SpatiaLite – Using SQL Views is really simple and easy
a very short introduction

Lots of people are frightened by terms such as: SQL,TABLE, SELECT, VIEW, JOIN
Following this step-by-step tutorial you'll quickly discover all this isn't at all so complex as you can fear.

Getting started:

You need first to get the latest SpatiaLite SW: then you have to get the **view-demo.sqlite** sample database in order to follow the present tutorial.

Go to the URL: http://www.gaia-gis.it/spatialite-2.4.0/

- download **spatialite-gui** and **spatialite-gis** [the one appropriate for the platform you are actually using] from the **precompiled binaries** section
- download the **view-demo.sqlite** DB from the **sample Dbs** section
- then copy these files into your local filesystem

Understanding the sample DB layout:

You can now launch the **spatialite-gui** and start exploring the DB layout. You'll easily discover it contains the following relevant tables:

- the **Regions** table depicts the Italy's topmost administrative level [*Regioni*]
- the **Counties** table depicts the Italy's intermediate administrative level [*Province*]
- and the **LocalCouncils** depicts the Italy's lowermost administrative level [*Comuni*]

All this one represents a classic hierarchic structure:

- each LocalCouncil belongs to some County
- and each County belongs to some Region
- so, implicitly, each LocalCouncil belongs to some Region [*via* its own County]

Let now examine in some detail the layout of each table.

Regions:

```
CREATE TABLE Regions (
RegId INTEGER PRIMARY KEY NOT NULL,
RegName TEXT NOT NULL,
RegGeom MULTIPOLYGON
)
```

- **RegId** is an unique identifier acting as Primary Key for this table
- **RegName** contains the Region's name
- **RegGeom** contains the Region boundary [as a MultiPolygon Geometry]

Under SQL rules, a **PRIMARY KEY** is an unique identifier used to avoid ambiguity.
Each table row has its own unique id, and this allows to reference individual rows in an completely unambiguous way.

Counties:

```
CREATE TABLE Counties (
CntyId INTEGER PRIMARY KEY NOT NULL,
CntyName TEXT NOT NULL,
PlateCode TEXT NOT NULL,
RegId INTEGER NOT NULL,
CntyGeom MULTIPOLYGON,
CONSTRAINT fk_cnty_reg FOREIGN KEY (RegId)
  REFERENCES Regions (RegId)
)
```

- **CntyId** is an unique identifier acting as Primary Key for this table
- **CntyName** contains the County's name
- **PlateCode** contains a two-chars code [used for car plates, under the Italian law]
- **RegId** contains the unique id identifying the Region to which a County belongs: please note, this column is declared as a Foreign Key.
- **CntyGeom** contains the County boundary [as a MultiPolygon Geometry]

Under SQL rules, a **FOREIGN KEY** [*aka* export key] is used to reference other tables *via* their unique identifier [i.e. a Foreign Key must match exactly some corresponding Primary Key into the referenced table]
Such cross-table correspondence is generally known as a **JOIN**

Usually, the table holding the Primary Key is known as the mother table, and the table holding the Foreign Key is known as the doughter table.
The following relations may exist:

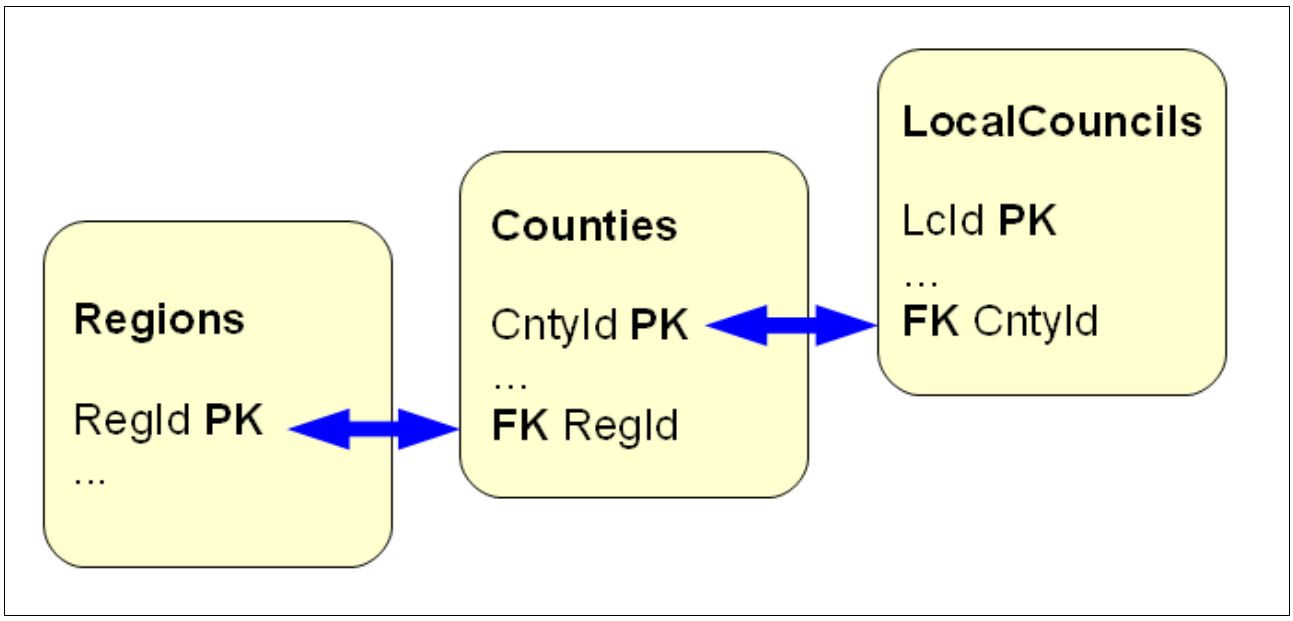
- **one-to-one**: each mother-side row may correspond to a single daughter-side row
- **one-to-many**: each mother-side row may correspond to many daughter-side rows

Quite obviously, the County-Region relation is of the type one-to-many, because a single Region contains several Counties.

LocalCouncils:

```
CREATE TABLE LocalCouncils (
LcId INTEGER PRIMARY KEY NOT NULL,
LcName TEXT NOT NULL,
CntyId INTEGER NOT NULL,
LcGeom MULTIPOLYGON,
CONSTRAINT fk_lc_county FOREIGN KEY (CntyId)
  REFERENCES Counties (CntyId)
)
```

- **LcId** is an unique identifier acting as Primary Key for this table
- **LcName** contains the Local Council's name
- **CntyId** contains the unique id identifying the County to which a Local Council belongs: please note, this column is declared as a Foreign Key.
- **LcGeom** contains the Local Council boundary [as a MultiPolygon Geometry]



This simple diagram may help you to understand the Primary Key / Foreign Key relations we are going to use in the next steps of this tutorial.

Disclaimer: the above materials [*Regions*, *Counties* and *LocalCouncils*] simply represents a slightly modified version of the original data provided by **ISTAT** [the Italy's Statistic Authority]
You can download the original materials directly from: http://www.istat.it/ambiente/cartografia/

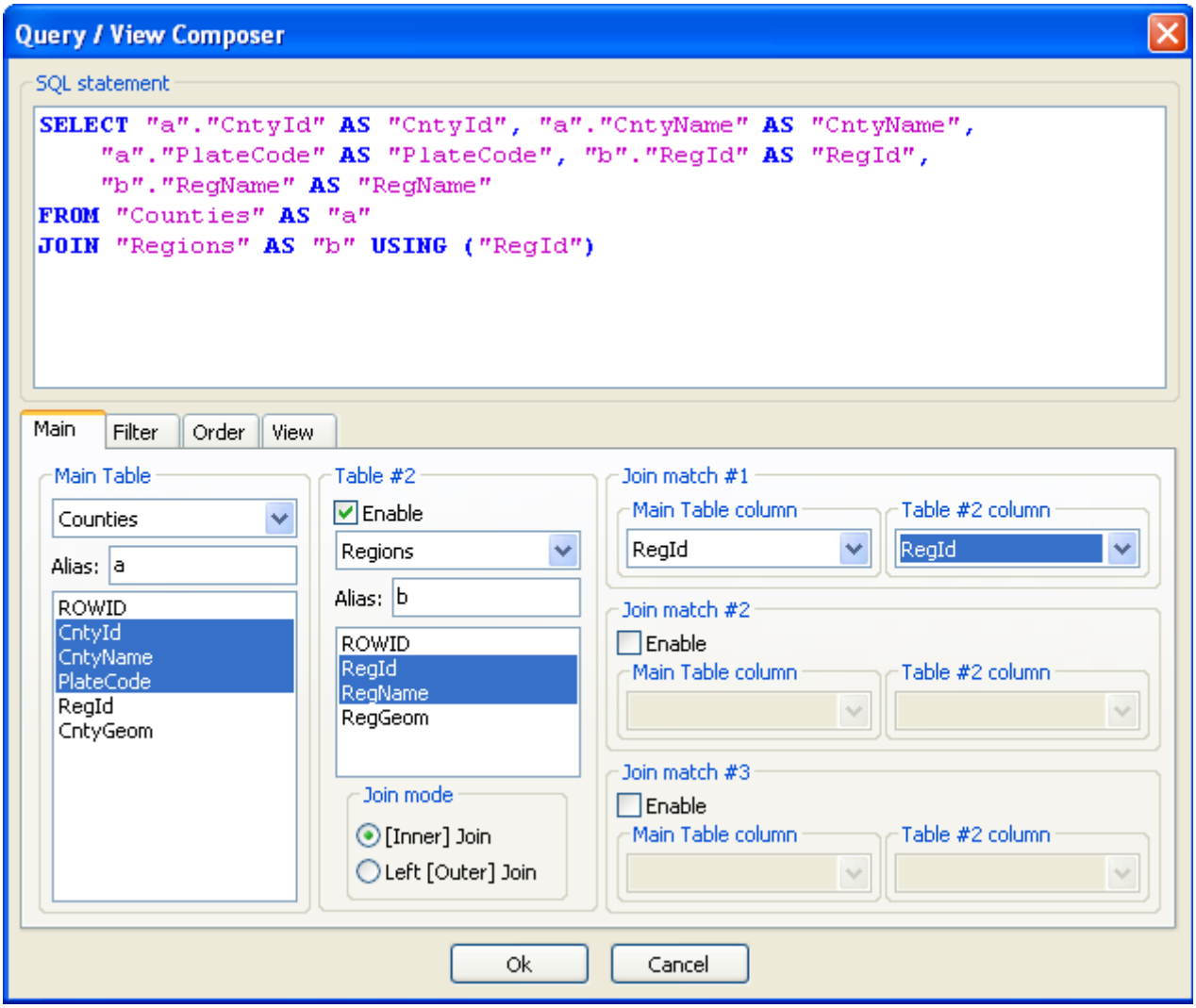
Problem #1:

Suppose you wish to get a table showing the following columns for each County:

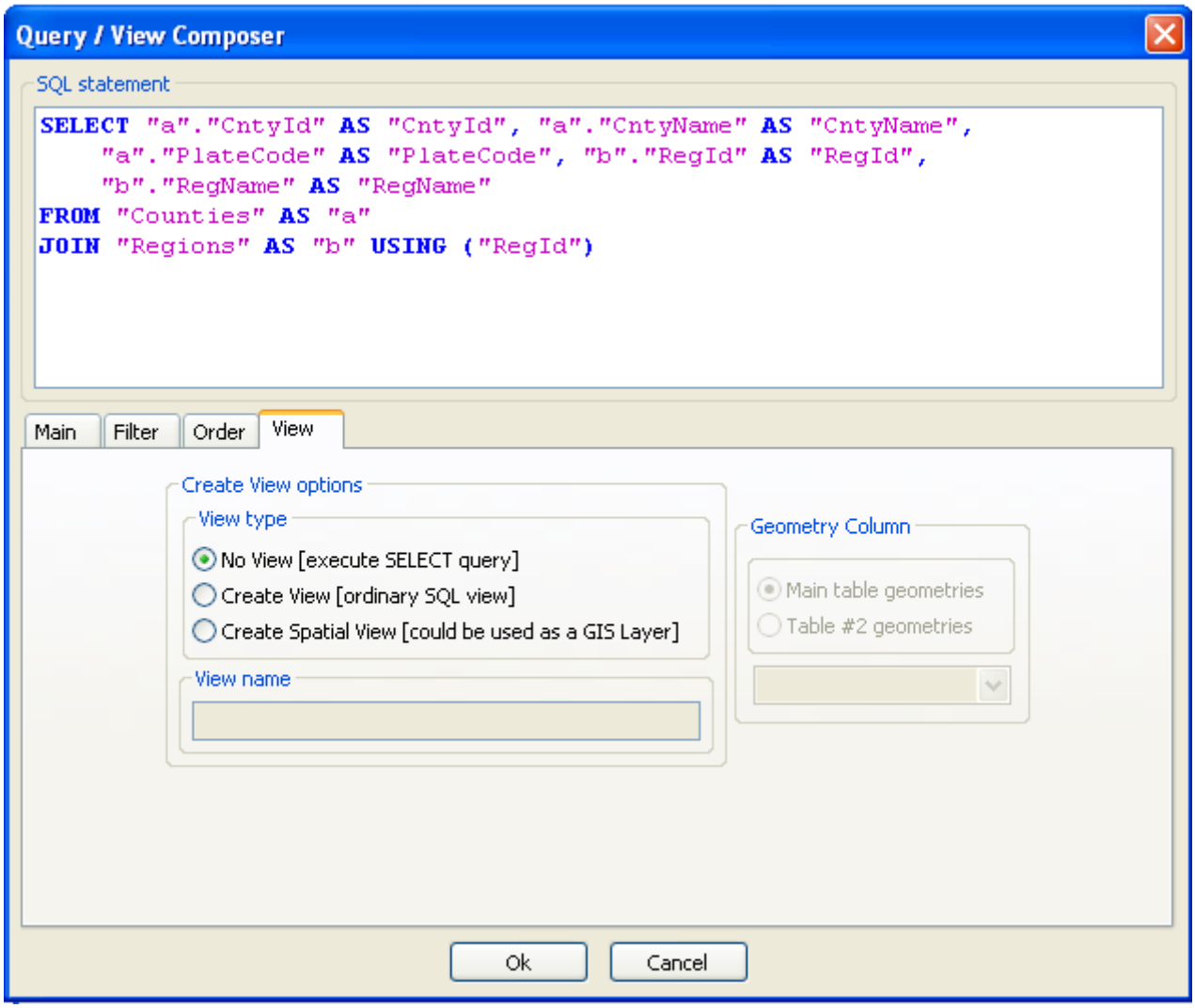
- the County unique Id: *CntyId*
- the County's name: *CntyName*
- the County's car plate code: *PlateCode*
- the Region unique Id: *RegId*
- and the Region's name to which the County belongs: *RegName*

This is a not-so-trivial task, because we are required to perform a relational JOIN operation between the Counties and the Regions tables in order to collect any interesting column.

Step #1.1:

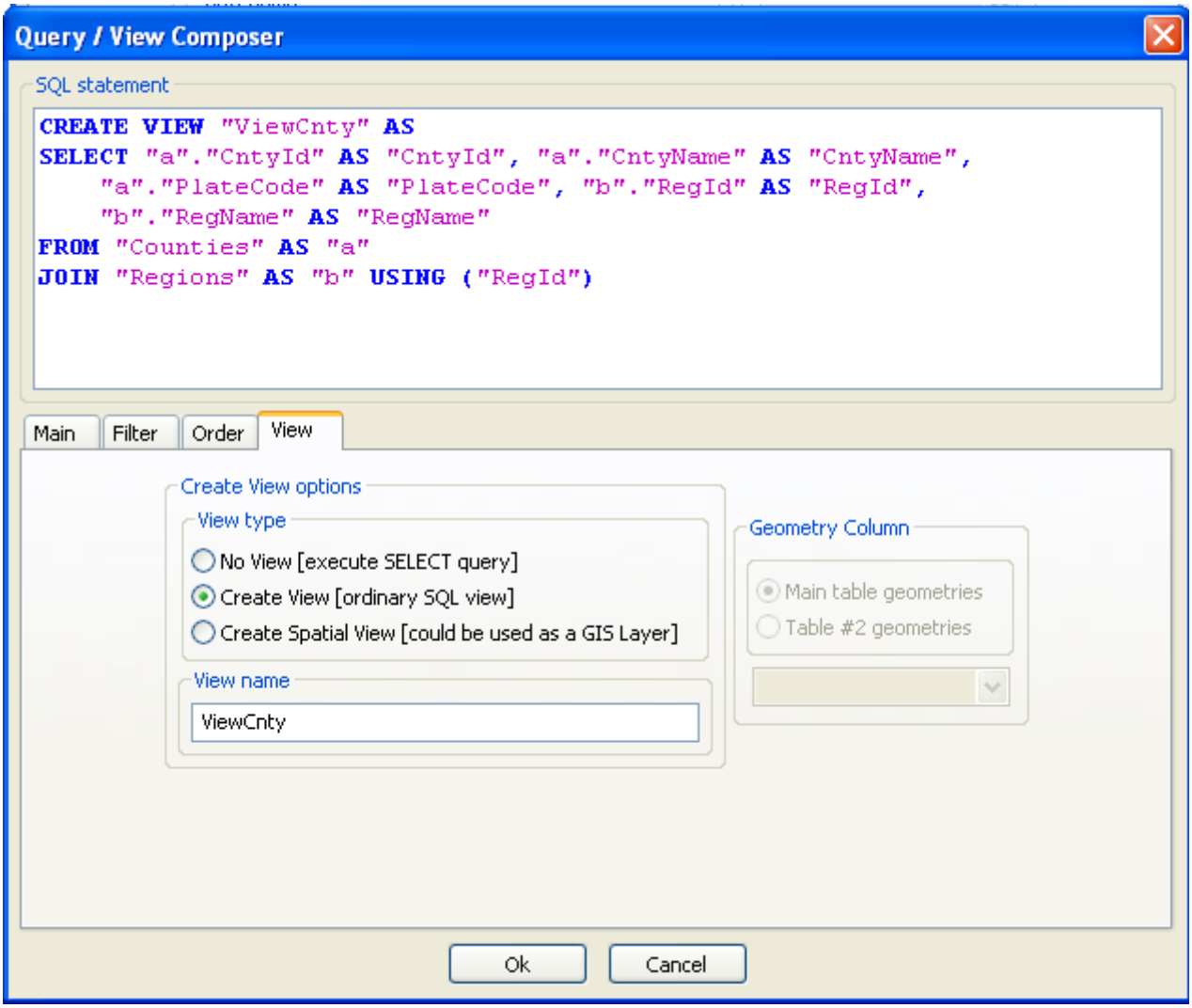


- we'll use the **Query / View Composer** tool implemented in the latest **spatialite-gui**
  - first we'll select the **Counties** table
    - declaring we are interested to get the **CntyId**, **CntyName** and **PlateCode** columns
  - and then we'll select the **Regions** table
    - declaring we intend to get the **RegId** and **RegName** columns
  - we'll require a plain **JOIN** [don't bother ... we'll see later what a Left Join means]
  - and finally we'll select the **RegId** column on both tables to establish the required match cryteria

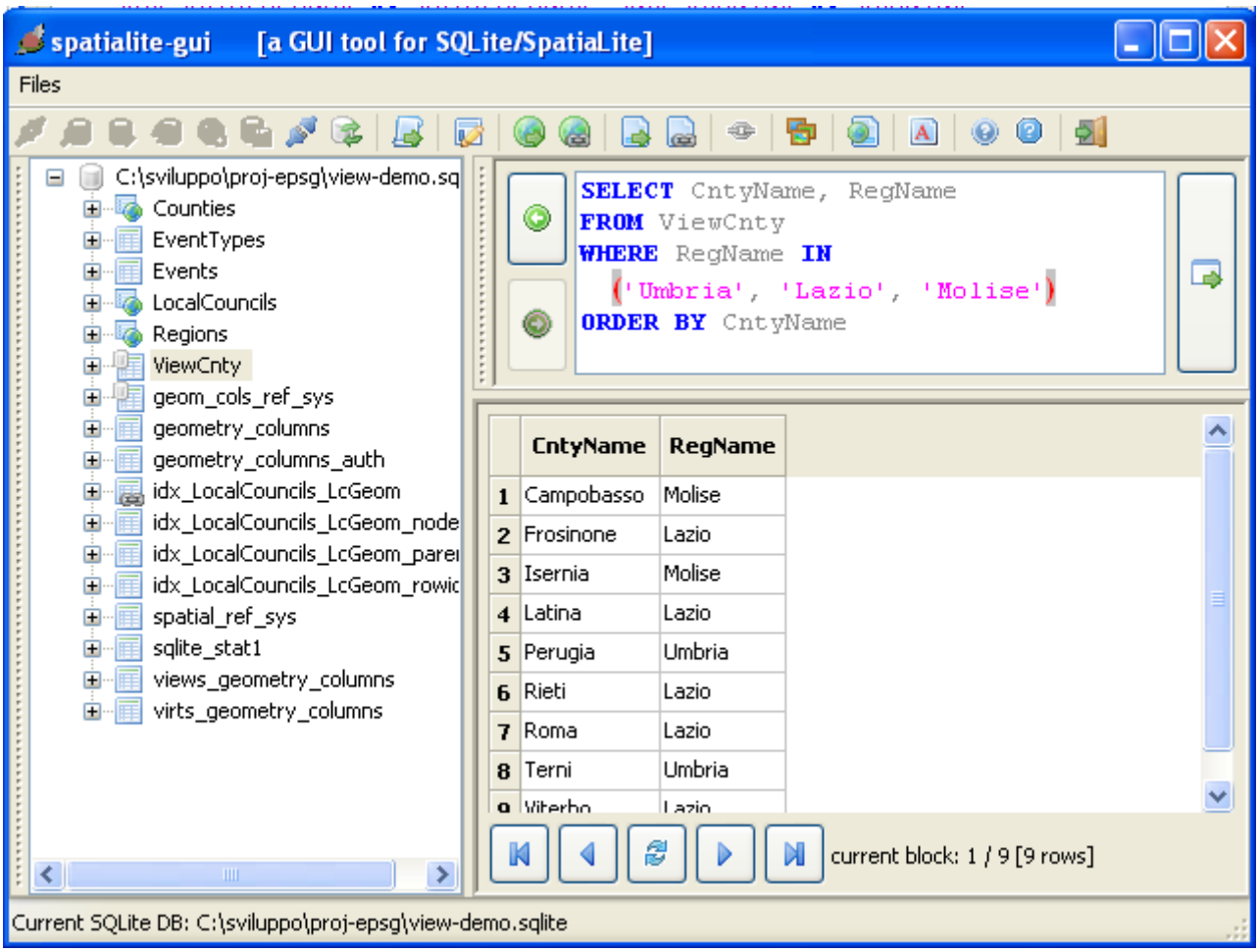


- as you can notice, the Composer tool will write for you the (quite exoteric and unfamiliar) SQL statement, following your directives step by step, as soon as you enter them.
- Once you have completed the SQL statement, you can go to the View tab and require some action to be actually performed
  - please, go slowly: at first, you'll simply require to **execute** the **SELECT query**
  - and then confirm your settings by pressing the **OK** button
  - as you can easily test, this query produces exactly the required result set

Step #1.2:



- repeat the above step: go to the **Main** tab and introduce the same identical settings as above
- but now you'll require [in the **View** tab] to **create** a **View** named **ViewCnty**



- as you can easily check, now the *ViewCnty* view figures as a DB permanent object
- and you can query this view exactly as if it was a plain, ordinary table.
- nevertherless, some important difference exists between tables and views:
  - a view is always a **read-only** object in SQLite, i.e.:
    - you can perform any SELECT op on a view
    - but you are never allowed to perform INSERT, UPDATE or DELETE ops on a view.

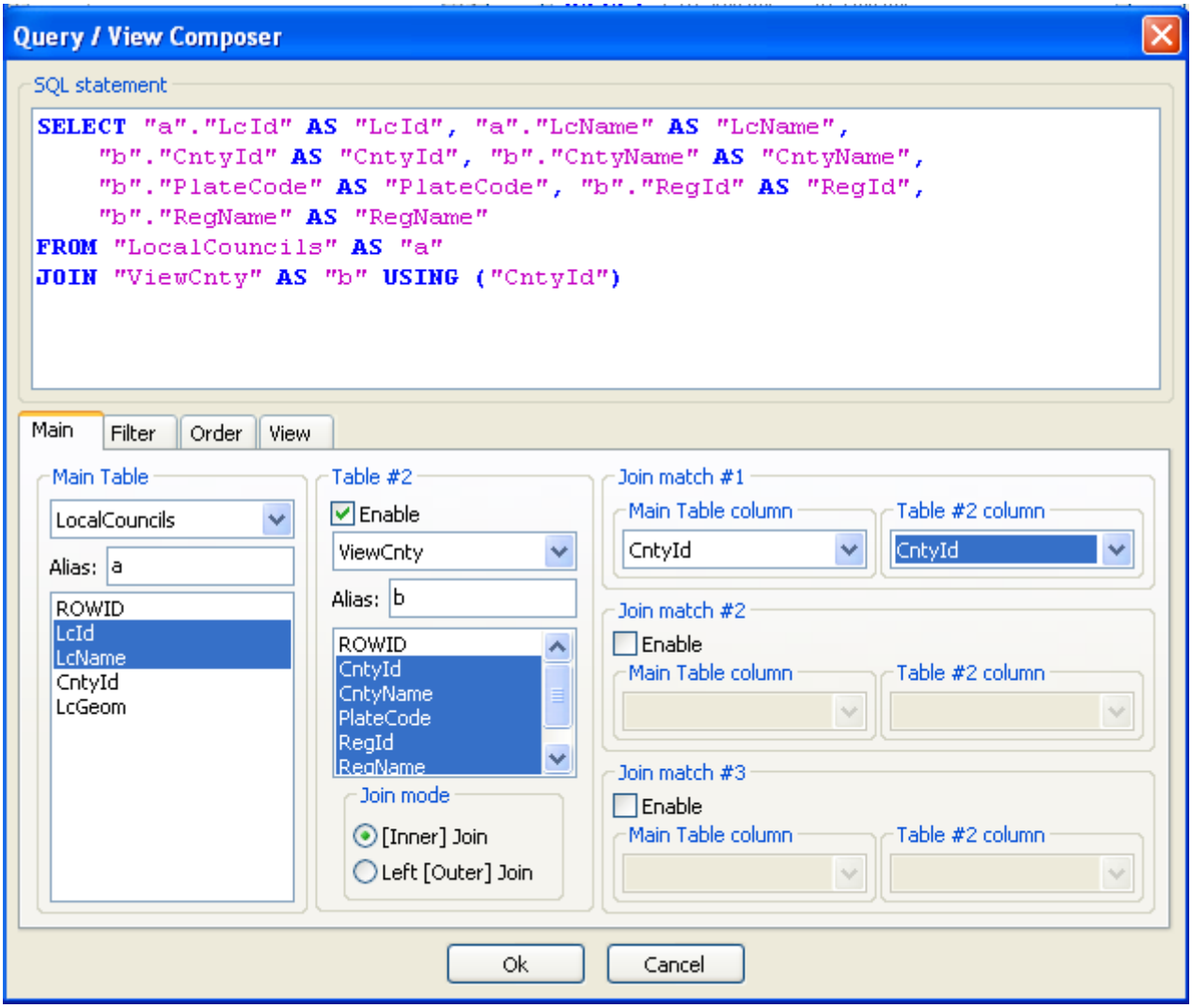
Problem #2:

Suppose you wish now to get a table showing the following columns for each Local Council:

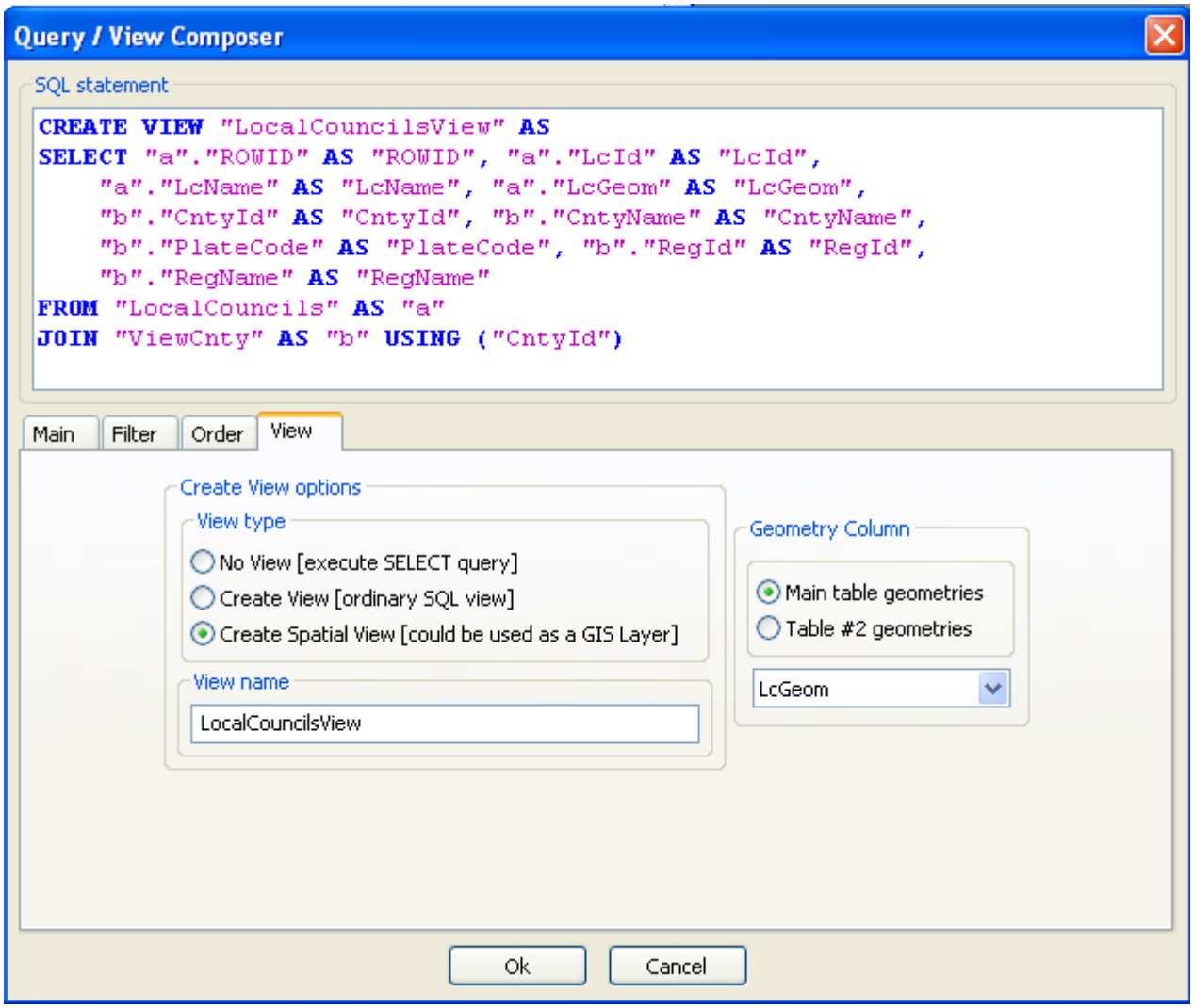
- the Local Council unique Id: *LcId*
- the Local Council's name: *LcName*
- the County unique Id: *CntyId*
- the County's name to which the Local Council belongs: *CntyName*
- the County's car plate code: *PlateCode*
- the Region unique Id: *RegId*
- the Region's name to which the Local Council belongs: *RegName*
- and suppose as well you wish to use all this as a GIS layer: so you have to include as well the *LcGeom* Geometry as well, in order to implement this option.

You have to JOIN three tables now: **LocalCouncils**, **Counties** and **Regions**: and this is a quite complex task.
But you've just defined the **ViewCnty** view: and this one resolves by itself the task of JOINing together the Counties and Regions tables.
So you can simply JOIN the **LocalCouncils** table and the **ViewCnty** view, in order to solve in the simplest and painless way this problem.

Step #2.1:



- we'll use again the **Query / View Composer** tool implemented in the latest **spatialite-gui**
  - first we'll select the **LocalCouncils** table
    - declaring we are interested to get the **LcId** and **LcName** columns
  - and then we'll select the **ViewCnty** table
    - declaring we intend to get the **CntyId**, **CntyName, PlateCode, RegId** and **RegName** columns
  - we'll require a plain **JOIN**
  - and finally we'll select the **CntyId** column on both tables to establish the required match cryteria



- this time you'll require [in the **View** tab] to **create** a **Spatial View** named **LocalCouncilsView**
- a full-featured **Spatial View** doesn't simply represents an ordinary view
- it supports a Geometry column as well, and can be used as a **GIS layer** [obviously, subjected to **read-only** restrictions]
- and that's not all: if you've already defined a **Spatial Index** based on the relevant Geometry column (or, if you'll define one in the after coming), a Spatial View will inherit such Spatial Index.

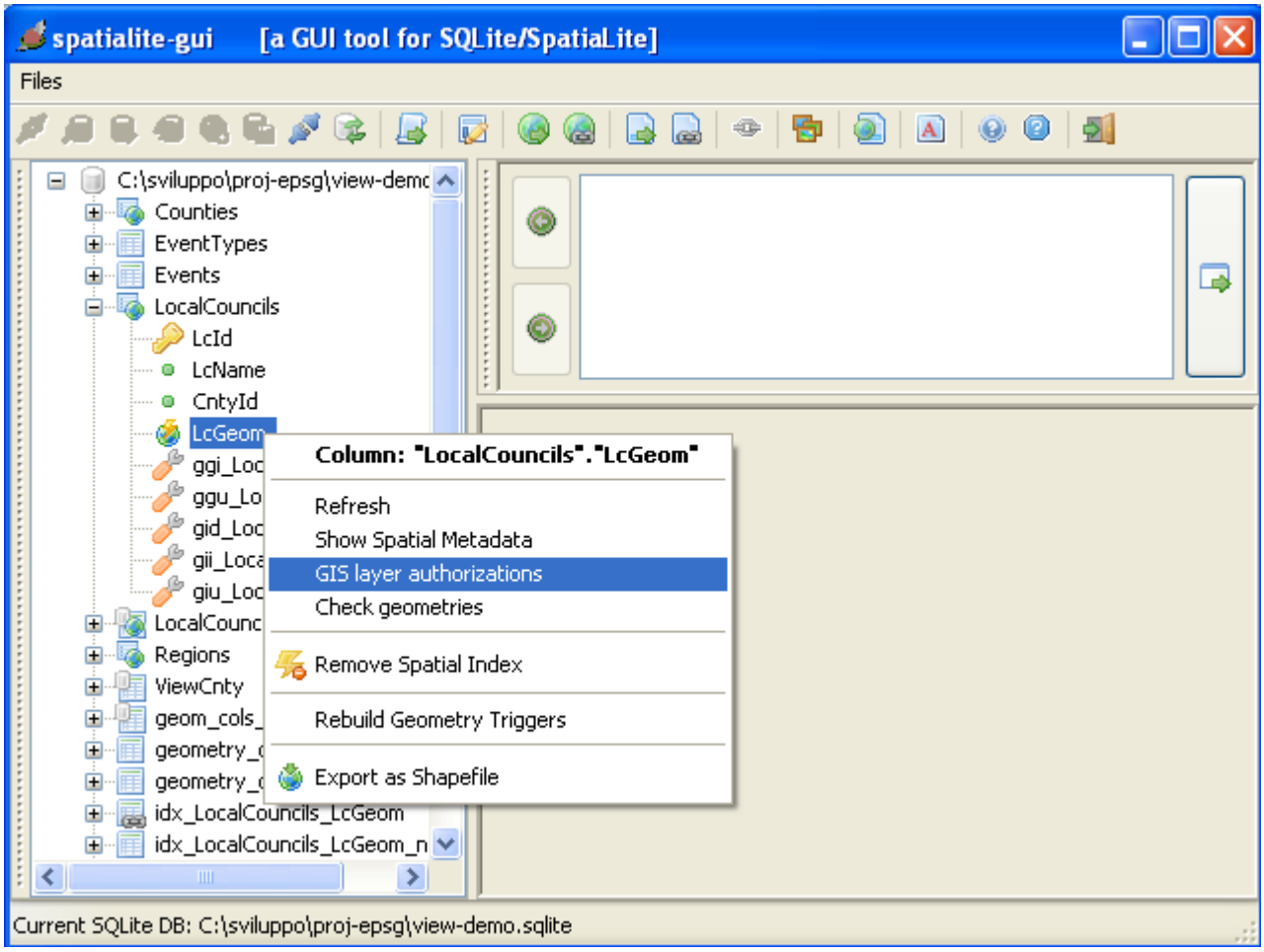
Step #2.2:

Please note: now the current DB contains two GIS layers representing the Local Councils:

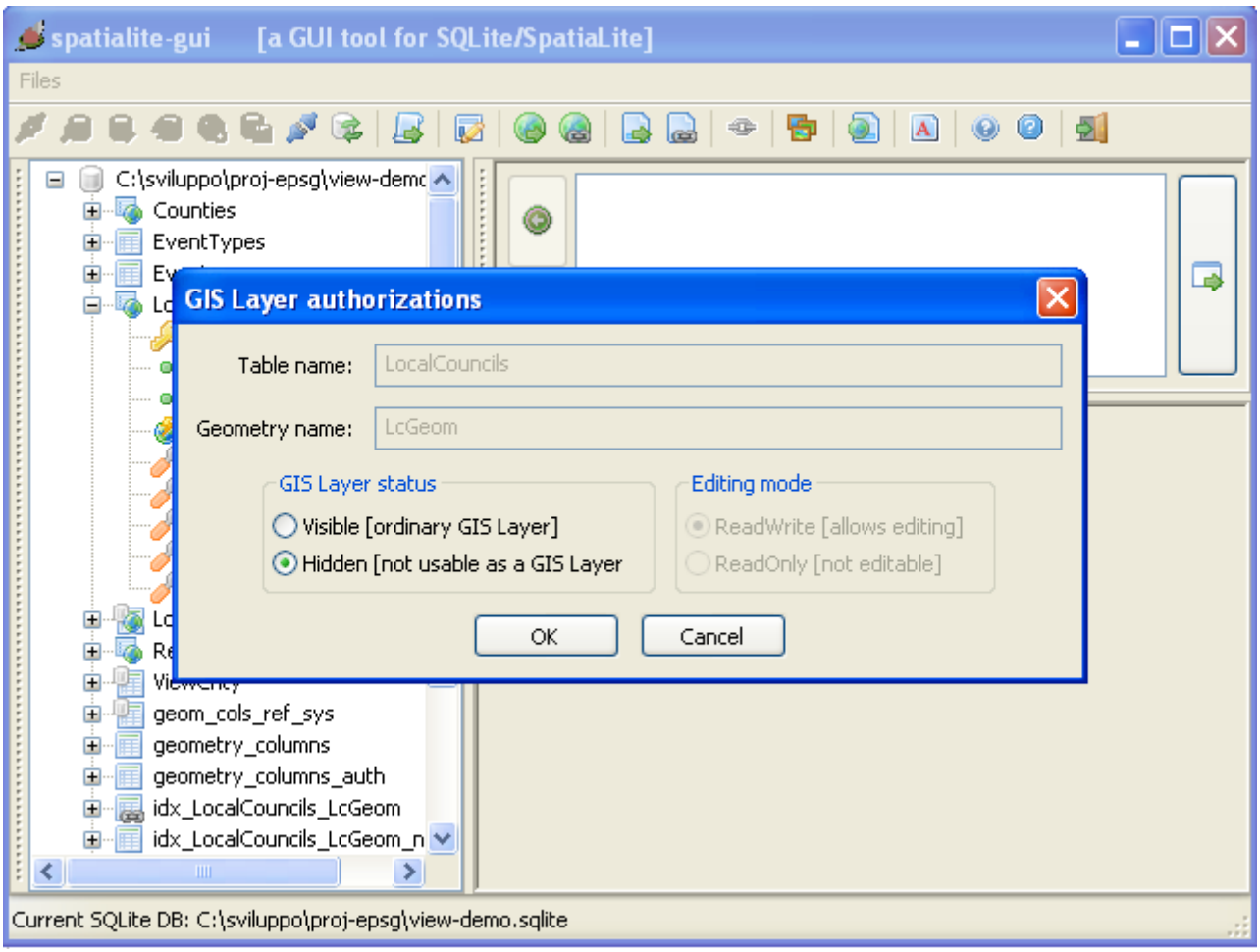
- the first one is the original **LocalCouncils** table
- and the second one is the **LocalCouncilsView** you've already created

It's not a good idea leaving things in such a state: this is because geometries are anyway the sames, but other columns change.
If you immediately try to show the DB Layers using any GIS app, you'll discover such an inconsistency.

But there is no real good reason to support any longer the *LocalCouncils* table as a GIS layer, because the *LocalCouncilsView* is a most rich and useful replacement.
You now have to hide the *LocalCouncils* table, so to avoid that GIS apps will try using it as a Layer.



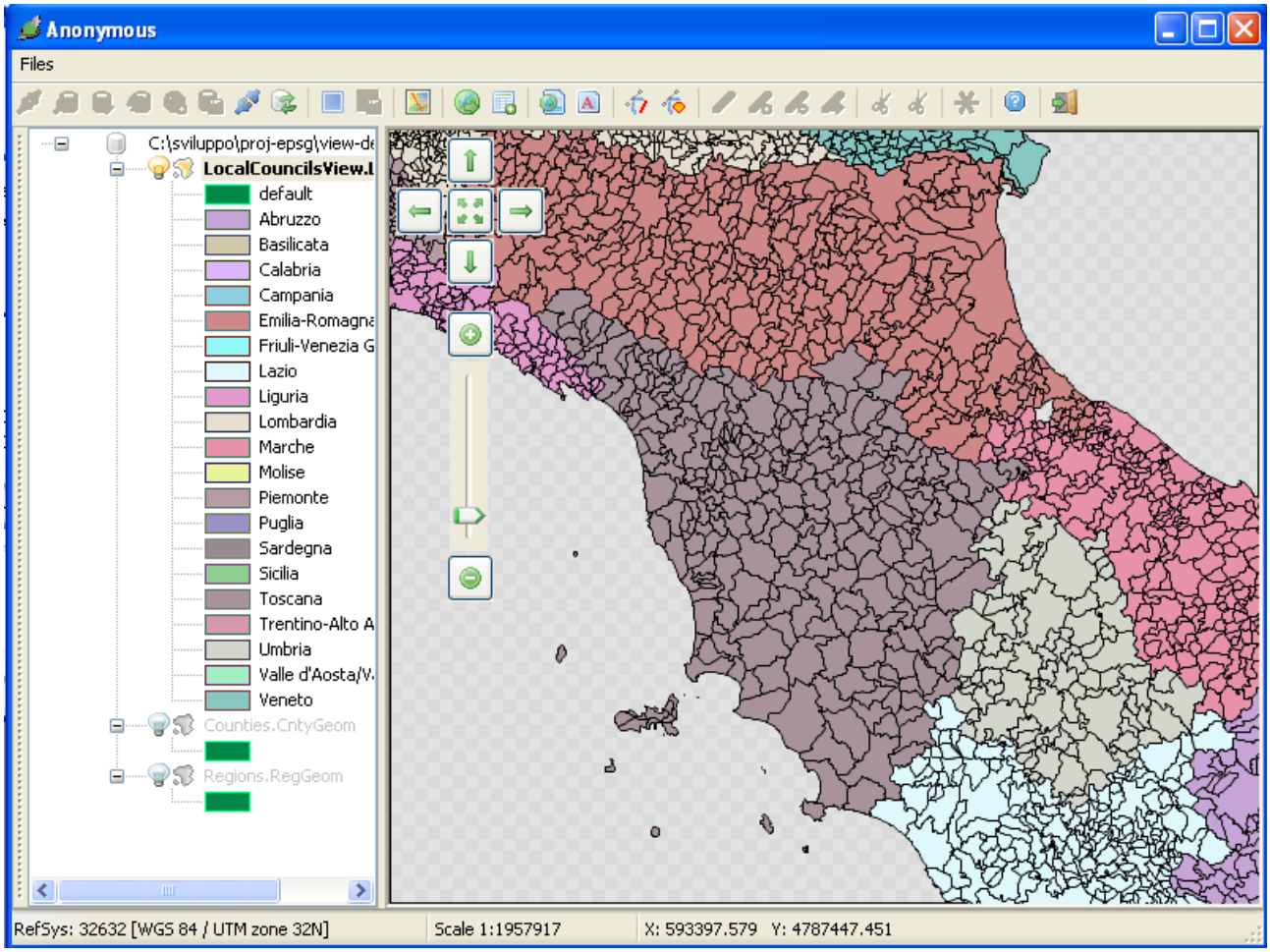
But accomplishing this step is a very simple task: you are simply required to invoke the **GIS layer authorizations** menu for the required table and geometry column.



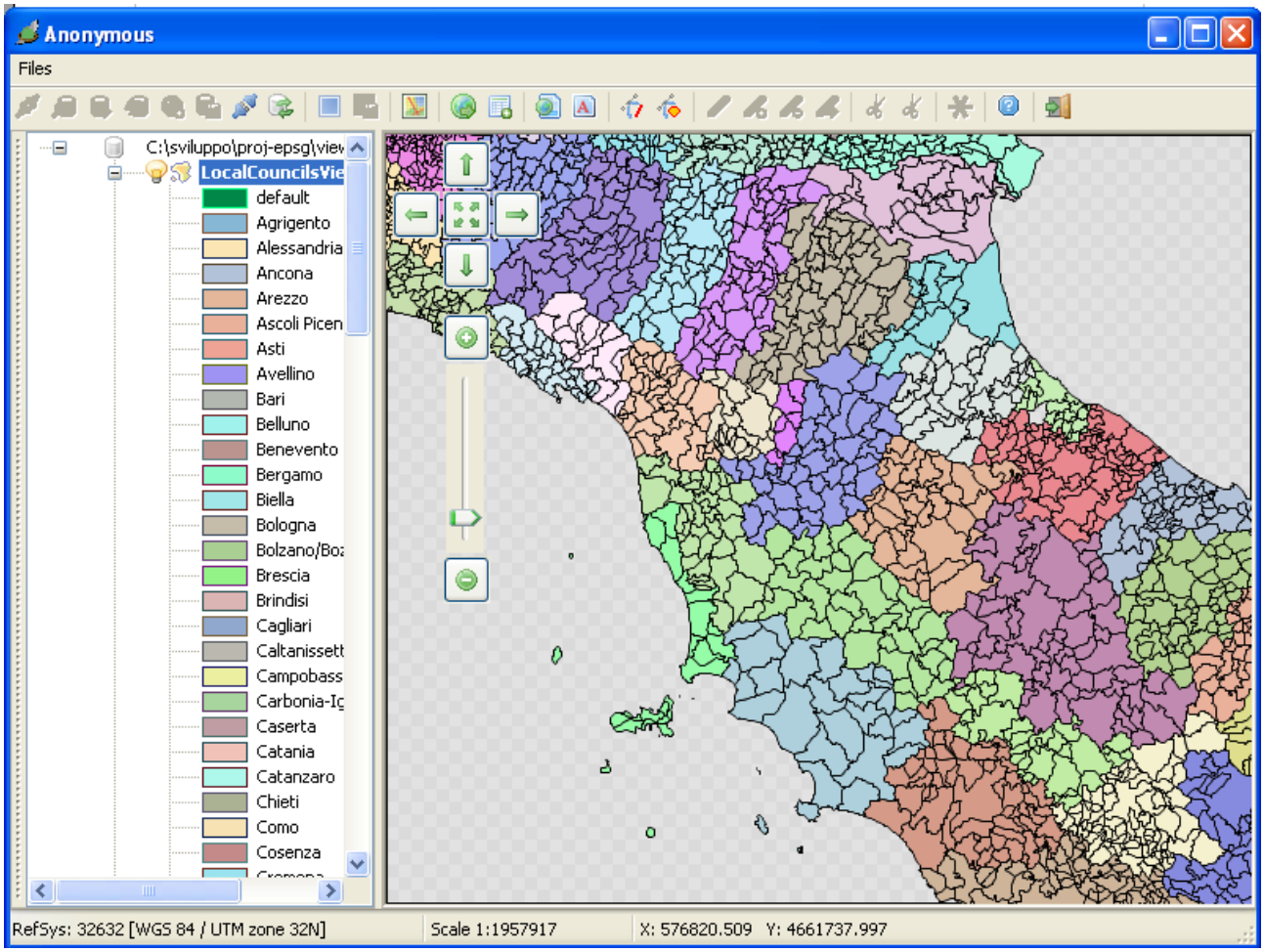
All right, you simply have to check the required option: now the *LocalCouncils* tables isn't any longer assumed to represent a GIS Layer.

Step #2.3:

Now you can perform some visual check using the **spatialite-gis** tool.



As you can see, it really easy to get a thematic representation based upon Regions ...



... or based upon Counties as well.

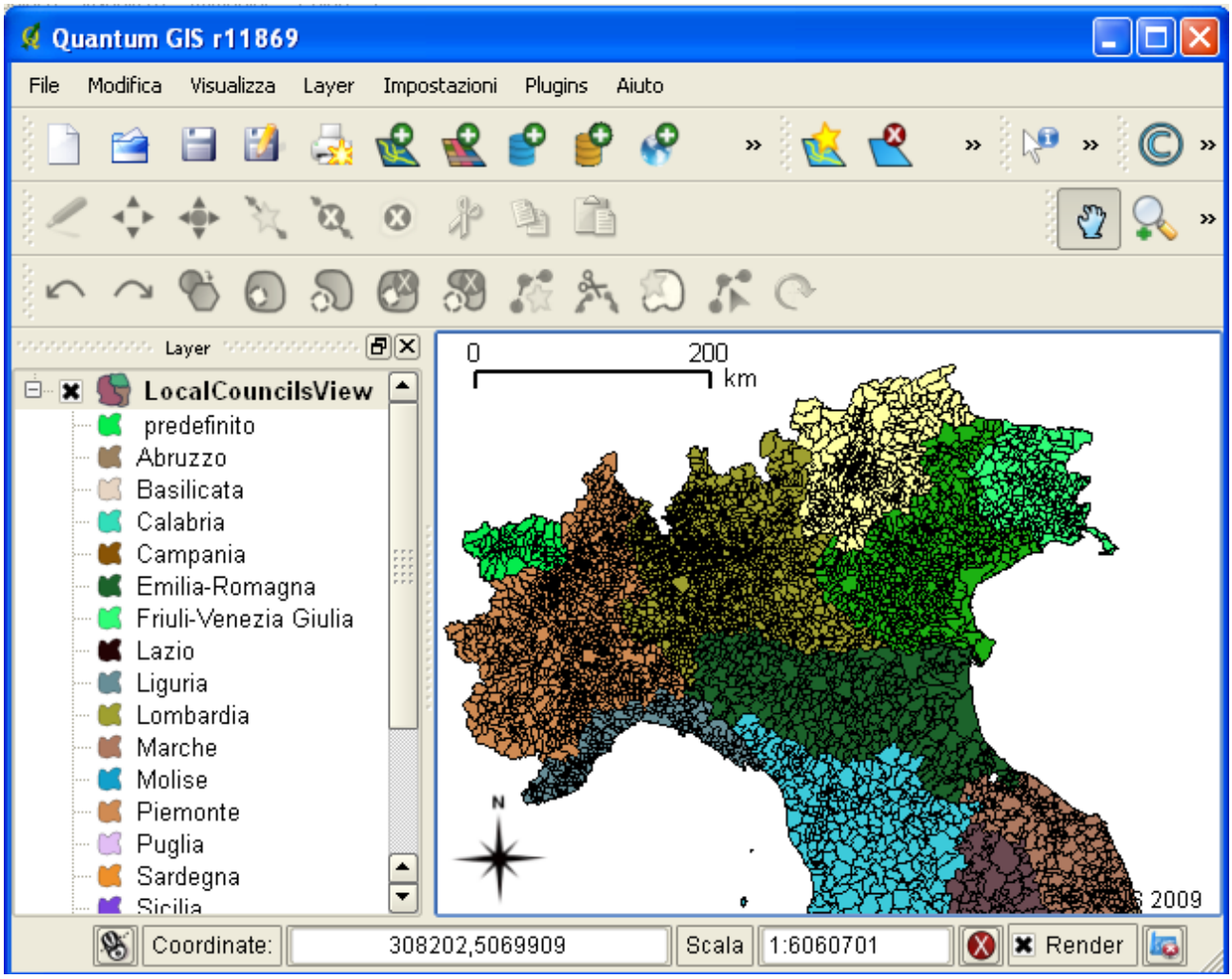
As you can quickly discover by yourself, using a Spatial View (instead of an ordinary Table) doesn't imposes quite any undesired overhead.

Even if you are running the test on a quite slow PC, the graphical rendering will be performed (more or less) in the same time as usual.

This is because the SQLite's data engine is smart enough to perform very fast even when accessing Views.

Caveat: a well designed View will perform in a very brilliant way. But a badly designed View will perform in a very poor way.
So you have to learn more about some useful performance hints. We'll examine this task in the after coming.

Step #2.4:



Please note: you can show a Spatial View using **QGis** as well, because the data provider for spatialite DBs supports exactly the same features as **spatialite-gis** does.

Performance hints

May well be one of your Views will actually perform in a very poor and slow (sluggish) way.
The easiest cause explaining for such an evenience is quite always the same: **you missed to define some required index**

When the SQL engine (query planner) detects an available index, an optimized data access strategy will be actually deployed, and the query will run in a very fast way.

But if some index is missing, then the SQL engine isn't allowed to apply any optimization, and consequently is forced to perform lots of stupid full table scans, thus producing sluggish queries.

Hint: always check each one of your Foreign Keys is properly supported by an opportune Index. Creating an index will require some extra disk space, but will grant a big performance boost.