SpatiaLite – Using SQL Views
advanced features for *power users*

You'll now continue the tutorial exploring some advanced (and most complex) features.

The sample DB **view-demo.sqlite** contains two further tables we have ignored in the above steps:

- **EventTypes**: contains a short list of happenings (Rock music concert, Classic music concert ..)
- **Events**: contains some 100,000+ rows of summer happenings. This one is a quite huge table, and this will impose some extra attention to be paid in order to maintain efficiency.

The layout for each table is as follows.

EventTypes:

```
CREATE TABLE EventTypes (
EvtId INTEGER PRIMARY KEY NOT NULL,
EvtType TEXT NOT NULL
)
```

- **EvtId** is an unique identifier acting as Primary Key for this table
- **EvtType** contains the event's category description

Events:

```
CREATE TABLE Events (
LcId INTEGER NOT NULL,
Year INTEGER NOT NULL,
Month INTEGER NOT NULL,
Day INTEGER NOT NULL,
EvtId INTEGER NOT NULL,
CONSTRAINT pk_evt PRIMARY KEY (LcId, Year, Month, Day),
CONSTRAINT fk_evt_lc FOREIGN KEY (LcId)
  REFERENCES LocalCouncils (LcId),
CONSTRAINT fk_evt_type FOREIGN KEY (EvtId)
  REFERENCES EventTypes (EvtId)
)

CREATE INDEX idx_events_date (Year, Month, Day)

CREATE INDEX idx_events_type (EvtId)
```

- **LcId** is the unique identifier referencing the Local Council who is organizing the happening.
- **Year**, **Month** and **Day** all together contain the happening's date.
- **EvtId** is the unique identifier referencing the Event Type.

> - this table has a peculiar Primary Key: this is based on four columns (*LcId*, *Year*, *Month* and *Day*). There is nothing wrong in defining a multi-column Primary Key. This is a plain, standard SQL feature. [please note: *we are arbitraily assuming each Local Council will host a single happening for each single day*]

- a first Foreign Key references the LocalCouncils table via the *LcId* column
- and a second Foreign Key references the EventTypes table via the *EvtId* column
- there is nothing wrong in defining more then one Foreign Key for the same table: this too is a plain, standard SQL feature
- please note: the *LcId* appears on both the Primary Key and as a Foreign Key: but this too isn't at all a wrong operation.
- this one is an huge table: so we've defined a couple of indexes in order to support efficient queries

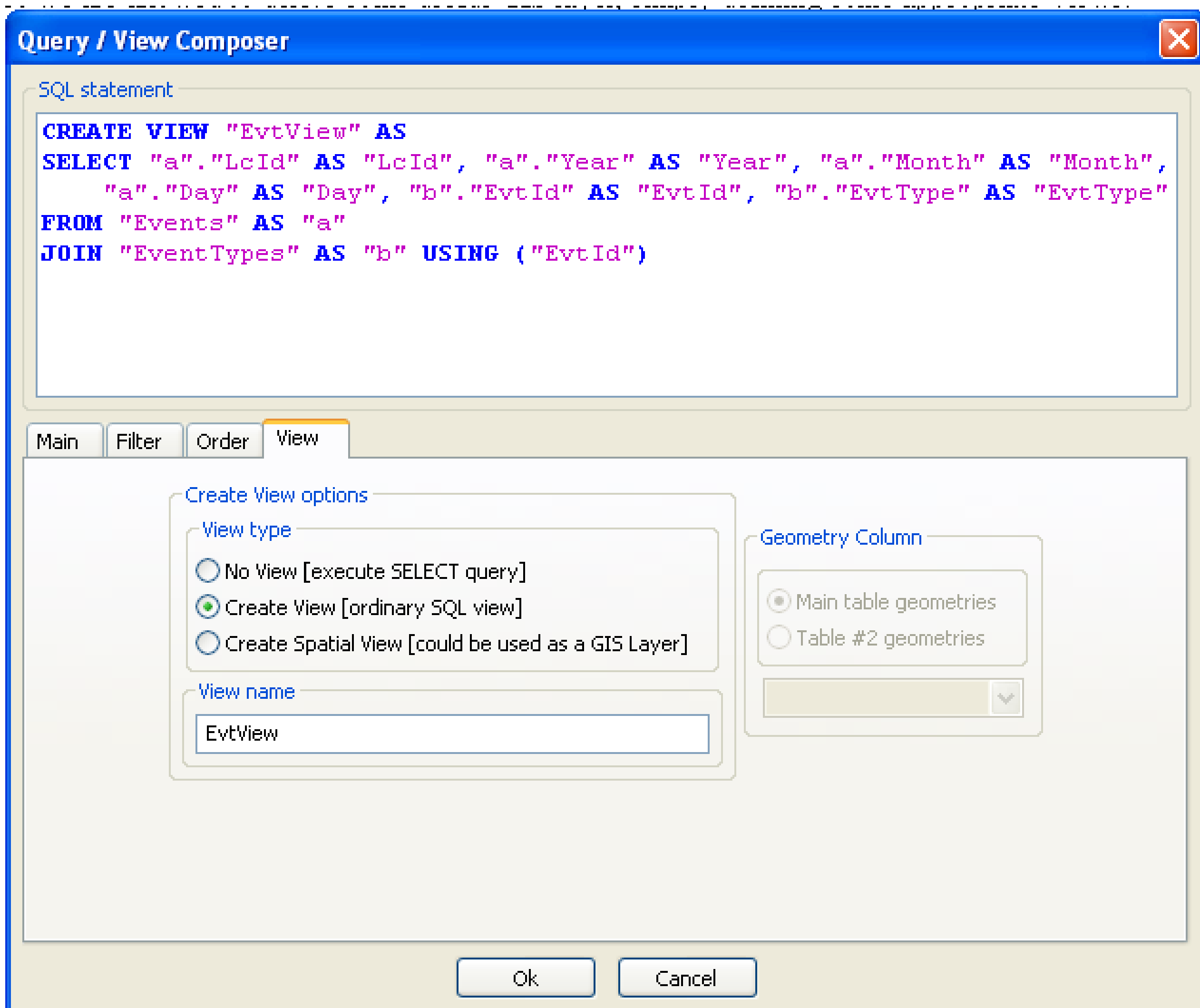
None of these tables has a Geometry on its own: but a reference to the *LocalCouncils* table exists, so we are allowed to derive some useful GIS layer, simply defining some appropriate Views.

Step #1:

You'll now create a first View resolving the JOIN between the *Events* and the *EventTypes* tables.

```
SELECT "a"."LcId" AS "LcId", "a"."Year" AS "Year", "a"."Month" AS "Month",
    "a"."Day" AS "Day", "b"."EvtId" AS "EvtId", "b"."EvtType" AS "EvtType"
FROM "Events" AS "a"
JOIN "EventTypes" AS "b" USING ("EvtId")
```

- nothing new in doing this: you simply define a JOIN between the two tables.



- as usual: you'll now **create** a new **View** named **EvtView**

Step #2:

You'll now create a first GIS layer showing any planned happening for the 2009-08-15 date.

Query / View Composer

SQL statement

```
SELECT "a"."LcId" AS "LcId", "a"."LcName" AS "LcName",
    "b"."Year" AS "Year", "b"."Month" AS "Month", "b"."Day" AS "Day",
    "b"."EvtId" AS "EvtId", "b"."EvtType" AS "EvtType"
FROM "LocalCouncils" AS "a"
JOIN "EvtView" AS "b" USING ("LcId")
```

Main | Filter | Order | View

Main Table: LocalCouncils
Alias: a
ROWID
LcId
LcName
CntyId
LcGeom

Table #2: Enable
EvtView
Alias: b
LcId
Year
Month
Day
EvtId
EvtType

Join mode
[Inner] Join
Left [Outer] Join

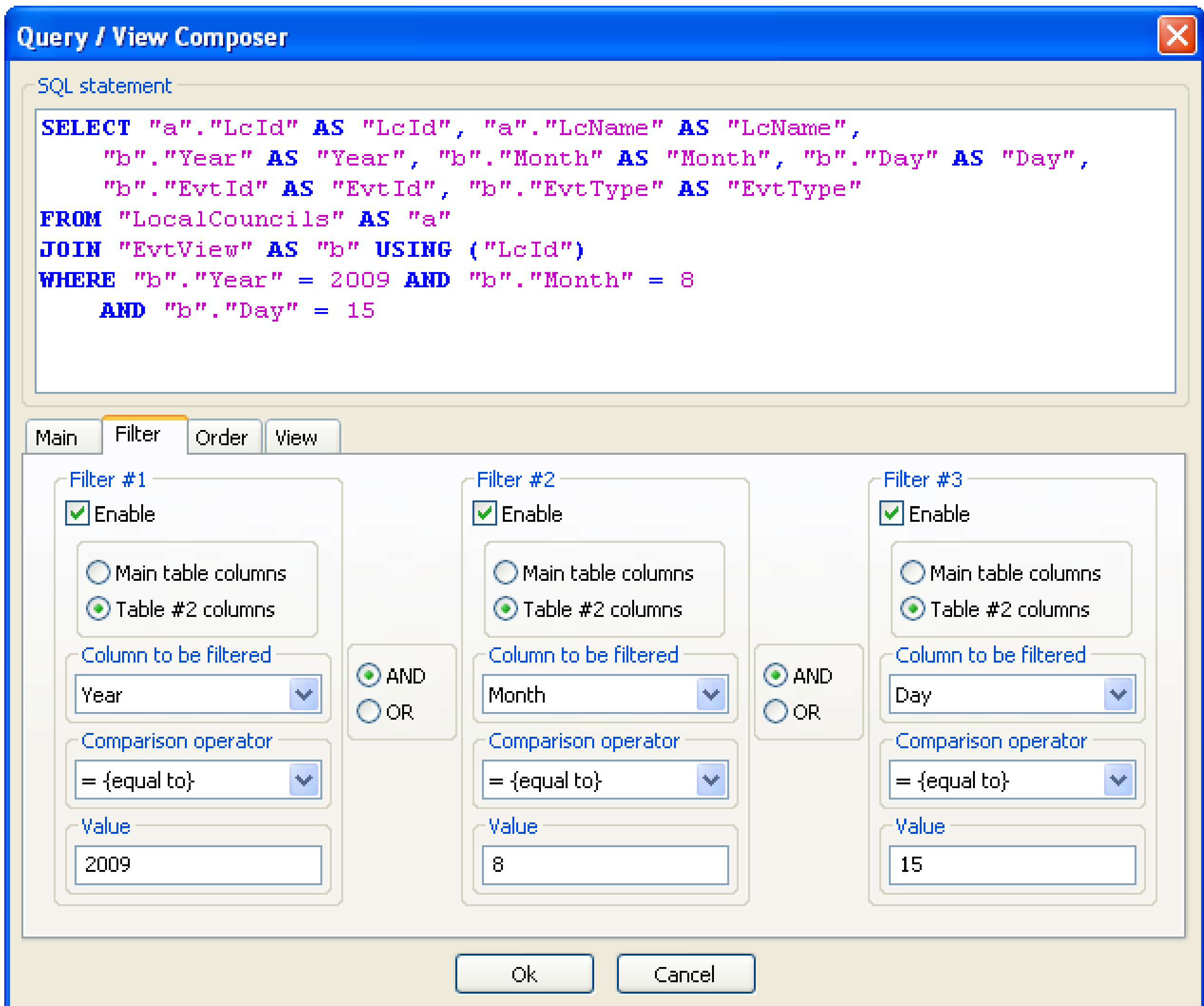
Join match #1
Main Table column: LcId
Table #2 column: LcId

Join match #2
Enable
Main Table column
Table #2 column

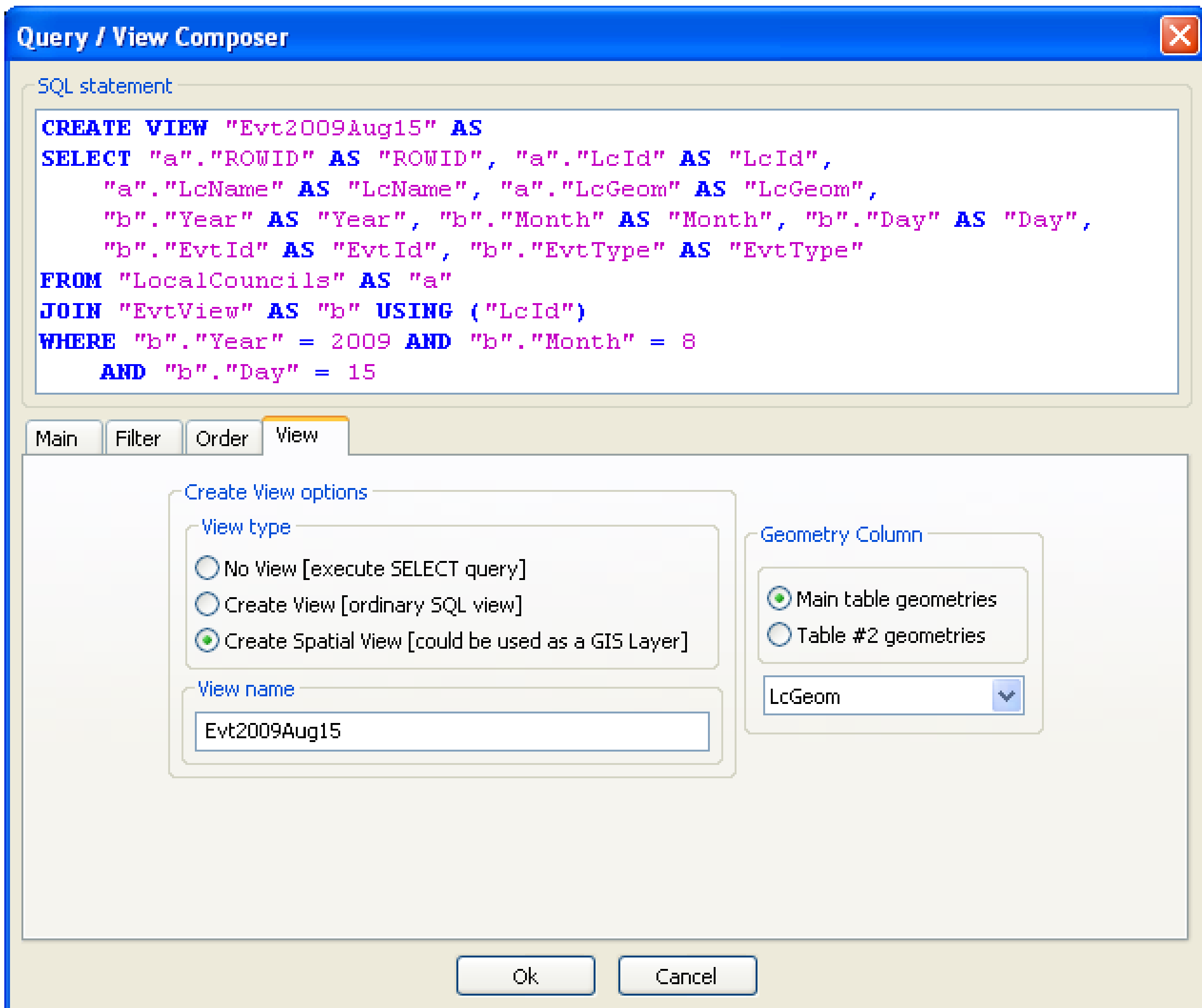
Join match #3
Enable
Main Table column
Table #2 column

Ok | Cancel

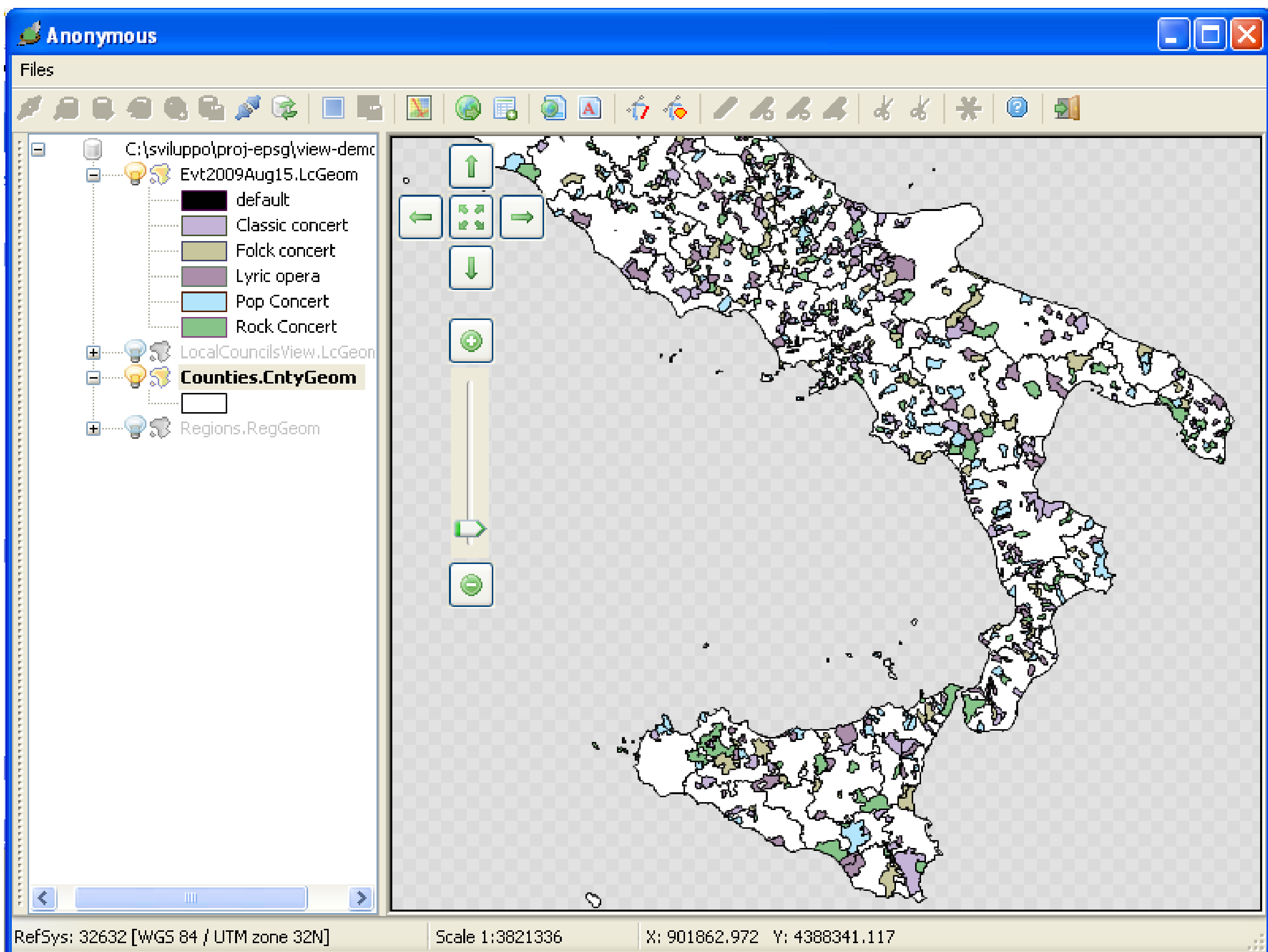
- once again, there is nothing new in doing this



- and finally you'll perform a new task never explained before
- you'll set a filter clause in order to extract only the happenings for the 2009-08-15 date



- now you'll **create** a **Spatial View** named **Evt2009Aug15**
- this one contains the *LocalCouncils* Geometry column, so you can directly use it as a GIS layer



- all right: here is the happening's map [2009-09-15] shown by **spatialite-gis**.

Please note: you are forced to show the Counties layer anyway, in order to get a decent map. Otherwise the map will be shown very badly [try by yourself: make the Counties layer to be invisible, and you'll immediately understand what I'm meaning).

This is easily explained: we used a simple **JOIN** op, so only the Local Councils actually performing some happening on 2009-08-15 will be included into the result set.
Any other Local Council [i.e., the many ones not performing any happening on 2009-08-15 will be simply ignored into the result set.

You can circumvent this issue quite easily: you have simply to use a **LEFT JOIN** op in order to include any *silent* Local Council into the result set as well.

And this requires some extra-care, because the SQLite query engine doesn't handles LEFT JOIN sas fast as you can hope, expecially when huge Geometries are involved.
You risk to get a sluggish query anyway, if you don't plan very carefully your own queries and/or views.

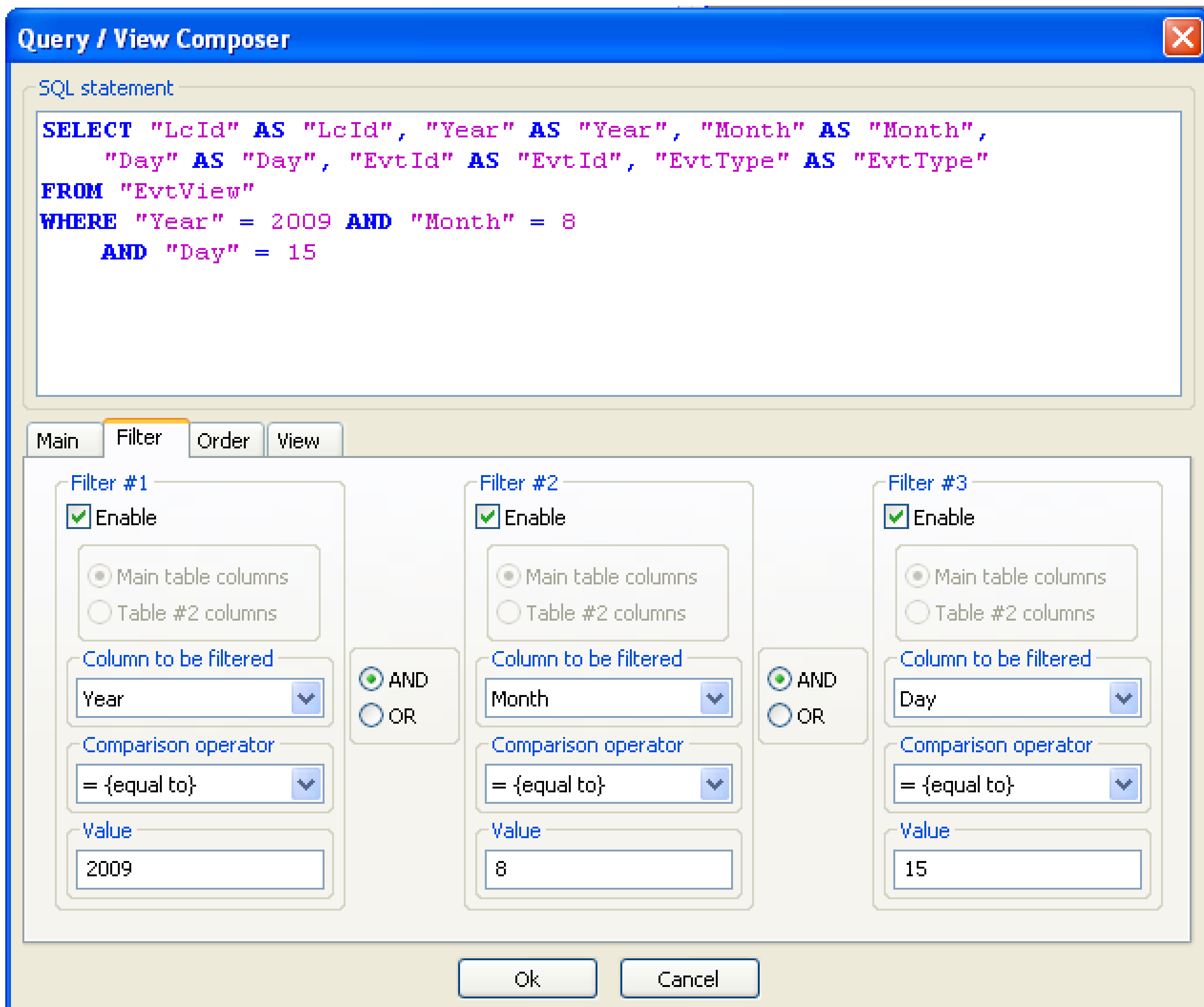
Step #3:

You'll now create a second GIS layer showing any planned happening for the 2009-08-15 date, but this time you'll use a LEFT JOIN in order to include the *silent* Local Councils into the result set as well.

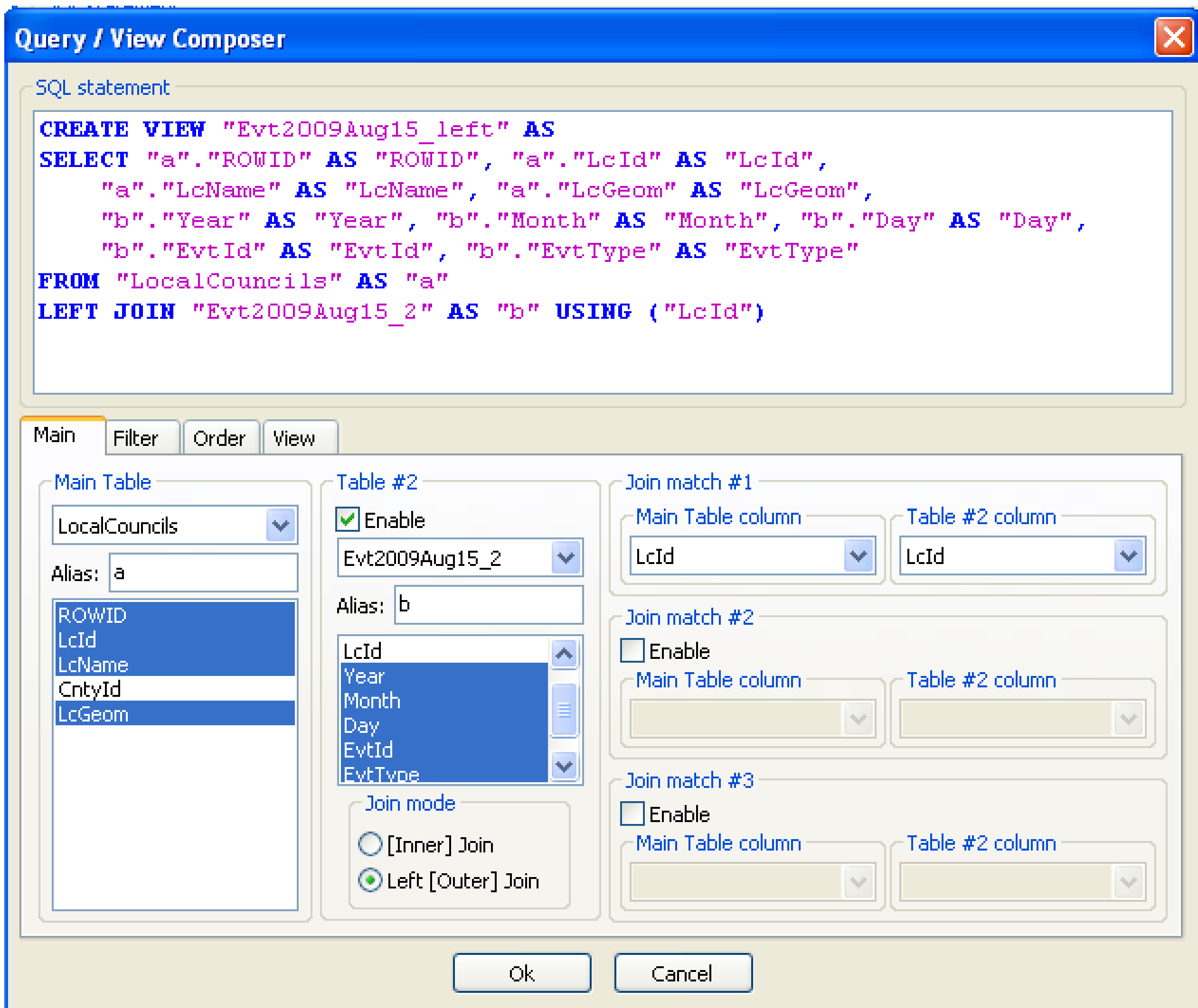
You cannot follow a straight way to get a LEFT JOIN Spatial View.

```
SELECT "LcId" AS "LcId", "Year" AS "Year", "Month" AS "Month",
    "Day" AS "Day", "EvtId" AS "EvtId", "EvtType" AS "EvtType"
FROM "EvtView"
```

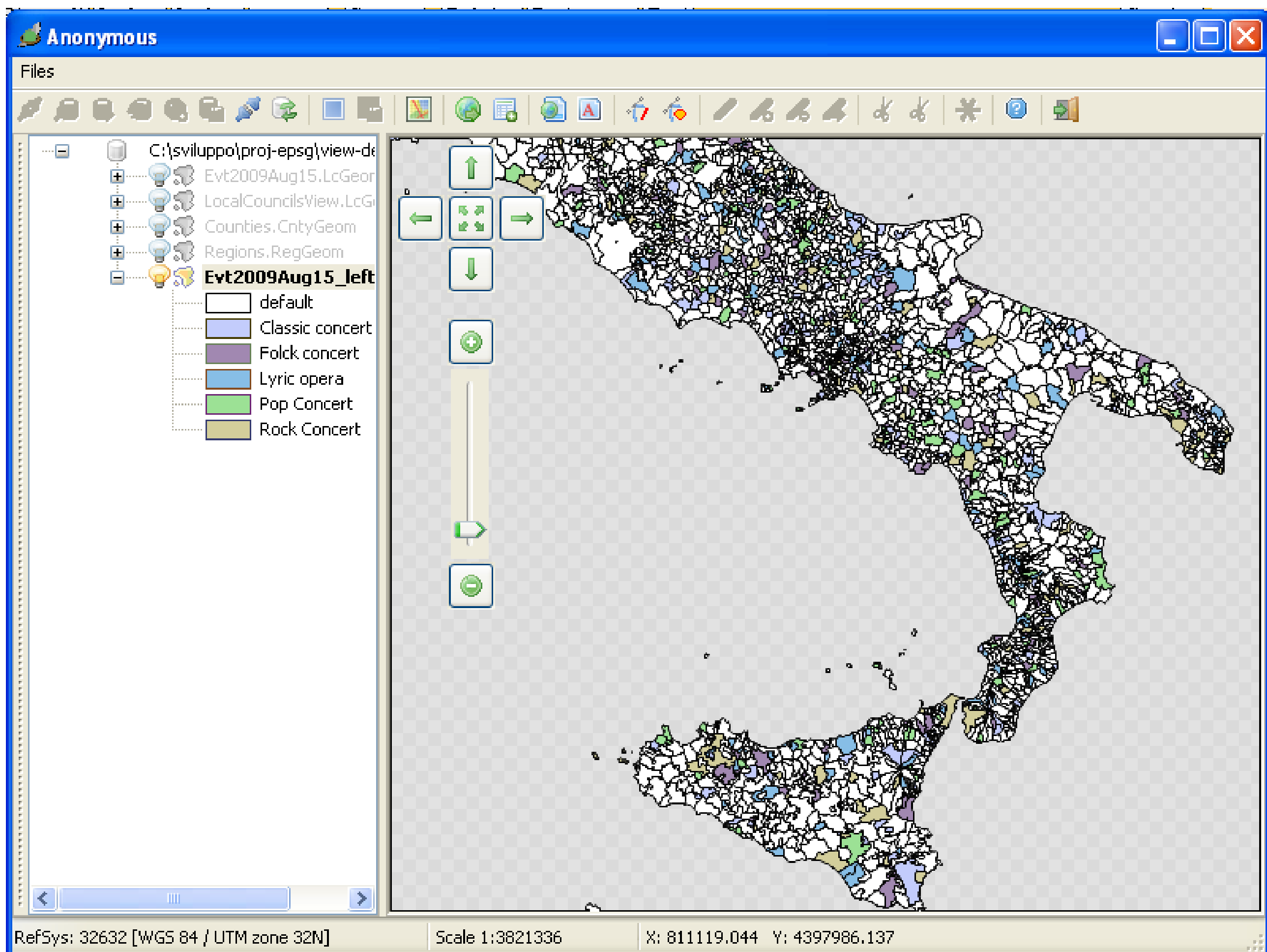
- you'll first define an intermediate View, whose role is simply the one to filter events by date
- so you'll simply select any relevant column from the *EvtView* view
- please note: you are not required at all to always use <u>two</u> tables when defining a View



- then you'll apply the appropriate filter clause in order to get only the 2009-08-15 happenings
- after this, you can create the Evt2009Aug15_2 view



- and finally you can create your LEFT JOINED Spatial View



- all right, this time you LEFT JOINed Spatial View correctly depicts any Local Council

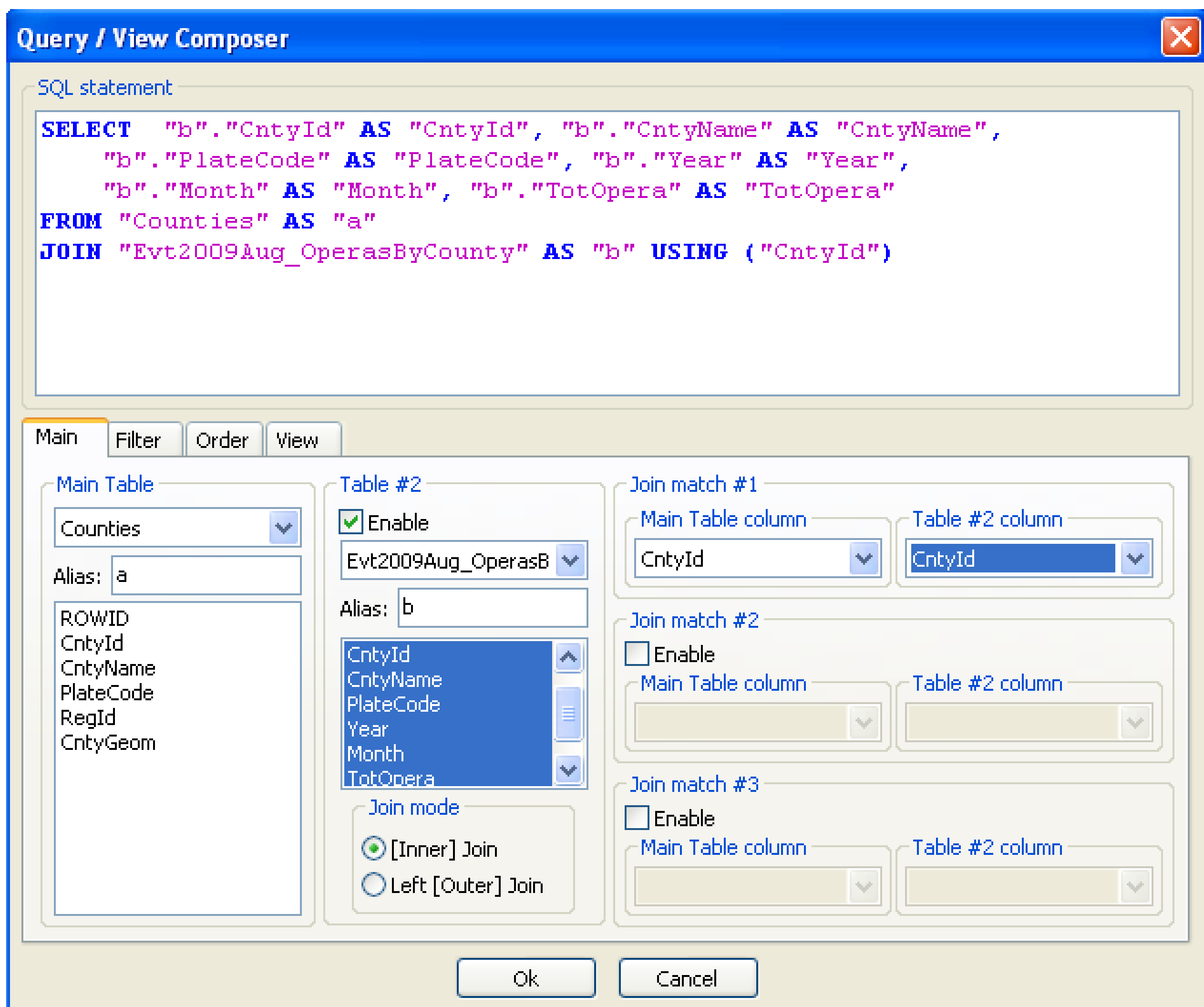
<u>Caveat:</u> using such an indirect approach you can get a (quite slow) but still effettive and usable layer.
Trying to adopt the direct approach (i.e., directly performing a LEFT JOIN and the date filtering all togetether in a single step) will produce an untolerably sluggish layer.
I mean, one you cannot use for any practical purpose.

Step #4:

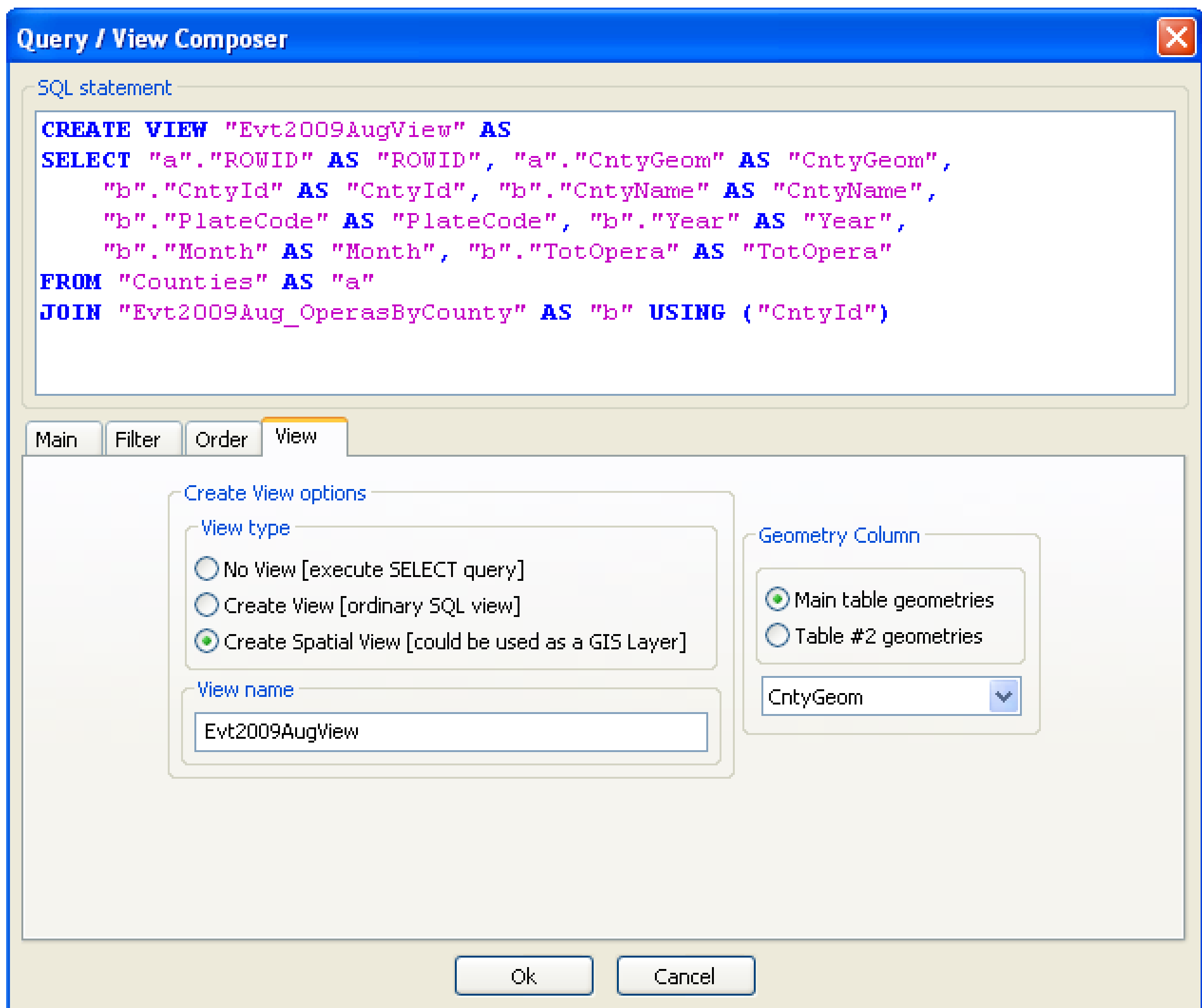
You'll now create a third GIS layer showing any planned happening of the 'Lyric Opera' type in August 2009: in order to make things a little bit more complex, you are required to aggregate the events by County.
In other worlds, you are required to show how many lyric operas have been played in each County during the whole August month.

```
CREATE VIEW Evt2009Aug_OperasByCounty AS
SELECT a.CntyId AS CntyId, a.CntyName AS CntyName,
  a.PlateCode AS PlateCode, b.Year AS Year, b.Month AS Month,
  Count(*) AS TotOpera
FROM Counties AS a, EvtView AS b, LocalCouncils AS c
WHERE c.CntyId = a.CntyId AND b.LcId = c.LcId
  AND b.Year = 2009 AND b.Month = 8 AND b.EvtType = 'Lyric opera'
GROUP BY a.CntyId
```

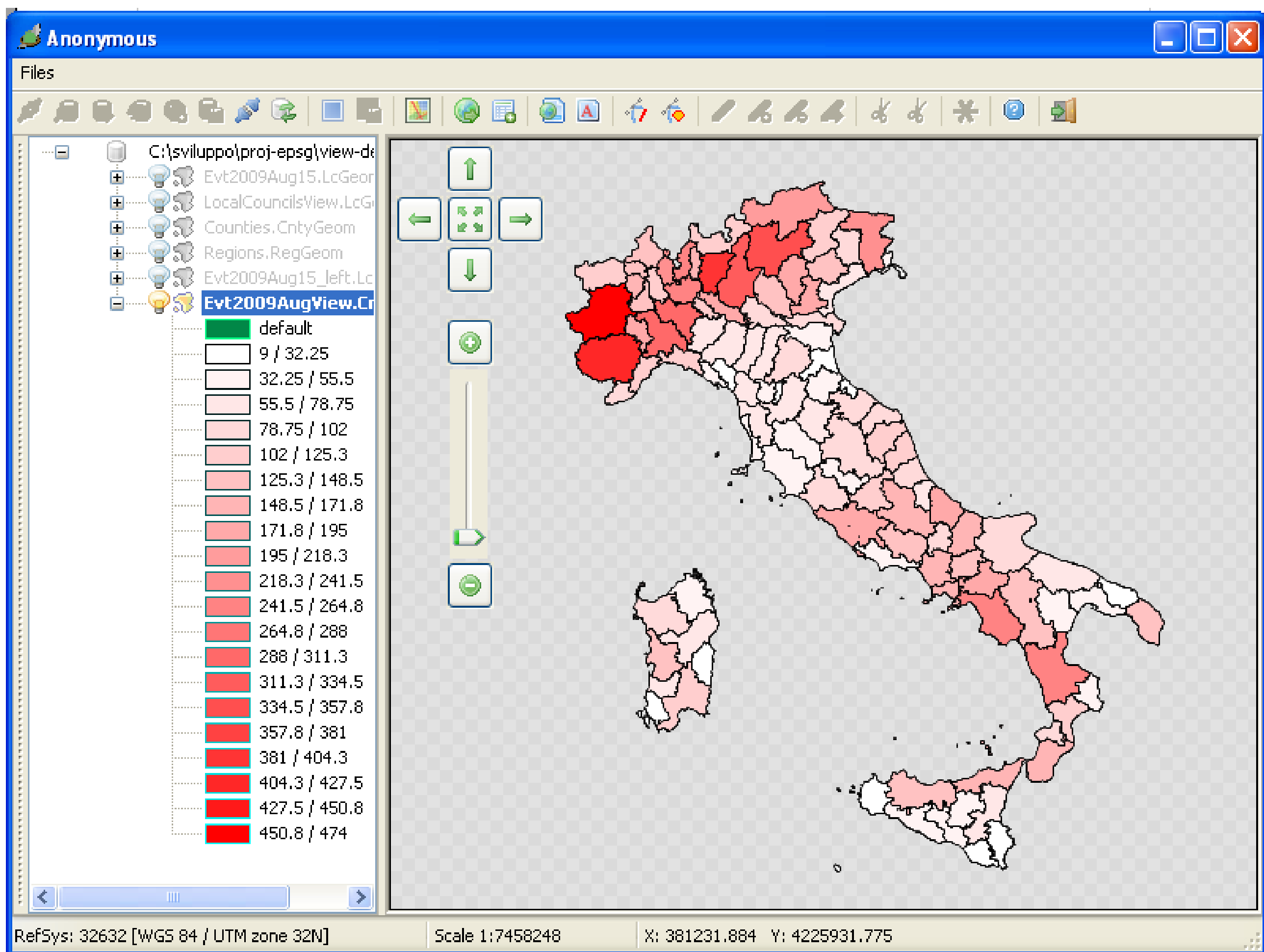
- yes, there is nothing wrong in this. Don't be lazy: you can create complex queries writing them completely by hand.
- don't fall victim of a nasty GUI-tools addiction: GUI tools are easiest and more comfortable to use, but using your own brain (*from time to time*) may well be a good exercise as well.



- and now you can revert to the *sybaritic* luxury offered by the GUI Query/Viewer Composer tool …



- finally creating a Spatial View you can then use as a GIS layer



- here we are: you've just performed a quite complex analysis in a very few steps
- and after all, it wasn't as difficult as you thought at first sight

Performance hints

The SQL data engine [*query optimizer*] implemented by SQLite seems to have some very specific hidiosyncrasies. Keep them always in mind, before planning your queries and/or views:

- complex View chains [i.e. defining a View based on a second View, and so on] are supported in a very efficient and brilliant way: at least, as long as you take care of defining any relevant index useful to quickly resolve the required JOIN conditions.
- but such complex View chains perform in a very poor (sluggish) way, if any Geometry column is involved. This issue may be less noticeable in the case of POINT Geometries, but it becomes a major issue in the case of (*possibly huge*) POLYGONs or LINESTRINGs Geometries.
- performance is noticeably slower when using LEFT JOINs.

All right, folks ... that all for tonight

I hope you've enjoined and found useful all this