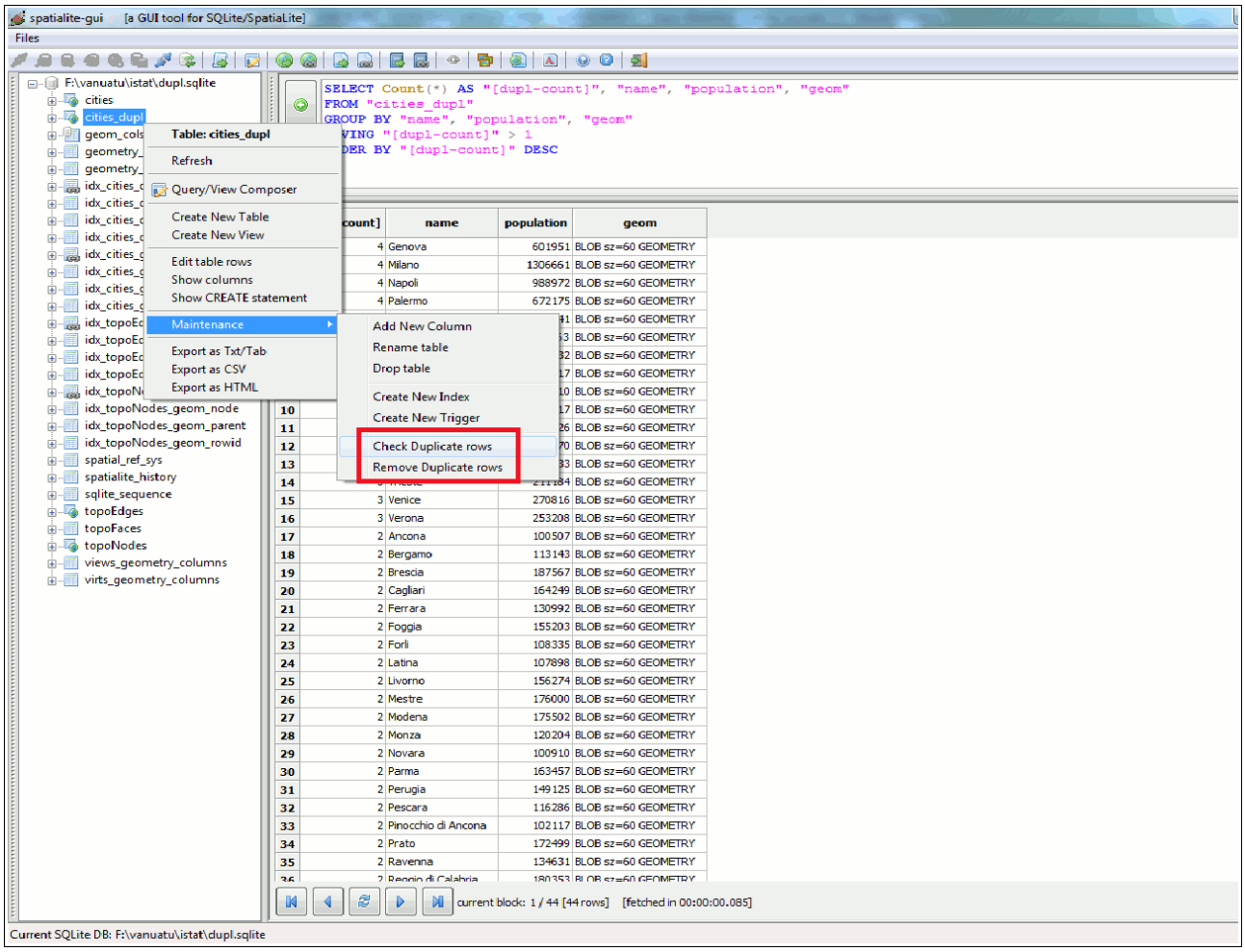
Internal event logging:

- metadata table creation and population: InitSpatialMetadata()
- geometry column creation: AddGeometryColumn(), RecoverGeometryColumn()
- Spatial Index: CreateSpatialIndex()
- evet description / timestamp
- versioning infos

#2 Checking / removing Duplicated Rows

Please see the **cities_dupl** table.
Simply the same as **cities**, but several duplicated rows were purposely inserted using the following SQL statements:

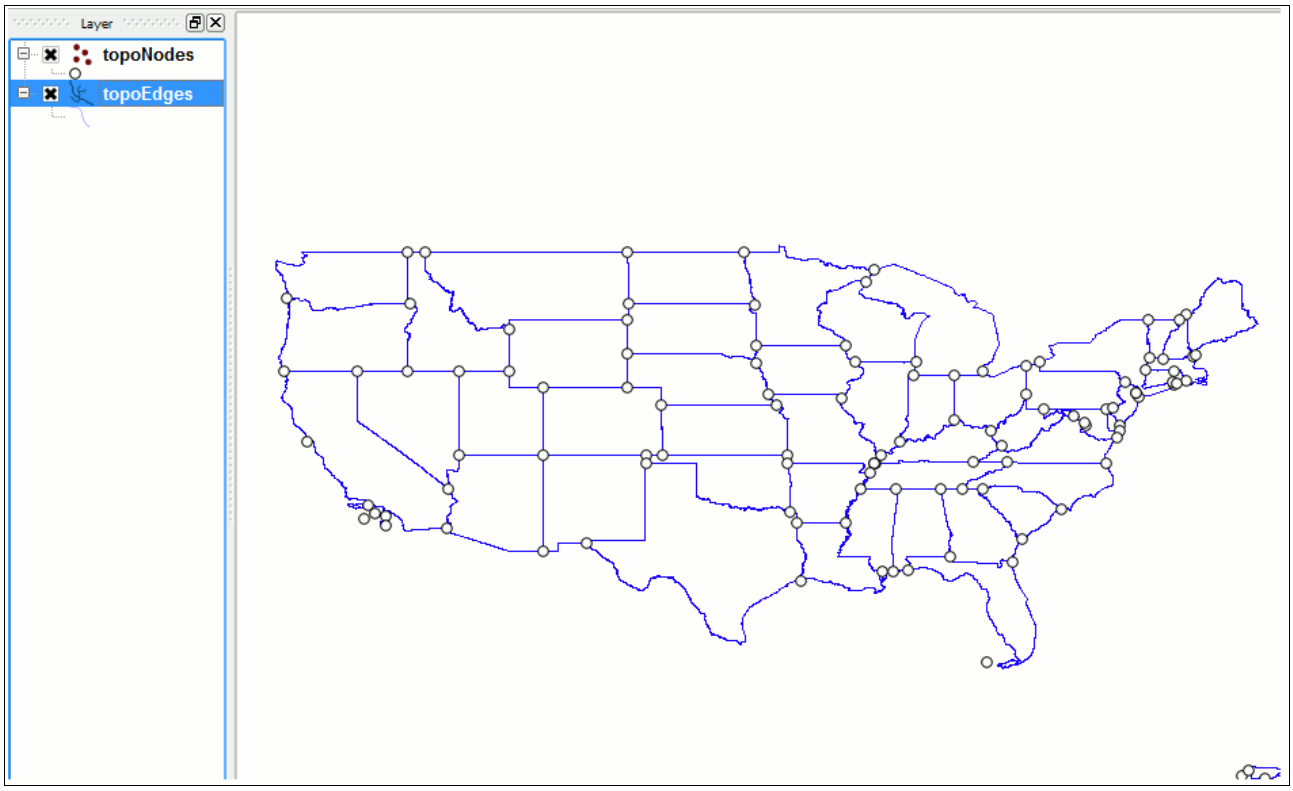
```
INSERT INTO cities_dupl (id, name, population, geom)
SELECT NULL, name, population, geom
FROM cities;
INSERT INTO cities_dupl (id, name, population, geom)
SELECT NULL, name, population, geom
FROM cities
WHERE population > 100000;
INSERT INTO cities_dupl (id, name, population, geom)
SELECT NULL, name, population, geom
FROM cities
WHERE population > 200000;
INSERT INTO cities_dupl (id, name, population, geom)
SELECT NULL, name, population, geom
FROM cities
WHERE population > 500000;
```



The **Check** tool will identify any duplicate row (please note: any **Primary Key** column will be ignored). And the **Remove** tool will eventually delete any duplicate row except the first one.

The same tool is supported by the **spatialite** CLI front end as **.chkdupl** and **.remdupl**

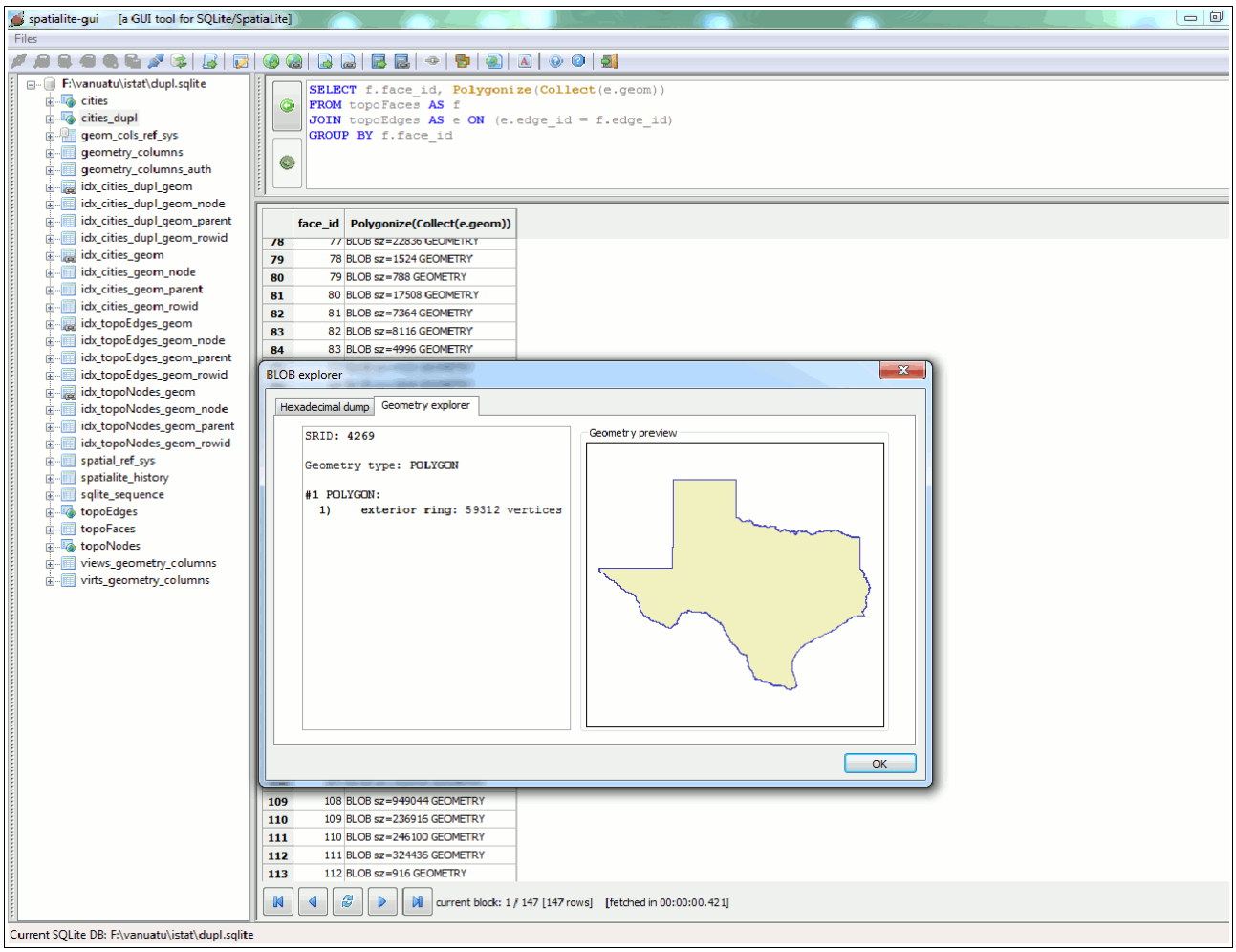
#3 Rudimentary Topology support



Please see the **topoNodes**, **topoEdges** and **topoFaces** tables.
They simply contain (alltogether) a topological representation of the United States (derived from the U.S. Census Bureau TIGER dataset).

As you can easily notice, there is no direct representation of *States as polygons*.

- an **Edge** is a LINESTRING representing a common boundary shared by to two adjacent States
- a **Node** is a POINT where two (or more) **Edges** intersects
- so each single State is represented by one (or more) **Faces**; but a **Face** simply is represented as a collection of **Edges** (i.e. the ones delimiting the face's own boundary). No explicit POLYGON is represented at the topolological level. And there is no GEOMETRY directly corresponding to a **Face**.




```
SELECT f.face_id, Polygonize(Collect(e.geom))
FROM topoFaces AS f
JOIN topoEdges AS e ON (e.edge_id = f.edge_id)
GROUP BY f.face_id;
```

Collect() (aggregate function) will create a MULTILINESTRING corresponding to the complete boundary delimiting a **Face**.
And then **Polygonize()** will reconstruct a POLYGON representing the same **Face**.

Nobody forbids us to create a further table representing State-Faces as POLYGONs:

```
CREATE TABLE state_polygs (
face_id INTEGER NOT NULL PRIMARY KEY);
SELECT AddGeometryColumn('state_polygs', 'geom', 4269, 'POLYGON',
'XY');
INSERT INTO state_polygs (face_id, geom)
SELECT f.face_id, Polygonize(Collect(e.geom))
FROM topoFaces AS f
JOIN topoEdges AS e ON (e.edge_id = f.edge_id)
GROUP BY f.face_id;
```

This is useful in order to test another two SQL functions recently introduced:

```
SELECT e.edge_id, e.geom
FROM state_polygs AS s
JOIN topoEdges AS e ON (CoveredBy(e.geom, s.geom))
WHERE s.face_id = 108;
```

This first query will identify any Edge covered by the Face corresponding to the Texas State.

```
SELECT s.face_id, s.geom
FROM topoEdges AS e
JOIN state_polygs AS s ON (Covers(s.geom, e.geom))
WHERE e.edge_id = 156;
```

And this second query will identify the two States sharing a common Edge (in this example: Texas and Louisiana).

#4 reconstruncting a GPS track from WayPoints

This time we'll use the **gps_track** table. As you can notice this table actually contains several GPS WayPoints:

```
SELECT MakeLine(MakePoint(longitude, latitude, 4326))
FROM gps_track
GROUP BY track_no;
```

Using the **MakeLine()** aggregate function you can easily get the whole GPS Track as a LINESTRING.

```
SELECT MakeLine(MakePoint(longitude, latitude, 4326))
FROM gps_track
WHERE gps_timestamp BETWEEN
  '2011-02-14T14:45' AND
  '2011-02-14T14:55'
GROUP BY track_no;
```

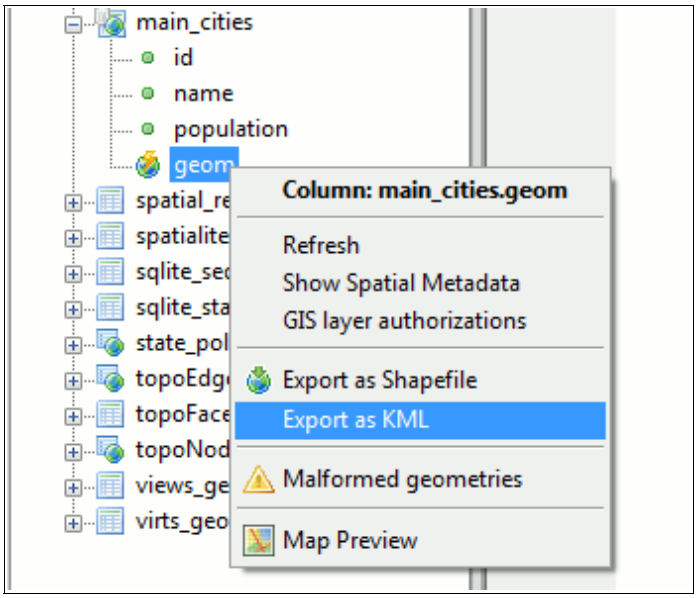
And you can obviously extract a specific portion of this GPS track setting an appropriate time interval: in this example the track walked on 2011-02-14 starting at 14:45 and ending at 14:55

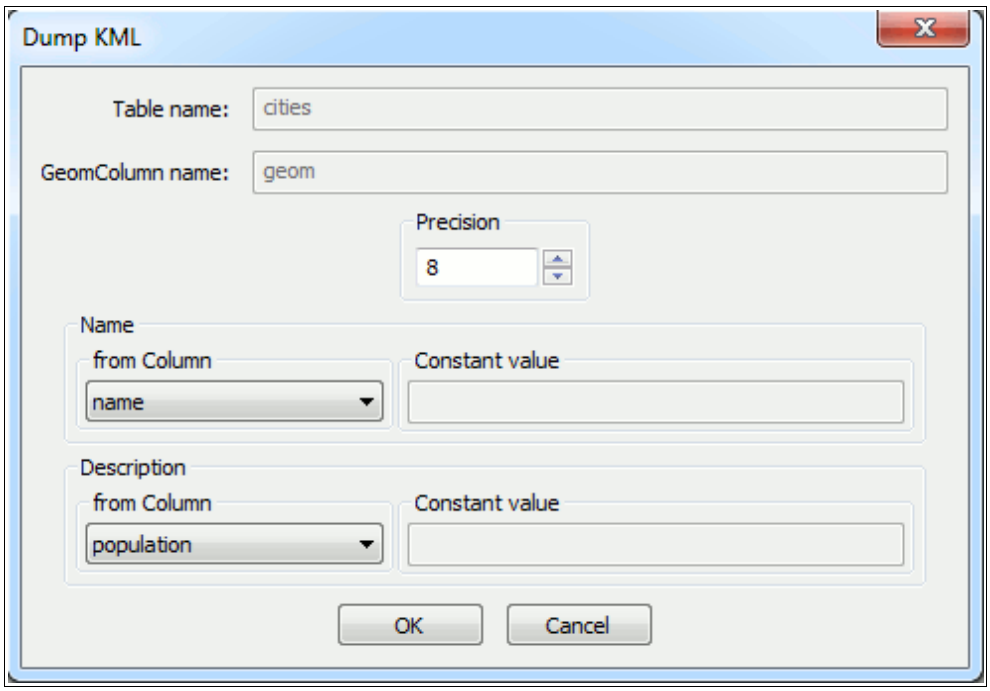
#5 exporting KML files

We'll start again from the **cities** table.

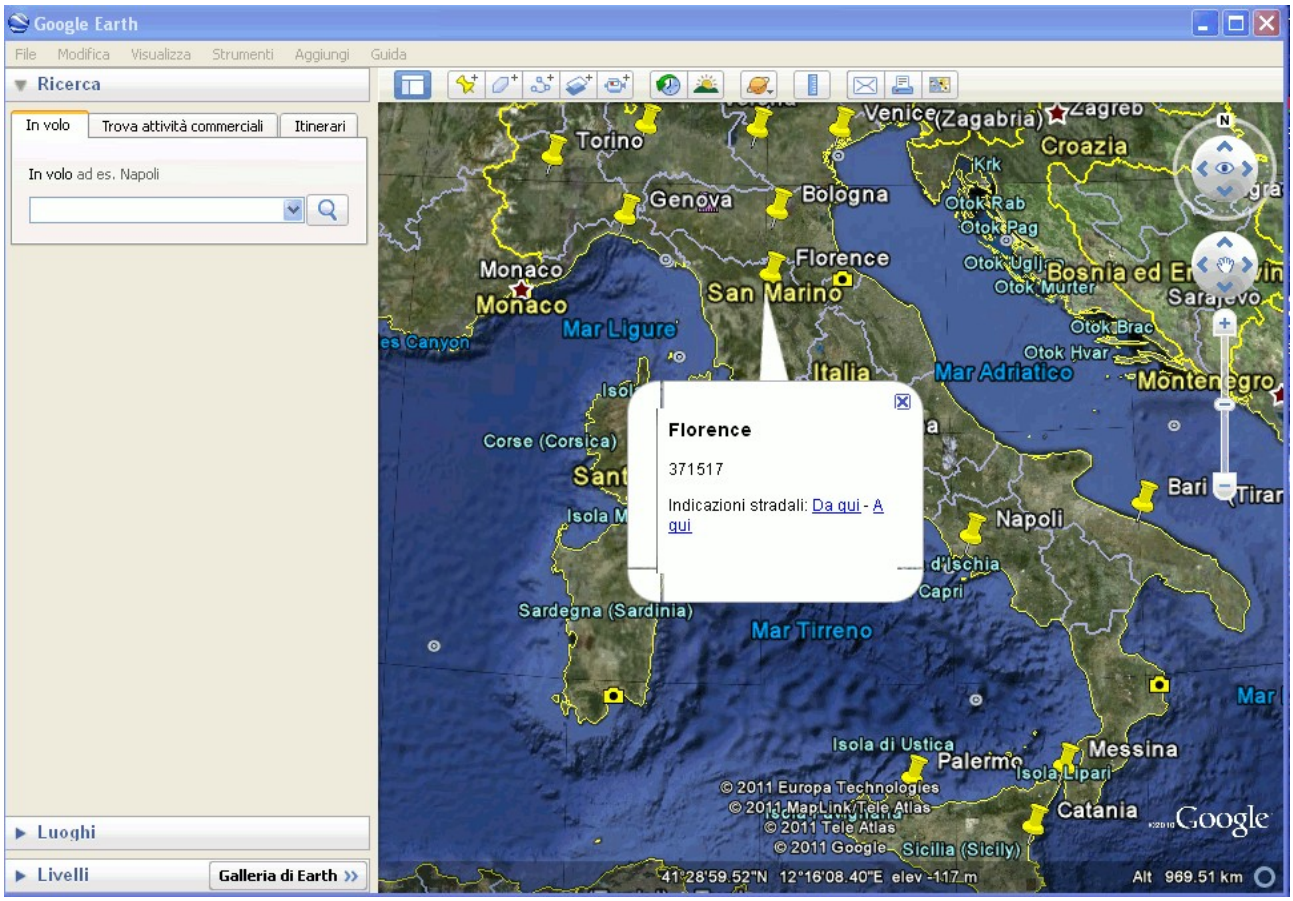
```
CREATE VIEW main_cities AS
SELECT id AS "id", name AS "name",
  population AS "population", geom AS "geom"
FROM cities
WHERE population > 250000;
INSERT INTO views_geometry_columns
(view_name, view_geometry, view_rowid, f_table_name,
f_geometry_column)
VALUES ('main_cities', 'geom', 'ROWID', 'cities', 'geom');
```

Just to make things a little bit difficulties, we'll create first a **main_cities** VIEW (filtering towns > 250,000 peoples). Then we'll properly register this VIEW into **view_geometry_columns**, so to get a real Spatial VIEW.





You must specify two columns: one corresponding to the **<name>** tag, the other corresponding to the **<descricption>** tag: anyway, you can specify a constant string if no such column exist.



Once you've exported the KML file, you can perform a direct check using **Google Earth** (or any other appropriate sw supporting KML).