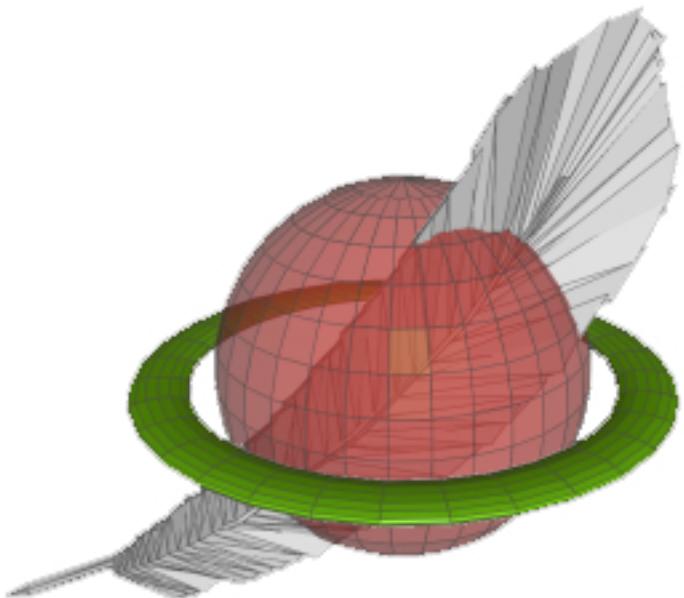


SpatiaLite Cookbook

Autore: Alessandro Furieri



Febbraio 2011



Autore: Alessandro Furieri a.furieri @ lqt.it

Questo lavoro è sotto licenza Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) della licenza.



È garantito il permesso di copiare, distribuire e / o modificare questo documento sotto i termini della GNU Free Documentation License , Versione 1.3 o ogni versione successiva pubblicata dalla Free Software Foundation; senza alcuna sezione non modificabile, senza testo di copertina e senza testo di quarta di copertina.

utilizzare SpatiaLite

una guida semplice e veloce per principianti



Febbraio 2011

Introduzione / Sommario

Dichiarazione di principio: in Informatica ci sono due parole sfortunate: **database e SQL** .

Entrambi i termini condividono una lunga e profondamente consolidata cattiva reputazione, la semplice pronuncia dei loro nomi causerà reazioni fortemente negative:

"*tropo complicato*", "*Non sarò mai in grado di capire tutto questo*", "*stranezze nerd (robe da informatici schizzati di testa)*" e così via

Se tutto questo è indiscutibilmente vero per il semplice SQL, che dire del più esotico **Spatial SQL** ? Ovviamente, questo suona di gran lunga molto più complesso e intimidatorio: si tratta di roba complicatissima esclusivamente per ultra-specialisti, che solo pochissimi possono confrontarsi con Spatial SQL, non è così?

Dimentica tutto questo, ed apri la tua mente!

(*Stavo quasi scrivendo: Cazzate! ma non credo che l'uso di termini volgari contrassegni positivamente l'inizio di un lavoro...*)

Suppongo che tutti i pregiudizi sono semplicemente la triste conseguenza delle politiche di marketing a lungo praticate.

Per molti anni il mercato dei DB è stato dominato da soluzioni proprietarie esageratamente costose: e lo Spatila DB costituisce soltanto un ristretto segmento specializzato (*ancora più esageratamente costoso*) all'interno del mercato generale dei DB.

Così il clima di aura sacrale volutamente indimitatorio che circonda le tecnologie DB è stato molto più un abusata storia di copertura per giustificare prezzi anormalmente elevati piuttosto che a un fatto tecnico oggettivo. Fortunatamente la verità reale è molto diversa: *qualsiasi persona con una normale preparazione informatica può facilmente imparare ed usare con successo sia l'SQL che lo Spatila SQL: non c'è nulla di difficile, oscuro o complicato nell'usarli.*

E quando dico **gente professionista** non intendo necessariamente *sviluppatori , ingegneri informatici o professionisti GIS* :

Personalmente sono ben consapevole che molti *ecologi, ingegneri del traffico, botanici, ingegneri ambientali, zoologi ,funzionari della pubblica amministrazione , geologi , geografi , archeologi* (e molti altri) possono usare con successo Spatial SQL nelle loro attività quotidiane.

Ora voi potete disporre, nel modo più facile e senza fatica, di un sofisticato, allineato agli standard ed efficace Spatial DB assolutamente libero.

E non è tutto: SpatiaLite è fortemente integrato nell'ecosistema del software libero. Inoltre potete connetervi immediatamente a uno Spatial DB ad es. usando QGis (una famosa e diffusa applicazione Gis per computer).

Non credete alle mie affermazioni? Beh, provate da soli: toccate con mano e vedete con i vostri occhi. Cosa potrebbe essere migliore di questo approccio per ottenere una prima esperienza neutrale, obiettiva e imparziale?

Seguire questa guida richiederà circa un'ora o due del vostro prezioso tempo: non è un compito troppo impegnativo e voi non siete chiamati a sforzi eccezionalmente complessi.

Spatialite è assolutamente *libero* (nel doppio significato: *free as a free beer and it's free as a free speech*) così potete dare uno sguardo veloce alle più aggiornate tecnologie di Spatial SQL nel modo più facile e senza fatica, efficace ed allineato agli standard.

Ed una volta che avrete la vostra opinione sulla base di esperienza diretta, potrete decidere da soli se lo Spatial DB può essere utile (in un modo o nell'altro) nella vostra attività quotidiana.

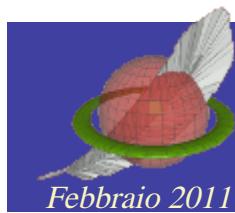
La struttura di questa guida: pensate a qualche libro di cucina. Da prima vi aspettate semplicemente di apprendere qualche piccola e semplice conoscenza circa:

- pentole e strumenti di cucina: *pentole, ciotole, fruste, coltelli, cucchiae, frullini*
- ingredienti più comuni: *uova, pesci, farina, verdure, spezie, frutta di stagione*
- tecniche elementari di cucina: *bollito, griglia, arrosti, frittura, zuppe*

Una volta che avete appreso queste basilari nozioni siete pronti a misurarvi con semplici ma nutrienti e gustosi piatti: potete chiamare questo livello **cucina familiare**. Per molte persone questo è lo sforzo massimo: è sufficiente per loro, ed abbandoneranno a questo punto.

Ma molti altri scopriranno che dopo tutto cucinare è piacevole e veramente interessante: questi sicuramente richiederanno qualcosa di molto più impegnativo.

Ogni buon libro di cucina si chiuderà presentando qualche complicato piatto di **alta cucina**. Può darsi che nessuno di voi diventerà mai uno **chef**, ma certamente vi divertirete e sarete orgogliosi della vostra abilità culinaria.



SpatiaLite Cookbook

Indice generale

1 Strumenti e tecniche culinarie

1.1	Breve introduzione tecnica.	pg 7
1.2	Per iniziare (installazione del software).	9
1.2.1	Scaricare l'applicazione.	9
1.2.2	Scaricare il set dati campione n. 1.	9
1.2.3	Scaricare il set dati campione n. 2.	9
1.2.4	Lanciamo spatialite_gui.	10
1.3	Costruire il tuo primo Spatial Database.	11
1.4	A proposito di ESRI Shapefilee Tabelle Virtuali.	16
1.4.1	Cos'è uno Shapefile?	16
1.4.2	Che cosa è uno Shapefile virtuale? (e tabelle virtuali....).	16
1.5	A proposito di codifiche Charset? (e perchè diavolo devo prendermi cura di queste cose....).	17
1.6	Cosa sono questi SRID?.. non ho mai sentito questo termine prima d'ora.	18
1.6.1	Tentativo di normalizzare il caos.	20
1.7	Eseguiamo le prime interrogazioni SQL .	23
1.8	I primi rudimenti sulle interrogazioni SQL	27
1.9	Comprendere le funzioni di aggregazione	31

2 Ingredienti di uso comune

2.1	La vostra prima interrogazione spaziale .	33
2.2	Ancora sulle interrogazioni spaziali: WKT e WKB	35
2.2.1	Avvertenze WKT and WKT	36
2.3	Le tabelle MetaData spaziali	40
2.4	Visualizzare un layer SpatiaLite in QGis	43

3 Cucina familiare

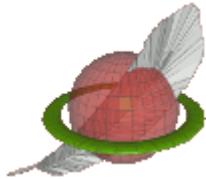
3.1	Ricetta # 1: Creare un DB ben fatto .	46
3.2	Ricetta # 2: Le vostre prime interrogazioni JOIN	51
3.3	Ricetta # 3: Maggiori informazioni su JOIN	55
3.4	Ricetta # 4: A proposito di VIEW	58
3.5	Ricetta # 5: Creazione di una nuova tabella (con annessi e connessi)	61
3.6	Ricetta # 6: Creare una nuova colonna GEOMETRY	65
3.6.1	SRID disponibili	65
3.6.2	Tipi Geometry disponibili.	66
3.6.3	Modelli Dimensionali disponibili.	66
3.7	Ricetta # 7: Inserire, aggiornare e cancellare	69
3.8	Ricetta # 8: Conoscere i vincoli	73
3.9	Ricetta # 9: ACIDity: conoscere le transazioni	78
3.10	Ricetta # 10: La bellezza dell'indice R*Tree spaziale.	82

4 Alta cucina

4.1 Ricetta # 11: Il Guiness dei primati.	88
4.2 Ricetta # 12: Dintorni	97
4.3 Ricetta # 13: Le isole	102
4.4 Ricetta # 14: Centri abitati e Comuni	104
4.5 Ricetta # 15: Centri abitati strettamente adiacenti.	107
4.6 Ricetta # 16: Ferrovia e Comuni.	109
4.7 Ricetta # 17: Ferrovie e centri abitati	101
4.8 Recitta # 18: Zone ferroviarie come Buffers.	117
4.9 Ricetta # 19: Unione [Merging] dei Comuni in Province e così via	120
4.10 Ricetta # 20: viste spaziali (Spatial Views).	125
4.11 Una raffinata esperienza culinaria: da Dijkstra	129

5 Dolci, liquori, the e caffè

5.1 Suggerimenti sul livello di prestazioni del sistema.	135
5.2 Importare/esportare Shapefile (DBF,TXT,...).	138
5.3 Linguaggi di collegamento(C / C + +, Java, Python, PHP ...)	149
5.3.1 C / C++.	151
5.3.2 Java	162
5.3.3 Python	169
5.3.4 PHP	173



Strumenti e tecniche culinarie

Febbraio 2011



Breve introduzione tecnica

Per iniziare (installazione del software)

Costruite il vostro primo Spatial DataBase

Sugli ESRI Shapefiles e Virtual Tables

Che cos'è una codifica di caratteri?

Cosa sono questi SRID?

Eseguiamo le prime interrogazioni SQL

I primi rudimenti sulle interrogazioni SQL

Comprendere le funzioni di aggregazione



Breve introduzione tecnica

Febbraio 2011

Breve introduzione tecnica: in termini tecnici **SpatiaLite** è uno **Spatial DBMS** che supporta standards internazionali quali **SQL92** e **OGC-SFS**.

Suppongo che tutti gli acronimi precedenti suoneranno un pò oscuri e (forse) vi disturberanno.

Non preoccupatevi: molto spesso oscuri gerghi tecnici nascondono concetti veramente facili da imparare:

- un **DBMS** (*Data Base Management System*) è un software progettato per memorizzare e ricercare dati generici nel modo più efficiente e generale possibile. Ed assai spesso si tratta di moli di dati veramente enormi.
- **SQL** (*Structured Query Language*) è un linguaggio strutturato adatto a manipolare i DB; attraverso comandi SQL potete decidere come organizzare i vostri dati. E potete inserire, cancellare e modificare i vostri dati nel DB. Ovviamente potete interrogare i vostri dati in modo molto flessibile (ed efficiente).
- **OGC-SFS.**(*Open Geospatial Consortium -Simple Feature Specification*) consente di estendere le potenzialità DBMS/SQL per gestire speciali tipi di dati **Geometry**, consentendo di disporre di uno Spatial DB.

SpatiaLite si appoggia e fa ampio uso del famoso **SQLite**, un DBMS leggero. Essi agiscono in coppia: SQLite implementa un motore SQL92, mentre SpatiaLite implementa il nucleo dello standard OGC-SFS. Usandoli combinati insieme si avrà un completo Spatial DB.

SQLite/SpatiaLite non si basano sulla architettura molto diffusa **client-server**, essi adottano una più semplice personal architecture. Cioè l'intero motore SQL è incorporato all'interno dell'applicazione.

Questa architettura semplice e poco sofisticata semplifica notevolmente ogni compito relativo alla gestione del database: potete semplicemente aprire (o creare) un **database-file** esattamente nello stesso modo in cui siete abituati ad aprire un documento di testo o un foglio elettronico.

Non c'è assolutamente nessuna complessità aggiuntiva in queste attività.

Un **Database** completo (magari uno contenente diversi milioni di dati) è semplicemente un file ordinario. Lo potete liberamente copiare (o anche cancellare) a vostro piacere senza nessun problema.

E non è tutto: quel **database-file** adotta un'architettura **universale** così potete trasferire l'intero database-file da un computer all'altro senza speciali accorgimenti.

I computer di provenienza e di destinazione possono ospitare anche sistemi operativi completamente differenti: questo non ha alcun effetto, perchè i database-files sono **portabili** su architetture diverse.

Chiaramente tutta questa semplicità e leggerezza ha un costo: il supporto di SQLite/SpatiaLite all'accesso concorrente (più accessi contemporanei) è piuttosto rudimentale e povero.

Questo è l'esatto significato di **personal DB**: il paradigma sottinteso è **singolo utente/singola applicazione/stazione di lavoro isolata**.

Se il supporto ad accessi concorrenti è il vostro problema principale, allora SQLite/SpatiaLite non è la scelta migliore per le vostre necessità: un più complesso DBMS client-server è fortemente consigliato.
In ogni caso SQLite/SpatiaLite è molto simile a **PostgreSQL/PostGis** (*uno spatial dbms molto potente e open source*): così potete liberamente passare da uno all'altro (*in modo relativamente indolore*) a seconda delle vostre reali esigenze, scegliendo ogni volta il miglior strumento da usare.

Qualche altro riferimento utile:

- [SQLite home](#)
- [SpatiaLite home](#)
- [SQL inteso da SQLite](#) (riferimento tecnico)
- [SpatiaLite manuale](#) (versione obsoleta, ma ancor oggi di utile lettura)
- [Old SpatiaLite Tutorial](#) (versione obsoleta, ma ancor oggi di utile lettura)



Febbraio 2011

Per iniziare

(Installazione del software)

Questo non è un manuale di teoria: questo è una guida pratica per principianti totali.

L'assunto sottostante è che voi ignoriate tutto rispetto a DBMS, SQL ed anche GIS.

Quindi cominceremo a scaricare alcune risorse dal Web ed inizieremo immediatamente la sessione di lavoro:

- primo scaricheremo il software **spatialite-gui**: questo è un semplice ma potente strumento (supporta la grafica, il mouse e così via ...) che vi consente di interagire con un database SpatiaLite
- poi scaricheremo qualche insieme di dati, disponibili al pubblico, necessari per costruire il DB da usare nel corso degli esercizi.

Scarica l'applicazione

Avviare il browser Web e accedere alla home **SpatiaLite**: <http://www.gaia-gis.it/spatialite>

La versione attuale (gennaio 2011) è v.2.4.0-RC4, e si può ottenere file binari eseguibili da:

<http://www.gaia-gis.it/spatialite-2.4.0-4/binaries.html>

Come ovvio l'organizzazione del sito dei download cambia di tanto in tanto, così gli indirizzi precedenti possono diventare facilmente obsoleti. Può succedere per gli utenti Linux di dover compilare i binari direttamente dai sorgenti: in tale caso leggete attentamente le note di rilascio prima di iniziare.

Scarica il set dati campione n. 1

Il primo insieme di dati che useremo è il censimento italiano 2001, cortesemente fornito dall' **ISTAT** (l' ente statistico italiano).

Punta il tuo browser web su <http://www.istat.it/ambiente/cartografia/> e quindi scaricare i seguenti file:

- Censimento 2001 - Regioni (*Regions*):
<http://www.istat.it/ambiente/cartografia/regioni2001.zip>
- Censimento 2001 - Province (*Counties*):
<http://www.istat.it/ambiente/cartografia/province2001.zip>
- Censimento 2001 - Comuni (*Local Councils*):
<http://www.istat.it/ambiente/cartografia/comuni2001.zip>

Scarica il set di dati campione n. 2

Il secondo insieme di dati necessario è **GeoNames**, una collezione mondiale di Centri Abitati (*Populated Places*). Ci sono diverse versioni di questi dati: noi useremo **cities-1000** (*ogni luogo nel mondo con più di 1000 persone*):
<http://download.geonames.org/export/dump/cities1000.zip>

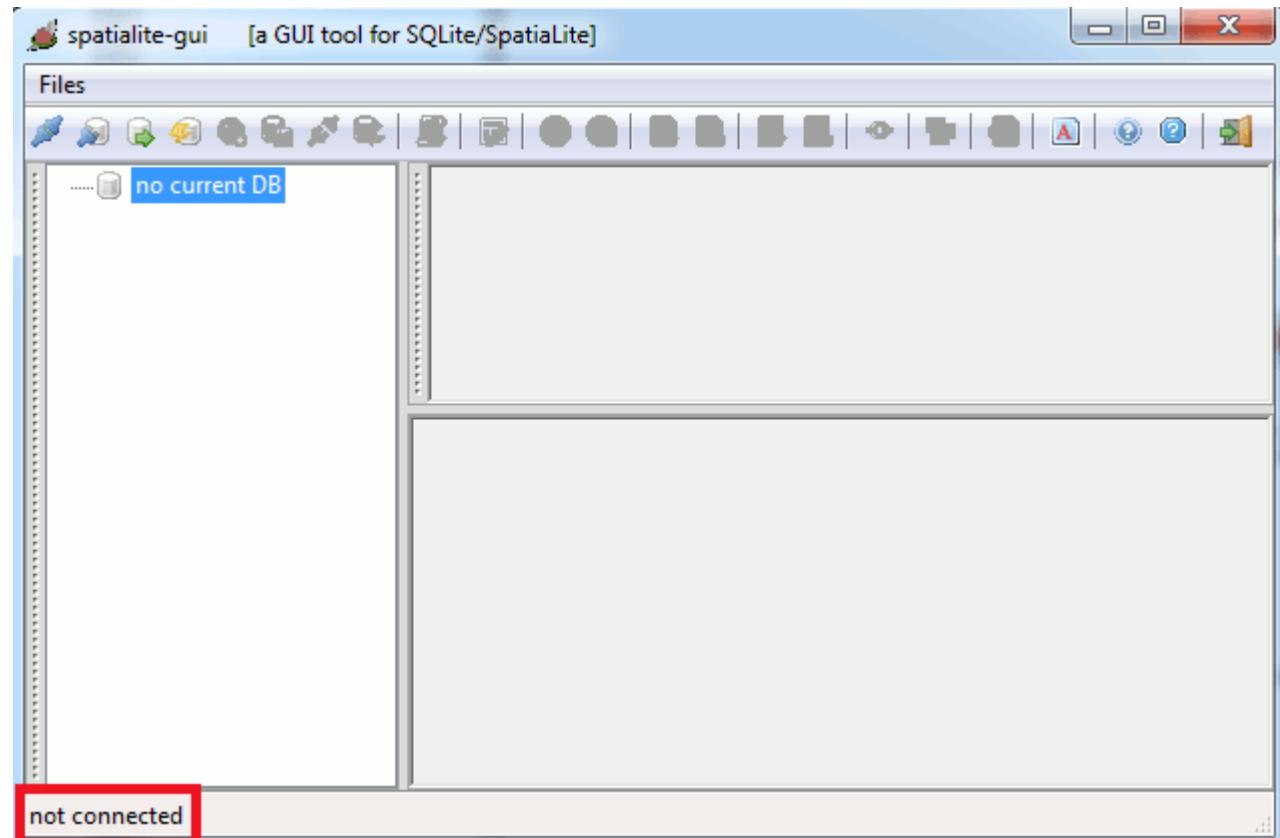
Lanciamo spatialite_gui

Il **spatialite_gui** non richiede alcuna installazione: è sufficiente *decomprimere* l'immagine compressa che avete appena scaricato e fare clic sull'icona di lancio. Questo è tutto.

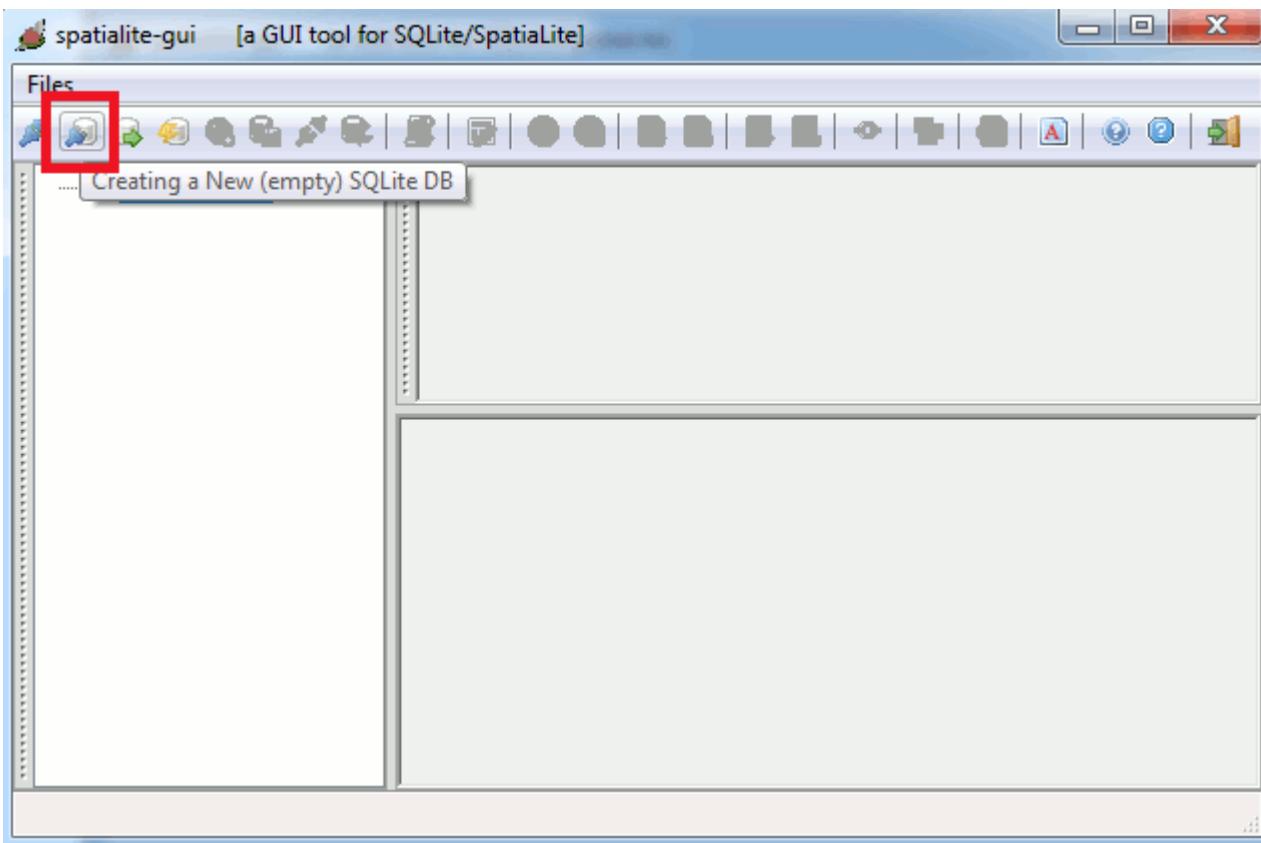


Febbraio 2011

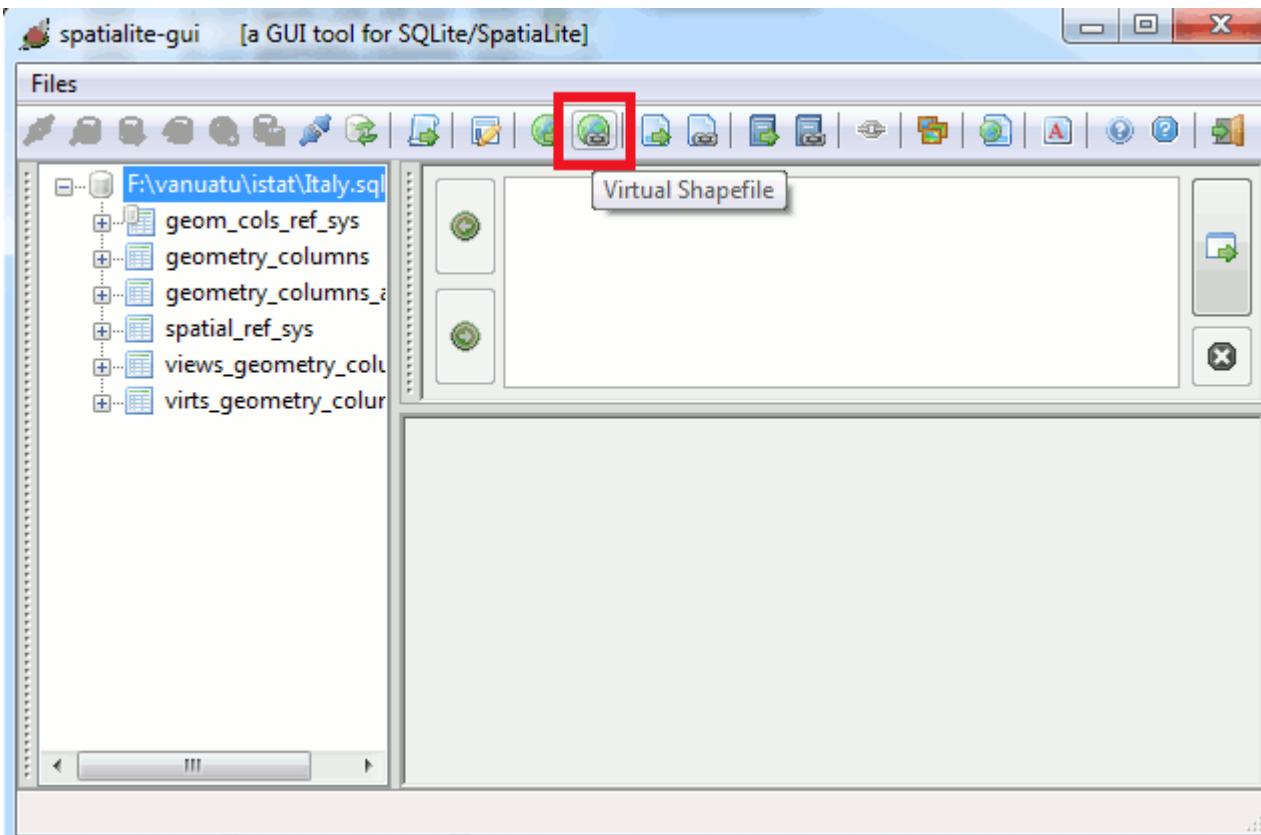
Costruire il tuo primo Spatial Database



Bene, avete appena avviato la vostra prima sessione di lavoro con SpatiaLite: come potete notare, non c'è nessun DB al momento connesso.

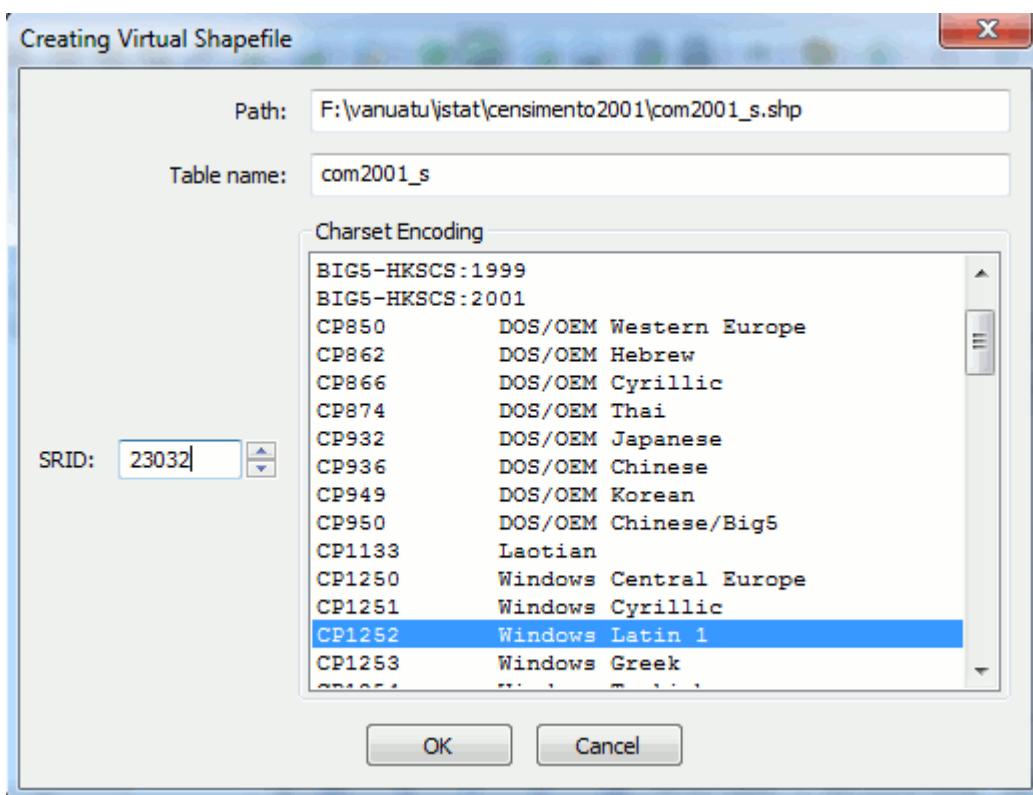


così creerete e vi connetterete al nuovo DB: dovete semplicemente premere il corrispondente bottone dalla barra degli strumenti (apparirà una *finestra di dialogo* per l'apertura di files) e definite un nome per il file. Solo per l'uniformità, si prega di chiamare questo DB come **Italy.sqlite**



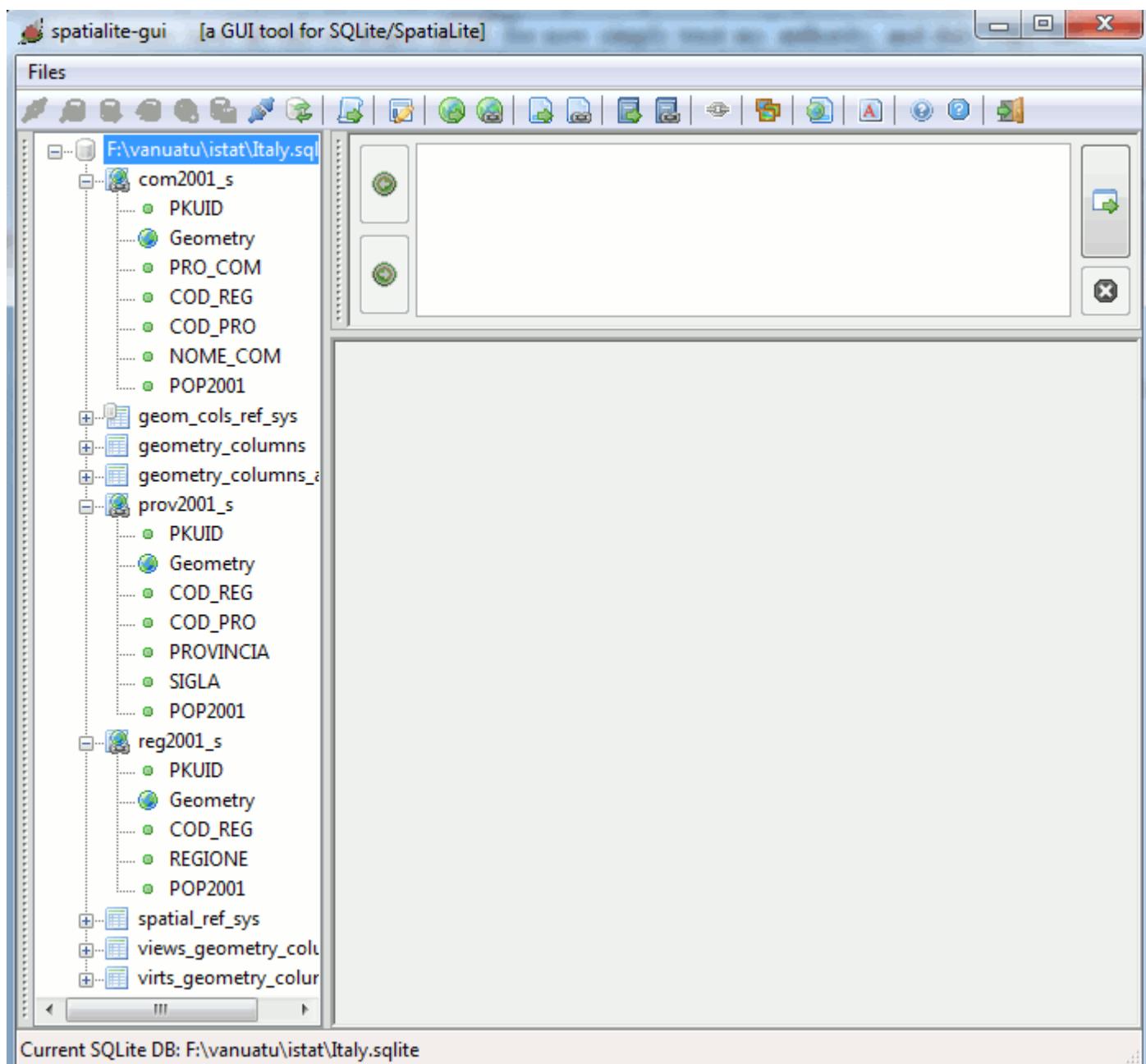
Come potete notare, subito dopo la creazione il DB contiene già parecchie tabelle: sono tutte **tabelle di sistema** (diciamo **metatabelle**), cioè tabelle necessarie a garantire l'amministrazione interna.

Al momento, la miglior scelta da fare è semplicemente ignorarle del tutto (siete dei principianti, no? siate pazienti, per favore). In ogni caso, ora siete connessi ad un DB valido, così potete ora caricare il primo insieme di dati: premete il bottone **Virtual Shapefile** nella barra degli strumenti, e quindi selezionate il file **com2001_s**.

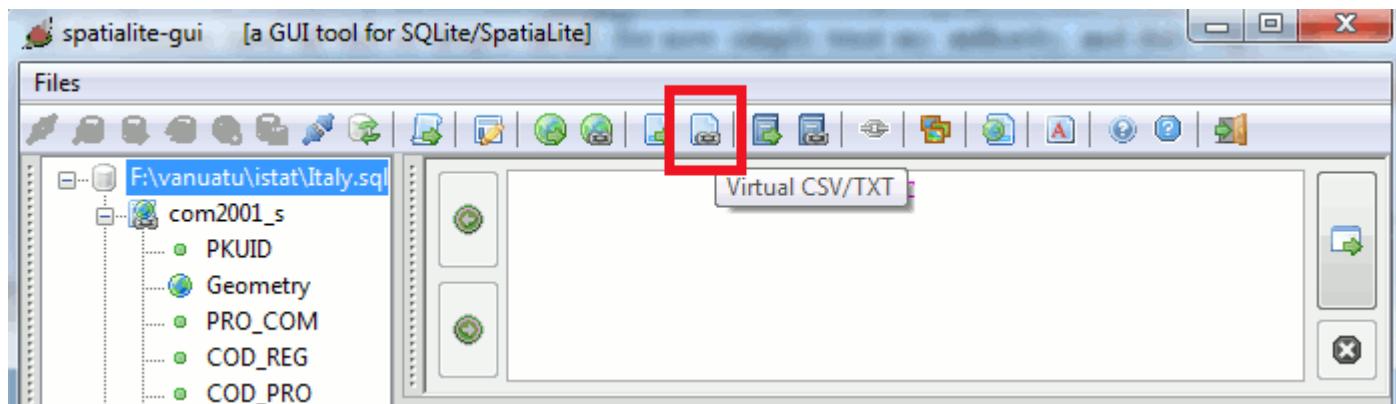


Apparirà una finestra di dialogo: selezionate per favore esattamente il settaggio evidenziato sopra e confermate. Esamineremo tutto questo in maggior dettaglio successivamente: per ora fidatevi della mia autorità, copiate diligentemente i valori suggeriti senza cercare di capire: sono dei valori magici per ora ed è tutto.

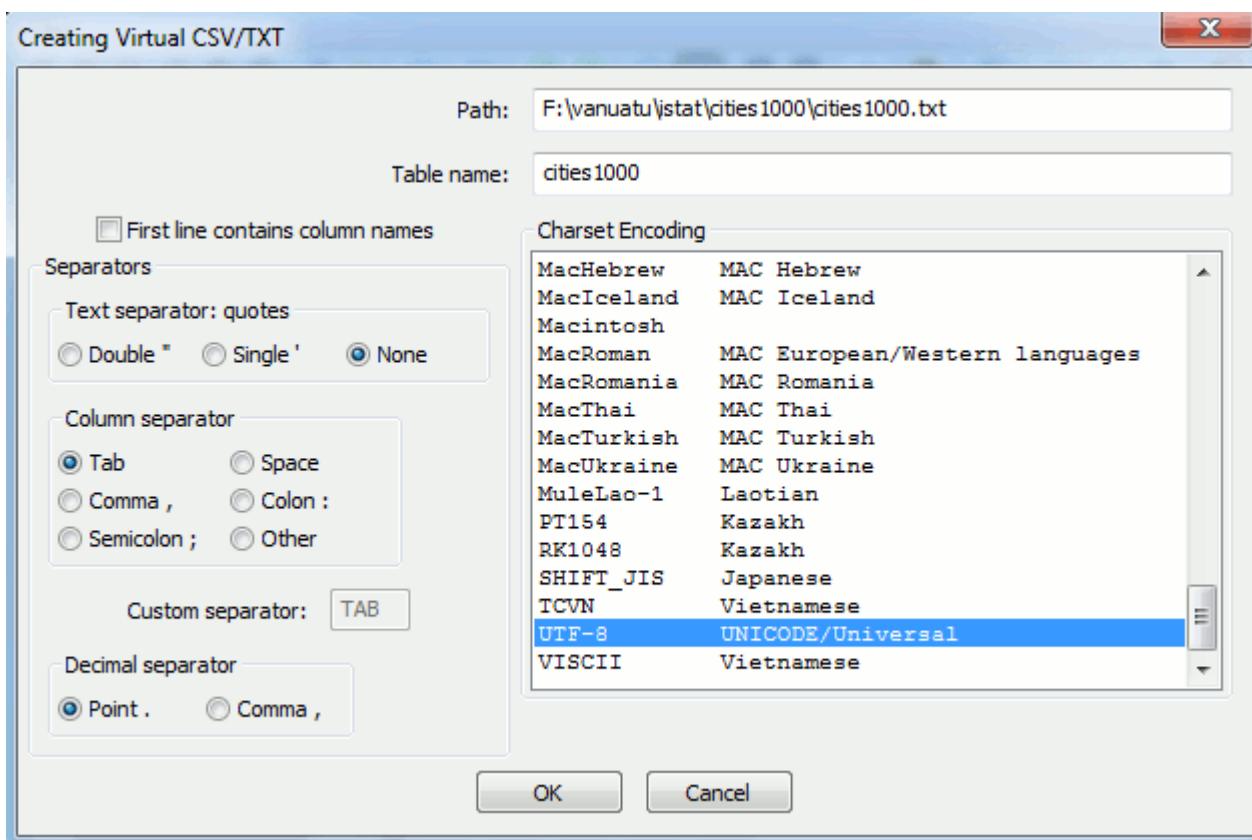
Una volta caricato l' insieme `com2001_s` potete continuare (usando sempre lo stesso settaggio) e caricare entrambi i files `prov2001_s` e `reg2001_s`.



Il vostro database dovrebbe assomigliare a questo: usando la vista ad albero alla sinistra del video è davvero facile esaminare le tabelle (e le colonne in ogni tabella).



Ora siete pronti per completare la messa a punto iniziale del DB: premete il bottone **Virtual CSV/TXT** e della barra degli strumenti, e selezionate il file `cities1000.txt`.



Apparirà una finestra di dialogo: selezionate per favore *esattamente* i settaggi visualizzati sopra e confermate. Questa finestra di dialogo assomiglia molto a quella che avete già usato per connettervi a Virtual Shapefiles, ma non è identica. Anche in questo caso esamineremo più tardi i dettagli relativi.

Bene: adesso avete tre insiemi di dati pronti da interrogare: ma è tempo di spiegare meglio cosa stiamo per fare.



A proposito di ESRI Shapefile e Tabelle Virtuali

Febbraio 2011

Cos'è uno Shapefile?

Shapefile è un semplice, elementare formato di file GIS (dati geografici) inventato molti anni orsono da ESRI: benché inizialmente nato in un ambiente proprietario, questo formato è stato successivamente reso pubblico e completamente documentato, così è praticamente quasi un formato *standard aperto*.

E' piuttosto vecchio adesso, ma è universalmente supportato.

Così rappresenta la lingua franca che ogni applicazione GIS sicuramente conosce: e non deve sorprendere, SHP è largamente usato per scambio neutrale di dati fra piattaforme diverse.

Il nome stesso è abbastanza fuorviante: dopo tutto, Shapefile non è un semplice file. Almeno 3 files distinti sono necessari (identificati dai suffissi **.shp .shx .dbf**): se anche un solo file manca (*malnominato / spostato / corrotto / qualcos' altro*), l'intero set è corrotto ed inusabile.

Qualche ulteriore utile riferimento:

- <http://en.wikipedia.org/wiki/Shapefile>(semplice introduzione)
- <http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf>(riferimenti tecnici)

Che cosa è un shapefile virtuale (e tabelle virtuali)?

SpatiaLite supporta un driver Virtual Shapefile: cioè ha la capacità di consentire accessi SQL (in modalità di sola lettura) per un Shapefile esterno, senza bisogno di caricare alcun dato nel DB stesso.

Questo è davvero utile durante i passi preliminari alla costruzione del database (come nel nostro caso). SQLite/SpatiaLite gestisce parecchi altri driver Virtual come il Virtual CSV/TXT, il Virtual DBF e così via...

Comunque, **attenti**: le Tabelle Virtual soffrono di parecchie limitazioni (e sono spesso molto più lente dei dati memorizzati all' interno del DB), pertanto non sono da prendere in cosiderazione per lavori di produzione seri.



A proposito di codifiche Charset

che cos'è una codifica di caratteri? (*e perchè diavolo devo prendermi cura di queste cose...*)

Detto molto semplicemente: ogni computer è in realtà una stupida macchina basata su un mucchio disordinato di silicio: possiede qualche capacità di capire semplici operazioni aritmetiche e booleane, ma è assolutamente incapace di padroneggiare il testo.

Potete avere memorizzato da qualche parte del vostro cervello l' errata convinzione che un computer possa realmente trattare del testo, ma questo non è esattamente vero.

Per essere più precisi, è piuttosto un gioco di prestigio finalizzata ad ingannare te, *stupido essere umano*, che ti fidi dei tuoi sensi limitati: ogni computer manipola semplicemente cifre, le unità periferiche (video, tastiera, stampante ..) sono espressamente progettate per darvi l' (illusoria) impressione che il vostro PC conosca effettivamente il testo.

Tutto questo è un **processo assolutamente convenzionale**: voi ed il vostro PC dovete essere d' accordo su quale **tabella di corrispondenza** debba essere utilizzata per trasformare oscure sequenze digitali in parole leggibili. In termini tecnici, questa convenzionale tabella di corrispondenza è nota come **Charset Encoding**. Quanti alfabeti differenti sono usati sulla Terra? centinaia e centinaia ... Latino, Greco, Cirillico, Ebraico, Arabo, Cinese, Giapponese e molti altri ... ed in conformità a questo, centinaia e centinaia di Charset Encodings diversi sono stati definiti durante gli anni (*voi sapete: l'industria elettronica / informatica è fondata su una incontrollata proliferazione di standard incongruenti*).

SQLite/SpatiaLite internamente usa sempre la **codifica UTF-8**, che è **universale** (cioè potete salvare con sicurezza qualsiasi alfabeto conosciuto nello stesso DB nello stesso momento): sfortunatamente, *Shapefiles* (e molti altri insiemi di dati, come i files SVC/TXT) non utilizzano la codifica UTF-8 (essi usano invece alcune codifiche nazionali), così siete obbligati a scegliere esplicitamente il Charset encoding da usare quando dovete importare (o esportare) qualsiasi dato. Sono dispiaciuto di questa situazione, ma purtroppo questa è la dura realtà dei fatti.

Comunque, consideriamo tutto questo non come una grande complicazione, ma come una grande risorsa: in questo modo sarete capaci di importare/esportare correttamente qualsiasi dataset da qualsiasi esotico paese non latino provenga, come Israele, Giappone, Vietnam, Grecia o Russia.

E dopo tutto, essere in grado di visualizzare stringhe di testo di molti alfabeti (come quello seguente) può far diventare i vostri amici verdi dall'invidia: **Roma, Ρόμη, Рим, ...**

Alcuni utili riferimenti ulteriori:

- http://en.wikipedia.org/wiki/Character_encoding (semplice introduzione)
- <http://www.gnu.org/software/libiconv/> (riferimento tecnico)



Febbraio 2011

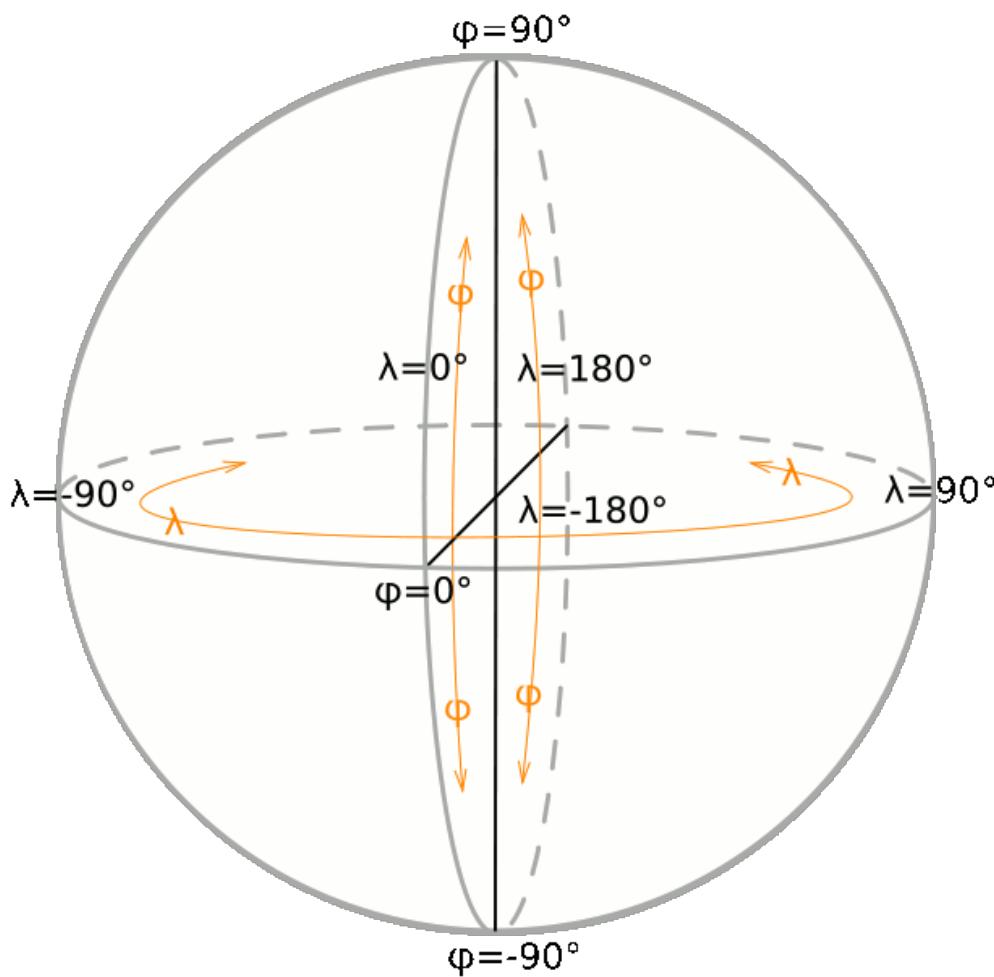
Cosa sono questi SRID?

... non ho mai sentito questo termine prima d'ora ...

Cosa sono questi SRID? ... Non ho mai sentito questo termine prima d'ora ...

Il pianeta Terra è una **sfera** ... non esattamente, il pianeta Terra ha una **forma ellisoidica** (leggermente schiacciata ai poli) ...oh no, è tutto sbagliato: il pianeta Terra non ha una forma geometrica regolare, in realtà si tratta di un **geoide**.

Tutte le affermazioni precedenti possono essere assunte per vere, ma a differenti livelli di approssimazione. Vicino all' Equatore le differenze fra la sfera e l' ellissoide sono piuttosto piccole e quasi non si vedono; ma vicino ai Poli tali differenze diventano più grandi e cominciano a farsi sentire.
Per molte necessità pratiche le differenze fra un ellissoide ed un geoide sono molto piccole: ma per la navigazione di un aereomobile a largo raggio (o peggio, per il posizionamento dei satelliti), questo è troppo grossolano ed offre una approssimazione inaccettabile.



In ogni caso, qualsiasi sia la vera forma della Terra, la posizione di ogni punto della superficie del pianeta può essere determinata precisamente misurando due **angoli: longitudine e latitudine**.

Per definire un **Sistema di Riferimento Spaziale** completo [Spatial Reference System, aka **SRS**] usiamo i **Poli e l'Equatore** (*che dopo tutto sono luoghi notevoli per le loro intrinseche proprietà astronomiche*): scegliere un **Meridiano** di riferimento [Meridiano Fondamentale] è d'altra parte assolutamente convenzionale: ma da molti secoli (Britannia rule the waves ...) adottare il **Meridiano di Greenwich** è una scelta ovvia.

Ogni SRS basato su coordinate **long-lat** è noto come **Sistema Geografico** [Geographic System].

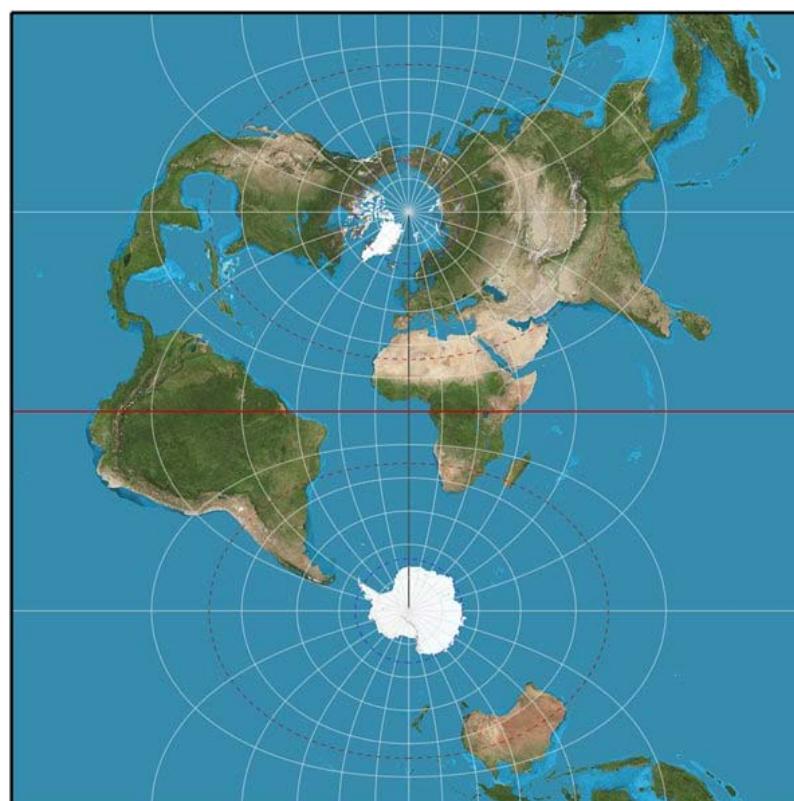
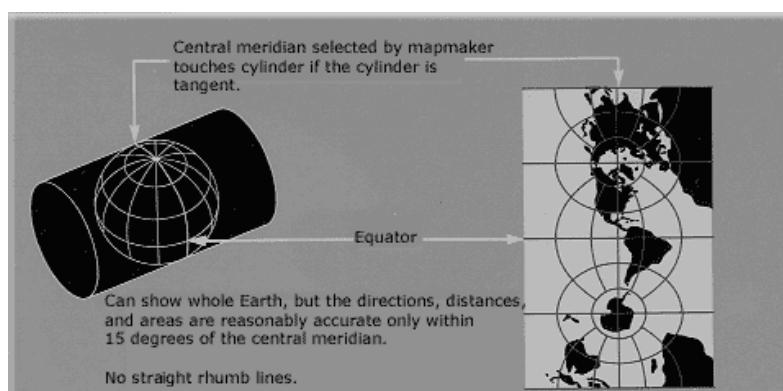
Usare un Sistema Geografico vi garantisce sicuramente la massima precisione ed accuratezza: ma sfortunatamente questo comporta fatalmente alcuni indesiderabili effetti collaterali:

- i fogli di carta (e gli schermi dei video) sono assolutamente piatti; essi non assomigliano per nulla ad una sfera
- l'uso degli **angoli** rende la misura delle distanze e delle aree molto difficile e poco intuitiva.

Perciò da molti secoli i cartografi hanno inventato parecchi sistemi (*convenzionali*) che permettono di rappresentare superfici sferiche su una piano: nessuno di loro è il *migliore* in assoluto.

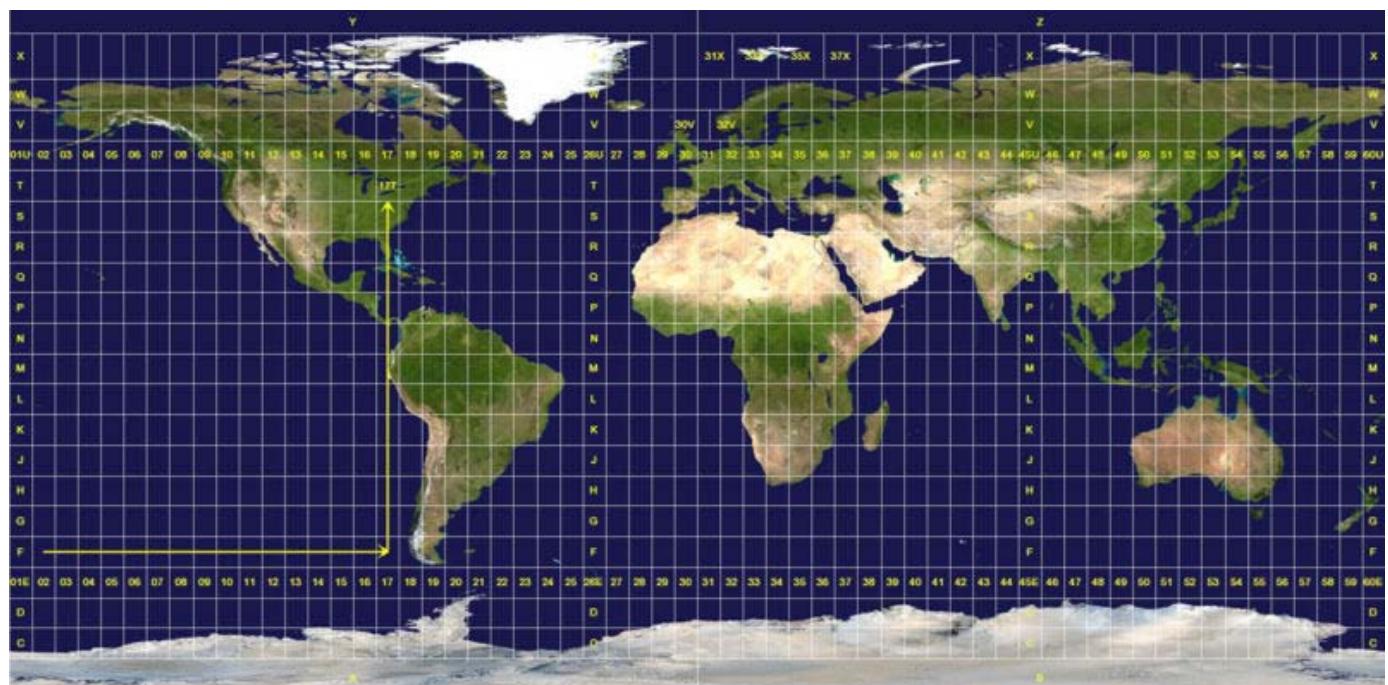
Tutti introducono qualche grado di approssimazione e di deformazione: la scelta dell'uno o dell'altro implica un percorso assolutamente *arbitrario e convenzionale*: una proiezione cartografica adatta a rappresentare piccole porzioni della Terra può facilmente essere inadatta a rappresentare territori molto vasti, e *viceversa*.

Ci addentriamo ad esaminare subito la proiezione cartografica **UTM** [*Universal Transverse Mercator*], semplicemente perchè è largamente usata.



Sembra che questa proiezione introduca gravi ed inaccettabili distorsioni: ma se prestate attenzione al fuso centrale, vi accorgerete immediatamente che UTM permette di avere una quasi perfetta proiezione piana di eccellente qualità.

In ogni caso questo ha un certo prezzo: il fuso centrale deve essere molto stretto (diciamo, deve estendersi per pochi gradi da ogni lato). Appena il fuso diventa più largo, le distorsioni diventano molto più grandi ed evidenti.



Per tutte le considerazioni precedenti, il sistema UTM definisce 60 zone standard, ognuna estesa per 6 gradi esatti di longitudine.

L'accorpamento di due fusi adiacenti (12 gradi) ovviamente riduce l'accuratezza, ma rimane accettabile per molte necessità pratiche: andare oltre questi limiti produce risultati davvero cattivi, e deve essere assolutamente evitata.

Tentativo di normalizzare il caos

Durante gli ultimi due secoli ogni Stato Nazionale ha introdotto almeno un (e molto spesso, più di uno) sistema di proiezione cartografica e relativo SRS: il risultato complessivo è assolutamente caotico (e davvero difficile da gestire).

Fortunatamente, uno standard internazionale è largamente usato per rendere più agevole la gestione dei Sistemi di Riferimento Spaiale: la **European Petroleum Survey Group [EPSG]** mantiene un grande archivio mondiale di oltre 3,700 voci diverse.

Molti di questi sono oggi obsoleti, e giocano semplicemente un ruolo storico; molti altri sono utili solo entro piccoli confini nazionali.

Comunque, questa è una straordinaria raccolta.

Ed ogni singola informazione dell'archivio EPSG è univocamente identificata dal suo **codice numerico ID** e da un **nome descrittivo**, così da evitare ogni possibile confusione e ambiguità.

Ogni Spatial DBMS richiede qualche **codice SRID** specificato per ogni Geometry: ma questo codice SRID è semplicemente il Riferimento Spaziale [**Spatial Reference ID**], e (fortunatamente) coincide con il corrispondente **EPSG ID**.

Proprio per facilitarvi a capire meglio questo caos degli SRID, questa è una lista quasi completa di SRID spesso usati in una (*piccola*) Nazione come l'**Italia**:

EPSG SRID	Nome	Note
4326	WGS 84	Geografico [<i>lung- at</i>]; mondiale; usato dagli strumenti GPS
3003 3004	Monte Mario / Italia zona 1 Monte Mario / Italia zona 2	obsoleto (1940), ma ancora comunemente usato
23032 23033	ED50 / UTM zone 32N ED50 / UTM zone 33N	superato e raramente utilizzato: European Datum 1950
32632 32633	WGS 84 / UTM zone 32N WGS 84 / UTM zone 33N	WGS84, adotta la proiezione piana UTM
25832 25833	ETRS89 / UTM zone 32N ETRS89 / UTM zone 33N	evoluzione di WGS84: standard ufficiali dell' UE

E gli esempi che seguono possono aiutare a comprendere ancora meglio:

Città	SRID	Coordinate	
		X (longitudine)	Y (latitudine)
Roma	4326	12.483900	41.894740
	3003	1789036.071860	4644043.280244
	23032	789036.071860	4644043.280244
	32632	789022.867800	4643960.982152
	35832	789022.867802	4643960.982036
Milano	4326	9.189510	45.464270
	3003	1514815.861095	5034638.873050
	23032	514815.861095	5034638.873050
	32632	514815.171223	5034544.482565
	35832	514815.171223	5034544.482445

Come si può facilmente notare:

- le coordinate WGS84 [4326] sono espresse in gradi decimali, perchè questo è un Sistema Geografico basato direttamente su angoli long-lat.
- invece ogni altro sistema adotta coordinate espresse in metri: sono tutti proiettati, cioè sistemi piani.
- i valori Y si assomigliano in tutti i sistemi piani SRS: non c' è da stupirsi, perchè questo valore rappresenta semplicemente la distanza dall' Equatore.
- i valori X sono molto diversi perchè i diversi SRS adottano origini (false easting) di comodo: cioè essi collocano il loro Meridiano di riferimento [Prime Meridian] in differenti posti (convenzionali).
- comunque, ogni SRS basato su UTM dà valori strettamente collegati, perchè tutti condividono lo stesso fuso UTM zona 32.
- le (piccole) differenze che potete notare fra diversi SRS basati su UTM possono essere facilmente spiegate: il fuso UTM zona 32 è sempre lo stesso, ma l' ellissoide sottostante cambia ogni volta.

Ottenerne una misura precisa per gli assi dell'ellissoide non è un compito facile: e, ovviamente, nel corso del tempo, diversi, sempre migliori e più accurate stime sono state progressivamente adottate.

Distanza tra intercurring Roma e Milano	
SRID	Calcolato Distanza
4326	4.857422
3003	477243.796305
23032	477243.796305
32632	477226.708868
35832	477226.708866
Great Circle	477109.583358
Geodetico	477245.299993

Ed ora possiamo vedere come l' uso dei diversi SRS influenza le distanze:

- le coordinate long-lat del *sistema geografico* WGS84 [4326] daranno una misura in **gradi decimali** [*non così utile, davvero ...*]
- tutti gli altri danno una misura della distanza espressa in **metri**: comunque, come potete vedere, i numeri non sono esattamente gli stessi.
- con il sistema **Great Circle** le distanze sono calcolate assumendo che la Terra sia esattamente una sfera: e questa ovviamente è la stima peggiore che possiamo avere.
- d' altro canto le distanze nel sistema **Geodesic** sono calcolate direttamente sull'Ellissoide di riferimento.

Conclusioni: Non ci sono misure esatte.

Ma questo non è per niente sorprendente nelle scienze fisiche e naturali: ogni valore misurato è intrinsecamente affetto da errori ed approssimazioni. Ed ogni valore calcolato sarà inesorabilmente affetto da errori di arrotondamento e troncamento. Così numeri assolutamente esatti semplicemente non esistono nel mondo reale: dovete essere coscienti che potete disporre soltanto di valori più o meno approssimati. Ma almeno, potete assicurarvi di ridurre tali approssimazioni nel miglior modo possibile.



Febbraio 2011

Eseguiamo le prime interrogazioni SQL

spatialite-gui [a GUI tool for SQLite/SpatiaLite]

Files

F:\vanuatu\istat\Italy.sqlite

SELECT *
FROM reg2001_s

Table: reg2001_s

Geometry COD_REG REGIONE POP2001

=67109 GEOMETRY	1	PIEMONTE	4214677
=17805 GEOMETRY	2	VALLE D'AOSTA	119548
=73756 GEOMETRY	3	LOMBARDIA	9032554
=43277 GEOMETRY	4	TRENTINO-ALTO ADIGE	940016
=53229 GEOMETRY	5	VENETO	4527694
=31549 GEOMETRY	6	FRIULI VENEZIA GIULIA	1183764
=49509 GEOMETRY	7	LIGURIA	1571783
=62257 GEOMETRY	8	EMILIA-ROMAGNA	3983346
=71901 GEOMETRY	9	TOSCANA	3497806
=35754 GEOMETRY	10	UMBRIA	825826
=38597 GEOMETRY	11	MARCHE	1470581
=55935 GEOMETRY	12	LAZIO	5112413
=32989 GEOMETRY	13	ABRUZZO	1262392
=28093 GEOMETRY	14	MOLISE	320601
=60155 GEOMETRY	15	CAMPANIA	5701931
=64409 GEOMETRY	16	PUGLIA	4020707
=33706 GEOMETRY	17	BASILICATA	597768
18 BLOB sz=46983 GEOMETRY	18	CALABRIA	2011466
19 BLOB sz=78470 GEOMETRY	19	SICILIA	4968991
20 BLOB sz=94235 GEOMETRY	20	SARDEGNA	1631880

Current block: 1 / 20 [20 rows] [fetched in 00:00:00.030]

Current SQLite DB: F:\vanuatu\istat\Italy.sqlite

Navigation icons: back, forward, search, refresh, first, last

Potete seguire due diverse metodologie per interrogare una tabella DB:

1. potete usare la voce di menu **Query Table** [Interrogare la tabella]
 - (a) questa è sicuramente la strada più facile e veloce, del tutto amichevole
 - (b) dovete semplicemente premere con il bottone destro del mouse sopra la tabella voluta, così apparirà un menu contestuale.
 - (c) potete semplicemente usare il bottone più in basso per scorrere i risultati su e giù a vostro piacimento
 - (d) comunque questo metodo è piuttosto meccanico, e non vi consentirà di sfruttare SQL al meglio delle sue possibilità.
2. in alternativa potete scrivere a mano qualsiasi comando SQL nel riquadro superiore, e quindi premere il bottone **Execute**
 - (a) questo è il metodo più difficile: voi siete responsabili di quello che state facendo (anche sbagliando...)
 - (b) ma in questo modo potete avvantaggiarvi della impressionante potenza di fuoco di SQL.

The screenshot shows the spatialite-gui application interface. At the top, there's a toolbar with various icons for file operations like Open, Save, and Import. Below the toolbar is a file browser showing a folder structure with files like 'Italy.sql', 'com2001_s', and 'geom_cols_ref_sys'. A main query editor window is open, displaying the following SQL command:

```
SELECT *
FROM reg2001_s
```

Below the query editor is a 'BLOB explorer' dialog box. It has two tabs: 'Hexadecimal dump' (selected) and 'Geometry explorer'. The 'Hexadecimal dump' tab shows the following output for SRID 23032:

```
SRID: 23032
Geometry type: MULTIPOLYGON
#14 POLYGONS:
1) exterior ring: 70 vertices
2) exterior ring: 16 vertices
3) exterior ring: 134 vertices
4) exterior ring: 36 vertices
5) exterior ring: 24 vertices
6) exterior ring: 20 vertices
7) exterior ring: 16 vertices
8) exterior ring: 9 vertices
9) exterior ring: 128 vertices
10) exterior ring: 49 vertices
11) exterior ring: 4068 vertices
12) exterior ring: 167 vertices
13) exterior ring: 100 vertices
14) exterior ring: 53 vertices
```

The 'Geometry explorer' tab shows a map preview of Italy with a yellow fill and a blue outline. At the bottom of the dialog is an 'OK' button.

Below the 'BLOB explorer' is a table view showing rows of data. One row is highlighted with a red box around the value '19 BLOB sz=78470 GEOMETRY'. The table columns include:

	18	18 BLOB sz=46983 GEOMETRY	18 CALABRIA	2011466
19	19 BLOB sz=78470 GEOMETRY	19 SICILIA	4968991	
20	20 BLOB sz=94235 GEOMETRY	BLOB explore	1631880	

At the very bottom of the application window, it says 'Current SQLite DB: F:\vanuatu\istat\Italy.sqlite'.

Avrete sicuramente notato nell' immagine precedente che la colonna Geometry riporta semplicemente un anonimo *BLOB GEOMETRY*: questo è lungi dall'essere soddisfacente.

Ma potete avere una più ricca presentazione di ogni Geometry semplicemente premendo il tasto destro del mouse sopra il valore corrispondente, quindi selezionando la voce di menu **BLOB explore**.

The screenshot shows the spatialite-gui interface. On the left, the file tree displays a database structure under 'F:\vanuatu\ista'. In the center, a SQL query window contains the following code:

```
SELECT COL002 AS name,
       COL006 AS longitude,
       COL005 AS latitude,
       MakePoint(COL006, COL005, 4326) AS geom
  FROM cities1000
 WHERE COL009 = 'IT'
```

Below the query window is a table viewer showing the results of the query. The table has columns: name, longitude, latitude, and geom. The data is as follows:

	name	longitude	latitude	geom
1	Zumpano	16.292090	39.310890	BLOB sz=60 GEOMETR
2	Zerfaliu	8.708890	39.958610	BLOB sz=60 GEOMETR
3	Zeddiani	8.595280	39.990830	BLOB sz=60 GEOMETR
4	Zambrone	15.990090	38.699180	BLOB sz=60 GEOMETR
5	Zagarise	16.663200	38.999190	BLOB sz=60 GEOMETR

At the bottom of the table viewer, it says 'current block: 1 / 500 [6192 rows] [fetched in 00:00:00.]'. Below the table viewer, the status bar indicates 'Current SQLite DB: F:\vanuatu\istat\Italy.sqlite'.

```
SELECT COL002 AS name,
       COL006 AS longitude,
       COL005 AS latitude,
       MakePoint(COL006, COL005, 4326) AS geom
  FROM cities1000
 WHERE COL009 = 'IT';
```

Potete provare come funziona SQL con il metodo manuale usando questo comando SQL: basta **copiarlo**, quindi **inserirlo** nell' apposito pannello, e premere il bottone **Execute**.

Una breve e veloce spiegazione:

- la tabella **cities1000** contiene ogni centro abitato [populated place] nel mondo
- vi sono molte colonne in questa tabella, ed il loro significato può risultare piuttosto oscuro (potete trovare una documentazione completa per questo su <http://www.geonames.org/>) ma per adesso potete fidarvi della mia autorità
- COL002 contiene il **nome** di ogni località abitata.
- COL006 contiene la corrispondente **longitudine** (espressa in gradi decimali)
- COL005 è la corrispondente **latitudine**
- MakePoint() è una funzione Spatial che costruisce una geometria di tipo punto dalle corrispondenti coordinate
- COL009 contiene il **codice del Paese** [Country code]
- detto molto semplicemente, la clausola **WHERE** filtrerà i risultati così da escludere tutte le località al di fuori dall' Italia.

Bene, si presume state sufficientemente abili per iniziare seriamente il lavoro.

Nella diapositiva successiva inizieremo ad esplorare il misterioso mondo di SQL e Spatial SQL.



Febbraio 2011

I primi rudimenti sulle interrogazioni SQL

I seguenti comandi SQL sono così semplici che potete verificare direttamente i risultati da soli.
Seguite semplicemente ogni esempio eseguendo il corrispondente comando SQL (usate **copia&incolla**).

```
SELECT *
FROM reg2001_s;
```

Questo è proprio il pons asinorum di SQL: tutte le colonne di ogni riga della tabella selezionata saranno visualizzate seguendo un ordine casuale.

```
SELECT pop2001, regione
FROM reg2001_s;
```

Non siete obbligati a visualizzare tutte le colonne: potete scegliere esplicitamente le colonne che volete includere nella ricerca, stabilendo anche il loro relativo posizionamento.

```
SELECT Cod_rEg AS code, REGIONE AS name,
       pop2001 AS "population (2001)"
FROM reg2001_s;
```

Potete anche decidere un nome più appropriato e comprensibile alle colonne, se trovate la cosa desiderabile.
Questo esempio mostra due importanti aspetti da notare assolutamente:

- i nomi delle tabelle e delle colonne in SQL non sono *case-sensitive* (minuscolo/maiuscolo): **REGIONE** e **regione** si riferiscono alla stessa tabella.
- i nomi SQL non possono contenere **caratteri proibiti** (come spazi, parentesi, due punti, accenti e così via). Non si possono inoltre usare parole chiave (ad es. non potete nominare una colonna con il nome **SELECT** o **FROM**, poiché questi creerebbero ambiguità con i corrispondenti comandi SQL)
- comunque potete esplicitamente **mascherare** ogni nome proibito, così da renderlo pienamente legittimo. Per applicare questo mascheramento, dovete semplicemente racchiudere l'intero nome fra **doppi apici**.
- potete ovviamente racchiudere fra apici indiscriminatamente qualsiasi nome SQL name: questo è completamente innocuo, ma non è strettamente richiesto.
- nel caso (*estremamente raro, ma no impossibile*) il vostro nome proibito contenga già uno o due apici, dovete inserire un doppio apice supplementare per ognuno di essi, ad es.: **A"Bc"D** deve essere correttamente mascherato come: **"A""Bc""D"**.

```
SELECT COD_REG, REGIONE, POP2001
FROM reg2001_s
ORDER BY regione;
```

SQL consente di ordinare i risultati nel modo più conveniente secondo le vostre necessità.

```
SELECT COD_REG, REGIONE, POP2001
FROM reg2001_s
ORDER BY POP2001 DESC;
```

Potete ordinare in ordine ascendente o descendente a vostra scelta: la specifica ASC è normalmente omessa, perchè è l' ordinamento predefinito:

- l'ordine ascendente significa dall'A alla Z per i dati testuali: e da minore a maggiore per i valori numerici.
- l'ordine decrescente significa dall'Z alla A per i dati testuali: e da maggiore a minore per i valori numerici.

```
SELECT COD_PRO, PROVINCIA, SIGLA
FROM prov2001_s
WHERE COD_REG = 9;
```

Usando la clausola **WHERE** potete restringere la ricerca: solo le righe che soddisfano la clausola **WHERE** saranno visualizzate nei risultati: in questo caso, saranno estratte le Province appartenenti alla Regione Toscana.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE COD_PRO = 48;
```

Come il precedente: questa volta sarà estratta la lista dei Comuni appartenenti alla Provincia di Firenze.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE COD_REG = 9 AND POP2001 > 50000
ORDER BY POP2001 DESC;
```

Potete combinare più condizioni nella stessa clausola **WHERE**: questa volta sarà estratta la lista dei Comuni con oltre 50,000 abitanti appartenenti alla Regione Toscana. Il risultato sarà ordinato in modo decrescente.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE nome_com = 'ROMA';
```

Potete ovviamente usare delle stringhe di testo come valori di confronto: nel puro stile SQL ogni stringa di testo va delimitata da **singoli apici**. [SQLite è abbastanza furbo da riconoscere anche le stringhe di testo fra doppi apici, ma vi sconsiglio vivamente dall' adottare questo cattivo stile come vostro preferito].

Vi prego di osservare bene: i confronti fra stringhe di testo per SQLite sono sempre case-sensitive(**minuscolo/maiuscolo**).

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE nome_com = 'L''AQUILA';
```

Quando qualche testo contiene l' *apostrofo*, dovete **mascherarlo**. Un **singolo apice** è richiesto anche per mascherare ogni apostrofo all'interno del testo: ad es.: **REGGIO NELL' EMILIA** deve essere mascherata correttamente in **'REGGIO NELL' 'EMILIA'**.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE nome_com LIKE 'roma';
```

Potete usare l'operatore di valutazione approssimata **LIKE** per rendere il confronto di stringhe insensibile al maiuscole /minuscole.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE nome_com LIKE '%maria%';
```

E potete usare l'operatore **LIKE** per eseguire confronti parziali, usando il carattere **%** come jolly: questa interrogazione estrarrà ogni Comune contenente la sottostringa '**maria**' nel suo nome.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE nome_com IN ('ROMA', 'MILANO', 'NAPOLI');
```

Alcune volte può essere utile usare una lista di valori, come nel caso precedente.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE POP2001 BETWEEN 1990 AND 2010;
```

Un altro criterio, non molto usato, ma alle volte utile, è quello di definire l'intervallo di valori da cercare.

```
SELECT PROVINCIA, SIGLA, POP2001
FROM prov2001_s
WHERE COD_REG IN (9, 10, 11, 12)
AND SIGLA NOT IN ('LI', 'PI')
AND (POP2001 BETWEEN 300000 AND 500000
OR POP2001 > 750000);
```

Usando SQL potete scrivere clausole **WHERE** di ogni complessità: non ci sono limiti imposti. E questo è proprio una fantastica prestazione, che apre potenzialmente la strada ad infiniti scenari. Proprio una breve spiegazione: la precedente richiesta produrrà:

- l'inclusione di ogni Provincia dell'Italia Centrale (Regioni: Toscana, Umbria, Marche e Lazio)
- con l'esclusione delle Province di Livorno e Pisa
- quindi il filtro della popolazione:
 - includendo quelle nell'intervallo da 300,000 a 500,000
 - includendo anche quelle con oltre 750,000 abitanti.

```
SELECT PROVINCIA, SIGLA, POP2001
FROM prov2001_s
WHERE COD_REG IN (9, 10, 11, 12)
AND SIGLA NOT IN ('LI', 'PI')
AND POP2001 BETWEEN 300000 AND 500000
OR POP2001 > 750000;
```

Notate bene anche: in SQL il connettore logico **OR** ha una bassissima priorità.

Provate da soli: omettendo di inserire la clausola **OR** fra le parentesi produce risultati molto diversi, no ?

```
SELECT *
FROM com2001_s
LIMIT 10;
```

Vi è in fine una utile (a volte) clausola **SELECT** da spiegare: usando **LIMIT** potete stabilire il numero massimo di righe da estrarre (*molto spesso non siete in realtà interessati a leggere completamente una tabella troppo densamente popolata: una piccola preview sarà sufficiente in molti casi*).

```
SELECT *
FROM com2001_s
LIMIT 10 OFFSET 1000;
```

E questo non è tutto: SQL non vi costringe a leggere un numero limitato di righe partendo necessariamente dall' inizio: potete decidere il punto di partenza dove volete, semplicemente usando **OFFSET** combinato con **LIMIT**.

Imparare SQL non è difficile dopo tutto. Ci sono davvero poche parole chiave, la sintassi del linguaggio è notevolmente regolare e prevedibile, ed i comandi di ricerca sono progettati per assomigliare all'inglese (nei limiti del possibile ...).

Ora si presume siate abili per cercare di scrivere dei (semplici) comandi SQL da soli.



Febbraio 2011

Comprendere le funzioni di aggregazione

Abbiamo visto sinora come SQL consente di ricercare singoli valori in una tabella. E' consentita anche una modalità diversa, calcolare **valori totali** per l' intera tabella, o per gruppi di righe selezionate. Questo implica qualche tipo speciale di funzioni, conosciute come **funzioni di aggregazione**.

```
SELECT Min(POP2001), Max(POP2001),
       Avg(POP2001), Sum(POP2001), Count(*)
FROM com2001_s;
```

Questa interrogazione fornirà una sola riga, contenendo qualcosa come un riepilogo dell' intera tabella:

- la funzione **Min()** restituirà il valore più piccolo trovato nella colonna interessata,
- la funzione **Max()** restituirà il valore più grande trovato nella colonna interessata,
- la funzione **Avg()** restituirà il valore medio relativo alla colonna interrogata,
- la funzione **Sum()** restituirà la somma totale dei valori della corrispondente colonna,
- la funzione **Count()** restituirà il numero delle righe trovate.

```
SELECT COD_PRO, Min(POP2001), Max(POP2001),
       Avg(POP2001), Sum(POP2001), Count(*)
FROM com2001_s
GROUP BY COD_PRO;
```

Potete usare la clausola **GROUP BY** per ottenere un risultato più dettagliato di aggregazione per sottogruppi. Questa interrogazione restituirà i valori richiesti, distinti per Provincia.

```
SELECT COD_REG, Min(POP2001), Max(POP2001),
       Avg(POP2001), Sum(POP2001), Count(*)
FROM com2001_s
GROUP BY COD_REG;
```

Potete trovare anche i riepiloghi per Regione cambiando semplicemente il criterio di raggruppamento (**GROUP BY**).

```
SELECT DISTINCT COD_REG, COD_PRO
FROM com2001_s
ORDER BY COD_REG, COD_PRO;
```

C'è un altro modo di aggregare le righe, ad es. usando la clausola **DISTINCT**.

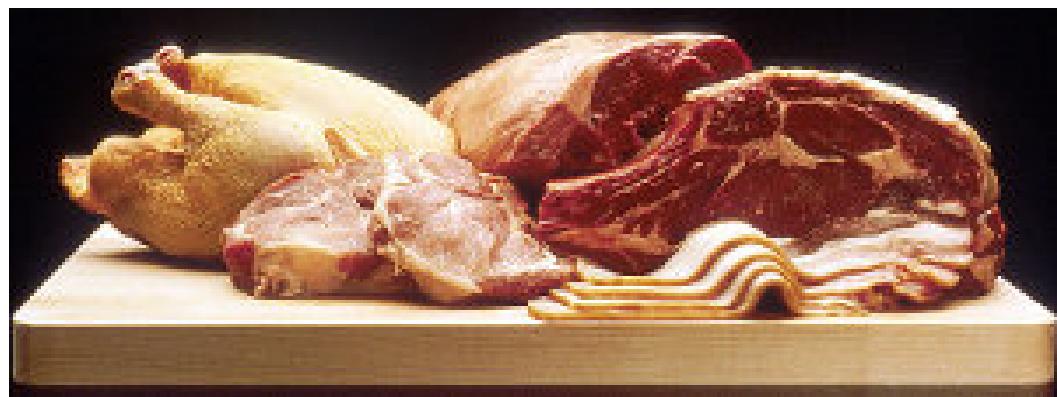
Prendete nota per favore: questo non è per niente uguale all' uso della clausola **GROUP BY**:

- la clausola **DISTINCT** sopprime semplicemente ogni riga duplicata, ma non ha niente a che vedere con le funzioni di aggregazione.
- la clausola **GROUP BY** è assolutamente necessaria ogni volta che aggregare le informazioni.



Ingredienti di uso comune

Febbraio 2011



La vostra prima interrogazione spaziale
Ancora sulle interrogazioni spaziali: WKT e WKB
Le tabelle MetaData spaziali
Visualizzare un layer SpatiaLite in QGis



La vostra prima interrogazione spaziale

SpatiaLite è uno **Spatial DBMS**, quindi è tempo di eseguire qualche interrogazione **Spatial SQL** [Spatial query]. Non c'è assolutamente niente di diverso in Spatial SQL: praticamente la stessa cosa dello standard SQL, la differenza è che adotti la esotica tipologia di dati **Geometry**. Normalmente non potete interrogare direttamente i valori Geometry (come abbiamo già visto essi sono semplicemente dei campi **BLOB** di tipo assolutamente generico e non qualificato): occorre usare alcune appropriate **funzioni spaziali** per accedere ai valori Geometry in modo significativo.

```
SELECT COD_REG, REGIONE, ST_Area(Geometry)
FROM reg2001_s;
```

La funzione **ST_Area()** è una delle funzioni Spatial; normalmente è facile riconoscere le funzioni spaziali perchè hanno tutte il prefisso **ST_**. Questa regola non è assoluta però: SpatiaLite è capace di interpretare l' alias **Area()** che identifica la stessa funzione.

Come si capisce dal nome, questa funzione calcola la superficie della corrispondente geometria [**Geometry**].

```
SELECT COD_REG AS code,
REGIONE AS name,
ST_Area(Geometry) / 1000000.0 AS "Surface (sq.Km)"
FROM reg2001_s
ORDER BY 3 DESC;
```

Come avrete sicuramente notato, l' interrogazione [**query**] precedente restituisce dei numeri molto grandi: questo è dovuto al fatto che l' insieme dei dati usati usa i **metri** come unità di lunghezza, e conseguentemente le superfici sono misurate in **m²**.

Ma basta applicare un appropriato fattore di scala per avere la più comoda unità in **km²**.

Notate per favore due caratteristiche SQL che introduciamo per la prima volta:

- SQL non impone la stampa diretta dei valori delle colonne nel risultato: voi potete definire liberamente qualsiasi valida espressione aritmetica a vostro piacimento.
- usare nella clausola ORDER BY espressioni complesse non è il massimo della praticità: potete invece riferirvi alle diverse colonne con l' identificativo della **relativa posizione** (la prima colonna ha codice 1 , e così via).

```
SELECT COD_REG AS code,
REGIONE AS name,
ST_Area(Geometry) / 1000000.0 AS "Surface (sq.Km)",
POP2001 / (ST_Area(Geometry) / 1000000.0)
AS "Density: Peoples / sq.Km"
FROM reg2001_s
ORDER BY 4 DESC;
```

E potete eseguire anche più complesse interrogazioni in SQL.

Questa interrogazione calcolerà la **densità di popolazione** (misurata come **abitanti / km²**).

Bene, ora avete acquisito una conoscenza basilare di SQL / Spatial SQL. Ora siete pronti per misurarvi con più complesse ed efficaci interrogazioni: ma questo richiede la costruzione di un database serio.

Ricordate ? finora abbiamo usato tabelle di Virtual Shapefiles; cioè una imitazione di vere tabelle Spatial (**memorizzate internamente**).

Così nei prossimi passi, prima creeremo e popoleremo un **DB ben costruito** (un compito non proprio banale), e poi vedremo ancora interrogazioni SQL più complesse e potenti.

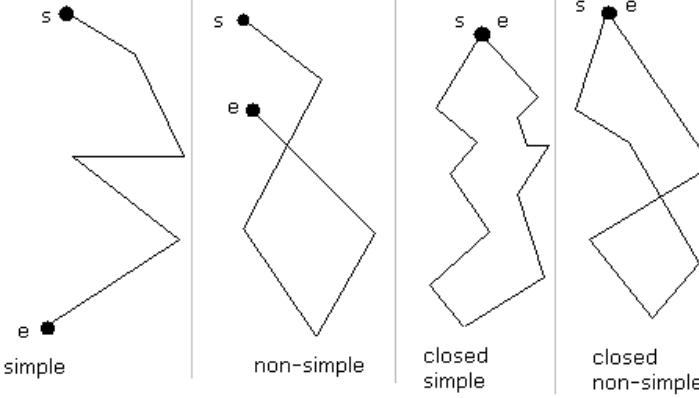
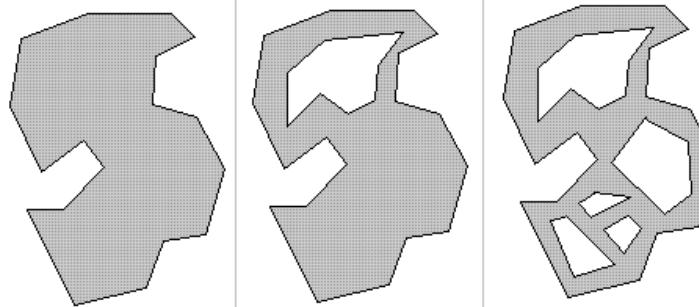


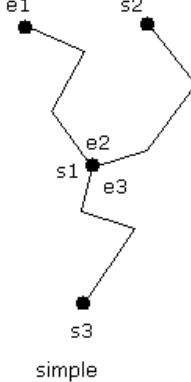
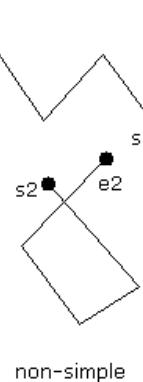
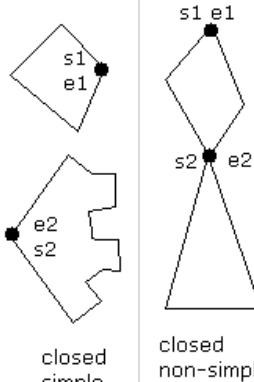
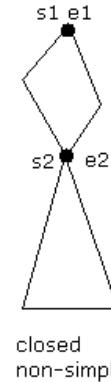
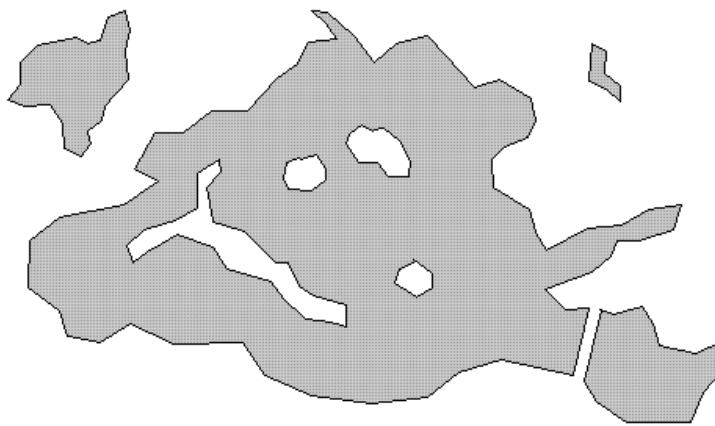
Febbraio 2011

Ancora sulle interrogazioni spaziali: WKT e WKB

SpatiaLite gestisce dati di tipo **Geometry** [geometrico] conformi allo standard internazionale **OGC-SFS** (*Open Geospatial Consortium - Simple Feature SQL*). <http://www.opengeospatial.org/standards/sfs>. Il tipo **Geometry** è un tipo **astratto** con sette sottoclassi.

Non potete usare direttamente il tipo **Geometry** (a causa della sua natura **astratta** alla quale non corrisponde alcuna implementazione): mentre potete usare liberamente qualsiasi **sottoclasse**.

Sub-Class	Esempio
POINT	 POINT
LINESTRING	 LINESTRINGS
POLYGON	 POLYGONS
MULTIPOINT	 MULTIPOINT

MULTILINESTRING				
	MULTILINESTRINGS			
MULTIPOLYGON				
GEOMETRYCOLLECTION	<p>qualsiasi arbitrario insieme di sottoclassi elementari.</p> <p>Notate, per favore: per qualche oscura ragione questa sembra la sottoclasse di gran lunga preferita dai <i>principianti</i>: tutti si basano su GEOMETRYCOLLECTION:</p> <ul style="list-style-type: none"> - la sottoclasse GEOMETRYCOLLECTION non è prevista dal formato <i>Shapefile</i>. - E questa sottoclasse non è generalmente gestita da normali applicativi GIS (<i>visualizzatori e così via</i>). <p>Per questo è molto raramente usata nel mondo professionale GIS.</p>			

Avvertenze WKT and WKB

Il tipo **Geometry** è un dato molto complesso: per questo, OGC-SFS definisce due standard alternativi per rappresentare valori Geometry:....

- il formato **WKT** (*Well Known Text*) è progettato per essere amichevole e facile da usare (*non così facile dopo tutto, ma almeno facilmente leggibile dagli esseri umani*)....
- il formato **WKB** (*Well Known Binary*) d' altro lato è più orientato per la precisa ed accurata **importazione/esportazione/scambio** di Geometries [Geometrie] fra piattaforme differenti.

Il formato **XY (2D)** (quello più comunemente usato):

Geometry Type	WKT example
POINT	POINT(123.45 543.21)
LINESTRING	LINESTRING(100.0 200.0, 201.5 102.5, 1234.56 123.89) <i>three vertices [tre vertici]</i>
POLYGON	POLYGON((101.23 171.82, 201.32 101.5, 215.7 201.953, 101.23 171.82)) <i>exterior ring, no interior rings [anello esterno, nessun anello interno]</i>
	POLYGON((10 10, 20 10, 20 20, 10 20, 10 10), (13 13, 17 13, 17 17, 13 17, 13 13)) <i>exterior ring, one interior ring [anello esterno, un anello interno]</i>
MULTIPOINT	MULTIPOINT(1234.56 6543.21, 1 2, 3 4, 65.21 124.78) <i>three points [tre punti]</i>
MULTILINESTRING	MULTILINESTRING((1 2, 3 4), (5 6, 7 8, 9 10), (11 12, 13 14)) <i>first and last linestrings have 2 vertices each one; [la priama e l'ultima hanno 2 vertici] the second linestring has 3 vertices [la seconda ha 3 vertici]</i>
MULTIPOLYGON	MULTIPOLYGON(((0 0, 10 20, 30 40, 0 0), (1 1, 2 2, 3 3, 1 1)), ((100 100, 110 110, 120 120, 100 100))) <i>two polygons: the first one has an interior ring [due poligoni il primo ha un anello interno]</i>
GEOMETRYCOLLECTION	GEOMETRYCOLLECTION(POINT(1 1), LINESTRING(4 5, 6 7, 8 9), POINT(30 30))

Il formato **XYZ (3D)**:

Geometry Type	WKT example
POINT	POINTZ(13.21 47.21 0.21)
LINESTRING	LINESTRINGZ(15.21 57.58 0.31, 15.81 57.12 0.33)
POLYGON	...
MULTIPOINT	MULTIPOINTZ(15.21 57.58 0.31, 15.81 57.12 0.33)
MULTILINESTRING	...
MULTIPOLYGON	...
GEOMETRYCOLLECTION	GEOMETRYCOLLECTIONZ(POINTZ(13.21 47.21 0.21), LINESTRINGZ(15.21 57.58 0.31, 15.81 57.12 0.33))

Il formato **XYM (2D + Misura)** (Avvertenza: questa non ha niente a che fare con il 3D.

M è un valore di **misura**, non una dimensione geometrica):

Geometry Type	WKT example
POINT	POINTM(13.21 47.21 1000.0)
LINESTRING	LINESTRINGM(15.21 57.58 1000.0, 15.81 57.12 1100.0)
POLYGON	...
MULTIPOINT	MULTIPOINTM(15.21 57.58 1000.0, 15.81 57.12 1100.0)
MULTILINESTRING	...
MULTIPOLYGON	...
GEOMETRYCOLLECTION	GEOMETRYCOLLECTIONM(POINTM(13.21 47.21 1000.0), LINESTRINGM(15.21 57.58 1000.0, 15.81 57.12 1100.0))

Il formato **XYZM (3D + Misura)** (Avvertenza: questa non ha niente a che fare con il 3D. **M** è un valore di **misura**, non una dimensione geometrica):

Geometry Type	WKT example
POINT	POINTZM(13.21 47.21 0.21 1000.0)
LINESTRING	LINESTRINGZM(15.21 57.58 0.31 1000.0, 15.81 57.12 0.33 1100.0)
POLYGON	...
MULTIPOINT	MULTIPOINTZM(15.21 57.58 0.31 1000.0, 15.81 57.12 0.33 1100.0)
MULTILINESTRING	...
MULTIPOLYGON	...
GEOMETRYCOLLECTION	GEOMETRYCOLLECTIONZM(POINTZM(13.21 47.21 0.21 1000.0), LINESTRINGZM(15.21 57.58 0.31 1000.0, 15.81 57.12 0.33 1100.0))

Vi sono parecchie funzioni Spatial SQL che gestiscono i formati **WKT** e **WKB**; esaminarle tutte una ad una è assolutamente noioso (*e nemmeno utile per l'utente medio*). Così vedremo brevemente solo le principali (*ed anche le più comunemente usate*).

```
SELECT Hex(ST_GeomFromText('POINT(1.2345 2.3456)'));

0001FFFFFFFF8D976E1283C0F33F16FBCBEEC9C302408D976E1283C0F33F16FBCBEEC9C302407C010000008D976E1283C0F33F16FBCBEEC9C30240FE

SELECT
ST_AsText(x'0001FFFFFFFF8D976E1283C0F33F16FBCBEEC9C302408D976E1283C0F33F16FBCBEEC9C302407C010000008D976E1283C0F33F16FBCBEEC9C30240FE');

POINT(1.2345 2.3456)

SELECT
Hex(ST_AsBinary(x'0001FFFFFFFF8D976E1283C0F33F16FBCBEEC9C302408D976E1283C0F33F16FBCBEEC9C302407C010000008D976E1283C0F33F16FBCBEEC9C30240FE'));

01010000008D976E1283C0F33F16FBCBEEC9C30240

SELECT Hex(ST_AsBinary(ST_GeomFromText('POINT(1.2345 2.3456)')));

01010000008D976E1283C0F33F16FBCBEEC9C30240

SELECT ST_AsText(ST_GeomFromWKB(x'01010000008D976E1283C0F33F16FBCBEEC9C30240'));

POINT(1.2345 2.3456)
```

Per favore, notate bene: entrambi i formati **WKT** and **WKB** sono fatti per gestire lo scambio dei dati (importazione/esportazione); ma il vero formato usato internamente da SpatialLite è differente, cioè **BLOB Geometry**.....Non vi dovete mai preoccupare di questo *formato interno*: dovete semplicemente usare le appropriate funzioni di conversione dagli standard **WKT** e **WKB**.....

- la funzione **Hex()** è una funzione standard SQL che consente di rappresentare valori binari come stringhe di testo in formato esadecimale.
- la funzione Spatial SQL **ST_GeomFromText()** converte ogni valida espressione **WKT** nel formato **BLOB Geometry** interno.
- la funzione **ST_GeomFromWKB()** converte ogni valida espressione **WKB** nel formato **BLOB Geometry** interno.
- la funzione **ST_AsText()** converte il valore dal formato interno **BLOB Geometry** nella corrispondente espressione **WKT**.
- la funzione **ST_AsBinary()** converte il valore dal formato interno **BLOB Geometry** nella corrispondente espressione **WKB**.

SELECT ST_GeometryType(ST_GeomFromText('POINT(1.2345 2.3456)'));
POINT
SELECT ST_GeometryType(ST_GeomFromText('POINTZ(1.2345 2.3456 10)'));
POINT Z
SELECT ST_GeometryType(ST_GeomFromText('POINT ZM(1.2345 2.3456 10 20)'));
POINT ZM

La funzione **ST_GeometryType()** restituirà il tipo di geometria [Geometry] dal formato interno **BLOB Geometry**.....

Annotate: quando usate dati non bidimensionali [dimensioni non 2D], dichiarare '**POINTZ**' o '**POINT Z**' è assolutamente lo stesso: SpatiaLite riconosce entrambe le notazioni.

SELECT ST_Srid(ST_GeomFromText('POINT(1.2345 2.3456)'));
-1
SELECT ST_Srid(ST_GeomFromText('POINT(1.2345 2.3456)', 4326));
4326

La funzione **ST_Srid()** restituisce il codice SRID dal formato BLOB Geometry.

Annotate: sia la funzione **ST_GeomFromText()** che la **ST_GeomFromWKB()** accetta un argomento **SRID** opzionale. Se il codice **SRID** non è specificato (*una pratica per niente corretta*), allora assume **-1**.

Errori più frequenti

"Ho dichiarato una geometria di tipo **MULTIPOINT**; ora devo inserire un semplice **POINT** [punto] nella tabella, ma ricevo un segnale di errore ..."

Ogni tipo **MULTIxxxx** può memorizzare elementi singoli: dovete solo usare la corretta sintassi **WKT**. Comunque esistono parecchie utili funzioni di **conversione di tipo**.

SELECT ST_GeometryType(ST_GeomFromText('MULTIPOINT(1.2345 2.3456)'));
MULTIPOINT
SELECT ST_AsText(CastToMultiLineString(ST_GeomFromText('LINESTRING(1.2345 2.3456, 12.3456 23.4567)')));
MULTILINESTRING((1.2345 2.3456, 12.3456 23.4567))
SELECT ST_AsText(CastToXYZM(ST_GeomFromText('POINT(1.2345 2.3456)')));
POINT ZM(1.2345 2.3456 0 0)
SELECT ST_AsText(CastToXY(ST_GeomFromText('POINT ZM(1.2345 2.3456 10 20)')));
POINT(1.2345 2.3456)



Le tabelle MetaData spaziali

Febbraio 2011

SpatiaLite richiede l' uso di parecchie ***tabelle metadata*** per funzionare correttamente. Non c'è assolutamente nulla di strano in queste tabelle; esse sono semplici tabelle come tutte le altre.

Esse sono chiamate complessivamente ***metadati*** perchè sono progettate per consentire un esteso e completo uso delle Geometries.

Quasi tutte le funzioni Spatial SQL si appoggiano su tali tabelle: per questo sono assolutamente necessarie per motivi di funzionamento interno.

Qualsiasi tentativo di ***modificare*** questa tabelle finirà per rendere il database corrotto (*e mal funzionante*).

C' è un solo modo sicuro di interagire con le ***tabelle metadata***, ad es. usando per quanto possibile le appropriate funzioni Spatial SQL.

L'uso diretto delle funzioni **`INSERT`**, **`UPDATE`** or **`DELETE`** è una modalità completamente **insicura e fortemente scoraggiata**.

```
SELECT InitSpatialMetaData();
```

La funzione **`InitSpatialMetaData()`** deve essere eseguita immediatamente dopo la creazione di un nuovo database, e prima di eseguire qualsiasi altra funzione Spatial SQL:

- la funzione di questo comando è esattamente quello di creare (*e popolare*) ogni *tabella metadata* richiesta da SpatiaLite per motivi interni.
- se qualche *tabella metadata* esiste già, la funzione non esegue alcuna operazione: quindi la ripetuta esecuzione di **`InitSpatialMetaData()`** è inutile ma completamente inoffensiva.
- **notate per favore:** l'applicativo **`spatialite_gui`** eseguirà automaticamente qualsiasi inizializzazione richiesta ogni volta che si crea un nuovo database: cioè, (*usando questo strumento*) non c' è alcun bisogno di chiamare esplicitamente questa funzione.

```
SELECT *
FROM spatial_ref_sys;
```

SRID	auth_name	auth_srid	ref_sys_name	proj4text	srs_wkt
2000	EPSG	2000	Anguilla 1957 / Grid British West Indies	+ Proj = tmerc + lat_0 = 0 + lon_0 = - 62 + k = 0,9995000000000001 + x_0 = 400.000 + y_0 = 0 + ellps = clrk80 + unità = m + no_defs	PROJCS "Anguilla 1957 / Grid British West Indies" [, GEOGCS ["Anguilla 1957", DATUM ["Anguilla_1957", Sferoide ["Clarke 1880 (RGS)", 6378249.145,293.465, AUTHORITY ["EPSG", "7.012"]], AUTHORITY ["EPSG", "6600"]], PRIMEM ["Greenwich", 0, AUTHORITY ["EPSG", "8.901"]], UNITA ["grado", 0,01745329251994328, AUTHORITY ["EPSG", "9.122"]], AUTHORITY ["EPSG", "4600"]], UNIT ["Meter", 1, AUTHORITY ["EPSG", "9001"]], PROIEZIONE ["Transverse_Mercator"], PARAMETER ["latitude_of_origin", 0], PARAMETER ["central_meridian", -62], PARAMETER ["scale_factor", 0,9,995 mila], PARAMETER ["false_easting", 400000], PARAMETER ["false_northing", 0], AUTHORITY ["EPSG", "2000"], AXIS ["Est", EST], AXIS ["Nord e Quota", a nord]]
...

La tabella **`spatial_ref_sys`** contiene l'intero **archivio EPSG** (*definizioni dei Spatial Reference System*).

- la colonna **SRID** è la chiave primaria (**PRIMARY KEY**) che identifica univocamente ciascun sistema.
- le colonne **auth_name < auth_srid e ref_sys_name** contengono normalmente un riferimento alla definizione originaria EPSG (*principalmente per motivi di documentazione*).
- la colonna **proj4text** contiene i **parametri geodesici** (???) richiesti dalla libreria **PROJ.4**.
 - questi parametri sono assolutamente necessari alla funzione **Transform()**, perchè ogni riproiezione di coordinate sarà eseguita invocando l' appropriata funzione **PROJ.4**.
- la colonna **srs_wkt** contiene una definizione completa del **SRS** usando il formato (*ovviamente verboso*) **WKT**.
 - SpatiaLite per sè non richiede che questa informazione sia presente: ma quando questa stringa **WKT** è disponibile, allora creerà un file **.PRJ** quando esporta in formato Shapefile (*molti applicativi GIS richiedono la presenza di un file .PRJ per ogni Shapefile*).
 - per evitare confusioni: questo formato **WKT** per i **SRS** non ha niente a che vedere con il più noto **WKT** usato per rappresentare le geometrie.
- **avviso importante:** non è sicuro e fortemente scoraggiato modificare la definizione EPSG originale, e deve essere assolutamente evitato. Comunque siete assolutamente liberi di inserire definizioni personalizzate a piacimento: in questo caso è fortemente consigliato usare codici **SRID > 32768**.

```
SELECT *
FROM geometry_columns;
```

f_table_name	f_geometry_column	type	coord_dimension	SRID	spatial_index_enabled
local_councils	geometry	MULTIPOLYGON	XY	23032	1
populated_places	geometry	POINT	XY	4326	1

La tabella **`geometry_columns`** descrive ogni Geometry column definita nel database:

- ogni colonna non definita da una corrispondente riga di questa tabella, non può essere considerata una Geometry corretta.
- **AVVISO IMPORTANTE:** ogni inquinamento di questa tabella causato dall' uso diretto dei comandi **INSERT**, **UPDATE** or **DELETE** finirà quasi sicuramente in un disastro (*ad es. un database corrotto e malfunzionante*). Usate piuttosto le appropriate funzioni SQL: **AddGeometryColumn()** , **RecoverGeometryColumn()** e così via.

La tabella **`geometry_columns`** è progettata per la gestione di tabelle normali. Inoltre esistono altre due tabelle simili:

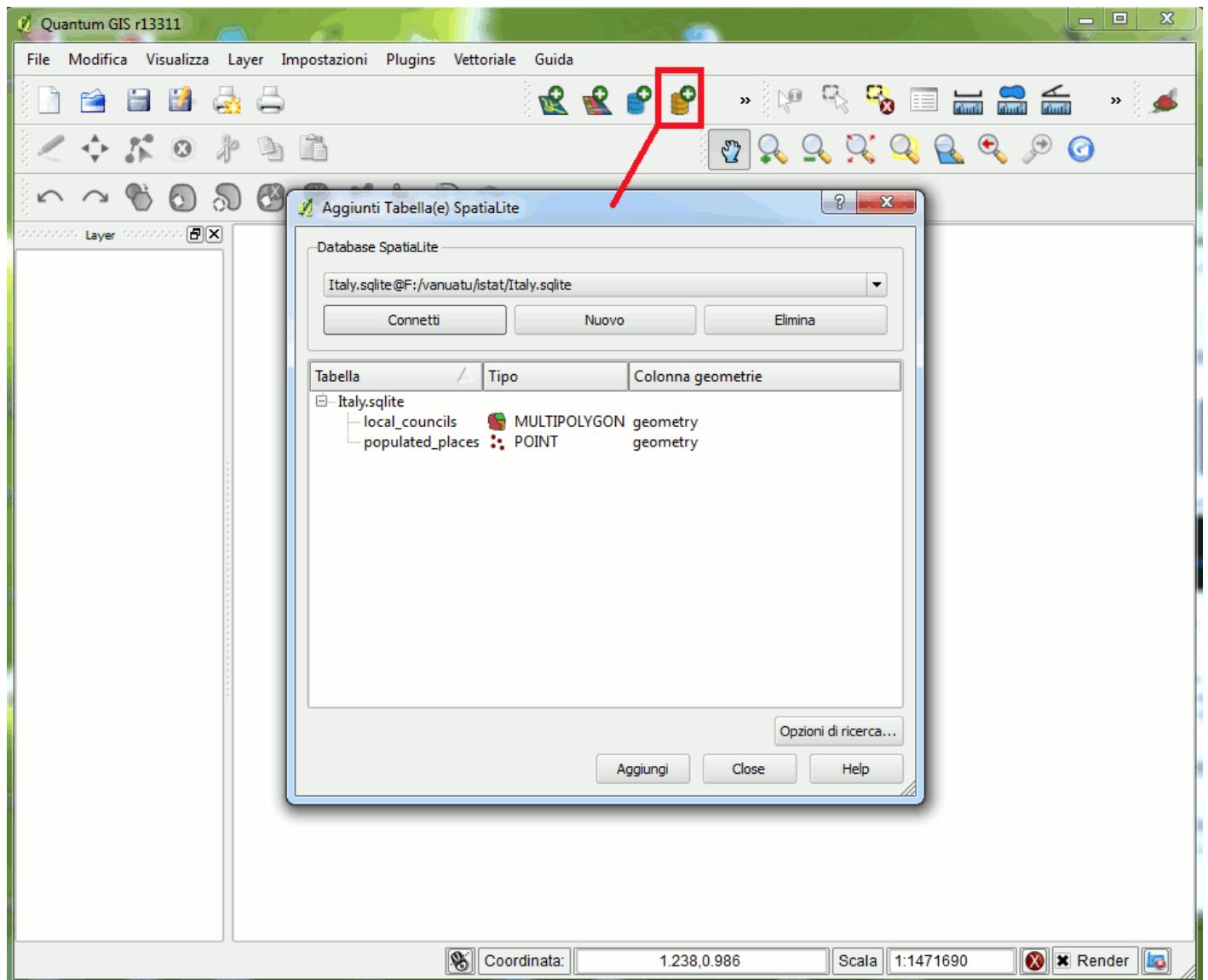
- la tabella **`views_geometry_columns`** è progettata per consentire le Geometry VIEW
- e la **`virts_geometry_columns`** per gestire i Virtual Shapefiles.



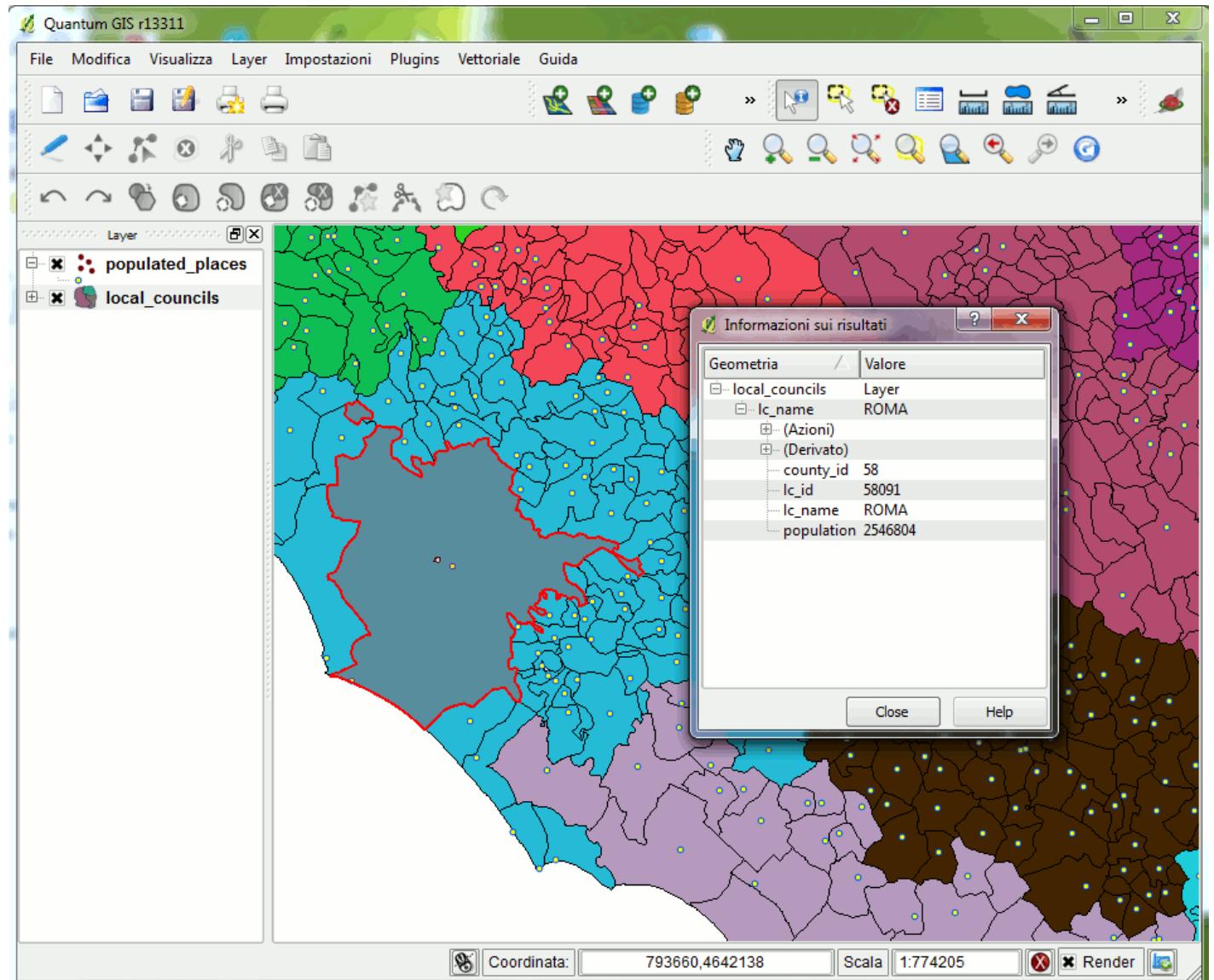
Visualizzare un layer SpatiaLite in QGis

QGIS è un applicativo GIS desktop popolare e largamente diffuso: potete scaricare l' ultima versione da: [http://www.qgis.org/.....](http://www.qgis.org/)

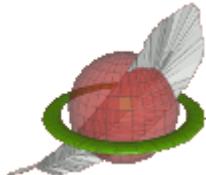
QGIS contiene un gestore interno di dati che abilita SpatiaLite: in questo modo è facile e semplice interagire con qualsiasi DB SpatiaLite usando un programma GIS classico per desktop.



Dovete semplicemente connettervi con il DB SpatiaLite, quindi scegliere lo/gli strato/i che volete usare.
Annotate: secondo la terminologia DBMS voi state usando delle **tabelle**. Ma nel gergo GIS si usa molto spesso il termine strato (**layer**) per indicare la stessa cosa.



Una volta connessi ai vostri strati [layers] del DB SpatiaLite potete immediatamente iniziare ad usare il vostro QGIS. E' tutto.



Febbraio 2011

Cucina familiare



Ricetta # 1: Creare un DB ben fatto

Ricetta # 2: Le vostre prime interrogazioni JOIN

Ricetta # 3: Maggiori informazioni su JOIN

Ricetta # 4: A proposito di VIEW

Ricetta # 5: Creazione di una nuova tabella (con annessi e connessi)

Ricetta # 6: Creare una nuova colonna GEOMETRY

Ricetta # 7: Inserire, aggiornare e cancellare

Ricetta # 8: Conoscere i vincoli

Ricetta # 9: ACIDity: conoscere le transazioni

Ricetta # 10: La bellezza dell'indice R*Tree spaziale



Ricetta # 1:

Creare un DB ben fatto

Febbraio 2011

Forma Normale

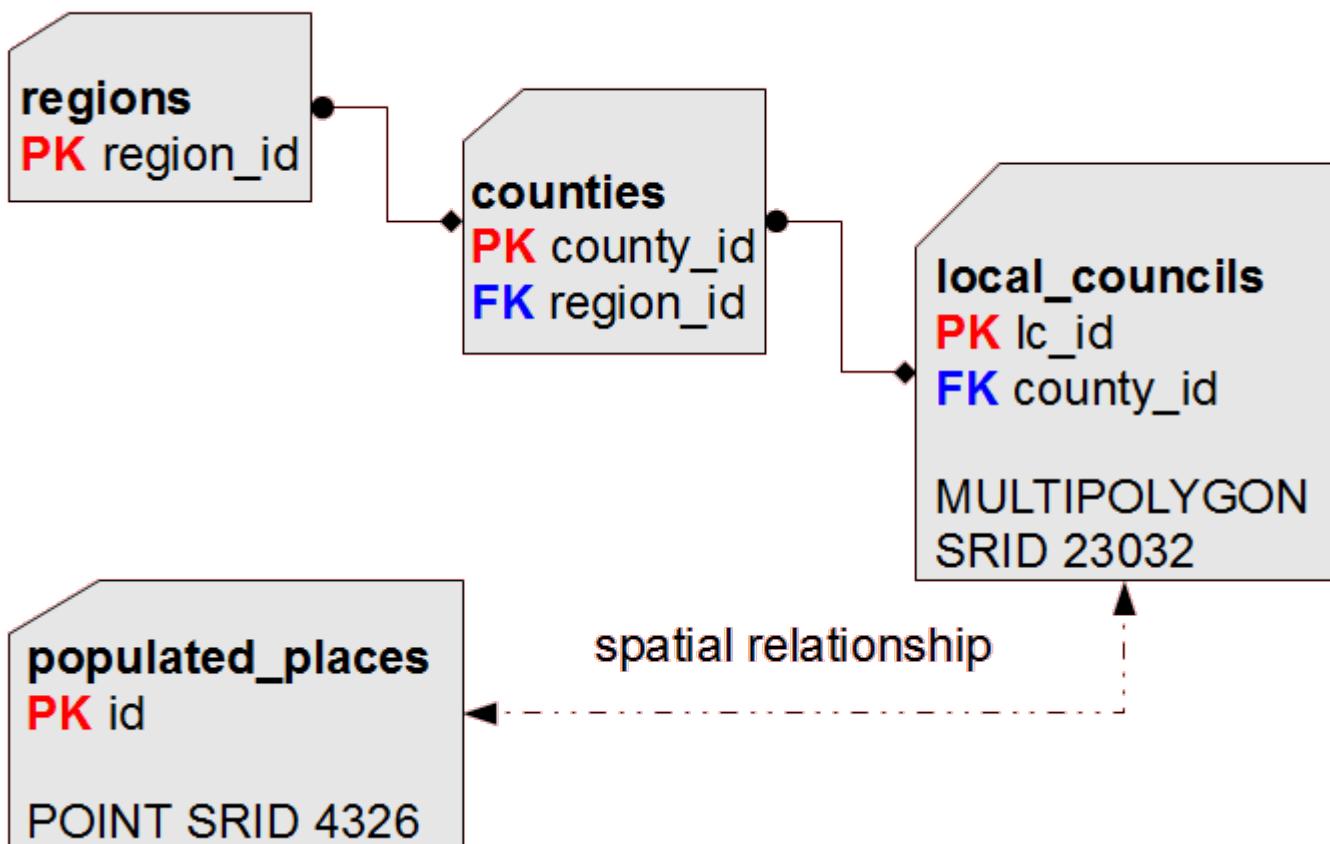
Ogni DB ben progettato aderisce ad un paradigma **relazionale**, ed implementa le cosiddette *Forme Normali* [Normal Form].

Spiegato molto semplicemente con parole chiare:

- prima cercate di identificare ogni distinta **categoria** (cioè classe) presente nel vostro insieme di dati
- nello stesso tempo dovete identificare ogni possibile **relazione** che collega le categorie.
- la **ridondanza** dei dati è fortemente sconsigliata, e deve essere evitata quando possibile.

Considerate i dati del **Censimento ISTAT 2001**; identificare le categorie e le relazioni è del tutto semplice:

- al più basso livello della gerarchia abbiamo ovviamente i Comuni [Local Councils].
- Ogni Comune appartiene sicuramente ad una Provincia [County]: quindi esiste una relazione fra Comuni e Provincia. Per essere più precisi, questa è una relazione **uno a molti** [one-to-many] (una singola Provincia - molti Comuni: mettere lo stesso Comune in più Province è assolutamente vietato).
- Lo stesso è vero per Province e Regioni.
- Non c'è un vero bisogno di collegare i Comuni con le Regioni perché possiamo stabilire questa relazione attraverso le Province.



Detto ciò, è abbastanza facile identificare diversi limitazioni concettuali nel formato originale Shapefile:

1. per ogni Comune, Provincia e Regione è presente un valore **POP2001**: bene, questa è chiaramente una ridondanza non necessaria. E' sufficiente conservare questa informazione al livello più basso (Comuni) poiché possiamo sempre calcolare il valore aggregato per le Province (o le Regioni).
2. Esiste una seconda ridondanza: non c'è necessità di conservare il doppio codice Provincia e Regione per ogni Comune. Il codice della Provincia è sufficiente, perché è possibile riferirsi alla Regione semplicemente attraverso la Provincia.
3. Sia le Province che le Regioni hanno una geometria associata: anche questa rappresenta una ridondanza inutile, poiché possiamo ottenere tali geometrie semplicemente aggregando quelle associate ai Comuni.

Poi abbiamo l'archivio **cities1000**: che arriva da una fonte completamente diversa (per questo non è disponibile una chiave per stabilire una relazione ad altre entità).

Inoltre questo archivio è nel sistema 4326 (**WGS84**), mentre il **Censimento ISTAT 2001** è nel sistema 23032 [**ED50 UTM zona 32**]; quindi per adesso possiamo usare questo archivio da solo.

Vedremo successivamente come in realtà possiamo integrare questo archivio con gli altri: dopo tutto, tutti rappresentano l' Italia, no? Di sicuro qualche relazione geografica deve esistere ...

```
CREATE TABLE regions (
    region_id INTEGER NOT NULL PRIMARY KEY,
    region_name TEXT NOT NULL);
```

Passo 1a) inizieremo a creare la tabella delle **regioni** (cioè quella situata al livello più alto della gerarchia).

Notate: abbiamo definito una **PRIMARY KEY**, cioè un identificatore unico (non duplicabile), assolutamente inequivocabile per ogni Regione.

```
INSERT INTO regions (region_id, region_name)
SELECT COD_REG, REGIONE
FROM reg2001_s;
```

Passo 1b) quindi riempiamo la tabella delle **regioni [regions]**.

Usare il comando **INSERT INTO ... SELECT ...** è più o meno come fare una copia: le righe estratte dalla tabella di input sono subito inserite nella tabella di output. Come potete vedere, le corrispondenti colonne sono identificate *per ordine*.

```
CREATE TABLE counties (
    county_id INTEGER NOT NULL PRIMARY KEY,
    county_name TEXT NOT NULL,
    car_plate_code TEXT NOT NULL,
    region_id INTEGER NOT NULL,
    CONSTRAINT fk_county_region
        FOREIGN KEY (region_id)
        REFERENCES regions (region_id));
```

```
INSERT INTO counties (county_id, county_name,
    car_plate_code, region_id)
SELECT cod_pro, provincia, sigla, cod_reg
FROM prov2001_s;
```

Passo 2a) creiamo (e popoliamo) la **tabella** delle provincie [counties]

Annotate: c'è una relazione fra **provincie e regioni**.

Definendo una appropriata **FOREIGN KEY** renderemo tale relazione esplicitamente chiara una volta per tutte.

```
CREATE INDEX idx_county_region
    ON counties (region_id);
```

Passo 2b) per motivi di prestazioni, dobbiamo anche creare un **INDEX** corrispondente ad ogni **FOREIGN KEY** che definiamo.

In parole povere: un **PRIMARY KEY** non è soltanto un vincolo logico.

La definizione di un **PRIMARY KEY** in SQLite implica l' automatica generazione di un indice implicito per gestire l' accesso rapido ad ogni singola riga. Ma d' altro lato la definizione di un **FOREIGN KEY** definisce un semplice vincolo logico: così se volete gestire l' accesso rapido ad ogni singola riga dovete creare esplicitamente l'indice corrispondente.

```
CREATE TABLE local_councils (
    lc_id INTEGER NOT NULL PRIMARY KEY,
    lc_name TEXT NOT NULL,
    population INTEGER NOT NULL,
    county_id INTEGER NOT NULL,
    CONSTRAINT fk_lc_county
        FOREIGN KEY (county_id)
    REFERENCES counties (county_id));
```

```
CREATE INDEX idx_lc_county
ON local_councils (county_id);
```

Passo 3a) creiamo adesso la tabella dei **comuni** [**local_councils**].

C'è una relazione che lega i **comuni** e le **provincie**.

Pertanto in questo caso definiamo un **FOREIGN KEY**, e poi creiamo il corrispondente indice.

Annotate: non abbiamo definito alcuna colonna di tipo Geometry, nonostante sia richiesta per i **comuni**; non è un errore, è assolutamente voluto.

```
SELECT AddGeometryColumn(
    'local_councils', 'geometry',
    23032, 'MULTIPOLYGON', 'XY');
```

Passo 3b) la creazione di una colonna Geometry non è lo stesso di ogni altra colonna normale.

Dobbiamo usare la funzione spaziale **AddGeometryColumn()**, specificando:

1. il nome della **tabella**
2. il nome della colonna di tipo **geometrico**
3. il codice **SRID** da usare.
4. la **classe di geometria** voluta
5. la dimensione del **modello** (in questo caso, semplice 2D).

```
INSERT INTO local_councils (lc_id,
    lc_name, population, county_id, geometry)
SELECT PRO_COM, NOME_COM, POP2001,
    COD_PRO, Geometry
FROM com2001_s;
```

Passo 3c) in seguito si può popolare la tabella dei **comuni** come di consueto.

```
CREATE TABLE populated_places (
    id INTEGER NOT NULL
        PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL);
```

```
SELECT AddGeometryColumn(
    'populated_places', 'geometry',
    4326, 'POINT', 'XY');
```

```
INSERT INTO populated_places (id,
    name, geometry)
SELECT NULL, COL002,
    MakePoint(COL006, COL005, 4326)
FROM cities1000
WHERE COL009 = 'IT';
```

Passo 4) dovete ora fare l'ultimo passo: creare (e popolare) la tabella **populated_places** [centri abitati].

Ci sono parecchi punti interessanti da studiare:

- abbiamo usato una clausola **AUTOINCREMENT** per il **PRIMARY KEY**.
- questo significa praticamente che SQLite può generare automaticamente il valore adatto per il **PRIMARY KEY**, quando non è stato esplicitamente definito il suo valore.
- per questo motivo, il comando **INSERT INTO** contiene un valore **NULL** in corrispondenza del **PRIMARY KEY**: e questo obbliga SQLite ad assegnare automaticamente i valori.
- l' archivio **cities1000** è distribuito in origine con due colonne numeriche per la **longitudine** [**COL006**] e la **latitudine** [**COL005**]: così abbiamo usato la funzione spaziale **MakePoint()** in modo da costruire una Geometry di tipo punto.
- usando lo **SRID 4326** collichiamo questa Geometry nell' **SRS WGS84** [Geographic System].

Giusto per ricapitolare:

- avete iniziato questa guida usando tabelle di **Virtual Shapefiles** (e di **Virtual CSV/TXT**)
- tali **Tabelle Virtuali** non sono delle vere tabelle DB: esse non sono *memorizzate internamente*. Esse sono semplicemente dei files esterni cui si accede tramite appropriati driver.
- Usando le Tabelle Virtuali potete da subito testare qualche semplice e basilare istruzione SQL.
- Ma per eseguire caratteristiche SQL più sofisticate i dati devono essere **adeguatamente importati** in un DBMS
- E questo passo richiede la creazione (ed il popolamento) delle *tabelle interne*, di pari passo con una struttura ben progettata.

```
DROP TABLE com2001_s;
DROP TABLE prov2001_s;
DROP TABLE reg2001_s;
DROP TABLE cities1000;
```

Passo 5) e finalmente potete eliminare qualsiasi **Tabella Virtuale**, perchè non sono più necessarie.

Prendete nota: eliminando un *Virtual Shapefile* o *Virtual CSV/TXT* non eliminate le corrispondenti sorgenti esterne, ma semplicemente rimuove la connessione con il database corrente.



Febbraio 2011

Ricetta # 2:

le vostre prime Interrogazioni JOIN

Avete appena imparato le nozioni base sulle interrogazioni SQL semplici.

Tutti gli esempi incontrati in precedenza interrogavano una singola tabella: ma SQL non impone dei limiti, quindi potere interrogare un numero arbitrario di tabelle nello stesso momento. Ma per fare ciò dovete ancora conoscere come usare correttamente un **JOIN**.

```
SELECT *
FROM counties, regions;
```

county_id	county_name	car_plate_code	region_id	region_id	region_name
1	TORINO	A	1	1	PIEMONTE
1	TORINO	A	1	2	VALLE D'AOSTA
1	TORINO	A	1	3	LOMBARDIA
1	TORINO	A	1	4	TRENTINO-ALTO ADIGE
1	TORINO	A	1	5	VENETO
...

Apparentemente questa interrogazione funziona alla grande; ma appena date uno sguardo ai risultati vi accorgerete che c'è qualcosa che in effetti non va: c'è un numero esagerato di righe ed ogni singola Provincia sembra appartenere a tutte le Regioni.

Ogni volta che SQL interroga contemporaneamente due differenti tabelle, restituisce il **Prodotto Cartesiano** di entrambi gli insiemi; cioè ogni riga di un insieme viene collegata con ogni riga dell'altro insieme. Questo è un procedimento combinatorio *alla cieca*, e molto difficilmente produrrà qualche risultato utile. Questo fatto inoltre può produrre un **risultato molto esteso**: deve essere assolutamente evitato, poiché:

- può richiedere *molto, molto (molto,molto)* tempo per essere portato a termine.
- può accadere di esaurire le risorse del sistema operativo prima della fine.

Detto questo, è quasi ovvio che adatte **condizioni di JOIN** devono essere definite per mantenere sotto controllo il Prodotto Cartesiano, così da ottenere solo informazioni utili.

```
SELECT *
FROM counties, regions
WHERE counties.region_id = regions.region_id;
```

Questa interrogazione è esattamente quella di prima, ma questa volta abbiamo introdotto la **condizione JOIN**. Qualche punto da notare:

- l'uso di due (*o più*) tabelle può facilmente portare ad *ambiguità*: ad es. in questo caso abbiamo due diverse colonne di nome **region_id**, una nella tabella **Provincie**, l'altra in quella **Regioni**.
- dovete usare dei *nomi completamente qualificati* per evitare possibili ambiguità:
ad es. **counties.region_id** identifica la **region_id** colonna appartenente alla **counties** table [tabella Regioni], in modo assolutamente inevocabile.
- definire il **WHERE counties.region_id = regions.region_id** clausola che impone un adeguato condizione di join.

Così il Prodotto Cartesiano sarà filtrato ed inserite nei risultati solo le righe che soddisfano la condizione imposta, tutte le altre saranno ignorate.

```
SELECT c.county_id AS county_id,
       c.county_name AS county_name,
       c.car_plate_code AS car_plate_code,
       r.region_id AS region_id,
       r.region_name AS region_name
  FROM counties AS c,
       regions AS r
 WHERE c.region_id = r.region_id;
```

county_id	county_name	car_plate_code	region_id	region_name
1	TORINO	A	1	PIEMONTE
2	VERCELLI	VC	1	PIEMONTE
3	NOVARA	NO	1	PIEMONTE
4	CUNEO	CN	1	PIEMONTE
5	ASTI	AT	1	PIEMONTE
6	ALESSANDRIA	AL	1	PIEMONTE
...

Questo è sempre come sopra, semplicemente in una forma un pò più chiara: l'uso abbondante della clausola **AS** per definire degli alias per le colonne e le tabelle rende la stesura delle condizioni **JOIN** molto più concisa e leggibile, e più facile da capire.

```

SELECT lc.lc_id AS lc_id,
       lc.lc_name AS lc_name,
       lc.population AS population,
       c.county_id AS county_id,
       c.county_name AS county_name,
       c.car_plate_code AS car_plate_code,
       r.region_id AS region_id,
       r.region_name AS region_name
FROM local_councils AS lc,
     counties AS c,
     regions AS r
WHERE lc.county_id = c.county_id
  AND c.region_id = r.region_id;

```

lc_id	LC_NAME	population	county_id	county_name	car_plate_code	region_id	region_name
1001	AGLIE '	2574	1	TORINO	A	1	PIEMONTE
1002	AIRASCA	3554	1	TORINO	A	1	PIEMONTE
1003	Ala di Stura	479	1	TORINO	A	1	PIEMONTE
...

Unire tre (*o anche più*) tabelle non è molto più difficile: dovete semplicemente applicare la **condizione JOIN** adatta alle necessità.

Analisi delle Prestazioni

L'esecuzione di interrogazioni [query] che coinvolgono numerose tabelle differenti può degenerare facilmente in un processo lentissimo.

Questo lo si può facilmente vedere con tabelle che contengono un alto numero di righe. Spiegare questo non è per niente difficile: per calcolare il Prodotto Cartesiano il motore SQL deve accedere molte e molte volte ad ogni tabella coinvolta nella interrogazione.

Il comportamento elementare è quello di passare tutta la tabella ogni volta: ed ovviamente scandire una tabella lunga numerosissime volte richiede molto tempo.

Quindi il punto chiave per ottimizzare le vostre interrogazioni è quello di evitare la scansione di tutta la tabella ogni volta che potete.

Questo è del tutto possibile, ed è facilmente realizzabile.

Ogni volta che **SQL-planner** (un componente interno del **motore SQL**) trova che è disponibile un INDEX adatto, non ha bisogno di scandire interamente la tabella, perchè ogni singola riga può essere raggiunta usando quell' Index.

E questo ovviamente sarà un processo molto più veloce.

Ogni colonna (o gruppo di colonne) usate frequentemente in clausole **JOIN** è un candidato adatto per un **INDEX** corrispondente.

Però la creazione di un Index implica parecchie conseguenze negative:

- la allocazione di memoria aumenterà lo spazio richiesto dal DB (alcune volte in modo drammatico).
- l'esecuzione di comandi **INSERT**, **UPDATE** e/o **DELETE** richiederà tempi più lunghi, perchè l'Index deve essere adeguatamente aggiornato. E questo ovviamente impone un ulteriore sovraccarico.

Pertanto (non stupitevi) è un processo di *messa a punto*: dovete valutare attentamente quando un INDEX è assolutamente richiesto, e cercate un giusto equilibrio, cioè un *compromesso* fra esigenze contrastanti, sotto varie condizioni ed in differenti casi specifici.

In altre parole, non vi è la *regola assoluta*: dovete trovare la soluzione ottimale *caso a caso* facendo diverse prove pratiche, finchè trovate la soluzione che soddisfa meglio tutte le vostre esigenze.



Febbraio 2011

Ricetta # 3:

Maggiori informazioni su JOIN

SQL consente una sintassi alternativa di descrivere clausole JOIN. Più o meno le due sono strettamente equivalenti, per cui usare una o l'altra è una questione di gusto personale nella maggioranza dei casi. Comunque, questo secondo metodo fornisce possibilità veramente interessanti altrimenti non disponibili.

```
SELECT lc.lc_id AS lc_id,
       lc.lc_name AS lc_name,
       lc.population AS population,
       c.county_id AS county_id,
       c.county_name AS county_name,
       c.car_plate_code AS car_plate_code,
       r.region_id AS region_id,
       r.region_name AS region_name
  FROM local_councils AS lc,
       counties AS c,.. regions AS r
 WHERE lc.county_id = c.county_id
   AND c.region_id = r.region_id;
```

Ora proverete una sensazione di *già visto*: ed è più che giusto, perchè avete già incontrato questa interrogazione nel precedente esempio.

```
SELECT lc.lc_id AS lc_id,
       lc.lc_name AS lc_name,
       lc.population AS population,
       c.county_id AS county_id,
       c.county_name AS county_name,
       c.car_plate_code AS car_plate_code,
       r.region_id AS region_id,
       r.region_name AS region_name
  FROM local_councils AS lc
 JOIN counties AS c ON (
    lc.county_id = c.county_id)
 JOIN regions AS r ON (
    c.region_id = r.region_id);
```

Ecco tutto, questa è la stessa identica interrogazione riscritta con la sintassi alternativa:

- l'uso della clausola **JOIN ... ON (...)** rende più esplicito quello che sta avvenendo.
- la condizione **JOIN** è scritta direttamente nell'espressione **ON (...)**: in questo modo il comando di interrogazione è meglio strutturato e più leggibile.
- comunque, tutto questo è semplicemente *uno zuccherino a livello di sintassi*: non c'è nessuna differenza fra i due comandi in termini *funzionali*.

```

SELECT r.region_name AS region,
       c.county_name AS county,
       lc.lc_name AS local_council,
       lc.population AS population
FROM regions AS r
JOIN counties AS c ON (
    c.region_id = r.region_id)
JOIN local_councils AS lc ON (
    c.county_id = lc.county_id
    AND lc.population > 100000)
ORDER BY r.region_name,
         county_name;

```

region	county	local_council	population
ABRUZZO	PESCARA	PESCARA	116286
CALABRIA	REGGIO DI CALABRIA	REGGIO DI CALABRIA	180353
CAMPANIA	NAPOLI	NAPOLI	1004500
CAMPANIA	SALERNO	SALERNO	138188
EMILIA-ROMAGNA	BOLOGNA	BOLOGNA	371217
...

Non c'è niente di strano in questa interrogazione:

- abbiamo semplicemente introdotto una ulteriore clausola **ON** (**... AND lc.population < 100000**), così da escludere i Comuni poco popolosi.

```

SELECT r.region_name AS region,
       c.county_name AS county,
       lc.lc_name AS local_council,
       lc.population AS population
FROM regions AS r
JOIN counties AS c ON (
    c.region_id = r.region_id)
LEFT JOIN local_councils AS lc ON (
    c.county_id = lc.county_id
    AND lc.population > 100000)
ORDER BY r.region_name,
         county_name;

```

region	county	local_council	population
ABRUZZO	CHIETI	NULL	NULL
ABRUZZO	L'AQUILA	NULL	NULL
ABRUZZO	PESCARA	PESCARA	116286
ABRUZZO	TERAMO	NULL	NULL
BASILICATA	MATERA	NULL	NULL
BASILICATA	POTENZA	NULL	NULL
...

Apparentemente questa interrogazione è la stessa della precedente. Ma c'è una significativa differenza:

- questa volta abbiamo usato la clausola **LEFT JOIN**: ed il risultato appare molto differente da quello precedente.
- la clausola **JOIN** normale includerà nel risultato solo le righe per le quali sia il *termine di sinistra* che quello di *destra* soddisfano positivamente la condizione.
- mentre la più sofisticata clausola **LEFT JOIN** includerà tutte le righe dell'elemento di *sinistra* anche se la relazione non è soddisfatta: in questo caso restituisce a destra il valore NULL.

Vi è una significativa differenza fra un normale **JOIN** e **LEFT JOIN**.

Ritornando all'esempio precedente, usando la clausola **LEFT JOIN** si ha la garanzia che ogni Regione ed ogni Provincia [County] saranno inserite nel risultato, anche quelle che non soddisfano il limite imposto della popolazione comunale.



Febbraio 2011

Ricetta # 4:

A proposito di VIEW

SQL dispone di una caratteristica veramente utile, le cosiddette **VIEW**.

In pochissime parole, una **VIEW** è qualcosa che sta a metà fra una **TABELLA** ed una interrogazione:

- la **VIEW** è un oggetto persistente (esattamente come le **TABELLE**).
- potete interrogare una **VIEW** esattamente allo stesso modo di una TABELLA: non c'è nessuna differenza che distingue una **VIEW** da una TABELLA dal punto di vista del comando **SELECT**.
- ma dopo tutto una **VIEW** assomiglia ad una specie di interrogazione “glorificata”. Una **VIEW** non ha assolutamente dati di per sé stessa. I dati che sembrano appartenere alla **VIEW** sono semplicemente presi da qualche altra tabella ogni volta che sono necessari.
- nella implementazione di SQLite ogni **VIEW** è un oggetto strettamente in sola lettura: potete liberamente fare riferimento ad ogni **VIEW** nei comandi **SELECT**. Ma non è consentito eseguire comandi **INSERT**, **UPDATE** o **DELETE** sulle **VIEW**.

In ogni caso, la cosa migliore è di fare qualche esercizio pratico per introdurre le Views [Viste].

```
CREATE VIEW view_lc AS
SELECT lc.lc_id AS lc_id,
       lc.lc_name AS lc_name,
       lc.population AS population,
       c.county_id AS county_id,
       c.county_name AS county_name,
       c.car_plate_code AS car_plate_code,
       r.region_id AS region_id,
       r.region_name AS region_name,
       lc.geometry AS geometry
  FROM local_councils AS lc
 JOIN counties AS c ON (
    lc.county_id = c.county_id)
 JOIN regions AS r ON (
    c.region_id = r.region_id);
```

E voila, questa è la vostra prima **VIEW**:

- banalmente, assomiglia esattamente alle **interrogazioni (query)** che avete visto finora.
- salvo questo; questa volta la prima linea è: **CREATE VIEW ... AS**
- e questa è l'unica differenza sintattica che transforma una semplice interrogazione in una **VIEW**.

```
SELECT lc_name, population, county_name
FROM view_lc
WHERE region_name = 'LAZIO'
ORDER BY lc_name;
```

lc_name	population	county_name
Accumoli	724	RIETI
Acquafondata	316	FROSINONE
ACQUAPENDENTE	5788	VITERBO
ACUTO	1857	FROSINONE
AFFILE	1644	ROMA
...

Adesso potete interrogare questa **VIEW**.

```
SELECT region_name,
       Sum(population) AS population,
       (Sum(ST_Area(geometry)) / 1000000.0)
           AS "area (sq.Km)",
       (Sum(population) /
       (Sum(ST_Area(geometry)) / 1000000.0))
           AS "popDensity (peoples/sq.Km)"
FROM view_lc
GROUP BY region_id
ORDER BY 4;
```

region_name	population	area (sq.Km)	popDensity (peoples/sq.Km)
VALLE D'AOSTA	119548	3258.405868	36.689107
BASILICATA	597768	10070.896921	59.355984
...
MARCHE	1470581	9729.862860	151.140979
TOSCANA	3497806	22956.355019	152.367656
...
LOMBARDIA	9032554	23866,529331	378.461144
CAMPANIA	5701931	13666.322146	417.224981

Potete eseguire interrogazioni anche molto complesse sulle **VIEW**.

```

SELECT v.lc_name AS LocalCouncil,
       v.county_name AS County,
       v.region_name AS Region
  FROM view_lc AS v
 JOIN local_councils AS lc ON (
    lc.lc_name = 'NORCIA'
    AND ST_Touches(v.geometry, lc.geometry))
 ORDER BY v.lc_name, v.county_name, v.region_name;

```

LocalCouncil	County	Region
Accumoli	RIETI	LAZIO
Arquata del Tronto	ASCOLI PICENO	MARCHE
CASCIA	PERUGIA	UMBRIA
Castelsantangelo SUL NERA	MACERATA	MARCHE
CERRETO DI SPOLETO	PERUGIA	UMBRIA
Cittareale	RIETI	LAZIO
Montemonaco	ASCOLI PICENO	MARCHE
PRECI	PERUGIA	UMBRIA

Potete eseguire in **JOIN** una **VIEW** e una TABELLA (o due **VIEW**, e così via ...)

Una piccola spiegazione: questo **JOIN** è in realtà basato sulle relazioni Spatial: il risultato rappresenta la lista dei comuni che confinano con quello di **Norcia**.

Potete vedere esempi più complessi [qui](#) (ricette di Alta Cucina).

L'opzione **VIEW** è una delle più potenti e brillanti meccanismi supportati da SQL.

E l' implementazione di SQLite di **VIEW** è sicuramente di primo livello. Dovreste usare le **VIEW** più spesso che potete: e vi accorgerete che maneggiare in questo modo strutture DB veramente complesse diventa un gioco da ragazzi.

Prendete nota: interrogare una **VIEW** può essere più veloce ed efficiente rispetto all' interrogazione di una **TABELLA**.

Ma la **VIEW** non può comunque essere più efficiente dell' interrogazione sottostante; una interrogazione progettata male e male ottimizzata si tradurrà certamente in una **VIEW** molto lenta.



Febbraio 2011

Ricetta # 5:

Creazione di una nuova tabella (con annessi e connessi)

Adesso siete consapevoli che le prestazioni e l'efficienza complessive di SQL sono strettamente legate alla **struttura sottostante al database**, ad es. le seguenti scelte progettuali sono critiche:

- la definizione delle **tabelle** (e delle **colonne**) nel modo più appropriato.
- l' identificazione delle **relazioni** di connessione delle diverse tabelle.
- l' uso per le relazioni di uso frequente di appropriati **indici**.
- la definizione di utili **vincoli**, così da preservare al massimo la correttezza e la consistenza dei dati.

E' ora di approfondire in dettaglio questi argomenti.

Avvertenza pedante: nel gergo **DBMS / SQL** tutto questo è chiamato complessivamente come **DDL** [*Data Definition Language - Linguaggio di Definizione dei Dati*], al contrario del **DML** [*Data Manipulation Language - Linguaggio di Manipolazione dei Dati*], come **SELECT**, **INSERT** e così via.

```
CREATE TABLE peoples (
    first_name TEXT,
    last_name TEXT,
    age INTEGER,
    gender TEXT, phone TEXT);
```

Questa istruzione creerà una semplice tabella di nome peoples:

- la definizione di ogni singola **colonna** deve almeno specificare il **tipo** di dato da usare, come **TEXT** o **INTEGER**
- **prendete nota:** la gestione dei tipi in SQLite differisce sensibilmente da quella di altri DMBS: ma lo vedremo più in dettaglio successivamente.

```

CREATE TABLE peoples2 (
    id INTEGER NOT NULL
        PRIMARY KEY AUTOINCREMENT,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    age INTEGER
        CONSTRAINT age_verify
            CHECK (age BETWEEN 18 AND 90),
    gender TEXT
        CONSTRAINT gender_verify
            CHECK (gender IN ('M', 'F')),
    phone TEXT);

```

Questa è una versione molto più sofisticata della stessa tabella:

- abbiamo aggiunto una colonna **id**, dichiarata come **PRIMARY KEY AUTOINCREMENT**
 - inserire l'opzione **PRIMARY KEY** su ogni tabella è davvero una *scelta fortemente consigliabile*
 - dichiarare una clausola **AUTOINCREMENT** chiederà a SQLite di generare automaticamente un valore univoco per questa chiave
- abbiamo aggiunto una clausola **NOT NULL** per le colonne **first_name** e **last_name**:
 - questo imporrà un primo tipo di **vincolo**: i valori **NULL** non saranno accettati per queste colonne
 - in altre parole, **first_name** e **last_name** devono assolutamente contenere qualche valore esplicito.
- abbiamo aggiunto un'istruzione **CONSTRAINT ... CHECK (...)** per le colonne **age** and **gender**.
 - questo definisce un secondo tipo di **vincolo**: i valori che non soddisfano il criterio di **CHECK (...)** saranno scartati.
 - la colonna **età** ora accetterà solo valori ragionevoli per l'età di persone *adulte*
 - e la colonna **gender** accetterà solo i valori '**M**' o '**F**'.
 - **attenzione**: non abbiamo dichiarato la clausola **NOT NULL**, così **age = NULL** e **gender = NULL** saranno considerati valori ammessi.

A proposito dei tipi di SQLite

Detto molto velocemente: SQLite non usa per niente tipi di dati. Siete assolutamente liberi di inserire qualsiasi tipo di dato in qualsiasi colonna: il tipo dichiarato per i dati della colonna ha solo funzione *estetica*, me non è controllato né obbligato. Questo non è per niente un **bug**: è una **specifica scelta di progetto**.

Comunque, tutti gli altri DBMS applicano precise qualificazione e controlli di tipo, perciò il comportamento di SQLite può apparire strano e sconcertante. **Siete avvertiti**.

In ogni caso SQLite gestisce internamente i seguenti tipi di dato:

- **NULL**: nessun dato.
- **INTEGER** : interi a **64bit**, in grado di gestire valori molto grandi.
- **DOUBLE**: tipo decimale, doppia precisione.
- **TEXT**: qualsiasi stringa di testo con codifica **UTF-8**, di lunghezza arbitraria.
- **BLOB** : qualsiasi Oggetto Binario Lungo [*Binary Long Object*], di lunghezza arbitraria ed illimitata.

Ricordate: qualsiasi *cella* (intersezione di *riga/colonna*) può memorizzare ogni tipo arbitrario di dato. Esiste una sola eccezione: le colonne dichiarate come **INTEGER PRIMARY KEY** richiedono assolutamente valori interi.

```
ALTER TABLE peoples2
    ADD COLUMN cell_phone TEXT;
```

Potete aggiungere in qualsiasi momento altre colonne alla tabella.

Ancora alcune specifiche **scelte di progetto** di SQLite.

- **non è consentita l' eliminazione** di colonne
- **non è consentita rinominare** colonne.

cioè una volta che avete creato la colonna non c' è modo per cambiare la definizione iniziale.

```
ALTER TABLE peoples2
    RENAME TO peoples_ok;
```

Siete invece assolutamente liberi di cambiare il nome alla tabella.

```
DROP TABLE peoples;
```

E questo cancellerà completamente la tabella (e tutto il suo contenuto) dal DB.

```
CREATE INDEX idx_peoples_phone
    ON peoples_ok (phone);
```

Questa istruzione creerà un indice.

```
DROP INDEX idx_peoples_phone;
```

E questo distruggerà lo stesso indice.

```
CREATE UNIQUE INDEX idx_peoples_name
ON peoples_ok (last_name, first_name);
```

- un indice può essere creato su più colonne.
- specificando la clausola **UNIQUE** si attiva un ulteriore **vincolo**: se il valore è già presente in tabella, non sarà consentita alcuna ulteriore immissione di quel valore.

```
PRAGMA table_info(peoples_ok);
```

cid	Name	typ	notnull	dflt_value	pk
0	id	INTEGER	1	NULL	1
1	first_name	TESTO	1	NULL	0
2	last_name	TESTO	1	NULL	0
3	età	INTEGER	0	NULL	0
4	di genere	TESTO	0	NULL	0
5	telefono	TESTO	0	NULL	0
6	cell_phone	TESTO	0	NULL	0

```
PRAGMA index_list(peoples_ok);
```

ss	Nome	unico
0	idx_peoples_phone	0
1	idx_peoples_name	1

```
PRAGMA index_info(idx_peoples_name);
```

seqno	cid	Nome
0	2	last_name
1	1	first_name

E usando le istruzioni **PRAGMA index_list(...)** e **PRAGMA index_info(...)** potete facilmente controllare l'organizzazione degli indici.



Febbraio 2011

Ricetta # 6:

Creare una nuova colonna Geometry

Ora esamineremo in maggior dettaglio come definire correttamente una colonna di tipo Geometry.

SpatiaLite segue molto da vicino l' approccio adottato da **PostgreSQL/PostGIS**; cioè non è consentita la creazione della Geometry contemporaneamente alla creazione della relativa tabella. Dovete sempre creare prima la tabella e poi, in un secondo momento, aggiungere la colonna Geometry con un atto separato.

```
CREATE TABLE test_geom (
    id INTEGER NOT NULL
        PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    measured_value DOUBLE NOT NULL);
```

```
SELECT AddGeometryColumn('test_geom', 'the_geom',
    4326, 'POINT', 'XY');
```

Questo è l'unico modo disponibile per avere una Geometry completamente valida. Qualsiasi metodo diverso produrrà una Geometry errata ed inaffidabile.

```
SELECT AddGeometryColumn('test_geom', 'the_geom',
    4326, 'POINT', 'XY', 0);
```

```
SELECT AddGeometryColumn('test_geom', 'the_geom',
    4326, 'POINT', 'XY', 1);
```

Per quanto la precedente sia la modalità sicuramente più usata, la forma completa disponibile di **GeometryColumn()** è questa:

- l'ultimo argomento (*opzionale*) al momento vale: **NOT NULL**
- selezionando il valore **ZERO** (*che è il valore predefinito in caso di omissione*) la colonna Geometry accetterà valori **NULL**.
- altrimenti saranno accettati solo valori **NOT NULL**.

SRID disponibili:

- qualsiasi SRID definito nella tabella metadata **spatial_ref_sys**.
- il valore **-1** indica uno SRS ignoto o non specificato.

Tipi Geometry disponibili

Geometry Type	Notes
POINT	sono le più usate; corrispondono alle specifiche Shapefile disponibili su ogni applicativo GIS per desktop;
LINESTRING	
POLYGON	
MULTIPOINT	
MULTILINESTRING	
MULTIPOLYGON	
GEOMETRYCOLLECTION	poco usata; non disponibile nei Shapefile e sugli applicativi GIS per desktop;
GEOMETRY	un contenitore generico che consente qualsiasi classe Geometry; poco usato; non disponibile nei Shapefile e sugli applicativi GIS per desktop;

Modelli Dimensionali disponibili

Dimension model	Alias	Notes
XY	2	coordinate X e Y (2D semplice)
XYZ	3	coordinate X, Y e Z (3D)
XYM		coordinate X e Y+ un valore M
XYZM		coordinate X,Y e Z + un valore M

Fate attenzione: questo è un errore molto frequente. Molti sviluppatori, professionisti GIS e simili pensano di essere più furbi e spesso tendono ad inventare qualche modalità molto fantasiosa di creare le proprie Geometries. ad es. pasticciare in qualche modo la tabella **geometry_columns** sembra essere una pratica molto popolare.

Può succedere che tali procedure creative possano funzionare con qualche particolare versione di SpatiaLite; ma è certo che prima o poi apparirà qualche grave incompatibilità ...

Siete avvisati: solo le Geometrie create usando l' istruzione **AddGeometryColumn()** sono pienamente accettate. Ogni diverso approccio è completamente insicuro (*e non supportate ..*).

Penso che controllando direttamente come **AddGeometryColumn()** incida sul database possa aiutarvi a capire meglio.

```
PRAGMA table_info(test_geom);
```

cid	Name	type	notnull	dflt_value	pk
0	id	INTEGER	1	NULL	1
1	Nome	TESTO	1	NULL	0
2	measured_value	DOPPIO	1	NULL	0
3	the_geom	PUNTO	0	NULL	0

Passo 1: una nuova colonna **test_geom** è stata aggiunta alla corrispondente tabella.

```
SELECT *
FROM geometry_columns
WHERE f_table_name LIKE 'test_geom';
```

f_table_name	f_geometry_column	tipo	coord_dimension	SRID	spatial_index_enabled
test_geom	the_geom	POINT	XY	4326	0

Passo 2: nella tabella metadata **geometry_columns** è stata inserita la riga relativa.

```
SELECT *
FROM sqlite_master
WHERE type = 'trigger'
AND tbl_name LIKE 'test_geom';
```

type	name	tbl_name	rootpage	sql
trigger	ggi_test_geom_the_geom	test_geom	0	CREATE TRIGGER "ggi_test_geom_the_geom" BEFORE INSERT ON "test_geom" FOR EACH ROW BEGIN SELECT RAISE(ROLLBACK, 'test_geom.the_geom violates Geometry constraint [geom-type or SRID not allowed]') WHERE (SELECT type FROM geometry_columns WHERE f_table_name = 'test_geom' AND f_geometry_column = 'the_geom' AND GeometryConstraints(NEW."the_geom", type, srid, 'XY') = 1) IS NULL END
trigger	ggu_test_geom_the_geom	test_geom	0	CREATE TRIGGER "ggu_test_geom_the_geom" BEFORE UPDATE ON "test_geom" FOR EACH ROW BEGIN SELECT RAISE(ROLLBACK, 'test_geom.the_geom violates Geometry constraint [geom-type or SRID not allowed]') WHERE (SELECT type FROM geometry_columns WHERE f_table_name = 'test_geom' AND f_geometry_column = 'the_geom' AND GeometryConstraints(NEW."the_geom", type, srid, 'XY') = 1) IS NULL END

Passo 3: la tabella **sqlite_master** è la tabella metadata principale usata da SQLite per memorizzare internamente gli oggetti.

Come potete notare, ogni Geometry richiede qualche **triggers** per avere completo supporto e buona integrazione nel funzionamento del DBMS.

Nessuna sorpresa, tutto è stato definito in modo stringente e consistente per consentire a SpatiaLite di lavorare come necessario.

Se qualche elemento manca o è male configurato, l' ovvia conseguenza sarà uno Spatial DBMS difettoso ed inaffidabile.

```
SELECT DiscardGeometryColumn('test_geom', 'the_geom');
```

Questa istruzione rimuoverà ogni **metadato** ed ogni **trigger** relativo alla data Geometry.....

Prendete nota: questo comunque lascerà inalterato qualsiasi valore geometrico salvato nella corrispondente tabella.....

Dopo l'invocazione del comando **DiscardGeometryColumn(...)** questi non saranno più geometrie qualificate, ma generici ed anonimi valori BLOB.

```
SELECT RecoverGeometryColumn('test_geom', 'the_geom',
    4326, 'POINT', 'XY');
```

Questa istruzione cerca di ripristinare i metadati ed i trigger connessi alla data Geometry. Se l' operazione termina positivamente, la colonna Geometry viene completamente ripristinata. In altre parole, non c' è alcuna differenza fra una Geometry creata con **AddGeometryColumn()** ed un'altra creata con

RecoverGeometryColumn(). Molto semplicemente:

AddGeometryColumn() è progettata per creare una colonna nuova e vuota.

RecoverGeometryColumn() è progettato per recuperare in un secondo momento una colonna già esistente (*e popolata*).

Problemi di compatibilità fra versioni differenti

Spatialite non è immutabile. Come qualsiasi altro prodotto umano ed ogni altro software SpatiaLite si evolve nel tempo; e SQLite anche.

Promessa solenne: siete assolutamente garantiti che ogni file di database generato da qualche versione precedente (**più vecchia**) può essere usata con sicurezza con qualsiasi versione successiva (**più recente**) sia di SQLite che di SpatiaLite.

Prendete bene nota: l' opposto non è necessariamente vero.

Il tentativo di usare un file di database generato da una versione più recente (**più nuova**) con una versione precedente (**più vecchia**) può risultare del tutto impossibile o può causare qualche problema più o meno grave.

Qualche volta l' aggiramento di problemi legati alle versioni è intrinsecamente impossibile: ad es. non c' è alcun modo di usare le geometrie 3D su versioni obsolete, poiché il supporto necessario è stato introdotto in tempi più recenti.

Mentre in numerosi altri casi tali problemi sono semplicemente causati da qualche funzione binaria, richiesta dai **triggers**, incompatibile.

Consiglio utile

Per risolvere qualsiasi incompatibilità legata ai trigger potete semplicemente provare a:

- prima rimuove tutti i trigger: il miglior modo per farlo è l'uso dell'istruzione
DiscardGeometryColumn()
- e quindi ricrearlo usando il comando **AddGeometryColumn()**

Questo garantirà che ogni *informazione nei metadati ed i trigger* rispetteranno le attese della versione corrente della vostra libreria binaria.



Febbraio 2011

Ricetta # 7:

Inserire, Aggiornare e Cancellare

Finora abbiamo principalmente visto come interrogare le tabelle. SQL non è ovviamente un linguaggio di *sola lettura*: sono consentiti in modo flessibile anche l'inserimento di nuove righe, la cancellazione di quelle esistenti e l' aggiornamento dei valori. E' venuta l' ora di vedere questi argomenti in maggior dettaglio.

```
CREATE TABLE test_geom (
    id INTEGER NOT NULL
        PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    measured_value DOUBLE NOT NULL);

SELECT AddGeometryColumn('test_geom', 'the_geom',
    4326, 'POINT', 'XY');
```

Niente di nuovo fin qui: è esattamente la stessa tabella che abbiamo costruito nell'esempio precedente.

```
INSERT INTO test_geom
    (id, name, measured_value, the_geom)
VALUES (NULL, 'first point', 1.23456,
    GeomFromText('POINT(1.01 2.02)', 4326));

INSERT INTO test_geom
VALUES (NULL, 'second point', 2.34567,
    GeomFromText('POINT(2.02 3.03)', 4326));

INSERT INTO test_geom
    (id, name, measured_value, the_geom)
VALUES (10, 'tenth point', 10.123456789,
    GeomFromText ('POINT(10.01 10.02)', 4326));

INSERT INTO test_geom
    (the_geom, measured_value, name, id)
VALUES (GeomFromText('POINT(11.01 11.02)', 4326),
    11.123456789, 'eleventh point', NULL);

INSERT INTO test_geom
    (id, measured_value, the_geom, name)
VALUES (NULL, 12.123456789, NULL, 'twelfth point');
```

L'istruzione **INSERT INTO (...) VALUES (...)** fa esattamente quello che dice il suo nome:

- la prima lista elenca le colonne per nome,
- mentre la seconda lista contiene i valori da inserire: la corrispondenza fra colonne e valori è *data dalla posizione*.
- potete omettere del tutto la lista delle colonne (*guardate la seconda istruzione INSERT, per favore*): però questa non è una buona prassi, perchè questa istruzione presuppone un certo ordine delle colonne, e questo non è un criterio interamente affidabile.
- un altro punto interessante: questa tabella dichiara un **PRIMARY KEY AUTOINCREMENT**:
 - come criterio generale abbiamo passato un valore NULL, in modo che sia SQLite a generare automaticamente il valore.
 - ma nella terza istruzione INSERT abbiamo dato esplicitamente un valore (*per favore, controllate nel paragrafo seguente cosa effettivamente succede in questo caso*).

```
SELECT *
FROM test_geom;
```

id	name	measured_value	the_geom
1	first point	1.234560	BLOB sz=60 GEOMETRY
2	second point	2.345670	BLOB sz=60 GEOMETRY
10	tenth point	10.123457	BLOB sz=60 GEOMETRY
11	eleventh point	11.123457	BLOB sz=60 GEOMETRY
12	twelfth point	12.123457	NULL

Appena un veloce controllo prima di continuare ...

```
INSERT INTO test_geom
VALUES (2, 'POINT #2', 2.2,
GeomFromText('POINT(2.22 3.33)', 4326));
```

Questa ulteriore istruzione **INSERT** fallirà miseramente, provocando una eccezione di vincolo violato (**constraint failed**).

Non è molto difficile rendersi conto di questo: la clausola **PRIMARY KEY** controlla sempre la condizione di unicità. Ed effettivamente una riga con **indice 2 (id=2)** esiste già nella tabella.

```
INSERT OR IGNORE INTO test_geom
VALUES (2, 'POINT #2', 2.2,
GeomFromText('POINT(2.22 3.33)', 4326));
```

Specificando l'opzione **OR IGNORE** questa istruzione adesso fallirà **in silenzio** (*per lo stesso motivo visto sopra*).

```
INSERT OR REPLACE INTO test_geom
VALUES (2, 'POINT #2', 2.2,
GeomFromText('POINT(2.22 3.33)', 4326));
```

C'è un'altra variante: specificando la clausola **OR REPLACE** l'istruzione si comporterà come un comando **UPDATE**.

```

REPLACE INTO test_geom
  (id, name, measured_value, the_geom)
VALUES (3, 'POINT #3', 3.3,
       GeomFromText('POINT(3.33 4.44)', 4326));

-----  

REPLACE INTO test_geom
  (id, name, measured_value, the_geom)
VALUES (11, 'POINT #11', 11.11,
       GeomFromText('POINT(11.33 11.44)', 4326));

```

In realtà è disponibile un'altra sintassi, ad es. usando l'istruzione **REPLACE INTO**: ma questo non è altro che un *alias* di **INSERT OR REPLACE**.

```

SELECT *
FROM test_geom;

```

id	name	measured_value	the_geom
1	first point	1.234560	BLOB sz=60 GEOMETRY
2	POINT #2	2.200000	BLOB sz=60 GEOMETRY
3	POINT #3	3.300000	BLOB sz=60 GEOMETRY
10	tenth point	10.123457	BLOB sz=60 GEOMETRY
11	POINT #11	11.110000	BLOB sz=60 GEOMETRY
12	twelfth point	12.123457	NULL

Ancora un piccolo controllo ...

```

UPDATE test_geom SET
  name = 'point-3',
  measured_value = 0.003
WHERE id = 3;

-----  

UPDATE test_geom SET
  measured_value = measured_value + 1000000.0
WHERE id > 10;

```

l'aggiornamento dei dati non è molto più complesso ...

```

DELETE FROM test_geom
WHERE (id % 2) = 0;

```

e lo stesso per eliminare le righe, ad es. questa istruzione **DELETE** cancellerà tutti le righe con **id** pari.

```
SELECT *
FROM test_geom;
```

id	name	measured_value	the_geom
1	first point	1.234560	BLOB sz=60 GEOMETRY
3	point-3	0.003000	BLOB sz=60 GEOMETRY
11	POINT #11	1000011.110000	BLOB sz=60 GEOMETRY

Un ultimo controllo ...

Avviso molto importante

State attenti: eseguire una istruzione **UPDATE** o **DELETE** senza specificare una corrispondente clausola **WHERE** è perfettamente legale in SQL.

Però SQL interpreta che la modifica agisce indiscriminatamente su ogni riga all' interno della tabella: e qualche volta questo è proprio quello che intendete fare.

Ma (*molto più spesso*) questo è uno splendido modo per distruggere o corrompere i vostri dati senza averne l' intenzione: principianti, fate molta attenzione.



Ricetta # 8:

Conoscere i Vincoli

Febbraio 2011

Imparare a capire **i vincoli** è un compito molto facile seguendo un semplice approccio **concettuale**. Ma d'altro canto capire perché alcune istruzioni SQL falliscono generando errori di **violazione dei vincoli** non è un affare molto semplice.

Per aiutarvi ad imparare meglio questo paragrafo è strutturato come un quiz:

- prima troverete le domande.
- le relative **risposte** le trovate sotto.

Avviso importante:

E' fortemente suggerita la creazione di un altro database in modo da conservare incontaminato il vostro database **campione**.

COME INIZIARE

```

CREATE TABLE mothers (
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    CONSTRAINT pk_mothers
        PRIMARY KEY (last_name, first_name));

SELECT AddGeometryColumn('mothers', 'home_location',
    4326, 'POINT', 'XY', 1);

CREATE TABLE children (
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    mom_first_nm TEXT NOT NULL,
    mom_last_nm TEXT NOT NULL,
    gender TEXT NOT NULL
        CONSTRAINT sex CHECK (
            gender IN ('M', 'F')),
    CONSTRAINT pk_childs
        PRIMARY KEY (last_name, first_name),
    CONSTRAINT fk_childs
        FOREIGN KEY (mom_last_nm, mom_first_nm)
        REFERENCES mothers (last_name, first_name));

```

```

INSERT INTO mothers (first_name, last_name, home_location)
VALUES ('Stephanie', 'Smith',
       ST_GeomFromText('POINT(0.8 52.1)', 4326));

INSERT INTO mothers (first_name, last_name, home_location)
VALUES ('Antoinette', 'Dupont',
       ST_GeomFromText('POINT(4.7 45.6)', 4326));

INSERT INTO mothers (first_name, last_name, home_location)
VALUES ('Maria', 'Rossi',
       ST_GeomFromText('POINT(11.2 43.2)', 4326));

INSERT INTO children
(first_name, last_name, mom_first_nm, mom_last_nm, gender)
VALUES ('George', 'Brown', 'Stephanie', 'Smith', 'M');

INSERT INTO children
(first_name, last_name, mom_first_nm, mom_last_nm, gender)
VALUES ('Janet', 'Brown', 'Stephanie', 'Smith', 'F');

INSERT INTO children
(first_name, last_name, mom_first_nm, mom_last_nm, gender)
VALUES ('Chantal', 'Petit', 'Antoinette', 'Dupont', 'F');

INSERT INTO children
(first_name, last_name, mom_first_nm, mom_last_nm, gender)
VALUES ('Henry', 'Petit', 'Antoinette', 'Dupont', 'M');

INSERT INTO children
(first_name, last_name, mom_first_nm, mom_last_nm, gender)
VALUES ('Luigi', 'Bianchi', 'Maria', 'Rossi', 'M');

```

Niente di troppo complicato: abbiamo semplicemente creato due tabelle:

- la tabella **mothers** contiene una colonna Geometry
- fra **mamme** e **figli** esiste una relazione e di conseguenza è stato creato un **FOREIGN KEY**
- un punto da notare: in questo esempio usiamo un **PRIMARY KEY** esteso su due colonne: ma non c'è nulla di strano in ciò ... è un'opzione pienamente legittima in SQL.

E quindi abbiamo inserito alcune righe nelle tabelle.

```

SELECT m.last_name AS MomLastName,
       m.first_name AS MomFirstName,
       ST_X(m.home_location) AS HomeLongitude,
       ST_Y(m.home_location) AS HomeLatitude,
       c.last_name AS ChildLastName,
       c.first_name AS ChildFirstName,
       c.gender AS ChildGender
FROM mothers AS m
JOIN children AS c ON (
    m.first_name = c.mom_first_nm
    AND m.last_name = c.mom_last_nm);

```

MomLastName	MomFirstName	HomeLongitude	HomeLatitude	ChildLastName	ChildFirstName	ChildGender
Smith	Stephanie	0.8	52.1	Brown	George	M
Smith	Stephanie	0.8	52.1	Brown	Janet	F
Dupont	Antoinette	4.7	45.6	Petit	Chantal	F
Dupont	Antoinette	4.7	45.6	Petit	Henry	M
Rossi	Maria	11.2	43.2	Bianchi	Luigi	M

Un piccolo controllo, poi siamo pronti per partire.

DOMANDE

Q1: perchè questa istruzione SQL fallirà, sollevando un'eccezione di **violazione di vincolo**?

```

INSERT INTO children
(first_name, last_name, mom_first_nm, mom_last_nm)
VALUES ('Silvia', 'Bianchi', 'Maria', 'Rossi');

```

Q2: ...stessa domanda...

```

INSERT INTO children
(first_name, last_name, mom_first_nm, mom_last_nm, gender)
VALUES ('Silvia', 'Bianchi', 'Maria', 'Rossi', 'f');

```

Q3: ...stessa domanda...

```

INSERT INTO children
(first_name, last_name, mom_first_nm, mom_last_nm, gender)
VALUES ('Silvia', 'Bianchi', 'Giovanna', 'Rossi', 'F');

```

Q4: ...stessa domanda...

```
INSERT INTO children
(first_name, last_name, mom_first_nm, mom_last_nm, gender)
VALUES ('Henry', 'Petit', 'Stephanie', 'Smith', 'M');
```

Q5: ...stessa domanda...

```
INSERT INTO mothers (first_name, last_name, home_location)
VALUES ('Pilar', 'Fernandez',
ST_GeomFromText('POINT(4.7 45.6)'));
```

Q6: ...stessa domanda...

```
INSERT INTO mothers (first_name, last_name, home_location)
VALUES ('Pilar', 'Fernandez',
ST_GeomFromText('MULTIPOINT(4.7 45.6, 4.75 45.32)', 4326));
```

Q7: ...stessa domanda...

```
INSERT INTO mothers (first_name, last_name)
VALUES ('Pilar', 'Fernandez');
```

Q8: ...stessa domanda...

```
INSERT INTO mothers (first_name, last_name, home_location)
VALUES ('Pilar', 'Fernandez',
ST_GeomFromText('POINT(4.7 45.6)', 4326));
```

Q9: ...stessa domanda...

```
DELETE FROM mothers
WHERE last_name = 'Dupont';
```

Q10: ...stessa domanda...

```
UPDATE mothers SET first_name = 'Marianne'
WHERE last_name = 'Dupont';
```

RISPOSTE

A1: manca o non è definito il genere (**gender**): per cui è assunto un valore **NULL**. Ma è stato imposto un valore **NOT NULL** per la relativa colonna.

A2: valore '**f**' del genere (**gender**) errato: le stringhe di testo in SQLite sono **case-sensitive**. Il controllo sulla colonna **sex** può accettare solo valori '**M**' o '**Ff**' non va bene.

A3: è stato violato il vincolo **FOREIGN KEY**. La coppia di valori { '**Rossi**' , '**Giovanna**' } non è stata trovata nella tabella delle madri [**mothers**].

A4: è stato violato il vincolo PRIMARY KEY. Il bimbo {'Petit','Henry'} esistono già nell'archivio dei figli [**children**].

A5: manca o non è definito lo **SRID** : quindi è implicitamente assunto il valore **-1**. Ma è stato creato un vincolo per la colonna **Geometry**; il sistema si aspetta uno **SRID 4326** per qualsiasi località [**home_location**] geografica.

A6: tipo **Geometry** errato: il controllo sulla colonna Geometry [**home_location**] accetta solo tipi **POINT**.

A7: manca o non è specificata la **home_location**: pertanto è assunto un valore **NULL**, in contrasto con il vincolo definito sulla colonna [**home_location**].

A8: l'espressione **WKT** è errata: **ST_GeomFromText()** ritornerà il valore **NULL** (stesso problema visto sopra).

A9: è stato violato il vincolo **FOREIGN KEY**: la tabella delle madri [**mothers**] non ha **FOREIGN KEY**. Invece la tabella dei figli [**children**] possiede **FOREIGN KEY**. Eliminando questi dati dalla tabella delle madri [**mothers**] spezzerà l'*integrità referenziale*, per questo l'operazione non è ammessa.

A10: è stato violato il vincolo **FOREIGN KEY**: più o meno come prima. Modificando un valore **PRIMARY KEY** nella tabella delle madri [**mothers**] verrà spezzata l'*integrità referenziale*, quindi l'operazione non è ammessa.

Lezione da imparare #1

Un uso appropriato dei vincoli SQL vi aiuta a preservare interamente i vostri dati in uno stato corretto ed assolutamente consistente.

Ovviamente la definizione di troppi vincoli può facilmente trasformare il vostro database in una specie di fortezza inespugnabile circondato da trincee, casematte, filo spinato e campi minati.

Cioè in qualcosa che nessuno di sicuro definirà amichevole. Usate molto buonsenso, ed evitate possibilmente ogni eccesso.

Lezione da imparare #2

Ogni volta che il motore SQL riscontra qualche violazione di un vincolo, viene immediatamente segnalata una eccezione.

Ma questo è una condizione di errore del tutto generica: dovete usare tutta la vostra esperienza ed astuzia per capire perfettamente (e possibilmente risolvere) ogni possibile pasticcio.



Febbraio 2011

Ricetta # 9: ACIDity: conoscere le transazioni

L'acronimo ACID non ha nulla a che fare con la chimica (**pH**, *idrogeno, ioni di idrossido, e così via ..*)
Nel contesto dei DBMS questo acronimo significa:

- Atomicità
- Consistenza
- Isolamento
- Durabilità.

Detto molto semplicemente:

- un DBMS è progettato per contenere dati complessi: deve attentamente controllare sofisticate relazioni e complessi vincoli. La consistenza dei dati deve essere assolutamente garantita
- ogni volta che elabora una istruzione **INSERT**, **UPDATE** or **DELETE**, la consistenza dei dati è messa a rischio. Se anche una sola modifica fallisce (*qualsiasi sia la ragione*), questo lascia l' intero archivio DB in uno stato inconsistente.
- un DBMS conforme alle specifiche **ACID** risolve in modo brillante ogni potenziale rischio.

Il modello concettuale sotteso è basato sull' approccio delle transazioni [**TRANSACTION**]:

- una **TRANSACTION** racchiude un gruppo arbitrario di istruzioni SQL.
- il sistema garantisce l' esecuzione della **TRANSACTION** in una unità operativa, detta **atomica**, consistente in o tutto o niente
 - se tutte le istruzioni racchiuse nella **TRANSACTION** vengono portate a termine con successo, allora la **TRANSACTION** stessa è considerata completata con successo
 - ma se una sola istruzione fallisce, l'intera **TRANSACTION** fallirà: e l'intero DB verrà conservato nello stato precedente, come era prima dell' inizio della **TRANSACTION**
- e non è tutto: qualsiasi modifica operata in un contesto transazionale è assolutamente invisibile alle altre connessioni al DBMS, in quanto la **TRANSACTION** definisce un contesto privato, *isolato*

Comunque, l'esecuzione di qualche prova diretta è certamente il modo più semplice di vedere come funzionano le **TRANSACTION**.

```
BEGIN;

CREATE TABLE test (
    num INTEGER,
    string TEXT);

INSERT INTO test (num, string)
VALUES (1, 'aaaa');

INSERT INTO test (num, string)
VALUES (2, 'bbbb');

INSERT INTO test (num, string)
VALUES (3, 'cccc');
```

L'inizio della **TRANSACTION** è dichiarata dall' istruzione **BEGIN**:

- dopo questa dichiarazione, ogni istruzione seguente apparterrà al contesto transazionale.
- potete anche usare l' espressione **BEGIN TRANSACTION**, ma questo è inutilmente prolioso, e poco usato.
- SQLite vieta l' uso di *transazioni nidificate*: in un dato momento ci deve essere una sola **TRANSACTION** attiva.

```
SELECT *
FROM test;
```

Potete controllare il vostro lavoro: non c' è nulla di strano? Assolutamente tutta appare come dovrebbe.

Però, dalla dichiarazione iniziale **BEGIN** deriva qualche conseguenza rilevante:

- intanto avete una **TRANSACTION** attiva (*non finita, incompleta*)
- potete fare un primo semplice controllo:
 - aprite una seconda sessione **spatialite_gui** e connettetevi allo stesso DB;
 - siete in grado di vedere la tabella di **test** ?
- NO: perchè la tabella è stata creata in un contesto privato (*isolato*) della prima sessione **spatialite_gui**, e quindi questa tabella semplicemente *non esiste* per ogni altra connessione.
- e potete fare anche un secondo controllo:
 - chiudete entrambe le sessioni **spatialite_gui**.
 - quindi avviate di nuovo **spatialite_gui**.
 - la tabella di **test** non c'è per nulla: sembra sparita, completamente svanita.
 - ma tutto questo ha una spiegazione semplice: la **TRANSACTION** non è mai stata **confermata**.
 - e quando la vecchia sessione è stata chiusa, SQLite ha annullato ogni operazione racchiusa nella **TRANSACTION**, lasciando il DB esattamente com' era in precedenza.

`COMMIT;``ROLLBACK;`

Quando iniziate [**BEGIN**] una **TRANSACTION**, ogni istruzione successiva rimane in uno stato di *sospensione (non confermato)*.

Prima o poi dovete:

- chiudere positivamente (confermare) la **TRANSACTION**, dichiarando una istruzione **COMMIT**.
 - ogni modifica al DB sarà confermata e applicata in modo definitivo.
 - queste modifiche diventeranno immediatamente visibili alle altre connessioni.
- chiudere negativamente (*rifiutare*) la **TRANSACTION**, dichiarando una istruzione **ROLLBACK**.
 - ogni modifica al DB sarà scartata: il DB sarà riportato allo stato precedente.
- se omettete la dichiarazione **COMMIT** o **ROLLBACK**, allora SQLite presumerà prudenzialmente che la **TRANSACTION** è invalida, e sarà implicitamente eseguita una istruzione **ROLLBACK**.
- se durante l'esecuzione della **TRANSACTION** si riscontra un errore o viene elevata una eccezione, allora l' intera **TRANSACTION** è scartata ed eseguita una istruzione **ROLLBACK**.

Considerazione sulle prestazioni

Gestire le **TRANSACTION** vi sembra troppo complicato? quindi starete pensando "*Ignoro semplicemente tutto questo ...*".

Bene, fate molta attenzione al fatto che SQLite è un **ACID DBMS** totale, pertanto è volutamente progettato per trattare le **TRANSACTION**. E non è tutto.

SQLite non è minimamente capace di agire al di fuori di un contesto **TRANSACTION**.

Ogni volta che dimenticate di dichiarare esplicitamente qualche istruzione **BEGIN / COMMIT**, allora SQLite attiva implicitamente la modalità cosiddetta **AUTOCOMMIT**:

- ogni singola istruzione sarà trattata come una **TRANSACTION**.

quando eseguite un semplice comando **INSERT INTO ...**, SQLite lo trasforma silenziosamente:

```
BEGIN;
INSERT INTO ...;
COMMIT;
```

Notate bene: questo è un modo assolutamente sicuro ed accettabile quando state inserendo poche righe a mano nel DB.

Ma quando qualche processo C / C++ / Java / Python cerca di eseguire **INSERT** per numerose e numerose righe (magari molti **miloni dirighe**), questo causa un inaccettabile sovraccarico.

In altre parole, la vostra applicazione funzionerà molto male, richiedendo tempi inutilmente lunghi per portare a termine l'attività: e tutto questo semplicemente perchè non è stata **esplicitamente dichiarata** una **TRANSACTION**.

Il *modo fortemente consigliato* di eseguire veloci **INSERT (UPDATE, DELETE, ...)** è il seguente:

- iniziare esplicitamente una **TRANSACTION (BEGIN)**
- eseguire tutti gli **INSERT** necessari
- confermare la **TRANSACTION** in corso (**COMMIT**).

E questa semplice astuzia vi garantirà delle prestazioni brillantissime.

Stranezze dei connettori...racconti di vita vera

Sviluppatori, state attenti: *linguaggi differenti, connettori differenti, configurazioni iniziali differenti ...*

Gli sviluppatori **C/C++** useranno direttamente le API di SQLite: in questo ambiente lo sviluppatore deve esplicitamente dichiarare la **TRANSACTION** come necessario, chiamando:

- `sqlite3_exec (db_handle, "BEGIN", NULL, NULL, &err_msg);`
- `sqlite3_exec (db_handle, "COMMIT", NULL, NULL, &err_msg);`

I connettori **Java/JDBC** seguono più o meno lo stesso approccio: lo sviluppatore deve chiudere esplicitamente la modalità **AUTOCOMMIT**, e poi dichiarare un **COMMIT** quando necessario e corretto:

- `conn.setAutoCommit(false);`
- `conn.commit();`

Brevemente detto: in C/C++ and Java lo sviluppatore deve iniziare una **TRANSACTION** per poter eseguire dei veloci **INSERT** nel DB.

L' omissione di questo passo causerà delle prestazioni molto lente.

Ma almeno ogni modifica verrà sicuramente applicata al DB.

Python segue un approccio completamente diverso: una **TRANSACTION** è silenziosamente attiva in ogni momento.

Le prestazioni sono sempre ottimali.

Ma se ci si dimentica un esplicito `conn.commit()` prima di chiudere, ogni modifica andrà persa per sempre immediatamente dopo aver terminato la connessione.

E questo sicuramente farà impazzire i principianti, *penso*.



Febbraio 2011

Ricetta # 10: la bellezza dell'indice **R*Tree spaziale**

Uno **Spatial Index** si comporta più o meno come ogni altro indice: infatti il compito di ogni indice è quello di consentire ricerche molto veloci degli elementi selezionati da un grande archivio.

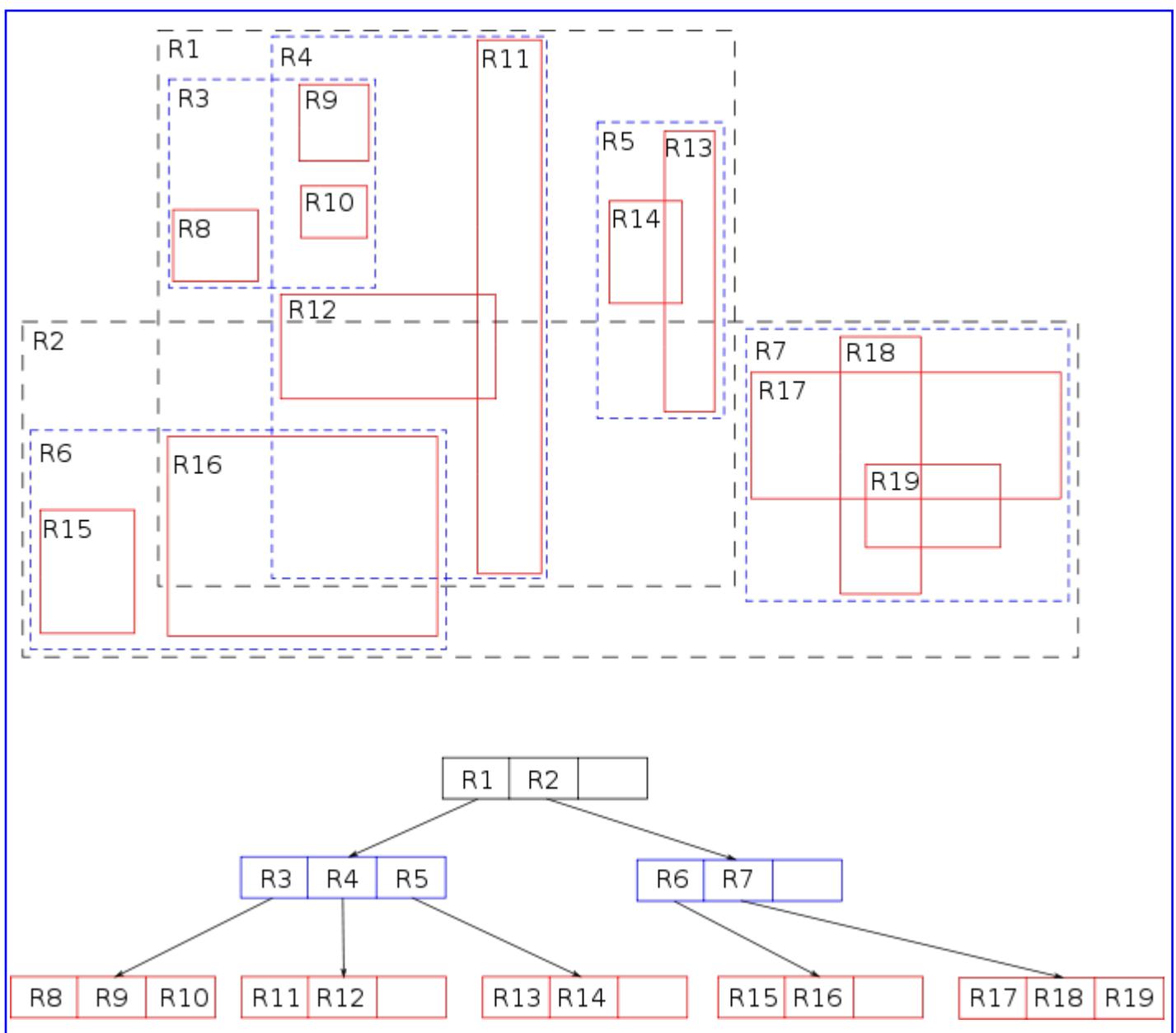
Pensate solo a qualche grande libro di testo: cercare un argomento specifico leggendo l'intero libro è un pò faticoso e può richiedere un tempo molto lungo. Ma in realtà potete leggere l'indice del libro, e quindi saltare direttamente alla pagina giusta.

Ogni indice DB gioca esattamente il medesimo ruolo.

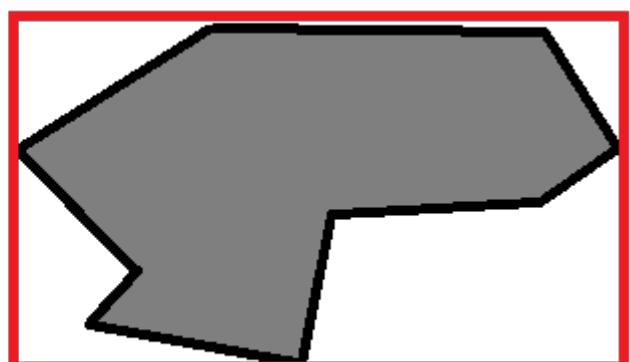
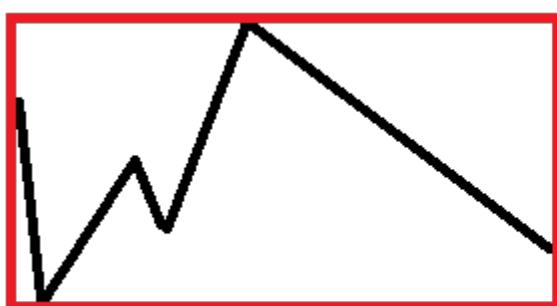
Tuttavia, cercare delle Geometries che cadono all'interno di una certa zona non è la stessa cosa di cercare una stringa di testo o un numero: per fare questo è necessario un differente tipo di indice, lo **Spatial Index** appunto.

Negli anni scorsi sono stati implementati diversi algoritmi per gestire lo Spatial Index.

L'indice spaziale di SQLite è basato sull'algoritmo **R*Tree**.



Detto in parole povere, R*Tree definisce una struttura *simile ad un albero* basata su rettangoli (la R di R*Tree sta esattamente per **Rettangolo**).



Qualsiasi Geometry arbitraria può essere rappresentata con un rettangolo, indipendentemente dalla sua effettiva sagoma: potete usare l' **MBR** (*Minimum Bounding Rectangle -Minor Rettangolo Circoscritto*) che contiene quella Geometry.

E' probabile che il termine **BBOX** (*Bounding Box*) vi sia più familiare: i due termini sono esattamente sinonimi.

Dovrebbe essere ora intuitivo capire come R*Tree in realtà lavori:

- le interrogazioni Spatial definiscono un arbitrario riquadro di ricerca (anche questo un rettangolo)
- l'indice R*Tree è immediatamente esaminato, identificando ogni segnale di sovrapposizione
- alla fine verrà riportata ogni singola Geometry che ricade nella zona di ricerca.

Pensate al noto problema dell'**ago nel pagliaio**: l' R*Tree è un'eccellente soluzione che vi consente di trovare l'ago in un tempo davvero breve, anche quando il pagliaio è impressionante per grandezza.

Luoghi comuni ed errori

"Ho una tabella che contiene milioni (o miliardi?) di punti disseminati intorno al mondo: disegnare una mappa è davvero faticoso e richiede molto tempo. Allora ho trovato un trucco molto utile, ho creato uno Spatial Index su questa tabella. E adesso le mie mappe sono disegnate molto velocemente, come caso generale. Tuttavia sono profondamente confuso, perché disegnare l'intera mappa del mondo richiede ancora un tempo lunghissimo. Perchè lo Spatial Index non funziona con la mappa globale ?"

La risposta è particolarmente semplice: lo Spatial Index velocizza un processo solo quando si cerca in una piccola zona dell'archivio.

Ma quando si cerca nell'intero archivio (o in una parte molto grande di esso), lo Spatial Index non può ovviamente dare alcun beneficio di velocità.

Ad essere pedanti, in queste condizioni l'uso dello Spatial Index introduce ulteriore **lentezza**, poiché interrogare l' R*Tree impone un pesante sovraccarico.

Conclusioni: lo Spatial Index non è la **bacchetta magica**.

Lo Spatial Index è dopotutto uguale ad un filtro:

- quando l'area di ricerca copre una regione molto piccola dell'archivio, l'uso dello Spatial Index implica un *profittevole guadagno*.
- quando l'area di ricerca copre una regione molto grande, l'uso dello Spatial Index implica un *guadagno modesto*.
- ma quando l'area di ricerca copre l'intero archivio (o quasi), l'uso dello Spatial Index implica un *ulteriore costo*.

Dettagli dell'implementazione dell' R*Tree in SQLite

SQLite implementa un R*Tree di prima categoria: tuttavia, alcuni dettagli implementativi possono sembrare *strani* agli utenti abituati ad altri Spatial DBMS (come ad es. PostGIS e così via).

Ogni R*Tree su SQLite richiede **quattro** tabelle strettamente correlate:

- ***rtreebasename_node*** salva (*in formato binario*) i nodi R*Tree.
- ***rtreebasename_parent*** salva le relazioni che collegano i genitori ed i nodi figli.
- ***rtreebasename_rowid*** salva valori ROWID che collegano un nodo R*Tree alla corrispondente riga della tabella indicizzata.
 - nessuna di queste tre tabelle è fatta per accedervi direttamente: esse sono riservate per uso interno
- ***rtreebasename*** è in realtà una tabella virtuale [Virtual Table], ed offre l' R*Tree per ogni accesso esterno.
 - **AVVISO IMPORTANTE:** non cercate mai di accedere direttamente qualche tabella relativa all' R*Tree potreste fare un pasticcio; quasi sicuramente questo causerà corruzioni irreversibili al R*Tree.

Siete avvisati.

```
SELECT *
FROM rtreebasename;
```

pkuid	Miny	maxx	Miny	Maxy
1022	313361.000000	331410.531250	4987924.000000	5003326.000000
1175	319169.218750	336074.093750	4983982.000000	4998057.500000
1232	329932.468750	337638.812500	4989399.000000	4997615.500000
...

Ogni tabella R*Tree assomiglia a questa:

- la colonna **pkid** contiene i valori **ROWID**
- **minx, maxx, miny and maxy** definisco i vertici dell' **MBR**

La logica interna all' R*Tree è *magicamente* implementata dalla Virtual Table.

La gestione dell' R*Tree in SpatiaLite

Ogni Spatial Index di SpatiaLite si appoggia completamente sul corrispondente R*Tree di SQLite. Comunque SpatiaLite integra in modo dolce l' R*Tree, in modo da semplificare completamente la manipolazione delle tabelle:

- ogni volta che eseguite un comando **INSERT**, **UPDATE** o **DELETE** che tocca la *tabella principale*, SpatiaLite si prende automaticamente cura di aggiornare il corrispondente R*Tree.
- qualche triggers garantirà la sincronizzazione..
- quindi, una volta definito uno Spatial Index, potete dimenticarvi completamente di lui.

Ogni Spatial Index di SpatiaLite adotta sempre la seguente convenzione sui nomi:

- ammettiamo di avere la tabella **local_councils** che contiene la colonna **geometria**.
- il corrispondente Spatial Index sarà chiamato **idx_local_councils_geometry**.
- e **idx.local_councils.pkid** punterà a **local_councils.ROWID**.

In ogni caso l'uso di uno Spatial Index di SpatiaLite per accelerare le interrogazioni Spatial è un pò più difficile di altri Spatial DBMS, perchè non c'è una stretta integrazione fra *tabella principale* e R*Tree corrispondente: dalla prospettiva di SQLite si tratta semplicemente di due distinte tabelle.

In conseguenza di ciò, l'uso di Spatial Index richiede l'esecuzione di un JOIN, e (magari) la definizione di una sotto-interrogazione.

Potete trovare molteplici esempi sull'uso di Spatial Index in SpatiaLite nella [sezione Alta Cuicina](#).

```
SELECT CreateSpatialIndex('local_councils', 'geometry');

SELECT CreateSpatialIndex('populated_places', 'geometry');
```

Questa semplice dichiarazione è quanto richiesto per creare uno Spatial Index su qualche colonna Geometry. Ed è tutto.

```
SELECT DiscardSpatialIndex('local_councils', 'geometry');
```

E questo rimuove uno Spatial Index:

- **attenzione prego**: questo non eliminerà [DROP] lo Spatial Index (*dovete eseguire l'operazione con un comando separato*)
- comunque le informazioni relative sono aggiornate per scartare lo Spatial Index, e tutti i **TRIGGER** collegati saranno automaticamente rimossi.

SpatiaLite gestisce un secondo, alternativo Spatial Index basato sul **caching dell' MBR**. Ma questo solo per ragioni storiche, perchè l'uso del caching dell' MBR è **fortemente sconsigliato**.



Alta Cucina

Febbraio 2011



Ricetta # 11: Il Guiness dei primati

Ricetta # 12: Dintorni

Ricetta # 13: Le isole

Ricetta # 14: Centri abitati e Comuni

Ricetta # 15: Centri abitati strettamente adiacenti

Ricetta # 16: Ferrovia e Comuni

Ricetta # 17: Ferrovie e centri abitati

Ricetta # 18: Zone ferroviarie come Buffers

Ricetta # 19: Unione [Merging] dei Comuni in Province e così via ...

Ricetta # 20: viste spaziali (Spatial Views)

Una raffinata esperienza culinaria: da Dijkstra



Ricetta # 11

il Guiness dei primati

Febbraio 2011

Local Council	County	Region
ATRANI	SALERNO	CAMPANIA
BARDONECCHIA	TORINO	PIEMONTE
BRIGA ALTA	CUNEO	PIEMONTE
CASAVATORE	NAPOLI	CAMPANIA
LAMPEDUSA E LINOSA	AGRIGENTO	SICILIA
LU	ALESSANDRIA	PIEMONTE
MORTERONE	LECCO	LOMBARDIA
NE	GENOVA	LIGURIA
OTRANTO	LECCE	PUGLIA
PINO SULLA SPONDA DEL LAGO MAGGIOR	VARESE	LOMBARDIA
PREDOI	BOLZANO	TRENTINO-ALTO ADIGE
RE	VERBANO-CUSIO-OSSOLA	PIEMONTE
RO	FERRARA	EMILIA-ROMAGNA
ROMA	ROMA	LAZIO
SAN VALENTINO IN ABRUZZO CITERIORE	PESCARA	ABRUZZO
VO	PADOVA	VENETO

La precedente lista di Comuni è una specie di **libro Guinness dei primati**. Per una ragione o l'altra ognuno di essi ha qualcosa di veramente eccezionale e degno di nota.

E' probabile che siate un pò confusi e sorpresi leggendo ciò, visto che nessuno di essi (*ad eccezione di Roma*) è fra i luoghi più rinomati d'Italia. Comunque, la spiegazione è molto semplice:

- **Roma** ha il più grande numero di abitanti: **2,546,804**
- **Morterone** ha il numero più basso: **33** abitanti
- **Roma** (ancora) ha la superficie maggiore: **1,287 km²**
- **Atrani** quella più piccola: **0.1 km²**
- **Casavatore** ha la più alta densità abitativa: **13,627.5** abitanti/km²
- **Briga Alta** la più bassa: **1.2** abitanti/km²
- **Pino sulla sponda del Lago Maggiore** e **San Valentino in Abruzzo Citeriore** condividono il privilegio di avere il nome più lungo.
- per contro, **Lu**, **Ne**, **Re**, **Ro** and **Vo** quello più corto.
- **Predoi** è quello situato più a nord
- **Lampedusa e Linosa** quello più a sud
- **Bardonecchia** quello più ovest
- **Otranto** quello più a est.

La stampa al quanto (*in*)utile di questo Guinness dei primati è abbastanza facile. Dovete semplicemente eseguire la seguente interrogazione SQL:

```

SELECT lc.lc_name AS LocalCouncil,
       c.county_name AS County,
       r.region_name AS Region,
       lc.population AS Population,
       ST_Area(lc.geometry) / 1000000.0 AS "Area_sqKm",
       lc.population / (ST_Area(lc.geometry) / 1000000.0)
              AS "PopDensity [peoples/sqKm]",
       Length(lc.lc_name) AS NameLength,
       MbrMaxY(lc.geometry) AS North,
       MbrMinY(lc.geometry) AS South,
       MbrMinX(lc.geometry) AS West,
       MbrMaxX(lc.geometry) AS East
FROM local_councils AS lc
JOIN counties AS c ON (c.county_id = lc.county_id)
JOIN regions AS r ON (r.region_id = c.region_id)
WHERE lc.lc_id IN (
    SELECT lc_id
    FROM local_councils
    WHERE population IN (
        SELECT Max(population)
        FROM local_councils
    UNION
        SELECT Min(population)
        FROM local_councils)
    UNION
    SELECT lc_id
    FROM local_councils
    WHERE ST_Area(geometry) IN (
        SELECT Max(ST_Area(geometry))
        FROM local_councils
    UNION
        SELECT Min(ST_Area(geometry))
        FROM local_councils)
    UNION
    SELECT lc_id
    FROM local_councils
    WHERE population / (ST_Area(geometry) / 1000000.0) IN (
        SELECT Max(population / (ST_Area(geometry) / 1000000.0))
        FROM local_councils
    UNION
        SELECT MIN(population / (ST_Area(geometry) / 1000000.0))
        FROM local_councils)
)

```

```

UNION
  SELECT lc_id
  FROM local_councils
  WHERE Length(lc_name) IN (
    SELECT Max(Length(lc_name))
    FROM local_councils
  UNION
    SELECT Min(Length(lc_name))
    FROM local_councils)
  UNION
    SELECT lc_id
    FROM local_councils
    WHERE MbrMaxY(geometry) IN (
      SELECT Max(MbrMaxY(geometry))
      FROM local_councils)
  UNION
    SELECT lc_id
    FROM local_councils
    WHERE MbrMinY(geometry) IN (
      SELECT Min(MbrMinY(geometry)))
    FROM local_councils)
  UNION
    SELECT lc_id
    FROM local_councils
    WHERE MbrMaxX(geometry) IN (
      SELECT Max(MbrMaxX(geometry)))
    FROM local_councils)
  UNION
    SELECT lc_id
    FROM local_councils
    WHERE MbrMinX(geometry) IN (
      SELECT Min(MbrMinX(geometry)))
    FROM local_councils));

```

Certo, questa non è sicuramente una interrogazione semplice e facile. Ma adesso state consultando le **ricette di Alta Cucina**.

Quindi penso che avevate voglia di provare qualche interrogazione SQL gustosa e piccante: e adesso l'avete. Divertitevi.

Dopo tutto l' interrogazione precedente è solo apparentemente complicata, ma la sua vera struttura è sorprendentemente semplice.

Proviamo a riscriverla in una forma semplificata:

```

SELECT lc.lc_name AS LocalCouncil,
       c.county_name AS County,
       r.region_name AS Region,
       lc.population AS Population,
       ST_Area(lc.geometry) / 1000000.0 AS "Area_sqKm",
       lc.population / (ST_Area(lc.geometry) / 1000000.0)
          AS "PopDensity [peoples/sqKm]",
       Length(lc.lc_name) AS NameLength,
       MbrMaxY(lc.geometry) AS North,
       MbrMinY(lc.geometry) AS South,
       MbrMinX(lc.geometry) AS West,
       MbrMaxX(lc.geometry) AS East
FROM local_councils AS lc
JOIN counties AS c ON (c.county_id = lc.county_id)
JOIN regions AS r ON (r.region_id = c.region_id)
WHERE lc.lc_id IN (... qualche lista di valori ...);

```

Suppongo che adesso siate in grado di capire facilmente il significato di questa interrogazione SQL. Non c'è assolutamente nulla di troppo difficile o complicato in essa:

- la tabella dei comuni [**local_councils**] è collegata [**JOIN**] alla tabella delle provincie [**counties**]
- questa è collegata [secondo **JOIN**] con la tabella delle regioni [**regions**]
- la funzione **ST_Area()** è usata per calcolare la superficie: è applicato un fattore di scala per trasformare i valori da **m²** a **km²**
- una espressione numerica (*per niente difficile o complicata*) è usata per calcolare la densità di popolazione in **abitanti/km²**
- una clausola **WHERE ... IN (...)** è infine usata per filtrare in qualche modo i risultati.

Ma conoscete già tutto ciò e suppongo siate poco interessati ad ulteriori dettagli.

Sarete ovviamente più interessati a sapere cosa succede nella condizione **WHERE ... IN (...)**; ed è proprio lì che concentreremo la nostra attenzione.

```

...
SELECT Max(population)
FROM local_councils
...
SELECT Min(population)
FROM local_councils
...

```

Questa porzione è veramente facile : ogni istruzione calcola il valore **Min** e **Max**.

```
...
SELECT Max(population)
FROM local_councils
UNION
SELECT Min(population)
FROM local_councils
...
...
```

Questa è la prima volta che introduciamo l'istruzione **UNION**:

- questa consente di **mescolare** (merging) due risultati, per averne uno solo.
- occorre soddisfare una condizione: le istruzioni **SELECT** che ci sono da entrambi i lati devono restituire lo stesso numero di colonne e queste devono corrispondersi per tipo.

```
...
SELECT lc_id
FROM local_councils
WHERE population IN (
    SELECT Max(population)
    FROM local_councils
UNION
    SELECT Min(population)
    FROM local_councils)
...
...
```

SQL dispone di un bellissimo meccanismo: quello conosciuto come sotto-interrogazione [**sub-query**]. Voi potete definire una interrogazione **interna** (*che è eseguita prima*), ed usare quindi i valori restituiti nella interrogazione più **esterna** (*principale*).

Adesso il nostro pezzetto SQL non dovrebbe più apparire tanto misterioso:

- in questo caso l'interrogazione **interna** [**sub-query**] è una interrogazione **UNION**.
- l'istruzione di sinistra fornirà il valore **Max** di abitanti.
- l'istruzione di destra fornirà il valore **Min**
- l'opzione **UNION** mescolerà i valori in un unico insieme: e questo è il risultato dell'interrogazione interna [**sub-query**]
- così la condizione **WHERE population IN (...)** dell'interrogazione principale riceverà due valori della popolazione da controllare
- dopo tutto, interrogazione **esterna** restituirà un insieme di valori che è in realtà una semplice lista di codici di comuni [**lc_id**]...

```

...
SELECT lc_id
FROM local_councils
WHERE population IN (
    SELECT Max(population)
    FROM local_councils
UNION
    SELECT Min(population)
    FROM local_councils)
UNION
SELECT lc_id
FROM local_councils
WHERE ST_Area(geometry) IN
    SELECT Max(ST_area(geometry))
    FROM local_councils
UNION
    SELECT Min(ST_Area(geometry))
    FROM local_councils)
...

```

Niente impedisce di nidificare le clausole **UNION**: è una opzione perfettamente legittima. In conformità a questo principio, la porzione SQL precedente va interpretata come segue:

- abbiamo già spiegato la interrogazione di *sinistra* nel paragrafo precedente: questa restituirà la lista dei codici dei comuni che hanno i valori **Min** e **Max** di abitanti
- la parte di *destra* fa un lavoro simile: restituirà come risultato un'altra lista di comuni [**lc_id**] che hanno il **Min** ed il **Max** di superficie (Max/Min ST_Area(geometry)-values)
- così il secondo livello di **UNION** restituirà semplicemente una lista più lunga di comuni, cioè quella contenente tutti i risultati intermedi
- e così via ...

```

SELECT lc.lc_name AS LocalCouncil,
c.county_name AS County,
r.region_name AS Region,
lc.population AS Population,
ST_Area(lc.geometry) / 1000000.0 AS "Area_sqKm",
lc.population / (ST_Area(lc.geometry) / 1000000.0)
    AS "PopDensity [peoples/sqKm]",
Length(lc.lc_name) AS NameLength,
MbrMaxY(lc.geometry) AS North,
MbrMinY(lc.geometry) AS South,
MbrMinX(lc.geometry) AS West,
MbrMaxX(lc.geometry) AS East

```

```

FROM local_councils AS lc
JOIN counties AS c ON (c.county_id = lc.county_id)
JOIN regions AS r ON (r.region_id = c.region_id)
WHERE lc.lc_id IN (
-- 
-- questa complessa interrogazion
-- restituirà una lista di comuni
-- 

    SELECT lc_id
    FROM local_councils
    WHERE population IN (
-- 

-- tquesta sotto-interrogazione (sub-query)
-- restituisce la popolazione Min/Max
-- 

        SELECT Max(population)
        FROM local_councils
        UNION
        SELECT Min(population)
        FROM local_councils)
UNION -- questo risultato verrà aggiunto a quello di primo livello
      SELECT lc_id
      FROM local_councils
      WHERE ST_Area(geometry) IN (
-- 
-- questa sotto-interrogazione restituisce
-- la superficie Min/Max
-- 

        SELECT Max(ST_area(geometry))
        FROM local_councils
        UNION
        SELECT Min(ST_Area(geometry))
        FROM local_councils)
UNION -- questo risultato verrà aggiunto a quello di primo livello
      SELECT lc_id
      FROM local_councils
      WHERE population / (ST_Area(geometry) / 1000000.0) IN (
-- 
-- questa sotto-interrogazione restituisce
-- la densità abitativa Min/Ma
-- 

```

```

SELECT Max(population / (ST_Area(geometry) / 1000000.0))
FROM local_councils

UNION

SELECT MIN(population / (ST_Area(geometry) / 1000000.0))
FROM local_councils)

UNION -- questo risultato verrà aggiunto a quello di primo livello

SELECT lc_id
FROM local_councils
WHERE Length(lc_name) IN (

-- 

-- questa sotto-interrogazione restituisce
-- il nome di lunghezza Min/Max

-- 

SELECT Max(Length(lc_name))
FROM local_councils

UNION

SELECT Min(Length(lc_name))
FROM local_councils)

UNION -- questo risultato verrà aggiunto a quello di primo livello

SELECT lc_id
FROM local_councils
WHERE MbrMaxY(geometry) IN (

-- 

-- questa sotto-interrogazione restituisce
-- il comune più a Nord

-- 

SELECT Max(MbrMaxY(geometry))
FROM local_councils)

UNION -- questo risultato verrà aggiunto a quello di primo livello

SELECT lc_id
FROM local_councils
WHERE MbrMinY(geometry) IN (

-- 

-- this further sub-query will return
-- Max SOUTH

-- 

SELECT Min(MbrMinY(geometry))
FROM local_councils)

UNION -- merging into first-level sub-query

SELECT lc_id
FROM local_councils
WHERE MbrMaxX(geometry) IN (

-- 

-- questa sotto-interrogazione restituisce
-- il comune più a Sud

-- 

```

```

SELECT Min(MbrMinY(geometry))
  FROM local_councils)

UNION -- questo risultato verrà aggiunto a quello di primo livello
      SELECT lc_id
        FROM local_councils
       WHERE MbrMaxX(geometry) IN (
-- 

-- questa sotto-interrogazione restituisce
-- il comune più a Ovest
-- 

      SELECT Max(MbrMaxX(geometry))
        FROM local_councils)

UNION -- questo risultato verrà aggiunto a quello di primo livello
      SELECT lc_id
        FROM local_councils
       WHERE MbrMinX(geometry) IN (
-- 

-- questa sotto-interrogazione restituisce
-- il comune più a Est
-- 

      SELECT Min(MbrMinX(geometry))
        FROM local_councils));

```

In conformità alla sintassi SQL syntax, *usando due tratti consecutivi* (--) potete scrivere un commento, cioè ogni testo successivo ai trattini e fino al prossimo fine linea è assolutamente ignorato dal SQL parser.

Mettendo appropriati commenti all'interno delle interrogazioni SQL complesse migliora sicuramente la leggibilità,

Conclusioni

SQL è uno splendido linguaggio, che dispone di una sintassi regolare e facilmente prevedibile. Ogni volta che incontrerete qualche interrogazione SQL complessa ed inquietante, nessuna paura e niente panico: cercate semplicemente di suddividere l'interrogazione globale in blocchi più piccoli e semplici, e vedrete presto che la complessità era più apparente che reale.

Consiglio utile

Cercare di validare istruzioni SQL molto complesse è ovviamente un compito non facile e dispendioso. L'Approccio migliore da seguire è di suddividere l'istruzione complessa in pezzi più piccoli, e poi testare ognuno di loro separatamente.



Ricetta # 12

Dintorni

Febbraio 2011

Pobлема

- Ogni comune ovviamente condivide un confine in comune con i comuni limitrofi: [*non è comunque una regola assoluta: ad es. delle piccole isole sono auto contenute*].
- Useremo Spatial SQL per identificare ogni coppia di comuni adiacenti.
- giusto per aggiungere un pò di complicazione, centeremo la nostra attenzione sui confini della regione Toscana.

Tuscan Local Council	Tuscan County	Neighbour LC	County	Region
ANGHIARI	AREZZO	CITERNA	PERUGIA	UMBRIA
AREZZO	AREZZO	MONTE SANTA MARIA TIBERINA	PERUGIA	UMBRIA
BIBBIENA	AREZZO	BAGNO DI ROMAGNA	FORLI' - CESENA	EMILIA-ROMAGNA
CHIUSI DELLA Verna	AREZZO	BAGNO DI ROMAGNA	FORLI' - CESENA	EMILIA-ROMAGNA
CHIUSI DELLA Verna	AREZZO	VERGHERETO	FORLI' - CESENA	EMILIA-ROMAGNA
...

```
SELECT lc1.lc_name AS "Local Council",
       lc2.lc_name AS "Neighbour"
  FROM local_councils AS lc1,
       local_councils AS lc2
 WHERE ST_Touches(lc1.geometry, lc2.geometry);
```

Questa interrogazione è davvero semplice:

- la funzione **ST_Touches (geom1, geom2)** valuta la relazione spaziale fra coppie di comuni
 - la tabella dei comuni è scandita due volte, in modo da implementare un **JOIN** (implicito); in altre parole questo consente di valutare un comune rispetto ad ogni altro, esaminando tutte le combinazioni
 - questo può ovviamente causare qualche ambiguità.
- Dobbiamo solo definire un alias adatto (**AS lc1 / AS lc2**) in modo da identificare in modo univoco ogni dati delle due tabelle.

Comunque, un approccio così semplicistico impone parecchi problemi (ardui, severi):

- questa interrogazione restituirà certamente un risultato corretto: ma il tempo di elaborazione sarà parecchio lungo [*in realtà, così lungo da renderlo del tutto inusabile per ogni proposito concreto*]
- spiegarlo è veramente facile: la funzione **ST_Touches()** implica una quantità di complesse elaborazioni, così questo è un passo veramente pesante (*lento*).
- seguendo una logica puramente combinatoria genera molti milioni di coppie da esaminare.
- conclusione: eseguire molti milioni di volte una operazione lenta porta sicuramente al disastro.

```
SELECT lc1.lc_name AS "Local Council",
       lc2.lc_name AS "Neighbour"
  FROM local_councils AS lc1,
       local_councils AS lc2
 WHERE lc2.ROWID IN (
   SELECT pkid
     FROM idx_local_councils_geometry
    WHERE pkid MATCH RTreeIntersects(
      MbrMinX(lc1.geometry),
      MbrMinY(lc1.geometry),
      MbrMaxX(lc1.geometry),
      MbrMaxY(lc1.geometry)));

```

Per fortuna possiamo eseguire queste interrogazioni spaziali in modo molto veloce ed efficiente:

- dobbiamo usare lo **Spatial Index [R*Tree]**
- questo aggiungerà qualche piccola difficoltà alla nostra istruzione SQL, ma comporterà un lucroso aumento di velocità.
- come lavora: per primo controllato l' **R*Tree**, in modo da calcolare il minimo rettangolo circoscritto [MBR] di ogni geometria.
 - questo è un passo molto veloce da eseguire, e consente di scartare una quantità di coppie che sicuramente non hanno confini in comune
 - così il comando **ST_Touches()** verrà eseguito un numero di volte notevolmente inferiore.
 - e questo riduce enormemente il tempo complessivo di elaborazione.
- A livello di sintassi SQL, l'uso dello Spatial Index richiede semplicemente la definizione di una **sotto-interrogazione (sub-query)**.
 - l' R*Tree Spatial Index in SQLite è collocato in una tabella separata
 - il nome della tabella è strettamente correlato: lo Spatial Index corrispondente alla tabella **myTbl** con la colonna geometria **myGeom** avrà sempre nome **idx_myTbl_myGeom**
 - per trovare velocemente ogni geometria si adopera la clausola **MATCH RTreeIntersects()** che controlla il suo **MBR**
 - per identificare i limiti estremi dei MBR si usano i comandi **MbrMinX()** e simili.

Per spiegare meglio cosa sta avvenendo, potete immaginare che questa interrogazione SQL procede con i seguenti passi:

- viene presa una geometria dalla tabella dei comuni: **lc1.geometry**
- allora interviene l' *R*Tree Spatial Index* in modo da identificare ogni altra geometria dalla seconda copia della tabella dei comuni: **lc2.geometry**
- attraverso lo Spatial Index saranno selezionate solo i casi di intersezione dei rettangoli MBR
- e quindi sarà eseguita la funzione **ST_Touches()**: ma questa volta sarà applicata ad un numero limitato e ben selezionato di geometrie.

```

SELECT lc1.lc_name AS "Tuscan Local Council",
       c1.county_name AS "Tuscan County",
       lc2.lc_name AS "Neighbour LC",
       c2.county_name AS County,
       r2.region_name AS Region
FROM local_councils AS lc1,
     local_councils AS lc2,
     counties AS c1,
     counties AS c2,
     regions AS r1,
     regions AS r2
WHERE c1.county_id = lc1.county_id
  AND c2.county_id = lc2.county_id
  AND r1.region_id = c1.region_id
  AND r2.region_id = c2.region_id
  AND r1.region_name LIKE 'toscana'
  AND r1.region_id <> r2.region_id
  AND ST_Touches(lc1.geometry, lc2.geometry)
  AND lc2.ROWID IN (
    SELECT pkid
      FROM idx_local_councils_geometry
     WHERE pkid MATCH RTreeIntersects(
        MbrMinX(lc1.geometry),
        MbrMinY(lc1.geometry),
        MbrMaxX(lc1.geometry),
        MbrMaxY(lc1.geometry)))
ORDER BY c1.county_name, lc1.lc_name;

```

Bene, una volta che avete risolto la complicazione dello Spatial Index scrivere l'interrogazione complessiva SQL non è così difficile.

Comunque questa è una istruzione sicuramente complessa, così qualche spiegazione è certamente benvenuta:

- dobbiamo risolvere la relazione **JOIN** fra comuni [**local_councils**] e provincie [**counties**] e fra provincie [**counties**] e regioni [**regions**]
- ma noi usiamo due copie della tabella dei comuni [**local_councils**], quindi è necessario definire le relazioni **JOIN** separatamente per ogni copia
- definendo la clausola **r1.region_name LIKE 'toscana'** restringeremo la ricerca ai soli comuni della Toscana
- mentre definendo la clausola **r1.region_id <> r2.region_id** ci assicuriamo che il confronto sarà eseguito solo con comuni al di fuori della Toscana
- quindi con questa istruzione saranno ricercati solo comuni toscani aventi confini con comuni non toscani.

```

SELECT lc1.lc_name AS "Tuscan Local Council",
       c1.county_name AS "Tuscan County",
       lc2.lc_name AS "Neighbour LC",
       c2.county_name AS County,
       r2.region_name AS Region
  FROM local_councils AS lc1,
       local_councils AS lc2
 JOIN counties AS c1
   ON (c1.county_id = lc1.county_id)
 JOIN counties AS c2
   ON (c2.county_id = lc2.county_id)
 JOIN regions AS r1
   ON (r1.region_id = c1.region_id)
 JOIN regions AS r2
   ON (r2.region_id = c2.region_id)
 WHERE r1.region_name LIKE 'toscana'
   AND r1.region_id <> r2.region_id
   AND ST_Touches(lc1.geometry, lc2.geometry)
   AND lc2.ROWID IN (
      SELECT pkid
        FROM idx_local_councils_geometry
       WHERE pkid MATCH RTreeIntersects(
          MbrMinX(lc1.geometry),
          MbrMinY(lc1.geometry),
          MbrMaxX(lc1.geometry),
          MbrMaxY(lc1.geometry)))
 ORDER BY c1.county_name, lc1.lc_name;

```

In questa interrogazione potete anche adottare la seconda sintassi del comando JOIN: la differenza è solamente sintattica. E non implica alcuna differenza a livello funzionale e prestazionale.

Eseguire sofisticate interrogazioni spaziali non necessariamente è un compito semplice e facile.

Trattare interrogazioni SQL complesse è un pò difficile (*ma non del tutto impossibile*).

Ottimizzare tali complicate istruzioni SQL, così da farle lavorare velocemente, richiede certamente qualche cura ed attenzione particolari.

Ma Spatial SQL vi aiuta nel modo più efficiente (*eflessibile*): i risultati che potete raggiungere sono semplicemente fantastici.

Dopo tutto il **gioco vale certo la candela**.



Febbraio 2011

Ricetta # 13

Isole ('Isolati')

Problema

Molto vicino a quello precedente. Ora il problema è:

- identificare tutti i comuni [Local Council] isolati, cioè quelli che non confinano con altri comuni italiani.

Local Council	County	Region
CAMPIONE D'ITALIA	COMO	LOMBARDIA
CAPRAIA ISOLA	LIVORNO	TOSCANA
CARLOFORTE	CAGLIARI	SARDEGNA
FAVIGNANA	TRAPANI	SICILIA
ISOLA DEL GIGLIO	GROSSETO	TOSCANA
ISOLE TREMITI	FOGGIA	PUGLIA
LA MADDALENA	SASSARI	SARDEGNA
LAMPEDUSA E LINOSA	AGRIGENTO	SICILIA
LIPARI	MESSINA	SICILIA
PANTELLERIA	TRAPANI	SICILIA
PONZA	LATINA	LAZIO
PROCIDA	NAPOLI	CAMPANIA
USTICA	PALERMO	SICILIA
VENTOTENE	LATINA	LAZIO

Avvertenza: quasi tutti i comuni elencati sono piccole isole, con la rimarchevole eccezione di **Campione d'Italia**, che è un'isola di terra: cioè una piccola enclave italiana circondata dalla Svizzera.

```

SELECT lc1.lc_name AS "Local Council",
       c.county_name AS County,
       r.region_name AS Region
  FROM local_councils AS lc1
 JOIN counties AS c ON (
    c.county_id = lc1.county_id)
 JOIN regions AS r ON (
    r.region_id = c.region_id)
 LEFT JOIN local_councils AS lc2 ON (
    lc1.lc_id <> lc2.lc_id
    AND NOT ST_Disjoint(lc1.geometry, lc2.geometry)
    AND lc2.ROWID IN (
        SELECT pkid
          FROM idx_local_councils_geometry
         WHERE pkid MATCH RTreeIntersects(
            MbrMinX(lc1.geometry),
            MbrMinY(lc1.geometry),
            MbrMaxX(lc1.geometry),
            MbrMaxY(lc1.geometry))))
 GROUP BY lc1.lc_id
 HAVING Count(lc2.lc_id) = 0
 ORDER BY lc1.lc_name;

```

Niente di nuovo qui: questo è più o meno quello che abbiamo già visto nell'ultimo esempio. Solo alcune differenze sono degne di nota:

- stavolta abbiamo usato il comando **NOT ST_Disjoint()** per identificare ogni relazione spaziale disponibile fra coppie di comuni
- ed abbiamo usato l'opzione **LEFT JOIN** per la seconda ripetizione della tabella dei comuni (AS lc2): così ci siamo assicurati di inserire nel risultato ogni comune proveniente dall'insieme di sinistra (lc1) poiché **LEFT JOIN** fornisce un risultato anche quando non c'è alcun elemento dell'insieme di destra (lc2) che soddisfa la condizione definita
- la clausola **GROUP BY lc1.lc_id** è necessaria per costruire un gruppo distinto per ogni comune
- dopo tutto stiamo contando con la funzione **Count(lc2.lc_id)** il numero dei vicini di ogni comune: quindi un valore **ZERO** indica che quel comune è isolato
- ed infine abbiamo usato l'opzione **HAVING** per escludere i comuni non isolati.
- Guardate bene: la clausola **HAVING** non deve essere confusa con la clausola **WHERE**.

Esse sembrano simili, ma esiste una marcata differenza fra loro:

- l'opzione **WHERE** valuta immediatamente se la riga candidata deve essere inserita o meno nel risultato quindi una riga scartata da **WHERE** è completamente ignorata, ed in seguito non può più essere usata.
- d'altra parte la clausola **HAVING** è esaminata solo quando il risultato è completamente definito, appena prima di essere passato al processo chiamante.

Quindi **HAVING** è davvero utile per eseguire qualsiasi tipo di *post-processing*, come in questo caso.

Noi abbiamo semplicemente bisogno di *ridurre* l'elenco risultante (*cancellando tutti i comuni non isolati*), e la clausola **HAVING** fa esattamente al caso nostro.



Ricetta # 14

Centri abitati e Comuni

Febbraio 2011

Problema

Ricordate ?

- abbiamo lasciato la tabella dei Centri abitati [**populated_places**] in sospeso finora.
- quando abbiamo progettato lo schema del DB avevamo concluso che qualche relazione spaziale doveva esistere fra i centri abitati [**populated_places**] ed i comuni [**local_councils**]
- possiamo aspettarci qualche inconsistenza fra questi insiemi di dati perchè arrivano da fonti assolutamente diverse
- la tabella dei centri abitati ha una geometria di tipo **POINT** definiti nello **SRID 4236** (Geographic, **WGS84, long-lat**): mentre la tabella dei comuni è caratterizzata da geometrie **MULTIPOLYGON** nello **SRID 23032 (planar, ED50 UTM zone 32)**
- l'uso di due differenti **SRID** introduce per forza qualche ulteriore complicazione da risolvere.

E' venuto il momento di misurarci con questo problema non così semplice.

PopulatedPlaceId	PopulatedPlaceName	LocalCouncilId	LocalCouncilName	County	Region
...
12383	Acitrezza	NULL	NULL	NULL	NULL
12384	Lavinio	NULL	NULL	NULL	NULL
11327	Altino	69001	ALTINO	CHIETI	ABRUZZO
11265	Archi	69002	ARCHI	CHIETI	ABRUZZO
11247	Ari	69003	ARI	CHIETI	ABRUZZO
...

```

SELECT pp.id AS PopulatedPlaceId,
       pp.name AS PopulatedPlaceName,
       lc.lc_id AS LocalCouncilId,
       lc.lc_name AS LocalCouncilName
FROM populated_places AS pp,
     local_councils AS lc
WHERE ST_Contains(lc.geometry,
      Transform(pp.geometry, 23032))
  
```

Potete cominciare con questa semplice interrogazione:

- la funzione **ST_Contains(geom1, geom2)** valuta la relazione spaziale che esiste tra comuni e centri abitati
- non c'è nulla di strano in questa interrogazione: state semplicemente usando una condizione di JOIN basta su una relazione spaziale
- è quasi ovvio usare la funzione **Transform()** per riproiettare le coordinate nello stesso **SRID**
- comunque siete avvisati: siete consapevoli che usando un approccio così semplicistico (cioè senza usare l'*R*Tree Spatial Index*) produrrete sicuramente una interrogazione molto lenta.

```

SELECT pp.id AS PopulatedPlaceId,
       pp.name AS PopulatedPlaceName,
       lc.lc_id AS LocalCouncilId,
       lc.lc_name AS LocalCouncilName
  FROM populated_places AS pp,
       local_councils AS lc
 WHERE ST_Contains(lc.geometry,
                   Transform(pp.geometry, 23032))
   AND lc.lc_id IN (
      SELECT pkid
        FROM idx_local_councils_geometry
       WHERE pkid MATCH RTreeIntersects(
          MbrMinX(
             Transform(pp.geometry, 23032)),
          MbrMinY(
             Transform(pp.geometry, 23032)),
          MbrMaxX(
             Transform(pp.geometry, 23032)),
          MbrMaxY(
             Transform(pp.geometry, 23032)));

```

Questa interrogazione è esattamente la stessa della prima, ad eccezione del fatto che quest'ultima sfrutta pienamente l'*R*Tree Spatial Index*.

Prendete nota: l'uso ripetuto della funzione **Transform()** è assolutamente necessario, per poter riproiettare ogni coordinata in uno **SRID** uniforme.

Comunque c'è ancora un aspetto non risolto nella interrogazione precedente: in questo modo tutti i centri abitati che non combaciano con qualche comune non saranno mai identificati.

Per controllare se qualche centro abitato sta al di fuori di ogni comune dovete assolutamente adottare una opzione **LEFT JOIN**.

```

SELECT pp.id AS PopulatedPlaceId,
       pp.name AS PopulatedPlaceName,
       lc.lc_id AS LocalCouncilId,
       lc.lc_name AS LocalCouncilName,
       c.county_name AS County,
       r.region_name AS Region
  FROM populated_places AS pp
 LEFT JOIN local_councils AS lc
    ON (ST_Contains(lc.geometry,
                     Transform(pp.geometry, 23032)))
   AND lc.lc_id IN (
      SELECT pkid
        FROM idx_local_councils_geometry
       WHERE pkid MATCH RTreeIntersects(
          MbrMinX(
             Transform(pp.geometry, 23032)),
          MbrMinY(
             Transform(pp.geometry, 23032)),
          MbrMaxX(
             Transform(pp.geometry, 23032)),
          MbrMaxY(
             Transform(pp.geometry, 23032))))))
LEFT JOIN counties AS c
  ON (c.county_id = lc.county_id)
LEFT JOIN regions AS r
  ON (r.region_id = c.region_id)
 ORDER BY 6, 5, 4;

```

Okay: questa è la versione finale e definitiva:

- avete semplicemente aggiunto ulteriori clausole LEFT JOIN, in modo da qualificare/identificare ogni comune che riporta la Provincia e la Regione corrispondente
- nessuna sorpresa per i venti centri abitati che non corrispondono ad alcun comune (*ve lo aspettavate questo, visto che i dati arrivano da fonti diverse*)
- potete usare **QGIS** per controllare visivamente questa incongruenze: e scoprirete subito che in ogni caso si tratta di città collocate vicino alla riva del mare.

Ed in effetti qualche (*piccolo*) errore di posizionamento esiste.



Ricetta # 15

Centri abitati strettamente adiacenti

Febbraio 2011

Problema

Ancora un problema sull'archivio dei centri abitati [**populated_places**]. Questa volta la domanda è:

- identificare tutte le possibili coppie di centri abitati che distano a meno di **1 Km**

Attenzione: questo problema nasconde una complicazione antipatica.

- l'archivio dei centri abitati [**populated_places**] è nel sistema [**SRID**] **4236** (geografico, WGS84, *long-lat*)
- di conseguenza le distanze sono misurate in **gradi decimali**
- ma il vincolo imposto è espresso in **metri/Km**.

PopulatedPlace #1	Distance (meters)	PopulatedPlace #2
Vallarsa	49.444299	Raossi
Raossi	49.444299	Vallarsa
Seveso	220.780551	Meda
Meda	220.780551	Seveso
...

```

SELECT ppl1.name AS "PopulatedPlace #1",
       GeodesicLength(
           MakeLine(ppl1.geometry, pp2.geometry))
           AS "Distance (meters)",
       pp2.name AS "PopulatedPlace #2"
FROM populated_places AS ppl1,
     populated_places AS pp2
WHERE GeodesicLength(
           MakeLine(ppl1.geometry, pp2.geometry)) < 1000.0
AND ppl1.id <> pp2.id
AND pp2.ROWID IN (
    SELECT pkid
    FROM idx_populated_places_geometry
    WHERE pkid MATCH RTreeDistWithin(
        ST_X(ppl1.geometry),
        ST_Y(ppl1.geometry), 0.02))
ORDER BY 2

```

Penso che ora sia assolutamente chiaro a tutti che l'uso dello Spatial Index è necessario per una interrogazione decentemente performante.

E che un JOIN fra due differenti copie della stessa tabella è necessario per eseguire questa analisi spaziale, e così via.....

Pertanto concentreremo la nostra attenzione sugli aspetti più importanti:

- la funzione **MakeLine()** costruisce un segmento fra due PUNTI [**POINT**]
- la funzione **GeodesicLength()** calcola la lunghezza totale (espressa in **metri**) per ogni LINESTRING in unità *long-lat*.

Questa funzione fornisce risultati molto accurati perchè opera direttamente sull'**ellissoide**. Sfortunatamente questo richiede una quantità di complessi calcoli, per cui calcolare una lunghezza geodesica [Geodesic length] è un compito intrinsecamente pesante (*e lento*).

Comunque un uso saggio dello Spatial Index riduce drasticamente la complessità.

- l'opzione **MATCH RTreeDistWithin()** consente una prima stima direttamente con lo Spatial Index.
Avvertenza: la costante **0.02** significa **2/100 di grado** (*circa 2Km*) sull'orizzonte [Great Circle]. Le coordinate dello Spatial Index sono in unità **long-lat** (*questo perchè l'archivio centri abitati [Populated Places] sono nello SRID 4326*), per cui qui è necessaria una misura angolare.
- l'uso della clausola **pp1.id <> pp2.id** evita di valutare la distanza di un centro abitato da se stesso (*che sarebbe evidentemente uguale a 0.0*).

L'esecuzione di una interrogazione spaziale [Spatial query] come questa in modo *naif* richiede un tempo estremamente lungo, anche usando le più recenti e potenti CPU.

Ma non è molto difficile applicare qualche piccola ottimizzazione. Ed una interrogazione SQL ben definita ed ottimizzata sicuramente funzionerà nel modo più veloce e dolce.



Ricetta # 16

Ferrovia e Comuni

Febbraio 2011

Problema

Questa volta useremo per la prima volta l'archivio della **ferrovia [railways]**. Per favore ricordate: questo è un archivio proprio piccolo che contiene due ferrovie: i dati sono forniti nel sistema **[SRID] 23032 [ED50 UTM zone 32]**. Il problema è:

- trovare i comuni intersecati da una linea ferrovia.

Nota importante: dovete eseguire alcuni passi preliminari.

E' necessario scaricare il file [railways.zip](#) (uno shapefile molto semplice, opportunamente derivato da OSM). E poi dovere caricare tale *shapefile* nella tabella delle ferrovie **[railways]**.

Railway	LocalCouncil	County	Region
...
Ferrovia Adriatica	SILVI	TERAMO	ABRUZZO
Ferrovia Adriatica	TORTORETO	TERAMO	ABRUZZO
Ferrovia Roma-Napoli	AVERSA	CASERTA	CAMPANIA
Ferrovia Roma-Napoli	CANCELLO ED ARNONE	CASERTA	CAMPANIA
...

```

SELECT rw.name AS Railway,
       lc.lc_name AS LocalCouncil,
       c.county_name AS County,
       r.region_name AS Region
  FROM railways AS rw
 JOIN local_councils AS lc ON (
    ST_Intersects(rw.geometry, lc.geometry)
    AND lc.ROWID IN (
        SELECT pkid
          FROM idx_local_councils_geometry
         WHERE pkid MATCH RTreeIntersects(
               MbrMinX(rw.geometry),
               MbrMinY(rw.geometry),
               MbrMaxX(rw.geometry),
               MbrMaxY(rw.geometry))))
 JOIN counties AS c
   ON (c.county_id = lc.county_id)
 JOIN regions AS r
   ON (r.region_id = c.region_id)
ORDER BY r.region_name,
         c.county_name,
         lc.lc_name

```

Esamineremo solamente alcuni punti interessanti:

- usiamo una clausola **JOIN** per cercare la provincia e la regione corrispondente ad ogni comune. Sapete già come questo funziona, perchè l'avete già usato in qualche esempio precedente.
- la cosa più importante in questa interrogazione è la prima clausola **JOIN**:
- la funzione **ST_Intersects()** valuta la relazione spaziale fra **comune** e la tabella delle **ferrovie**; comunque tutto ciò non sorprende per nulla, poiché avete già visto cose simili in precedenti esempi
- ed ancora una volta usiamo il **R*Tree Spatial Index** per velocizzare l'esecuzione della interrogazione.

Più o meno, questa è quasi la stessa cosa di prima, quando abbiamo esaminato le relazioni spaziali esistenti fra comuni e centri abitati.

Comunque questo conferma che usare **qualsiasi** tipo di relazione spaziale è un compito ragionevolmente facile e che potete usare con successo relazioni spaziali per risolvere *molteplici* casi pratici.



Ricetta # 17

Ferrovie e centri abitati

Febbraio 2011

Problema

Usiamo ancora una volta l'archivio delle ferrovie.

Ma questa è una ricetta davvero ***molto pepata***: preparatevi ad assaggiare sapori molto forti. Come potete facilmente immaginare da soli, calcolare distanze fra ferrovia e centri abitati non è difficile.

Così questo esercizio introduce un livello ulteriore di complessità (giusto per sfuggire alla noia e mantenere la vostra mente attiva e attenta).

Immaginate che per qualche buona ragione esista questa classificazione:

Class	Min. distanza	Max. distanza
A	0 Km	1 Km
B	1 Km	2.5 Km
C	2.5 Km	5 Km
D	5 Km	10 Km
E	10 Km	20 Km

Il problema che dovete risolvere è:

- individuare ogni centro abitato che si colloca in un raggio di 20km di distanza dalla ferrovia
- individuare la classe di distanza di ognuno dei centri abitati.

Railway	PopulatedPlace	A class [< 1Km]	B class [< 2.5Km]	C class [< 5Km]	D class [< 10Km]	E class [< 20Km]
Ferrovia Adriatica	Zapponeta	NULL	NULL	NULL	NULL	1
Ferrovia Adriatica	Villamagna	NULL	NULL	NULL	NULL	1
Ferrovia Adriatica	Villafonsina	NULL	NULL	NULL	1	0
Ferrovia Adriatica	Vasto	1	0	0	0	0
...

```

SELECT rw.name AS Railway,
       pp_e.name AS PopulatedPlace,
       (ST_Distance(rw.geometry,
                    Transform(pp_a.geometry, 23032)) <= 1000.0)
          AS "A class [< 1Km]",
       (ST_Distance(rw.geometry,
                    Transform(pp_b.geometry, 23032)) > 1000.0)
          AS "B class [< 2.5Km]",
       (ST_Distance(rw.geometry,
                    Transform(pp_c.geometry, 23032)) > 2500.0)
          AS "C class [< 5Km]",
       (ST_Distance(rw.geometry,
                    Transform(pp_d.geometry, 23032)) > 5000.0)
          AS "D class [< 10Km]",
       (ST_Distance(rw.geometry,
                    Transform(pp_e.geometry, 23032)) > 10000.0)
          AS "E class [< 20Km]"

FROM railways AS rw
JOIN populated_places AS pp_e ON (
    ST_Distance(rw.geometry,
                Transform(pp_e.geometry, 23032)) <= 20000.0
    AND pp_e.id IN (
        SELECT pkid
        FROM idx_populated_places_geometry
        WHERE pkid MATCH RTreeIntersects(
            MbrMinX(
                Transform(
                    ST_Envelope(rw.geometry), 4326)),
            MbrMinY(
                Transform(
                    ST_Envelope(rw.geometry), 4326)),
            MbrMaxX(
                Transform(
                    ST_Envelope(rw.geometry), 4236)),
            MbrMaxY(
                Transform(
                    ST_Envelope(rw.geometry), 4326)))))

LEFT JOIN populated_places AS pp_d ON (
    pp_e.id = pp_d.id
    AND ST_Distance(rw.geometry,
                    Transform(pp_d.geometry, 23032)) <= 10000.0

```

```
AND pp_d.id IN (
    SELECT pkid
    FROM idx_populated_places_geometry
    WHERE pkid MATCH RTreeIntersects(
        MbrMinX(
            Transform(
                ST_Envelope(rw.geometry), 4326)),
        MbrMinY(
            Transform(
                ST_Envelope(rw.geometry), 4326)),
        MbrMaxX(
            Transform(
                ST_Envelope(rw.geometry), 4236)),
        MbrMaxY(
            Transform(
                ST_Envelope(rw.geometry), 4326)))))

LEFT JOIN populated_places AS pp_c ON (
    pp_d.id = pp_c.id
    AND ST_Distance(rw.geometry,
        Transform(pp_c.geometry, 23032)) <= 5000.0
    AND pp_c.id IN (
        SELECT pkid
        FROM idx_populated_places_geometry
        WHERE pkid MATCH RTreeIntersects(
            MbrMinX(
                Transform(
                    ST_Envelope(rw.geometry), 4326)),
            MbrMinY(
                Transform(
                    ST_Envelope(rw.geometry), 4326)),
            MbrMaxX(
                Transform(
                    ST_Envelope(rw.geometry), 4236)),
            MbrMaxY(
                Transform(
                    ST_Envelope(rw.geometry), 4326)))))

LEFT JOIN populated_places AS pp_b ON (
    pp_c.id = pp_b.id
    AND ST_Distance(rw.geometry,
        Transform(pp_b.geometry, 23032)) <= 2500.0
```

```

AND pp_b.id IN (
    SELECT pkid
    FROM idx_populated_places_geometry
    WHERE pkid MATCH RTreeIntersects(
        MbrMinX(
            Transform(
                ST_Envelope(rw.geometry), 4326)),
        MbrMinY(
            Transform(
                ST_Envelope(rw.geometry), 4326)),
        MbrMaxX(
            Transform(
                ST_Envelope(rw.geometry), 4236)),
        MbrMaxY(
            Transform(
                ST_Envelope(rw.geometry), 4326)))))

LEFT JOIN populated_places AS pp_a ON (
    pp_b.id = pp_a.id
    AND ST_Distance(rw.geometry,
        Transform(pp_a.geometry, 23032)) <= 1000.0
    AND pp_a.id IN (
        SELECT pkid
        FROM idx_populated_places_geometry
        WHERE pkid MATCH RTreeIntersects(
            MbrMinX(
                Transform(
                    ST_Envelope(rw.geometry), 4326)),
            MbrMinY(
                Transform(
                    ST_Envelope(rw.geometry), 4326)),
            MbrMaxX(
                Transform(
                    ST_Envelope(rw.geometry), 4236)),
            MbrMaxY(
                Transform(
                    ST_Envelope(rw.geometry), 4326))))));

```

Sì, certo, questa interrogazione sembra proprio complessa e minacciosa. Comunque la complessità è molto più apparente che reale.

Voi ormai conoscete il trucco: dovete semplicemente spezzettare l'istruzione in blocchi più piccoli. E scoprirete in fretta che non c'è niente di realmente complicato e difficile.

Esaminiamo la struttura principale:

```
SELECT rw.name AS Railway, ...
FROM railways AS rw
JOIN populated_places AS pp_e ON (...)
LEFT JOIN populated_places AS pp_d ON (...)
LEFT JOIN populated_places AS pp_c ON (...)
LEFT JOIN populated_places AS pp_b ON (...)
LEFT JOIN populated_places AS pp_a ON (...);
```

- mettiamo in relazione le ferrovia (**JOIN AS rw**) ed i centri abitati (**AS pp_e**): non c'è niente di strano fin qui, no?
 - quindi usiamo un **LEFT JOIN** con i centri abitati (**AS pp_d**) una seconda volta: e voi certamente ricordate che il **LEFT JOIN** inserisce una riga nei risultati anche se non ha una valida corrispondenza con il termine di *destra*
 - e finalmente ripetiamo il comando **LEFT JOIN** per **pp_c**, **pp_b** e **pp_a**
 - cominciamo ovviamente verificando la distanza massima, perchè se questo confronto fallisce è certamente inutile qualsiasi ulteriore controllo
- E così via, seguendo la sequenza **decrescente** delle distanza.

```
...
JOIN populated_places AS pp_e ON (
    ST_Distance(rw.geometry,
        Transform(pp_e.geometry, 23032)) <= 20000.0
)
...
```

- ogni **JOIN** (o **LEFT JOIN**) valuta semplicemente la distanza che intercorre fra la linea ferroviaria ed i centri abitati, testando se questi cadono nell'intervallo di distanza della classe
- è richiesto l'uso della funzione **Transform()** poiché la ferrovia è nel sistema (**SRID**) **23032**, mentre i centri abitati sono nel sistema (**SRID**) **4326**.

```
... AND pp_e.id IN (
    SELECT pkid
    FROM idx_populated_places_geometry
    WHERE pkid MATCH RTreeIntersects(
        MbrMinX(
            Transform(
                ST_Envelope(rw.geometry), 4326)),
        MbrMinY(
            Transform(
                ST_Envelope(rw.geometry), 4326)),
        MbrMaxX(
            Transform(
                ST_Envelope(rw.geometry), 4236)),
        MbrMaxY(
            Transform(
                ST_Envelope(rw.geometry), 4326)))
    ...)
```

- l'uso come di consueto dello Spatial Index è sempre richiesto, in modo da consentire un veloce filtraggio della geometria dei centri abitati
- ovviamente dobbiamo applicare la funzione **Transform()** in modo da riproiettare la geometria della ferrovia nel sistema **4326** (quello usato dai centri abitati)
- **attenzione:** eseguire la trasformazione di coordinate di un singolo **POINT** può essere considerata un'operazione *leggera* ma trasformare (numerose e numerose volte) qualche complessa **LINESTRING** o **POLYGON** è un'operazione molto più pesante così usiamo ingegnosamente la funzione **ST_Envelope()** in modo da semplificare drasticamente ogni geometria che deve essere trasformata.

Okay, ora la struttura complessiva dell'interrogazione è assolutamente chiara:

- il primo **JOIN** includerà nei risultati ogni centro abitato che dista meno di **20 Km** dalla ferrovia
- ogni altro **LEFT JOIN** controllerà le distanze via via decrescenti, in accordo con i vincoli di classe imposti
- ed ogni **LEFT JOIN** controllerà attentamente se l'indice (**ID**) del centro abitato è lo stesso che ha soddisfatto la classe precedente, come in: **pp_d.id = pp_c.id**
- ogni volta che **LEFT JOIN fallisce**, viene inserito un valore **NULL** nei risultati.

```
SELECT rw.name AS Railway,
       pp_e.name AS PopulatedPlace,
       (ST_Distance(rw.geometry,
                    Transform(pp_a.geometry, 23032)) <= 1000.0)
          AS "A class [< 1Km]",
       ...

```

Ancora un ultimo elemento da spiegare brevemente:

- consideriamo una distanza ad es. di **3.8 Km**: questa determina l'inclusione in **classe C**, ma questa condizione soddisfa anche i criteri delle **classi D e E**
- dobbiamo quindi eseguire un ulteriore controllo
- fortunatamente SQL è un linguaggio proprio smaliziato. Ogni colonna da riportare nel risultato finale può rappresentare una espressione arbitraria
- ed in SQL una espressione logica è valutata:
 - **0** [FALSE]
 - **1** [TRUE]
 - **NULL** se qualcuno degli operatori è **NULL**.
- e questo è proprio tutto.

Ora potete giocare da soli, eseguendo altre prove su questa interrogazione.

Ad es. potete aggiungere qualche clausola **ORDER BY** o **WHERE** ben congegnata: ora è davvero facile, non è vero?



Febbraio 2011

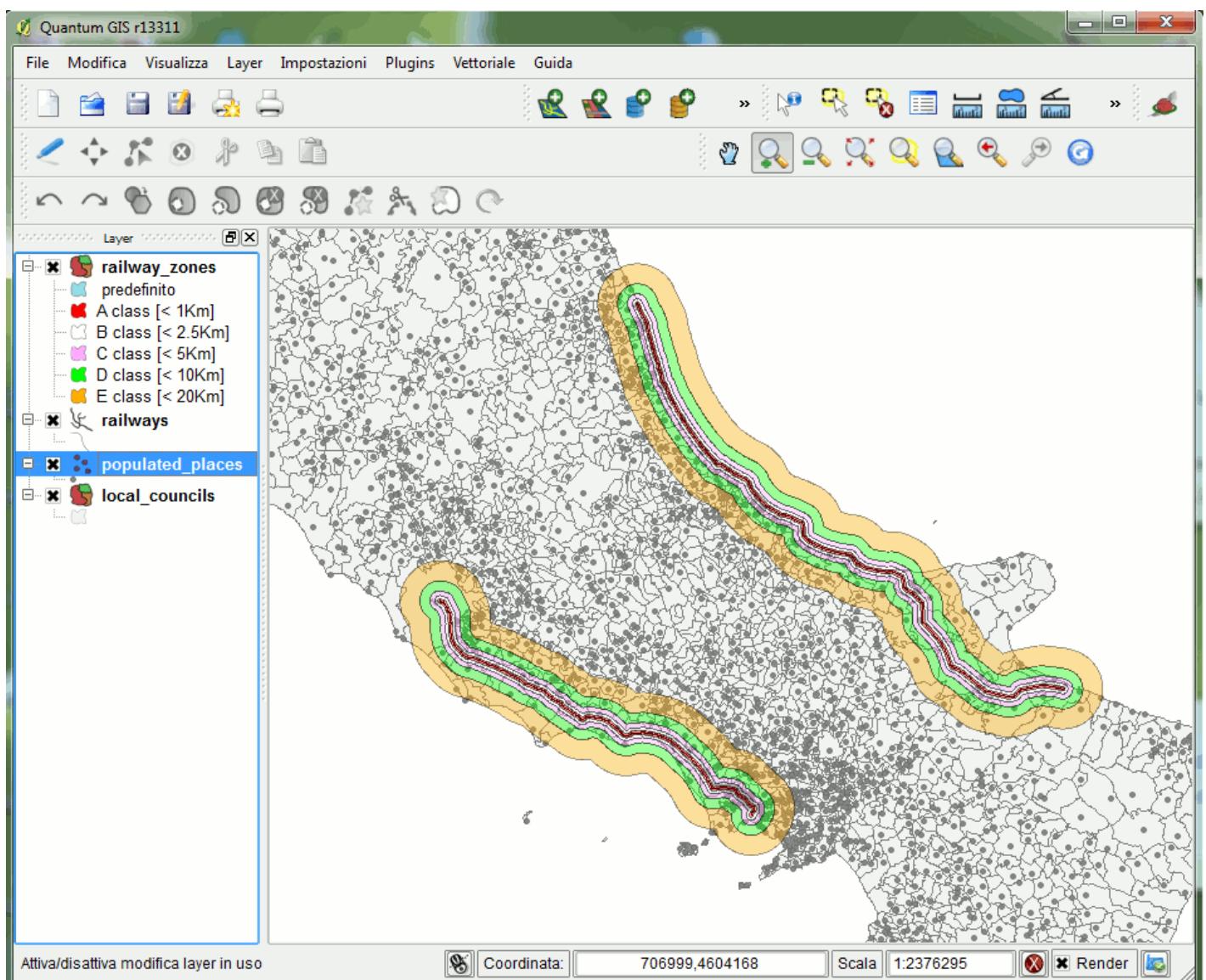
Ricetta # 18

Zone ferroviarie come Buffers

Problema

Questa è una sorta di conclusione *visiva* dell' ultimo esercizio. Il problema ora è:

- creare una **mappa adatta** a rappresentare le classi **A, B, C, D ed E** come le abbiamo definite in precedenza.



```
CREATE TABLE railway_zones (
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    railway_name TEXT NOT NULL,
    zone_name TEXT NOT NULL);
```

```
SELECT AddGeometryColumn('railway_zones', 'geometry',
    23032, 'MULTIPOLYGON', 'XY');
```

Stiamo creando una nuova tabella:

- come sempre, prima creiamo la tabella, tralasciando la colonna Geometry
- e quindi creiamo la colonna Geometry in un secondo momento, con il comando `AddGeometryColumn()`; ma adesso sapete già tutto questo
- adottare per questa nuova tabella il sistema (**SRID**) **23032 [ED50 UTM zone 32]** è una scelta assolutamente ovvia; dopo tutto la tabella originale delle ferrovie [railways] è nello stesso identico sistema
- la dichiarazione di una geometria MULTYPOLYGON è meno ovvio: ma vedremo in seguito perché questo è necessario.

```
INSERT INTO railway_zones
(id, railway_name, zone_name, geometry)
SELECT NULL, name, 'A class [< 1Km]',
CastToMultiPolygon(
    ST_Buffer(geometry, 1000.0))
FROM railways;
```

C'è poco di interessante in questa istruzione `INSERT ... SELECT ...` (*ancora, voi conoscete già tutto questo*). Eccetto il seguente argomento:

- useremo la funzione `ST_Buffer()` per creare un POLYGON corrispondente alla **classe A (<1Km)**
- **avvertenza**: non è per niente facile indovinare il tipo esatto di geometria generata dalla funzione `ST_Buffer()`; qualche volta questa funzione genera un **POLYGON**, ma altre volte può generare un **MULTIPOLYGON** (*questo dipende dalla sagoma esatta della linea di input, ed ovviamente il raggio di buffer ha, a sua volta, una forte influenza*)
- così, per evitare qualsiasi incongruenza sul tipo, abbiamo definito una geometria **MULTIPOLYGON** per questa tabella
- e stiamo obbligando il tipo di geometria ricorrendo esplicitamente alla funzione di conversione `CastToMultiPolygon()`.

```
INSERT INTO railway_zones
(id, railway_name, zone_name, geometry)
SELECT NULL, name, 'B class [< 2.5Km]',
CastToMultiPolygon(
    ST_Difference(
        ST_Buffer(geometry, 2500.0),
        ST_Buffer(geometry, 1000.0)))
FROM railways;
```

```
INSERT INTO railway_zones
(id, railway_name, zone_name, geometry)
SELECT NULL, name, 'C class [< 5Km]' ,
CastToMultiPolygon(
ST_Difference(
    ST_Buffer(geometry, 5000.0),
    ST_Buffer(geometry, 2500.0)))
FROM railways;
```

```
INSERT INTO railway_zones
(id, railway_name, zone_name, geometry)
SELECT NULL, name, 'D class [< 10Km]' ,
CastToMultiPolygon(
ST_Difference(
    ST_Buffer(geometry, 10000.0),
    ST_Buffer(geometry, 5000.0)))
FROM railways;
```

```
INSERT INTO railway_zones
(id, railway_name, zone_name, geometry)
SELECT NULL, name, 'E class [< 20Km]' ,
CastToMultiPolygon(
ST_Difference(
    ST_Buffer(geometry, 20000.0),
    ST_Buffer(geometry, 10000.0)))
FROM railways;
```

Creare ogni ulteriore zona non è difficile:

- usiamo semplicemente la funzione **ST_Difference()** in modo da avere la rappresentazione appropriata; in altre parole, dobbiamo creare un buco interno, in modo da escludere la sovrapposizione con ogni altra zona già creata

Potete adesso eseguire un *semplice controllo visivo* usando **QGIS**. Ed è tutto.



Febbraio 2011

Ricetta # 19

Unione [Merging] dei Comuni in Province e così via ...

Problema

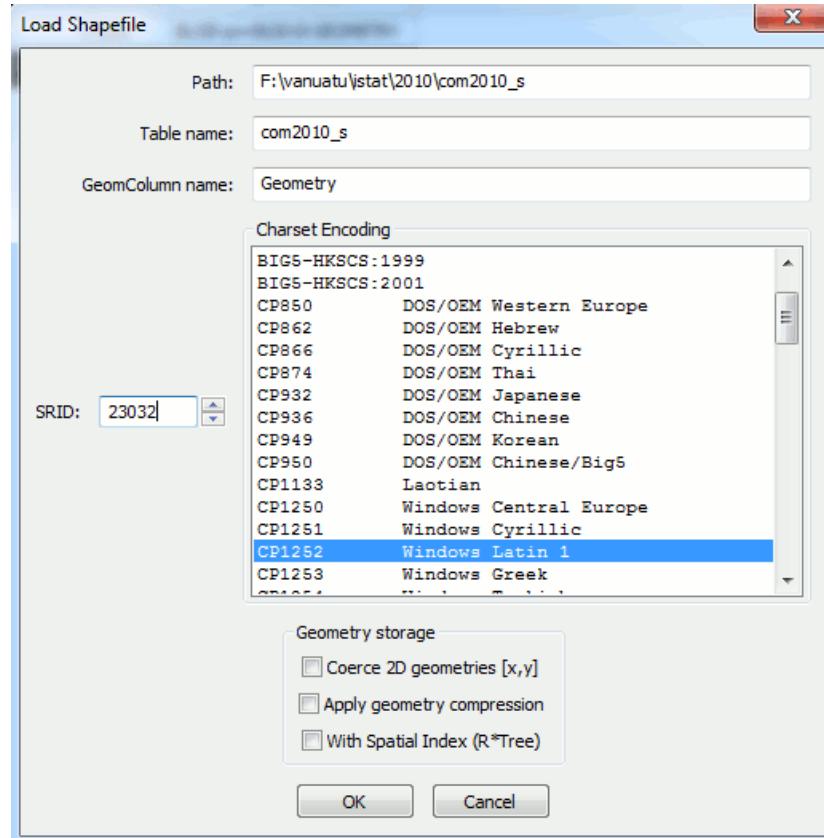
Comuni, Province e Regioni seguono una gerarchia ben definita. Per motivi amministrativi l'Italia è suddivisa in Regioni; le Regioni sono suddivise in Province e le Province sono suddivise in Comuni.

Seguendo corrette procedure con Spatial SQL potete partire dalle geometrie dei Comuni, quindi generare le geometrie delle corrispondenti Province. E così via....

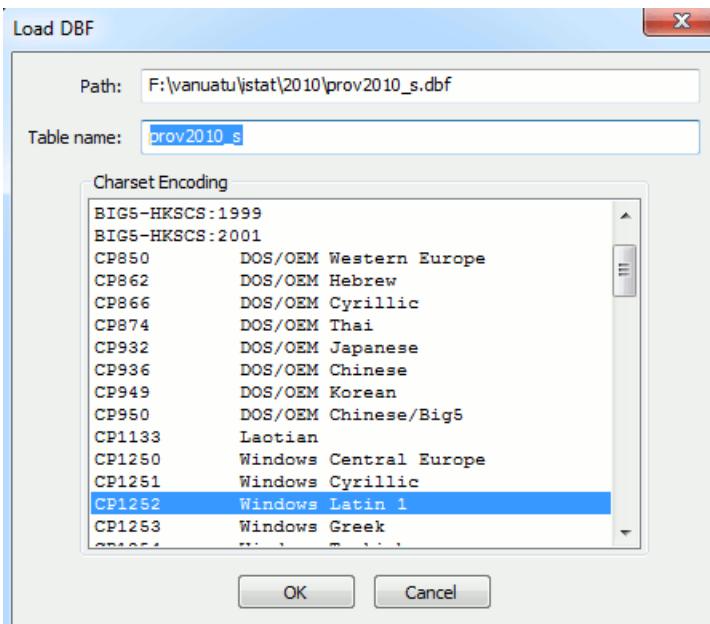
Avvertenza importante: il censimento ISTAT 2001 non è molto adatto allo scopo, poiché è affetto da numerose inconsistenze topologiche.

Usiamo invece gli ultimi dati ISTAT 2010, che presentano una qualità e coerenza molto migliori.

- <http://www.istat.it/ambiente/cartografia/comuni2010.zip>
- <http://www.istat.it/ambiente/cartografia/province2010.zip>
- <http://www.istat.it/ambiente/cartografia/regioni2010.zip>



Cominciate a creare un nuovo DB; quindi mediante **spatialite_gui** importate lo **shapefile com2010_s**.



Il prossimo passo è quello di caricare l'archivio **prov2010_s**: sì, anche questo è uno *shapefile*. Per i vostri specifici scopi potete ignorare del tutto le geometrie delle Province (*generare tutte queste è l'esercizio specifico assegnatovi questa volta, va bene?*).

Potete importare solo il file **.DBF**, importando tutti i dati ma tralasciando ed ignorando del tutto le relative geometrie.

Poi potete importare il file **.DBF** delle **Regioni**, esattamente come prima.

```
CREATE VIEW local_councils AS
SELECT c.cod_reg AS cod_reg,
       c.cod_pro AS cod_pro,
       c.cod_com AS cod_com,
       c.nome_com AS nome_com,
       p.nome_pro AS nome_pro,
       p.sigla AS sigla,
       r.nome_reg AS nome_reg,
       c.geometry AS geometry
FROM com2010_s AS c
JOIN prov2010_s AS p USING (cod_pro)
JOIN reg2010_s AS r USING(cod_reg)

SELECT * FROM local_councils
```

cod_reg	cod_pro	cod_com	nome_com	nome_pro	sigla	nome_reg	geometry
1	1	1	Agliè	Torino	TO	PIEMONTE	BLOB sz=1117 GEOMETRY
1	1	2	Airasca	Torino	TO	PIEMONTE	BLOB sz=1149 GEOMETRY
1	1	3	Ala di Stura	Torino	TO	PIEMONTE	BLOB sz=1933 GEOMETRY
...

Questo creerà la VIEW comuni [**local_councils VIEW**]; questa VIEW rappresenta una semplice tabella piatta, non normalizzata, in grado di rendere ogni successiva attività del tutto banale.

```

CREATE TABLE counties AS
SELECT cod_pro, nome_pro, sigla, cod_reg, nome_reg,
       ST_Union(geometry) AS geometry
FROM local_councils
GROUP BY cod_pro;

```

```

SELECT RecoverGeometryColumn('counties', 'geometry',
                           23032, 'MULTIPOLYGON', 'XY');

```

Ora creerete e popolerete la tabella delle **Province [countries]**:

- si usa la funzione SQL spaziale **ST_Union()** per unire/aggregare le geometrie
- usando la clausola **GROUP BY cod_pro**, la funzione **ST_Union()** agirà come una *funzione di aggregazione*, costruendo quindi la rappresentazione geometrica di ogni singola Provincia
- **attenzione**: dovete assolutamente eseguire il comando **RecoverGeometryColumn()** in modo da registrare correttamente la colonna **counties.geometry** nella tabella *metadata geometry_columns*.

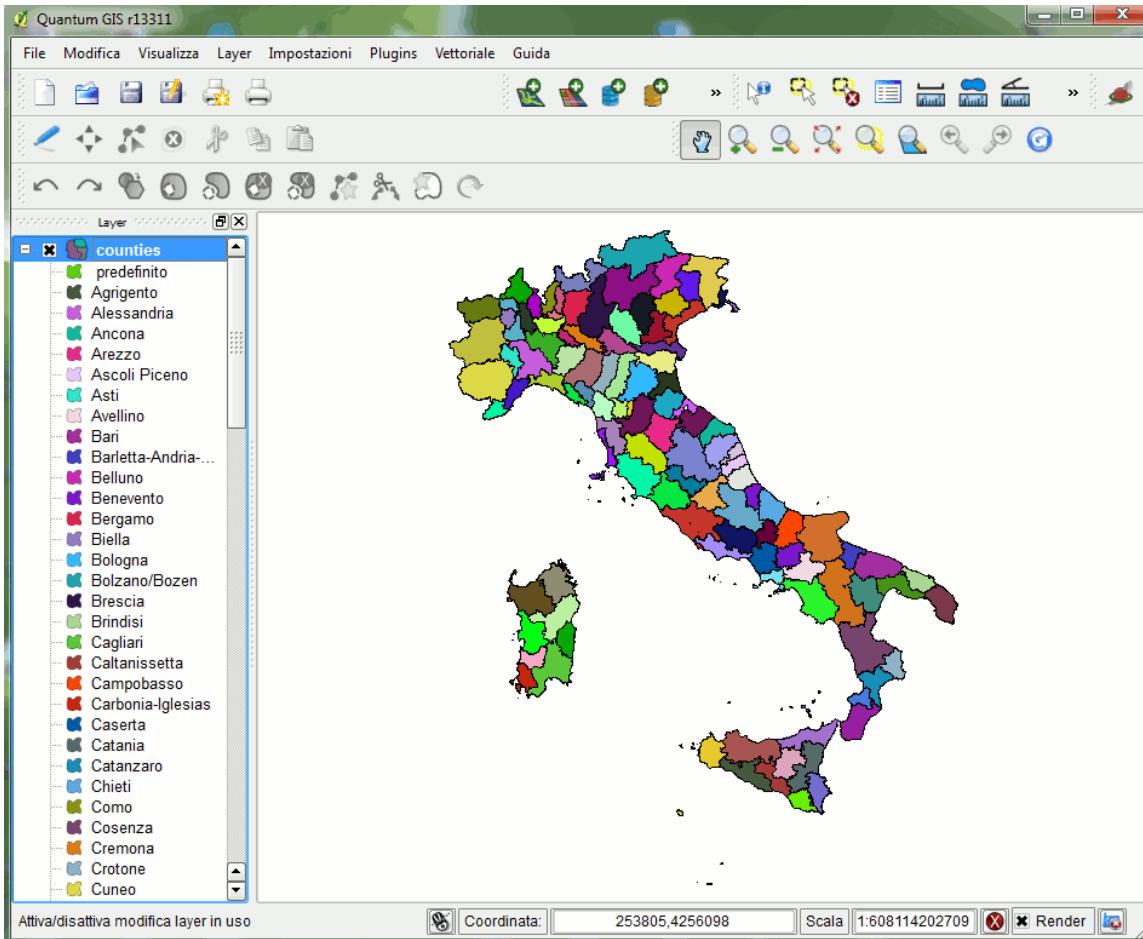
```

SELECT * FROM counties;

```

cod_pro	nome_pro	sigla	cod_reg	nome_reg	geometry
1	Torino	TO	1	PIEMONTE	BLOB sz=36337 GEOMETRY
2	Vercelli	VC	1	PIEMONTE	BLOB sz=27357 GEOMETRY
3	Novara	NO	1	PIEMONTE	BLOB sz=15341 GEOMETRY
...

Ed infine siete pronti a visualizzare la mappa delle provincie [counties] mediante QGIS.



```

CREATE TABLE regions (
    cod_reg INTEGER NOT NULL PRIMARY KEY,
    nome_reg TEXT NOT NULL);

SELECT AddGeometryColumn('regions', 'geometry',
    23032, 'MULTIPOLYGON', 'XY');

INSERT INTO regions (cod_reg, nome_reg, geometry)
SELECT cod_reg, nome_reg, ST_Union(geometry)
FROM counties
GROUP BY cod_reg;

```

Adesso creerete e popolerete la tabella delle regioni [**regions**]:

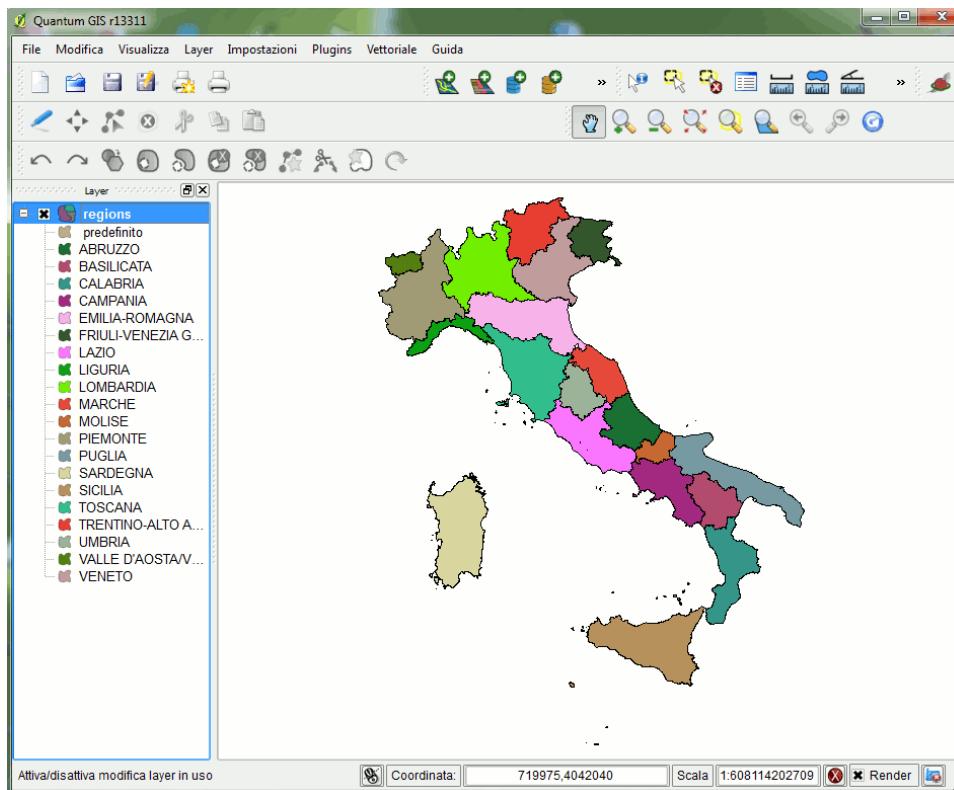
- come nel caso precedente userete la funzione **ST_Union()** e la clausola **GROUP BY** per costruire le geometrie delle regioni
- **notate bene:** in questo esempio avete creato esplicitamente la tabella delle regioni, e quindi con il comando **AddGeometryColumn()** avete creato la colonna **regions.geometry**. Infine con i comandi **INSERT INTO ... (...) SELECT ...** avete popolato la tabella.

Il procedimento è diverso, ma il risultato finale è esattamente lo stesso dell'esempio precedente.

```
SELECT * FROM regions;
```

cod_reg	nome_reg	geometry
1	PIEMONTE	BLOB sz=75349 GEOMETRY
2	VALLE D'AOSTA/VALLÉE D'AOSTE	BLOB sz=18909 GEOMETRY
3	LOMBARDIA	BLOB sz=83084 GEOMETRY
...

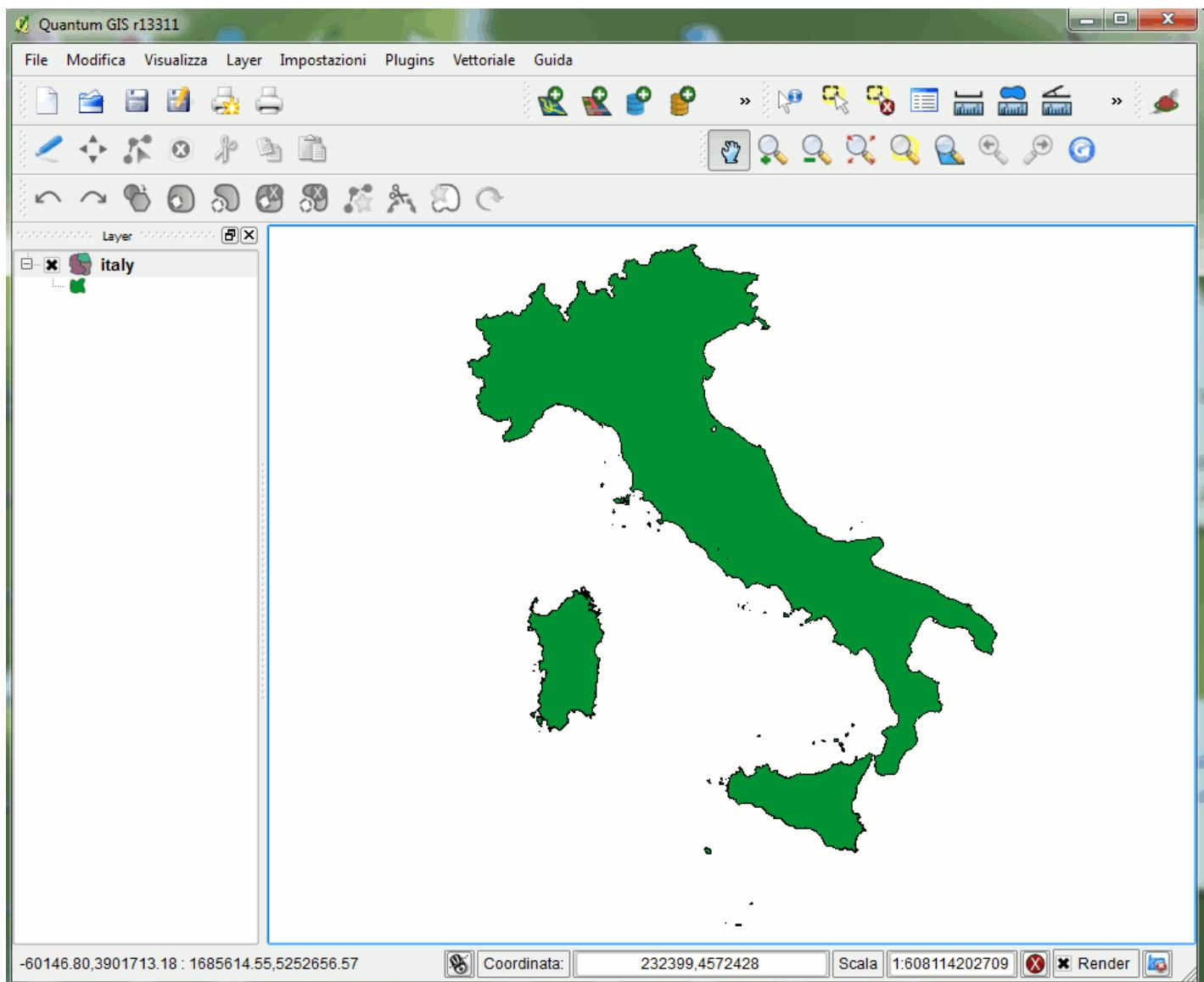
Un controllo veloce ...



```
CREATE TABLE italy AS
SELECT 'Italy' AS country,
       ST_Union(geometry) AS geometry
FROM regions;

SELECT RecoverGeometryColumn('italy', 'geometry',
  23032, 'MULTIPOLYGON', 'XY');
```

Come passo finale potrete ora creare la tabella Italia [**Italy**] con i confini internazionali dell'intera Repubblica Italiana.



E adesso potete visualizzare la mappa con QGis... ed è tutto.



Febbraio 2011

Ricetta # 20

Spatial Views

SpatiaLite gestisce le Spatial Views: una spatial view correttamente definita può essere usata come ogni altra mappa, ad es. può essere visualizzata con QGIS.

Avvertenza: le SQLite VIEWS possono essere usate in sola lettura (**SELECT**) ovviamente tale limitazione si applica anche alle Spatial View (i comandi **INSERT**, **DELETE** o **UPDATE** non sono ammessi).

Il costruttore di interrogazioni

Spatialite_gui fornisce un *costruttore di interrogazioni* [*query composer tool*]; in questo primo esempio useremo proprio questo.

The screenshot shows the Spatialite GUI's Query / View Composer window. The SQL statement pane contains the following code:

```

SELECT "a"."lc_id" AS "lc_id", "a"."lc_name" AS "lc_name",
       "a"."population" AS "population", "a"."geometry" AS "geometry",
       "b"."county_id" AS "county_id", "b"."county_name" AS "county_name",
       "b"."car_plate_code" AS "car_plate_code"
  FROM "local_councils" AS "a"
 JOIN "counties" AS "b" USING ("county_id")

```

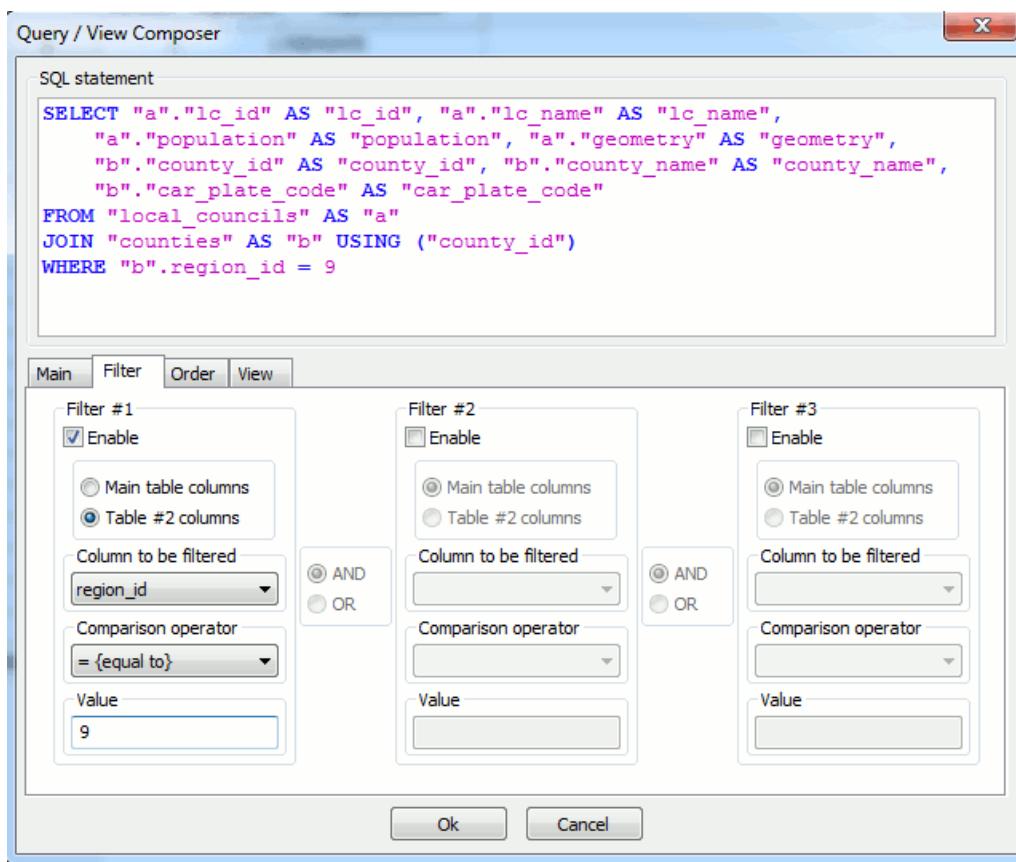
The interface includes tabs for Main, Filter, Order, and View. Under the Main tab, the Main Table is set to "local_councils" with alias "a". The columns selected are ROWID, lc_id, lc_name, population, county_id, and geometry. The Table #2 is set to "counties" with alias "b". The Join mode is set to [Inner] Join. Three join conditions are defined:

- Join match #1: Main Table column county_id, Table #2 column county_id
- Join match #2: Main Table column (empty), Table #2 column (empty)
- Join match #3: Main Table column (empty), Table #2 column (empty)

At the bottom are Ok and Cancel buttons.

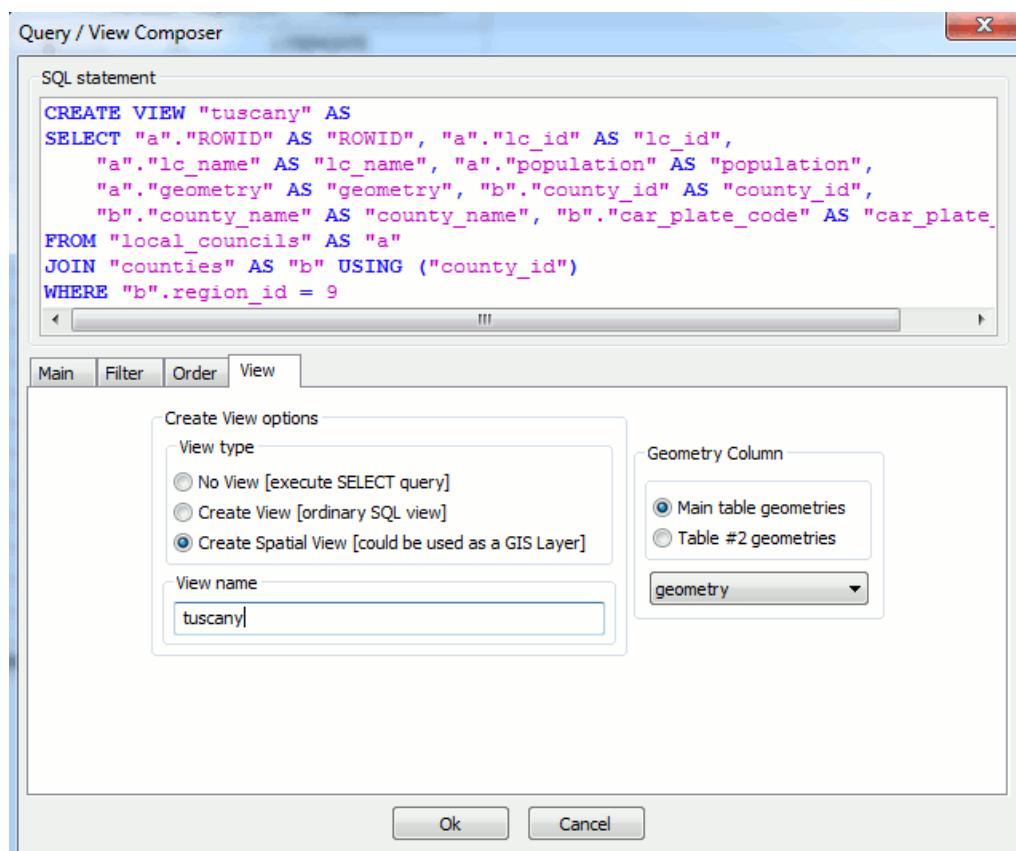
Passo 1: selezionare le tabelle e le colonne necessarie e definire le condizioni JOIN

In questo primo esempio collegheremo (**JOIN**) le tabelle dei comuni [**local_councils**] e delle provincie [counties].



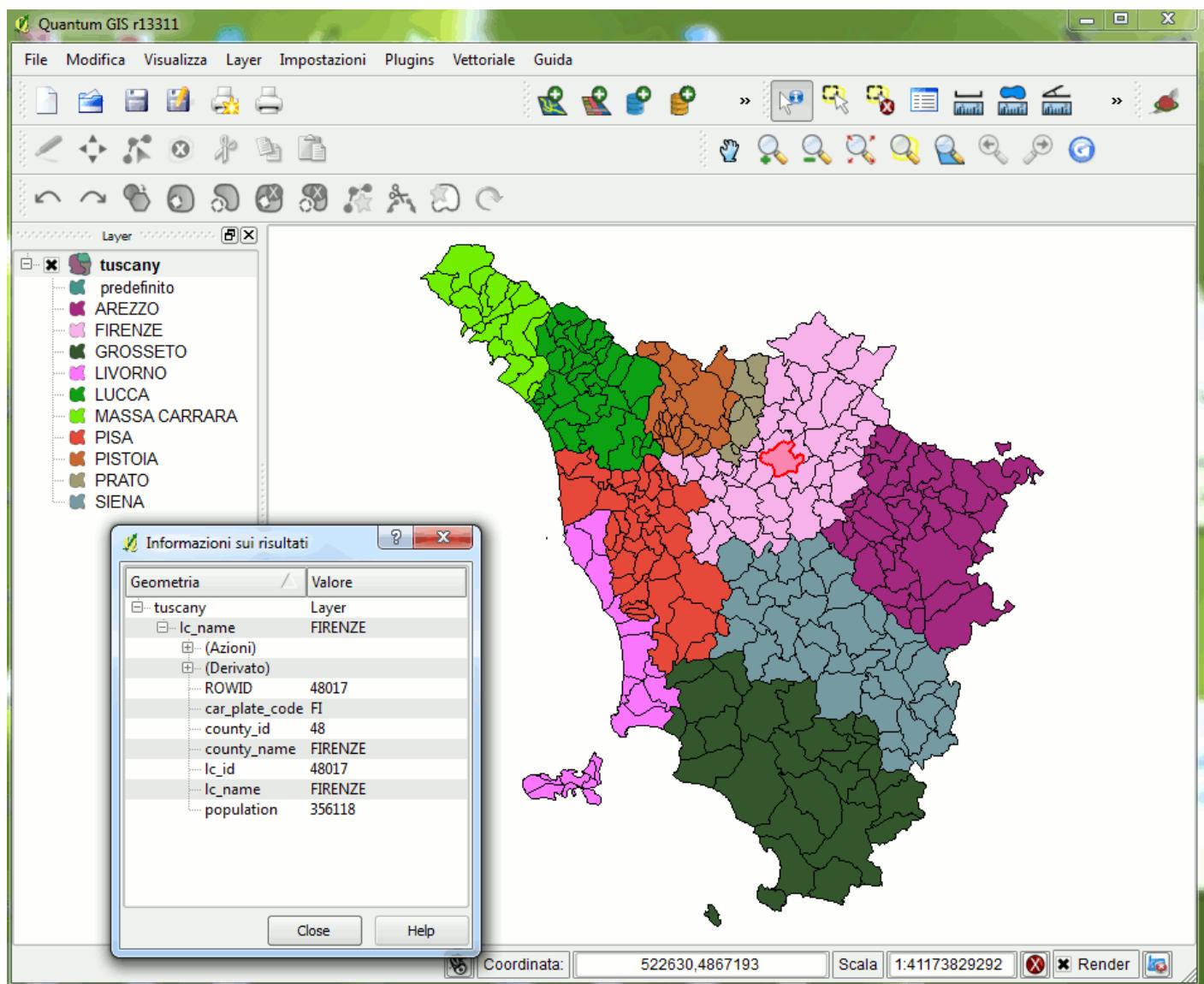
Passo 2: specificare una clausola *filtro*

In questo caso estrarremo solo i comuni e le provincie appartenenti alla Regione Toscana (`region_id = 9`).



Passo 3: diamo un nome alla **VIEW**

durante quest'ultima fase sceglieremo la colonna Geometry corrispondente a questa **VIEW**.



Siamo adesso in grado di visualizzare la Spatial View mediante QGIS (abbiamo applicato un appropriato **rendering tematico** [tematismo] per evidenziare le Province).

Scrivere a mano la vostra Spatial VIEW

```

CREATE VIEW italy AS
SELECT lc.ROWID AS ROWID,
       lc.lc_id AS lc_id,
       lc.lc_name AS lc_name,
       lc.population AS population,
       lc.geometry AS geometry,
       c.county_id AS county_id,
       c.county_name AS county_name,
       c.car_plate_code AS car_plate_code,
       r.region_id AS region_id,
       r.region_name AS region_name
FROM local_councils AS lc
JOIN counties AS c ON (lc.county_id = c.county_id)
JOIN regions AS r ON (c.region_id = r.region_id)
    
```

Non siete obbligati ad usare il *costruttore di interrogazioni*. Siete assolutamente liberi di definire ogni arbitraria VIEW da usare come Spatial View.

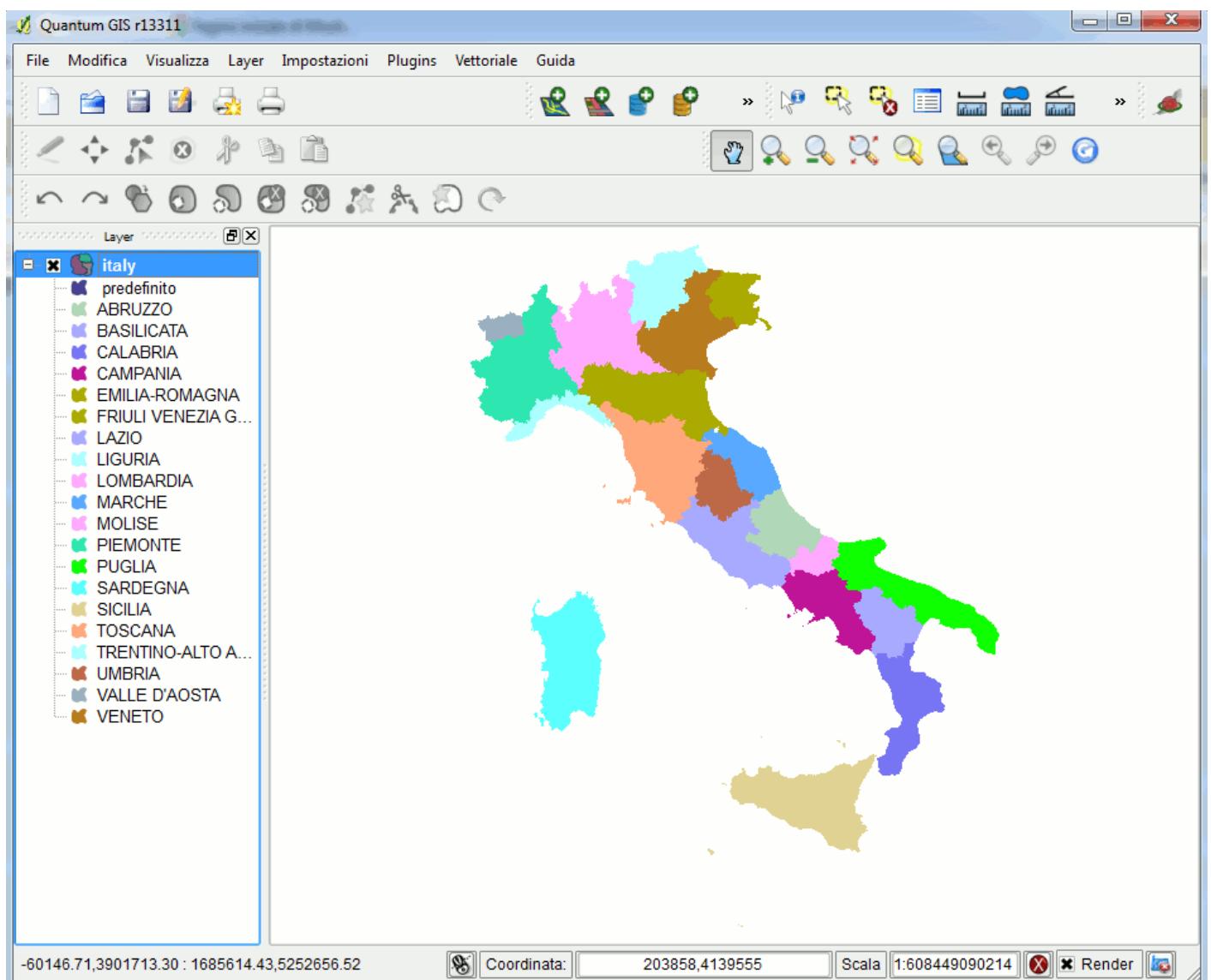
```
INSERT INTO views_geometry_columns
  (view_name, view_geometry, view_rowid, f_table_name, f_geometry_column)
VALUES ('italy', 'geometry', 'ROWID', 'local_councils', 'geometry');
```

In ogni caso dovete registrare questa **VIEW** nelle **views_geometry_columns** per farla diventare una vera *e propria Spatial View*.

```
SELECT * FROM views_geometry_columns;
```

view_name	view_geometry	view_rowid	f_table_name	f_geometry_column
tuscany	geometry	ROWID	local_councils	geometry
italy	geometry	ROWID	local_councils	geometry

Un piccolo controllo ...



E finalmente possiamo visualizzare questa Spatial View con QGIS (abbiamo applicato un rendering tematico [tematismo] per evidenziare le Regioni).



Una raffinata esperienza culinaria: da Dijkstra

SpatiaLite gestisce un modulo di “**routing**” denominato **VirtualNetwork** (rete virtuale). Lavorando su una rete arbitraria questo modulo consente di identificare le **connessioni di percorso** minimo (shortest path) con una semplice interrogazione SQL.

Il modulo **VirtualNetwork** si appoggia su algoritmi sofisticati ed altamente ottimizzati, così è veramente veloce ed efficiente anche nel caso di reti di grande dimensione.

Nozioni basilari sulle reti

Non potete presumere che qualsiasi generica *mappa di strade* corrisponda ad una rete. Una *vera rete* deve soddisfare parecchi requisiti specifici, ad es. deve essere un **grafo**.

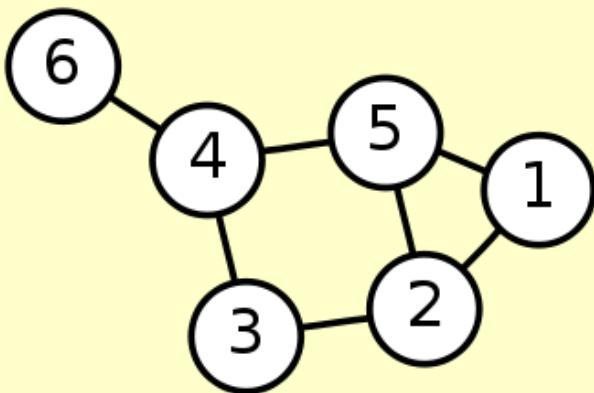
La **teoria dei Grafi** è un'ampia e complessa area della matematica; se siete interessati a ciò, qui potete trovare ulteriori dettagli:

[Teoria Dei Grafi \(Graph Theory\)](#)

[Problema del Percorso Minimo \(Shortest Path Problem\)](#)

[Algoritmo di Dijkstra \(Dijkstra's Algorithm\)](#)

[Algoritmo A* \(A* Algorithm\).](#)



Spiegato in poche parole:

- una rete è un insieme di **archi**
- ogni arco connette due **nodi**
- ogni arco ha una **direzione** univoca: ad es. l'arco dal nodo A al nodo B non è necessariamente lo stesso dell'arco che va dal nodo B al nodo A;
- ogni arco ha un “**costo**” conosciuto (ad es. lunghezza, tempo di percorrenza, capacità,)
- archi e nodi devono essere **univocamente identificati** da etichette
- la geometria del grafo (archi e nodi) deve soddisfare una **forte coerenza topologica**.

Partendo da una **rete** (o **grafo**) sia l'algoritmo di **Dijkstra** che l'algoritmo **A*** possono individuare il **percorso minimo** (*connessione con il minimo costo*) che connette ogni coppia arbitraria di punti.

Vi sono parecchie fonti che distribuiscono dati di tipo rete. Una delle più note e largamente usate è **OSM** [*Open Street Map*], un archivio di dimensione planetaria completamente libero. Vi sono parecchi siti dove scaricare OSM; tanto per citarne qualcuno:

- <http://www.openstreetmap.org/>
- <http://download.geofabrik.de/osm/>
- <http://downloads.cloudmade.com/>

Comunque per l'esempio seguente abbiamo scaricato l'archivio OSM necessario da www.gfoss.it. Più precisamente, abbiamo scaricato l'archivio **TOSCANA.osm.bz2**.

Passo 1: Dovete decomprimere l'archivio OSM. Questo file è compresso usando l'algoritmo **bzip2**, largamente fornito da numerosi strumenti di software libero: ad es. potete usare **7-zip** per scompattare il file: www.7-zip.org

Passo 2: Tutti gli archivi OSM sono semplicemente dei file **XML** (*potete aprire questo file usando un banale editor di testi di vostra scelta*). SpatiaLite fornisce uno specifico strumento CLI che consente di caricare un archivio OSM in un DB: **spatialite_osm_net** .

```
>spatialite_osm_net -o TOSCANA.osm -d tuscany.sqlite -T tuscany -m
SQLite version: 3.7.4
SpatiaLite version: 2.4.0-RC5
using IN-MEMORY database
Loading OSM nodes ... wait please ...
    Loaded 1642867 OSM nodes
Verifying OSM ways ... wait please ...
    Verified 60893 OSM ways
Disambiguating OSM nodes ... wait please ...
    Found 40 duplicate OSM nodes - fixed !!!
Loading network ARCs ... wait please ...
    Loaded 121373 network ARCs
Dropping temporary table 'osm_tmp_nodes' ... wait please ...
    Dropped table 'osm_tmp_nodes'
Dropping index 'from_to' ... wait please ...
    Dropped index 'from_to'
exporting IN_MEMORY database ... wait please ...
    IN_MEMORY database successfully exported
VACUUMing the DB ... wait please ...
    All done: OSM graph was successfully loaded
>
```

Spiegazione veloce:

- **-o TOSCANA.osm** seleziona l'archivio OSM di input
- **-d tuscanysqlite** seleziona il DB da creare e popolare
- **-T tuscanysqlite** creerà la tabella Geometry che memorizza l'archivio OSM
- **-m** forzerà l'uso di un database in-memory, in modo da eseguire l'importazione nel modo più veloce.

```
SELECT *
FROM tuscanysqlite;
```

id	osm_id	class	node_from	node_to	name	oneway_from_to	oneway_to_from	length	cost	geometry
...
2393	8079944	tertiary	659024545	659024546	Via Cavour	1	1	7.468047	0.537699	BLOB sz=80 GEOMETRY
2394	8079944	tertiary	659024546	156643876	Via Cavour	1	1	12.009911	0.864714	BLOB sz=96 GEOMETRY
2395	8083989	motorway	31527668	319386487	Autostrada del Sole	1	0	424.174893	13.882087	BLOB sz=80 GEOMETRY
2396	8083990	motorway	31527665	31527668	Autostrada del Sole	1	0	130.545183	4.272388	BLOB sz=112 GEOMETRY
...

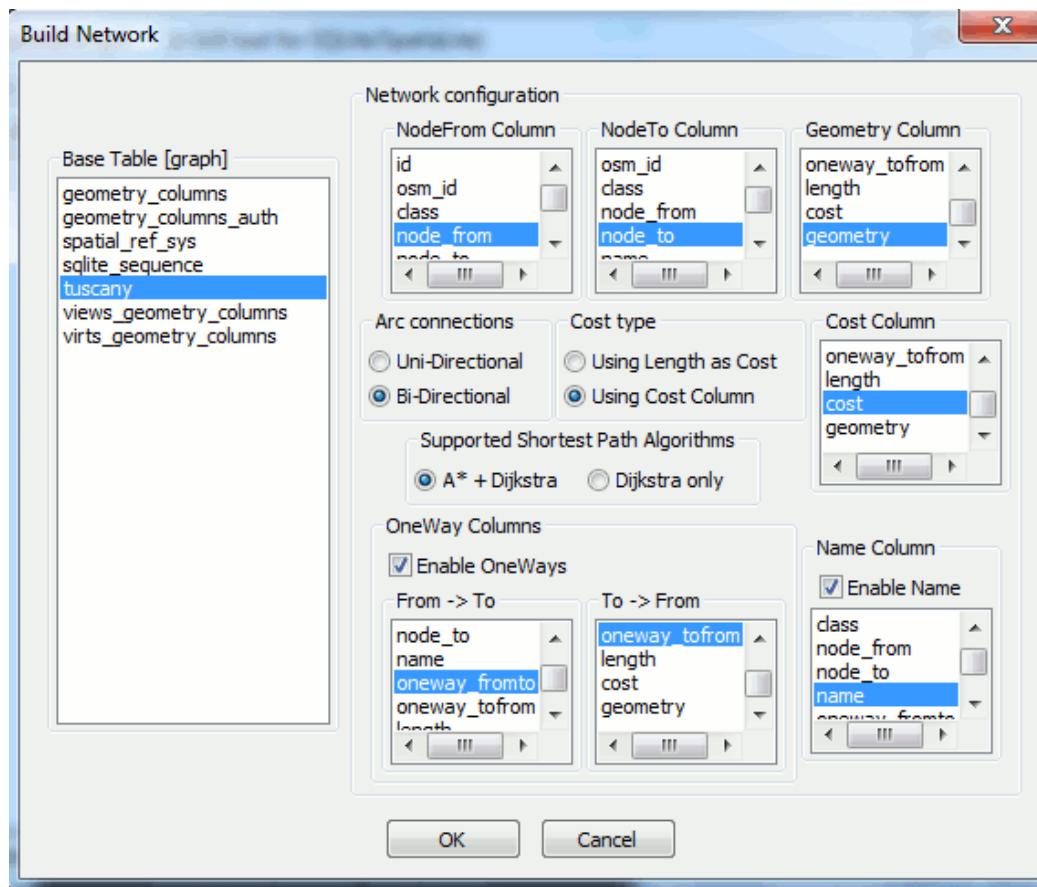
Un piccolo controllo:

- **spatialite_osm_net** crea una sola tabella Toscana nel DB
- ogni riga corrisponde ad un singolo **arco di rete**
- i nodi connessi dall'arco sono identificati da **node_from** e **node_to**
- **oneway_from_to** e **oneway_to_from** determinano se l'arco può essere percorso nelle due direzioni o no
- **lunghezza** (length) è la lunghezza geometrica dell'arco (misurata in **metri**)
- **costo** (cost) è il tempo di percorrenza stimato in **secondi**
- **geometria** è la **LINESTRING** che rappresenta l'arco.

Avvertenza 1: i **nodi** non sono rappresentati in modo separato, ma sono indirettamente recuperati dal corrispondente arco.

Avvertenza 2: questa è senza dubbio una **rete reale**, ma in questa forma non può consentire l'interrogazione dei percorsi (**query routing**).

E' necessario ancora un passo, cioè la creazione della tabella **VirtualNetwork**.



Useremo **spatialite_gui** per creare la tabella VirtualNetwork. Comunque la stessa operazione è consentita anche dallo strumento **spatialite_network** CLI (inoltre CLI fornisce estese capacità diagnostiche, utili per identificare eventuali problemi).

```
SELECT *
FROM tuscanynet
WHERE NodeFrom = 267209305
AND NodeTo = 267209702;
```

Algorithm	ArcRowid	NodeFrom	NodeTo	Cost	Geometry	Name
Dijkstra	NULL	267209305	267209702	79.253170	BLOB sz=272 GEOMETRY	NULL
Dijkstra	11815	267209305	250254381	11.170037	NULL	Via Guelfa
Dijkstra	11816	250254381	250254382	8.583739	NULL	Via Guelfa
Dijkstra	11817	250254382	250254383	12.465016	NULL	Via Guelfa
Dijkstra	16344	250254383	256636073	15.638407	NULL	Via Cavour
Dijkstra	67535	256636073	270862435	3.147105	NULL	Piazza San Marco
Dijkstra	25104	270862435	271344268	5.175379	NULL	Piazza San Marco
Dijkstra	25105	271344268	82591712	3.188657	NULL	Piazza San Marco
Dijkstra	11802	82591712	267209666	4.978328	NULL	Piazza San Marco
Dijkstra	20773	267209666	267209702	14.906501	NULL	Via Giorgio La Pira

E finalmente potete controllare la vostra prima interrogazione di navigazione (query routing):

- dovete semplicemente definire la clausola **WHERE NodeFrom = ... AND NodeTo = ...**
- ed il risultato rappresenterà la soluzione di **percorso minimo**
- la *prima riga* del risultato sintetizza l'intero percorso e contiene la corrispondente geometria
- le *altre righe* rappresentano i singoli archi da percorrere, nel giusto ordine, per andare dall'origine alla destinazione.

```
UPDATE tuscanynet SET Algorithm = 'A*';
```

```
UPDATE tuscanynet SET Algorithm = 'Dijkstra';
```

Le tabelle **VirtualNetwork** di SpatiaLite forniscono due diversi algoritmi:

- l'algoritmo **Dijkstra's shortest path (percorso più breve)** è un *classico* algoritmo di navigazione, basato su estesi assunti matematici, e troverà sicuramente la soluzione ottimale
- L'algoritmo A* è un metodo alternativo basata su assunti *euristici*: è normalmente più veloce di quello Dijkstra, ma in condizioni anomale può anche fallire, o restituire una soluzione non ottimale.
- comunque passare dall'uno all'altro è proprio semplice
- la selezione predefinita usa l'algoritmo di Dijkstra.

Una tabella **VirtualNetwork** rappresenta una *immagine statica* della rete in esame. Questo consente di adottare una rappresentazione binaria altamente efficiente (*in altre parole: consente di produrre risultati in tempi molto stretti*), ma non consente ovviamente modifiche dinamiche.

Ogni volta che la rete in esame cambia, la corrispondente tabella **VirtualNetwork** va eliminata e ricreata, così da riflettere con esattezza l'ultimo stato della rete.

In molte situazioni questo non per nulla un problema; ma in *situazioni altamente dinamiche* questo può essere una scocciatura. **Siate ben consapevoli di questi limiti.**



Dolci, liquori, the e caffè

Febbraio 2011



Suggerimenti sul livello di prestazioni del sistema
Importare/esportare Shapefile (DBF,TXT,...)
Linguaggi di collegamento(C / C ++, Java, Python, PHP ...)



Suggerimenti sul livello di prestazioni del sistema

Finora abbiamo esaminato vari aspetti legati alla ottimizzazione: ma tutto era principalmente inteso come “*scrivere con sapienza interrogazioni ben congegnate*”.

Sebbene la preparazione di interrogazioni SQL ben progettate è sicuramente il fattore principale per raggiungere prestazioni ottimali, questo non è tutto. Esiste un secondo livello di ottimizzazione prestazionale (*fine tuning*), ad es. quello relativo alle interazioni fra il DBMS ed i sottostanti *sistema operativo e file system*

DB pages / page cache

Ogni SQLite DB è semplicemente un file monolitico: tutti i dati e le relative informazioni sono memorizzate all'interno dei file. Come in molti altri DBMS, lo spazio disco non è allocato a caso, ma è strutturato in modo opportuno:

L'unità atomica è la **pagina**, pertanto un DB non è altro che una collezione ben organizzata di *pagine [pages]*. Tutte le pagine all'interno di un DB devono avere la stessa identica dimensione (tipicamente **1KB = 1024 bytes**):

- l'adozione di una dimensione maggiore può ridurre il traffico I/O, ma può causare un significativo spreco di spazio inusato
- una dimensione minore è fortemente scoraggiata poiché genererà sicuramente un traffico molto maggiore
- pertanto la dimensione standard di 1KB rappresenta una condizione media ben adatta alla grande maggioranza delle situazioni reali.

Leggere e scrivere dal disco una singola pagina ogni volta non è un metodo efficiente; quindi SQLite mantiene la pagina in una memoria interna temporanea [**page cache**] (*allocata in RAM*), consentendo un accesso molto veloce alle pagine più cercate.

Come è facile capire, l'adozione di memoria interna [**page cache**] più grande riduce il traffico I/O totale; e conseguentemente si può raggiungere una *maggior quantità di risultati [maggiore produttività]*

Normalmente SQLite adotta un approccio molto conservativo, tale da richiedere una quantità di memoria molto bassa; la memoria temporanea iniziale contiene 2000 pagine (corrispondenti ad una allocazione totale di soli 20MB).

Ma una così piccola quantità di memoria sicuramente sarà insufficiente per gestire correttamente **grandi DB** (ad es. quelli quantificabili in **numerosi GB**); questo può facilmente diventare un reale **collo di bottiglia** e causare prestazioni globali davvero cattive.

PRAGMA page_size;
1024
PRAGMA page_count;
31850
PRAGMA freelist_count;
12326

Avete a disposizione parecchie istruzioni PRAGMA per controllare lo stato delle pagine [pages] del DB corrente:

- l'istruzione **PRAGMA page_size**; riporterà la dimensione corrente della pagina
- **PRAGMA page_count**; riporterà il numero totale delle pagine allocate
- **PRAGMA freelist_count**; riporterà il numero totale di pagine non usate.

Avvertenza: ogni volta che eseguite dei comandi **DELETE** o **DROP [TABLE | INDEX]** in quantità, verranno lasciate nel DB parecchie pagine inutilizzate.

PRAGMA page_size = 4096;
PRAGMA page_size;
1024

Potete eseguire l'istruzione PRAGMA page_size per specificare una dimensione differente per la pagina (dovete specificare una *potenza di 2* come argomento, variando da **512** a **65536**):

- comunque la dimensione di pagina rimane immutata
- questo perchè è necessaria una riorganizzazione completa del DB perchè tali modifiche abbiano effetto.

VACUUM;

L'esecuzione del comando **VACUUM** implica l'effettuazione delle seguenti azioni:

- l'intero DB sarà controllato e completamente riscritto da capo
- ogni cambiamento strutturale (*ad es. il cambio della dimensione della pagina*) sarà allora eseguito
- ogni pagina inutilizzata sarà scartata, così il DB sarà effettivamente ricompattato
- **avvertenza:** l'operazione **VACUUM** su un DB di grandi dimensioni richiederà molto tempo.

PRAGMA page_size;
4096
PRAGMA page_count;
5197
PRAGMA freelist_count;
0

Giusto un controllo veloce: subito dopo l'esecuzione di VACUUM è applicata la nuova dimensione della pagina, e non ci sono più pagine inutilizzate.

PRAGMA cache_size;
1000
PRAGMA cache_size = 1000000;
PRAGMA cache_size;
1000000

Potete usare il comando **PRAGMA cache_size** per conoscere o definire la memoria temporanea;

- **avvertenza:** la dimensione è misurata come *numero di pagine* memorizzabili: quindi la corrispondente dimensione in *bytes* sarà **page_size * cache_size**
- l' allocazione di una memoria temporanea maggiore dovrebbe implicare prestazioni migliori in ogni caso considerate attentamente che:
 - una memoria esageratamente grande è completamente inutile: butterete semplicemente una grande quantità di preziosa RAM per nulla
 - quando la richiesta di memoria per la memorizzazione temporanea delle pagine eccede la RAM fisica disponibile, le prestazioni scenderanno in modo catastrofico, poiché questo causerà un enorme traffico I/O per lo *scambio (swapping) memoria - disco*
 - sulle piattaforme a 32bit esiste un ulteriore limite: in questo caso ogni processo non può gestire più di 4GB di memoria
 - ma per motivi pratici tale limite è più basso, a circa 1.5GB.

L'adozione di una memoria temporanea molto generosa (*ma saggiamente dosata, senza troppo esagerare*) normalmente garantisce incrementi di prestazione, in particolare quando state lavorando con DB molto grandi.

Potete modificare un'altra importante predisposizione tramite un'apposito comando **PRAGMA** fornito da SQLite:

- il comando **PRAGMA ignore_check_constraint** può essere usato per controllare, abilitare o disabilitare un controllo dei vincoli [**CHECK constraints**](*nota: disabilitare il controllo dei vincoli è pericoloso, ma può essere necessario durante il caricamento iniziale dei dati*).
- il comando **PRAGMA foreign_key** può essere usato per controllare, abilitare o disabilitare il vincolo **FOREIGN KEY** (*anche questo può essere utile o necessario durante l'iniziale caricamento dei dati*)
- il comando **PRAGMA journal_mode** può essere usato per controllare o affinare dettagli circa il journaling delle **TRANSACTION**

L'implementazione dei comandi **PRAGMA** cambia di tanto in tanto, quindi può essere utile consultare la [documentazione ufficiale SQLite](#).



Febbraio 2011

Importare/esportare Shapefile (DBF,TXT,...)

Ci sono parecchi formati che sono molto diffusi nel mondo professionale GIS.

Sono tutti *formati aperti* (cioè non sono confinati in qualche specifico software proprietario, non sono coperti da brevetti, e sono pubblicamente documentati)

Nessuna sorpresa: tali formati sono universalmente riconosciuti da ogni software collegato al GIS, e possono essere usati in sicurezza per lo scambio dei dati fra differenti piattaforme e sistemi.

- il formato **Shapefile** della ESRI rappresenta di fatto lo standard universale per lo scambio dei dati GIS
- il formato **DBF** è stato introdotto nei primissimi giorni del personal computing da **dBase** (il primo software *di tipo DBMS a conoscere una popolarità universale*) in ogni Shapefile è incluso un file DBF ma è abbastanza facile trovare file DBF 'nudi' (senza SHP corrispondente) usati per distribuire semplici tabelle.
- il formato **TXT/CSV** identifica qualsiasi *file strutturato di testo* (normalmente, le varianti più usate sono valori separati da caratteri di tabulazione o da punti e virgola)

SpatiaLite gestisce tutti i formati precedenti in import e/o export.

Avvertenza: altri formati di dati sono molto popolari e diffusi, ad es. i seguenti:

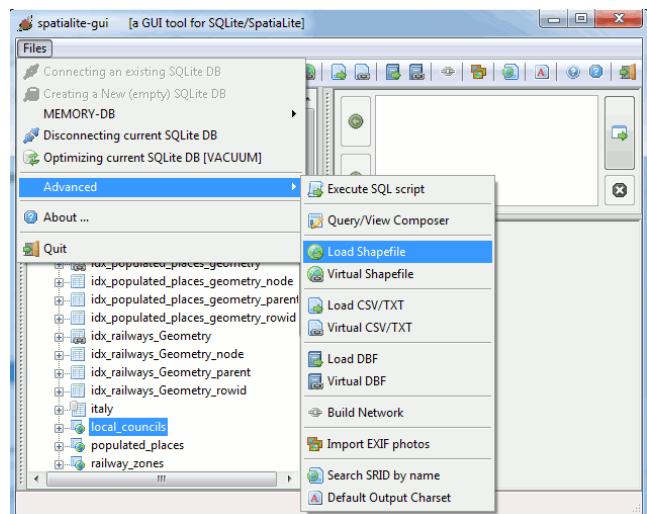
- il formato per fogli di calcolo Excel di Microsoft (**.xls**)
- il formato per database Access di Microsoft (**.mdb**)
- e molti, molti altri.

Tutti questi sono formati *chiusi (proprietari)*, cioè richiedono l'uso di qualche specifico software o sistema operativo proprietario, non dispongono di documentazione pubblicamente accessibile e/o sono coperti da brevetti. E tutto questo chiarisce perchè SpatiaLite (come molti altri programmi open source) non possono gestire questi *formati chiusi*.

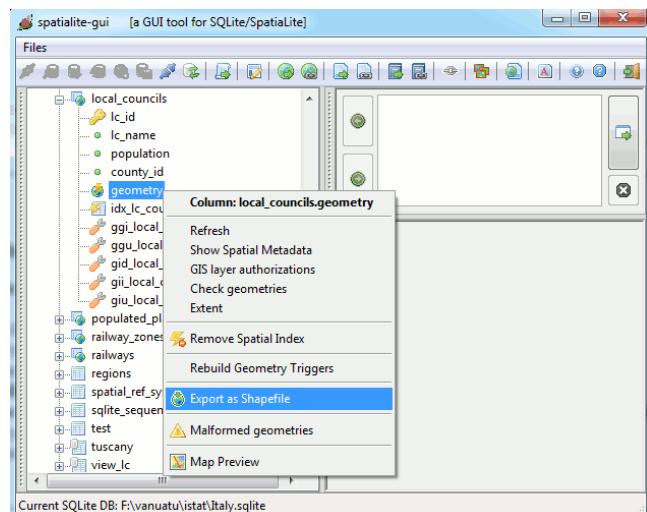
Shapefiles

SpatiaLite consente sia l'import che l'export degli Shapefiles:

- potete accedere direttamente ad ogni Shapefile esterno come tabella **VirtualShapefile** per mezzo SQL
- potete importare ogni Shapefile in una tabella DB:
 - usando **spatialite_gui** troverete una voce **Load Shapefile** nel menu principale e nella barra degli strumenti
 - usando il programma **front-end spatialite CLI** potete disporre della macro **.loadshp**
 - o potete usare la shell di comandi di **spatialite_tool**
- potete esportare qualsiasi tabella **Geometry** come Shapefile:
 - usando **spatialite_gui** la voce **Export As Shapefile** nel menu contestuale di ogni colonna **Geometry** all'interno della vista ad albero principale
 - usando il programma **front-end spatialite CLI** potete disporre della macro **.dumpshp**
 - oppure potete usare la shell di comandi di **spatialite_tool**.



spatialite_gui : importazione di uno shapefile



spatialite_gui : esportazione di uno shapefile

```
> spatialite counties.sqlite
SpatiaLite version ...: 2.4.0-RC5 Supported Extensions:
  - 'VirtualShape'          [direct Shapefile access]
  - 'VirtualDbf'            [direct DBF access]
  - 'VirtualText'           [direct CSV/TXT access]
  - 'VirtualNetwork'        [Dijkstra shortest path]
  - 'RTree'                  [Spatial Index - R*Tree]
  - 'MbrCache'               [Spatial Index - MBR cache]
  - 'VirtualFDO'              [FDO-OGR interoperability]
  - 'SpatiaLite'                [Spatial SQL - OGC]

PROJ.4 version .....: Rel. 4.7.1, 23 September 2009
GEOS version .....: 3.3.0-CAPI-1.7.0
SQLite version .....: 3.7.4
Enter ".help" for instructions

spatialite> .loadshp prov2010_s counties CP1252 23032
the SPATIAL_REF_SYS table already contains some row(s)
=====
Loading shapefile at 'prov2010_s' into SQLite table 'counties'

BEGIN
CREATE TABLE counties (
PK_UID INTEGER PRIMARY KEY AUTOINCREMENT,
"OBJECTID" INTEGER,
"COD_PRO" INTEGER,
"NOME_PRO" TEXT,
"SIGLA" TEXT)
SELECT AddGeometryColumn('counties', 'Geometry', 23032, 'MULTIPOLYGON', 'XY')
COMMIT

Inserted 110 rows into 'counties' from SHAPEFILE
=====
spatialite> .headers on
spatialite> SELECT * FROM counties LIMIT 5
PK_UID|OBJECTID|COD_PRO|NOME_PRO|SIGLA|Geometry
1|1|1|Torino|TO|
2|2|2|Vercelli|VC|
3|3|3|Novara|NO|
4|4|4|Cuneo|CN|
5|5|5|Asti|AT|
spatialite>
```

spatialite CLI front-end: importazione di uno shapefile

```

spatialite> .dumpshp counties Geometry exported_counties CP1252
=====
Dumping SQLite table 'counties' into shapefile at 'exported_counties'

SELECT * FROM "counties" WHERE GeometryAliasType("Geometry") = 'POLYGON'
OR GeometryAliasType("Geometry") = 'MULTIPOLYGON' OR "Geometry" IS NULL;

Exported 110 rows into SHAPEFILE
=====

spatialite> .quit

>

```

spatialite CLI front-end: esportazione di uno shapefile

```

> spatialite_tool -i -shp prov2010_s -d db.sqlite -t counties -c CP1252 -s 23032
SQLite version: 3.7.4
Spatialite version: 2.4.0-RC5
Inserted 110 rows into 'counties' from 'prov2010_s.shp'

>

```

shell di comandi spatialite_tool: importazione di uno shapefile

```

> spatialite_tool -e -shp exported_counties -d db.sqlite -t counties -g Geometry -c CP1252
SQLite version: 3.7.4
Spatialite version: 2.4.0-RC5
Exported 110 rows into 'exported_counties.shp' from 'counties'

>

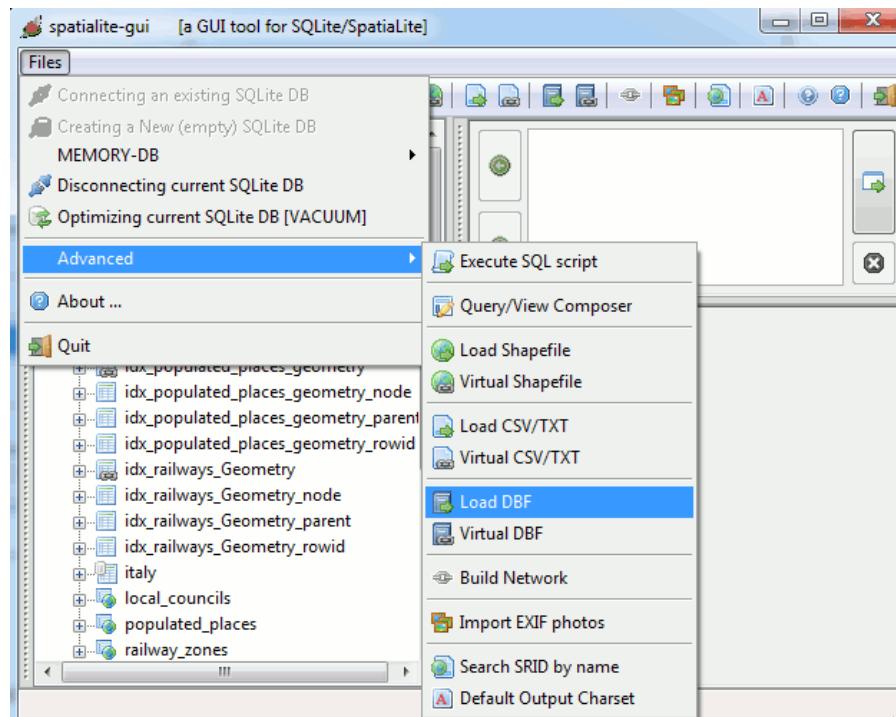
```

shell di comand spatialite_tool: shapefile export

File DBF

SpatiaLite consente solo l'importazione di file DBF:

- potete accedere ad ogni file DBF come tabella **VirtualDbf** per mezzo di SQL
- potete importare ogni file DBF in una tabella DB
 - usando **spatialite_gui** troverete una voce **Load DBF** nel menu principale e nella barra degli strumenti
 - usando il programma **front-end spatialite CLI** potete disporre **.loaddir**
 - oppure potete usare la shell di comandi di **spatialite_tool**.



spatialite_gui : importazione di file DBF

```

> spatialite local_councils.sqlite
SpatiaLite version ...: 2.4.0-RC5 Supported Extensions:
  - 'VirtualShape'          [direct Shapefile access]
  - 'VirtualDbf'            [direct DBF access]
  - 'VirtualText'           [direct CSV/TXT access]
  - 'VirtualNetwork'        [Dijkstra shortest path]
  - 'RTree'                  [Spatial Index - R*Tree]
  - 'MbrCache'               [Spatial Index - MBR cache]
  - 'VirtualFDO'              [FDO-OGR interoperability]
  - 'SpatiaLite'                [Spatial SQL - OGC]

PROJ.4 version .....: Rel. 4.7.1, 23 September 2009
GEOS version .....: 3.3.0-CAPI-1.7.0
SQLite version .....: 3.7.4
Enter ".help" for instructions

spatialite> .loaddbf com2010_s.dbf local_councils CP1252
=====
Loading DBF at 'com2010_s.dbf' into SQLite table 'local_councils'

BEGIN;
CREATE TABLE local_councils (
PK_UID INTEGER PRIMARY KEY AUTOINCREMENT,
"OBJECTID" INTEGER,
"COD_REG" INTEGER,
"COD_PRO" INTEGER,
"COD_COM" INTEGER,
"PRO_COM" INTEGER,
"NOME_COM" TEXT,
"NOME_ITA" TEXT,
"NOME_TED" TEXT);
COMMIT;

Inserted 8094 rows into 'local_councils' from DBF =====
spatialite> .headers on
spatialite> SELECT * FROM local_councils LIMIT 5 OFFSET 5000;
PK_UID|OBJECTID|COD_REG|COD_PRO|COD_COM|PRO_COM|NOME_COM|NOME_ITA|NOME_TED
5001|4958|12|58|54|58054|Manziana|Manziana|
5002|4959|12|58|55|58055|Marano Equo|Marano Equo|
5003|4960|12|58|56|58056|Marcellina|Marcellina|
5004|4961|12|58|57|58057|Marino|Marino|
5005|4962|12|58|58|58058|Mazzano Romano|Mazzano Romano|
spatialite> .quit

>
```

spatialite CLI front-end: importazione di file DBF

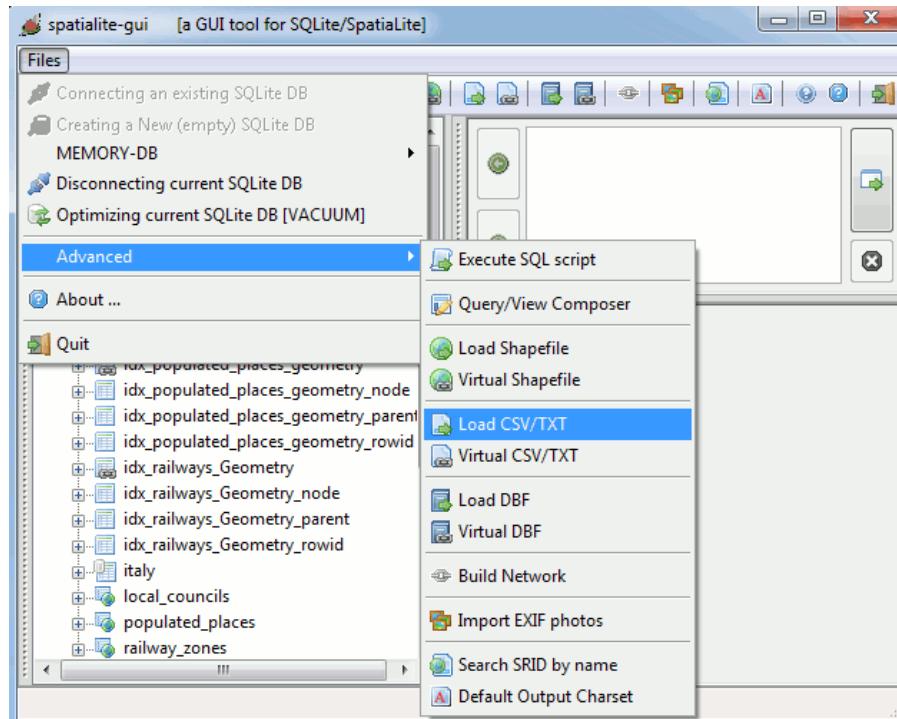
```
> spatialite_tool -i -dbf com2010_s -d db.sqlite -t local_councils -c CP1252
SQLite version: 3.7.4
SpatiaLite version: 2.4.0-RC5
Inserted 8094 rows into 'local_councils' from 'com2010_s.dbf'

>
shell di comandi spatialite_tool: DBF import
```

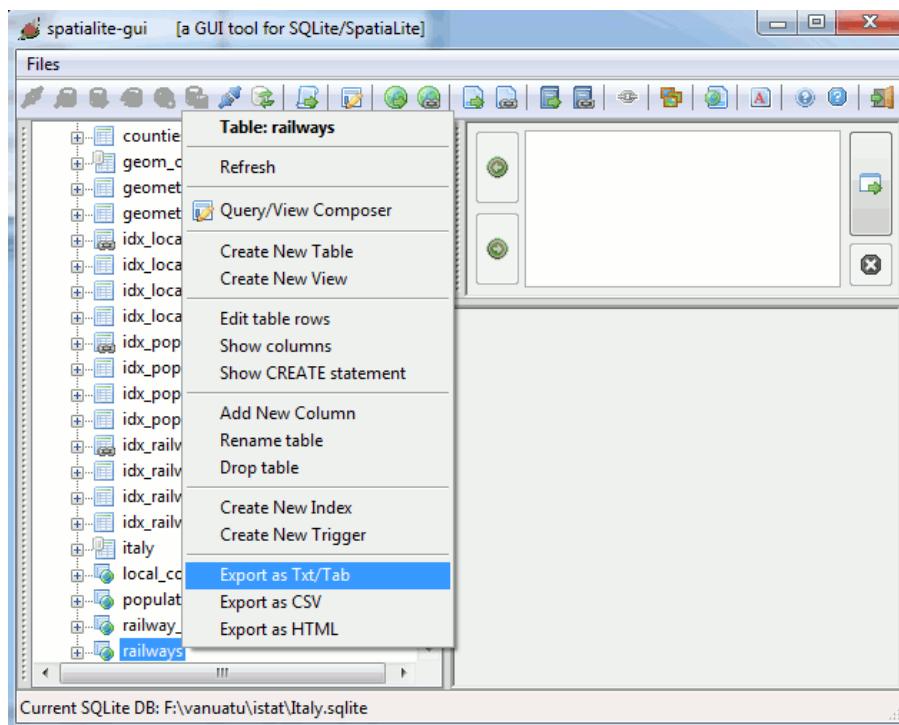
File TXT/CSV

SpatiaLite consente sia l'importazione che l'esportazione di file TXT/CSV:

- potete accedervi direttamente come tabelle **VirtualText** per mezzo di SQL
- potete importare ogni file TXT/CSV in una tabella DB
 - usando **spatialite_gui** troverete la voce **Load TXT/CSV** nel menu principale e nella barra degli strumenti
- potete esportare ogni tabella in un file TXT/CSV:
 - usando **spatialite_gui** troverete la voce **Export As TXT/CSV** nel menu contestuale in corrispondenza di ogni tabella all'interno della vista ad albero principale.



spatialite_gui: importazione di file TXT/CSV

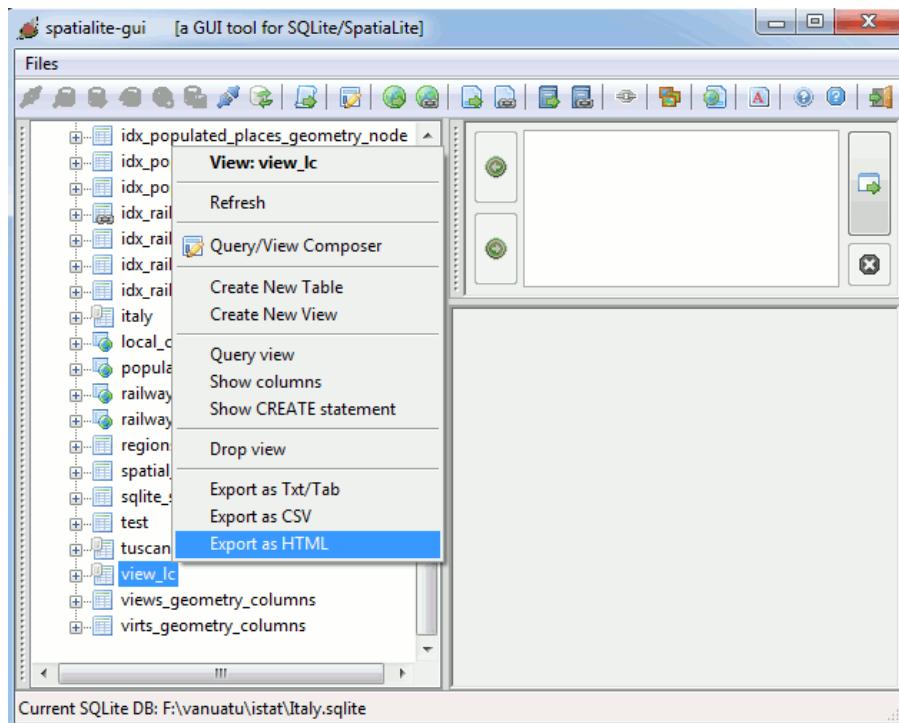


spatialite_gui: esportazione di file TXT/CSV

Altri formati di esportazione gestiti

Attraverso **spatialite_gui** potete anche esportare i vostri dati come:

- pagine web HTML
- immagini PNG (solo la geometria)
- documenti PDF (solo la geometria)
- file grafici vettoriali SVG (solo la geometria)



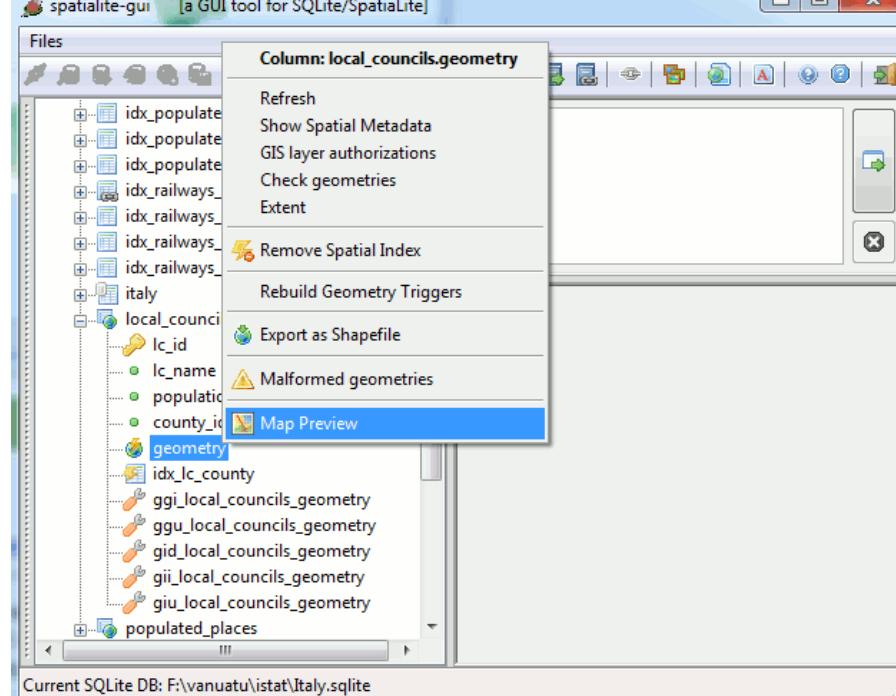
spatialite_gui : esportazione di pagine HTML

Table 'view_lc': from SQLite/SpatiaLite DB 'F:\vanuatu\istat\table.html' - Mozilla Firefox

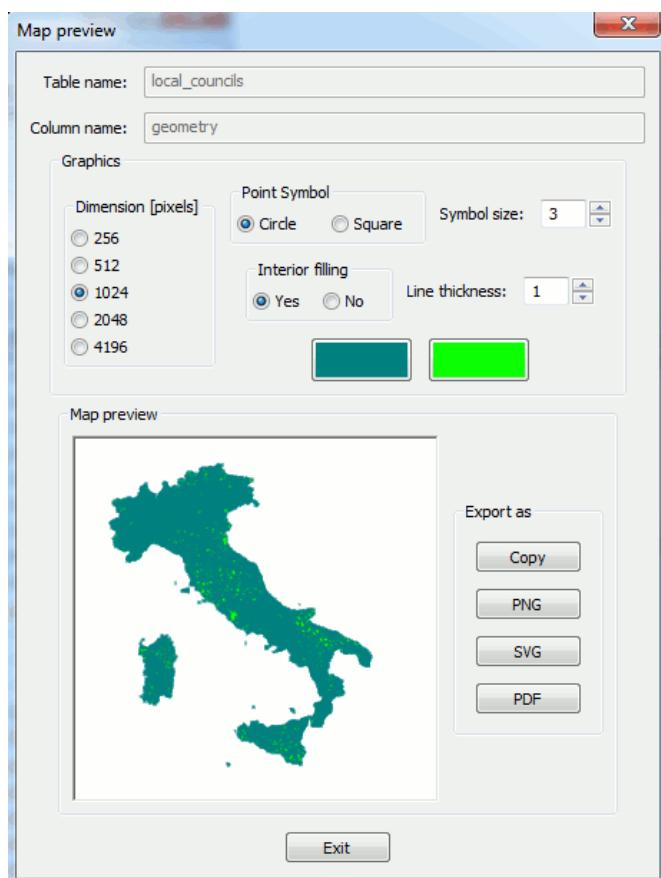
lc_id	lc_name	population	county_id	county_name	car_plate_code	region_id	region_name
1001	AGLIE'	2574	1	TORINO	TO	1	PIEMONTE
1002	AIRASCA	3554	1	TORINO	TO	1	PIEMONTE
1003	ALA DI STURA	479	1	TORINO	TO	1	PIEMONTE
1004	ALBIANO D'IVREA	1696	1	TORINO	TO	1	PIEMONTE
1005	ALICE SUPERIORE	616	1	TORINO	TO	1	PIEMONTE
1006	ALMESE	5658	1	TORINO	TO	1	PIEMONTE
1007	ALPETTE	300	1	TORINO	TO	1	PIEMONTE
1008	ALPIGNANO	16648	1	TORINO	TO	1	PIEMONTE
1009	ANDEZENO	1705	1	TORINO	TO	1	PIEMONTE
1010	ANDRATE	476	1	TORINO	TO	1	PIEMONTE
1011	ANGROGNA	777	1	TORINO	TO	1	PIEMONTE
1012	ARIGNANO	898	1	TORINO	TO	1	PIEMONTE
1013	AVIGLIANA	11070	1	TORINO	TO	1	PIEMONTE
1014	AZEGLIO	1274	1	TORINO	TO	1	PIEMONTE
1015	BAIRO	788	1	TORINO	TO	1	PIEMONTE
1016	BALANGERIO	3048	1	TORINO	TO	1	PIEMONTE
1017	BALDISSERO CANAVESE	513	1	TORINO	TO	1	PIEMONTE
1018	BALDISSERO TORINESE	3244	1	TORINO	TO	1	PIEMONTE
1019	BALME	101	1	TORINO	TO	1	PIEMONTE
1020	BANCHETTE	3427	1	TORINO	TO	1	PIEMONTE

Completato

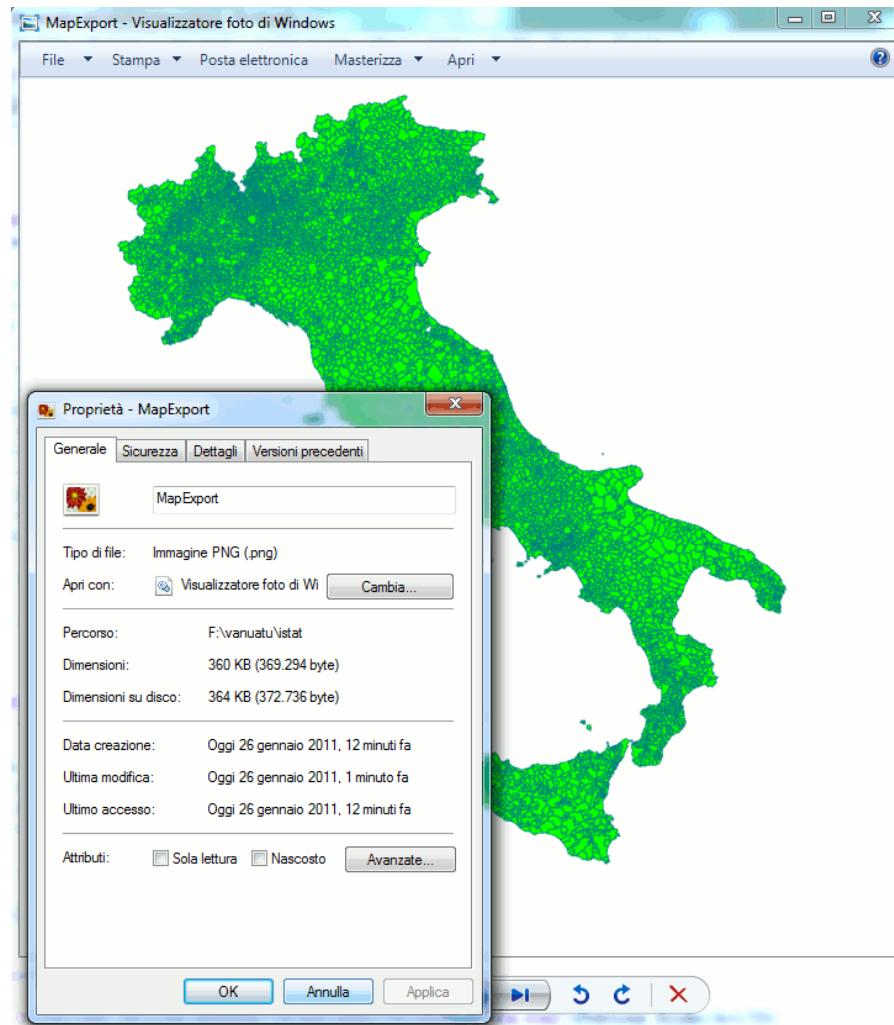
Esempio di pagina HTML



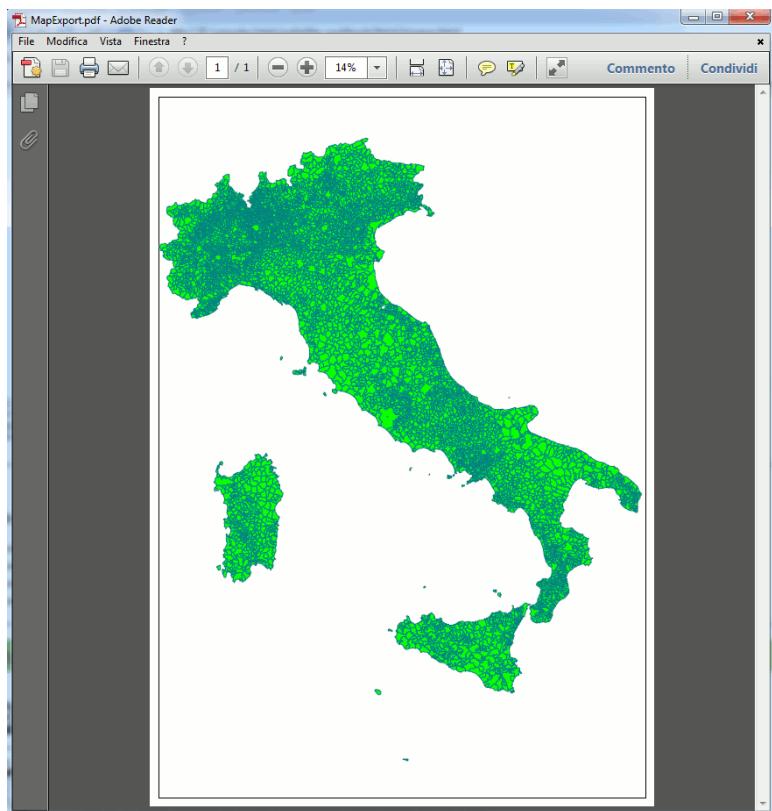
spatialite_gui : esportazione di immagini/documenti PNG / PDF / SVG



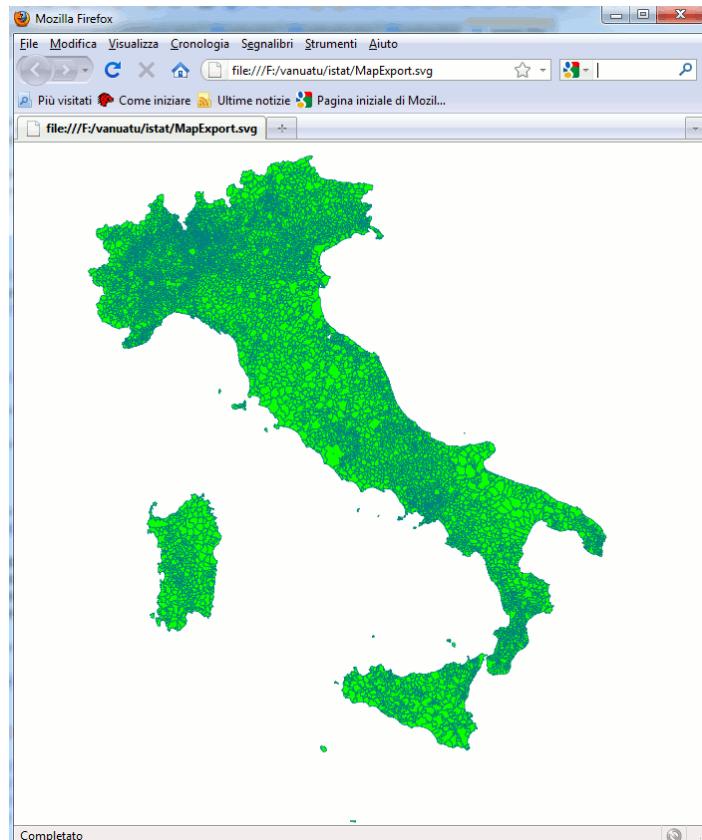
esempio di immagine/documento PNG / PDF / SVG



esempio di immagine PNG



esempio di documento PDF



esempio di SVG



Febbraio 2011

Linguaggi di collegamento: (C/C++, Java, Python, PHP ...)

Sia SQLite che SpatiaLite sono elementarmente semplici e veramente leggeri. Prtanto sono ovvi candidati se siete degli sviluppatori di software, e la vostra applicazione richiede una robusta ed efficiente gestione Spatial DBMS, ma state anche cercando di mantenere il tutto più semplice possibile, evitando possibilmente ogni inutile complessità.

Il miglior linguaggio che mette a disposizione SQLite e SpatiaLite è ovviamente il **C** [o il suo *brutto anatroccolo C++*] (*dopo tutto, sia SQLite che SpatiaLite sono interamente scritti in C*).

Mediante **C/C++** potete accedere direttamente alle **APIs** di entrambi, ed avere così il pieno e libero accesso ad ogni capacità, al più basso livello desiderato.

E non è tutto: con il **C/C++** è possibile adottare eventualmente il **collegamento statico**, così da contenere nell'eseguibile (*e con una dimensione incredibilmente piccola*) il completo motore DBMS.

E tale approccio certamente libera da ogni mal di testa legato all'installazione.

In ogni caso non siete obbligati ad usare **C/C++**, con SQLite e SpatiaLite sono messi a disposizione anche molti altri linguaggi, come **Java**, **Python**, **PHP** e *altri ancora*.

Approccio basilare

Ogni linguaggio che fornisce un generico **driver SQLite**, altrimenti detto **connettore**, può gestire completamente anche SpatiaLite.

SQLite consente il caricamento dinamico di estensioni: e SpatiaLite è semplicemente una di queste estensioni.

```
SELECT load_extension('path_to_extension_library');
```

L'esecuzione della precedente istruzione caricherà tutte le estensioni di SQLite: incluso anche SpatiaLite. Pero', troppo spesso questo è vero solo in *teoria*, ma la *realtà* è completamente diversa.

Esaminiamo brevemente gli argomenti principali che trasformano questo semplice approccio in un completo disastro:

- SQLite è altamente configurabile: fornisce molte e molte opzioni al build-time
Se pensate di essere più sicuri, potete disabilitare completamente il meccanismo di caricamento dinamico delle estensioni.
Purtroppo, per molti lunghi anni questa è stata la scelta preferita per la grande maggioranza dei sistemi di pacchettizzazione.
- SQLite sta crescendo molto rapidamente: normalmente ogni pochi mesi è rilasciato un aggiornamento principale.
Ma molti sistemi di pacchettizzazione continuano ancora a distribuire versioni obsolete di SQLite.
Ovviamente tali versioni obsolete non possono fornire supporto a SpatiaLite.
- Una volta che siete caduti in questa situazione da panico (estensioni disabilitate/ SQLite obsoleto) non potete più fare assolutamente nulla. Non potete far altro che dimenticare SpatiaLite , almeno per ora.
- Comunque, se questa fosse la vostra situazione attuale, non perdetevi d'animo: le cose si evolvono, e normalmente tendono ad evolvere nella direzione giusta.
Fino ad appena due anni fa pochi linguaggi davano supporto a SpatiaLite.
Oggi non è più così per i linguaggi di più largo uso (*almeno, usando le versioni di recente rilascio*).

Potete leggere la sezione corrispondente al linguaggio da voi preferito, così da essere pronti a partire nel più breve tempo.

Ogni sezione contiene un *programma di esempio completo*, e contiene anche ogni consiglio relativo alla *configurazione del sistema*, preparazione del *compilatore/linker* e così via:

- [C / C++](#)
- [Java / JDBC](#)
- [Python](#)
- [PHP.](#)



Febbraio 2011

Linguaggi di collegamento: C/C++

Programma di esempio in C/C++

`spatialite_sample.c`

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <float.h>

#ifndef SPATIALITE_AMALGAMATION
    #include <spatialite/sqlite3.h>
#else
    #include <sqlite3.h>
#endif

#ifndef SPATIALITE_EXTENSION
    #include <spatialite.h>
#endif

int
main (void)
{
    sqlite3 *db_handle;
    sqlite3_stmt *stmt;
    int ret;
    char *err_msg = NULL;
    char sql[2048];
    char sql2[1024];
    int i;
    char **results;
    int rows;
    int columns;
    int cnt;
    const char *type;
    int srid;
    char name[1024];
    char geom[2048];

#ifndef SPATIALITE_EXTENSION
/*
 * initializing SpatiaLite-Amalgamation
 *
 * any C/C++ source requires this to be performed
 * for each connection before invoking the first
 * SQLite/SpatiaLite call
 */
    spatialite_init (0);
    fprintf(stderr, "\n\n***** hard-linked libspatialite *****\n\n");
#endif

/* creating/connecting the test_db */
    ret =
        sqlite3_open_v2 ("test-db.sqlite", &db_handle,
                        SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE, NULL);
    if (ret != SQLITE_OK)
    {
        fprintf (stderr, "cannot open 'test-db.sqlite': %s\n",
                sqlite3_errmsg (db_handle));
        sqlite3_close (db_handle);
        db_handle = NULL;
    }
}
```

```

        return -1;
    }

#endif SPATIALITE_EXTENSION
/*
 * loading SpatiaLite as an extension
 */
sqlite3_enable_load_extension (db_handle, 1);
strcpy (sql, "SELECT load_extension('libspatialite.so')");
ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "load_extension() error: %s\n", err_msg);
    sqlite3_free (err_msg);
    return 0;
}
fprintf(stderr, "\n\n**** SpatiaLite loaded as an extension ***\n\n");
#endif

/* reporting version infos */
#ifndef SPATIALITE_EXTENSION
/*
 * please note well:
 * this process is physically linked to libspatialite
 * so we can directly call any SpatiaLite's API function
 */
    fprintf (stderr, "SQLite version: %s\n", sqlite3_libversion());
    fprintf (stderr, "SpatiaLite version: %s\n", spatialite_version());
#else
/*
 * please note well:
 * this process isn't physically linked to libspatialite
 * because we loaded the library as an extension
 *
 * so we aren't enabled to directly call any SpatiaLite's API functions
 * we simply can access SpatiaLite indirectly via SQL statements
 */
    strcpy (sql, "SELECT sqlite_version()");
    ret =
    sqlite3_get_table (db_handle, sql, &results, &rows, &columns, &err_msg);
    if (ret != SQLITE_OK)
    {
        fprintf (stderr, "Error: %s\n", err_msg);
        sqlite3_free (err_msg);
        goto stop;
    }
    if (rows < 1)
    {
        fprintf (stderr,
            "Unexpected error: sqlite_version() not found ??????\n");
        goto stop;
    }
    else
    {
        for (i = 1; i <= rows; i++)
        {
            fprintf (stderr, "SQLite version: %s\n",
                results[(i * columns) + 0]);
        }
    }
    sqlite3_free_table (results);
    strcpy (sql, "SELECT spatialite_version()");
    ret =
    sqlite3_get_table (db_handle, sql, &results, &rows, &columns, &err_msg);
    if (ret != SQLITE_OK)
    {
        fprintf (stderr, "Error: %s\n", err_msg);
        sqlite3_free (err_msg);
        goto stop;
    }
    if (rows < 1)
    {
        fprintf (stderr,
            "Unexpected error: spatialite_version() not found ??????\n");
        goto stop;
    }
    else
    {
        for (i = 1; i <= rows; i++)
        {

```

```

        fprintf (stderr, "SpatiaLite version: %s\n",
            results[(i * columns) + 0]);
    }
}
sqlite3_free_table (results);
#endif /* SpatiaLite as an extension */

/* initializing Spatialite's metadata tables */
strcpy (sql, "SELECT InitSpatialMetadata()");
ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "InitSpatialMetadata() error: %s\n", err_msg);
    sqlite3_free (err_msg);
    return 0;
}

/* creating a POINT table */
strcpy (sql, "CREATE TABLE test_pt ());
strcat (sql, "id INTEGER NOT NULL PRIMARY KEY,");
strcat (sql, "name TEXT NOT NULL");
ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}

/* creating a POINT Geometry column */
strcpy (sql, "SELECT AddGeometryColumn('test_pt', ");
strcat (sql, "'geom', 4326, 'POINT', 'XY')");
ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}

/* creating a LINESTRING table */
strcpy (sql, "CREATE TABLE test_ln ());
strcat (sql, "id INTEGER NOT NULL PRIMARY KEY,");
strcat (sql, "name TEXT NOT NULL");
ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}

/* creating a LINESTRING Geometry column */
strcpy (sql, "SELECT AddGeometryColumn('test_ln', ");
strcat (sql, "'geom', 4326, 'LINESTRING', 'XY')");
ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}

/* creating a POLYGON table */
strcpy (sql, "CREATE TABLE test_pg ());
strcat (sql, "id INTEGER NOT NULL PRIMARY KEY,");
strcat (sql, "name TEXT NOT NULL");
ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}

/* creating a POLYGON Geometry column */
strcpy (sql, "SELECT AddGeometryColumn('test_pg', ");
strcat (sql, "'geom', 4326, 'POLYGON', 'XY')");
ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}

```

```

        goto stop;
    }

/*
* inserting some POINTs
* please note well: SQLite is ACID and Transactional
* so (to get best performance) the whole insert cycle
* will be handled as a single TRANSACTION
*/
ret = sqlite3_exec (db_handle, "BEGIN", NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}
for (i = 0; i < 100000; i++)
{
    /* for POINTs we'll use full text sql statements */
    strcpy (sql, "INSERT INTO test_pt (id, name, geom) VALUES (");
    sprintf (sql2, "%d, 'test POINT #%"d, i + 1, i + 1);
    strcat (sql, sql2);
    sprintf (sql2, ", GeomFromText('POINT(%1.6f %1.6f)', i / 1000.0,
        i / 1000.0);
    strcat (sql, sql2);
    strcat (sql, ", 4326))");
    ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
    {
        fprintf (stderr, "Error: %s\n", err_msg);
        sqlite3_free (err_msg);
        goto stop;
    }
}
ret = sqlite3_exec (db_handle, "COMMIT", NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}

/* checking POINTs */
strcpy (sql, "SELECT DISTINCT Count(*), ST_GeometryType(geom), ");
strcat (sql, "ST_Srid(geom) FROM test_pt");
ret =
sqlite3_get_table (db_handle, sql, &results, &rows, &columns, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}
if (rows < 1)
{
    fprintf (stderr, "Unexpected error: ZERO POINTs found ??????\n");
    goto stop;
}
else
{
    for (i = 1; i <= rows; i++)
    {
        cnt = atoi (results[(i * columns) + 0]);
        type = results[(i * columns) + 1];
        srid = atoi (results[(i * columns) + 2]);
        fprintf (stderr, "Inserted %d entities of type %s SRID=%d\n",
            cnt, type, srid);
    }
}
sqlite3_free_table (results);

/*
* inserting some LINESTRINGS
* this time we'll use a Prepared Statement
*/
strcpy (sql, "INSERT INTO test_ln (id, name, geom)");
strcat (sql, "VALUES (?, ?, GeomFromText(?, 4326))");
ret = sqlite3_prepare_v2 (db_handle, sql, strlen (sql), &stmt, NULL);
if (ret != SQLITE_OK)
{

```

```

fprintf (stderr, "SQL error: %s\n%s\n", sql,
        sqlite3_errmsg (db_handle));
goto stop;
}
ret = sqlite3_exec (db_handle, "BEGIN", NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}
for (i = 0; i < 100000; i++)
{
    /* setting up values / binding */
    sprintf (name, "test LINESTRING #%d", i + 1);
    strcpy (geom, "LINESTRING(");
    if ((i % 2) == 1)
    {
        /* odd row: five points */
        strcat (geom, "-180.0 -90.0, ");
        sprintf (sql2, "%1.6f %1.6f, ", -10.0 - (i / 1000.0),
                 -10.0 - (i / 1000.0));
        strcat (geom, sql2);
        sprintf (sql2, "%1.6f %1.6f, ", -10.0 - (i / 1000.0),
                 10.0 + (i / 1000.0));
        strcat (geom, sql2);
        sprintf (sql2, "%1.6f %1.6f, ", 10.0 + (i / 1000.0),
                 10.0 + (i / 1000.0));
        strcat (geom, sql2);
        strcat (geom, "180.0 90.0");
    }
    else
    {
        /* even row: two points */
        sprintf (sql2, "%1.6f %1.6f, ", -10.0 - (i / 1000.0),
                 -10.0 - (i / 1000.0));
        strcat (geom, sql2);
        sprintf (sql2, "%1.6f %1.6f, ", 10.0 + (i / 1000.0),
                 10.0 + (i / 1000.0));
        strcat (geom, sql2);
    }
    strcat (geom, ")");
    sqlite3_reset (stmt);
    sqlite3_clear_bindings (stmt);
    sqlite3_bind_int (stmt, 1, i + 1);
    sqlite3_bind_text (stmt, 2, name, strlen (name), SQLITE_STATIC);
    sqlite3_bind_text (stmt, 3, geom, strlen (geom), SQLITE_STATIC);
    /* performing INSERT INTO */
    ret = sqlite3_step (stmt);
    if (ret == SQLITE_DONE || ret == SQLITE_ROW)
        continue;
    fprintf (stderr, "sqlite3_step() error: [%s]\n",
            sqlite3_errmsg (db_handle));
    goto stop;
}
sqlite3_finalize (stmt);
ret = sqlite3_exec (db_handle, "COMMIT", NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}

/* checking LINESTRINGS */
strcpy (sql, "SELECT DISTINCT Count(*), ST_GeometryType(geom), ");
strcat (sql, "ST_Srid(geom) FROM test_ln");
ret =
sqlite3_get_table (db_handle, sql, &results, &rows, &columns, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}
if (rows < 1)
{
    fprintf (stderr, "Unexpected error: ZERO LINESTRINGS found ??????\n");
    goto stop;
}

```

```

else
{
    for (i = 1; i <= rows; i++)
    {
        cnt = atoi (results[(i * columns) + 0]);
        type = results[(i * columns) + 1];
        srid = atoi (results[(i * columns) + 2]);
        fprintf (stderr, "Inserted %d entities of type %s SRID=%d\n",
            cnt, type, srid);
    }
}
sqlite3_free_table (results);

/*
* inserting some POLYGONS
* this time too we'll use a Prepared Statement
*/
strcpy (sql, "INSERT INTO test_pg (id, name, geom) ");
strcat (sql, "VALUES (?, ?, GeomFromText(?, 4326))");
ret = sqlite3_prepare_v2 (db_handle, sql, strlen (sql), &stmt, NULL);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "SQL error: %s\n%s\n", sql,
        sqlite3_errmsg (db_handle));
    goto stop;
}
ret = sqlite3_exec (db_handle, "BEGIN", NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}
for (i = 0; i < 100000; i++)
{
    /* setting up values / binding */
    sprintf (name, "test POLYGON #%d", i + 1);
    strcpy (geom, "POLYGON((");
    sprintf (sql2, "%1.6f %1.6f, ", -10.0 - (i / 1000.0),
        -10.0 - (i / 1000.0));
    strcat (geom, sql2);
    sprintf (sql2, "%1.6f %1.6f, ", 10.0 - (i / 1000.0),
        -10.0 - (i / 1000.0));
    strcat (geom, sql2);
    sprintf (sql2, "%1.6f %1.6f, ", 10.0 + (i / 1000.0),
        10.0 + (i / 1000.0));
    strcat (geom, sql2);
    sprintf (sql2, "%1.6f %1.6f, ", -10.0 - (i / 1000.0),
        10.0 - (i / 1000.0));
    strcat (geom, sql2);
    sprintf (sql2, "%1.6f %1.6f", -10.0 - (i / 1000.0),
        -10.0 - (i / 1000.0));
    strcat (geom, sql2);
    strcat (geom, ")");
    sqlite3_reset (stmt);
    sqlite3_clear_bindings (stmt);
    sqlite3_bind_int (stmt, 1, i + 1);
    sqlite3_bind_text (stmt, 2, name, strlen (name), SQLITE_STATIC);
    sqlite3_bind_text (stmt, 3, geom, strlen (geom), SQLITE_STATIC);
    /* performing INSERT INTO */
    ret = sqlite3_step (stmt);
    if (ret == SQLITE_DONE || ret == SQLITE_ROW)
        continue;
    fprintf (stderr, "sqlite3_step() error: [%s]\n",
        sqlite3_errmsg (db_handle));
    goto stop;
}
sqlite3_finalize (stmt);
ret = sqlite3_exec (db_handle, "COMMIT", NULL, NULL, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}

/* checking POLYGONS */
strcpy (sql, "SELECT DISTINCT Count(*), ST_GeometryType(geom), ");
strcat (sql, "ST_Srid(geom) FROM test_pg");
ret =

```

```

sqlite3_get_table (db_handle, sql, &results, &rows, &columns, &err_msg);
if (ret != SQLITE_OK)
{
    fprintf (stderr, "Error: %s\n", err_msg);
    sqlite3_free (err_msg);
    goto stop;
}
if (rows < 1)
{
    fprintf (stderr, "Unexpected error: ZERO POLYGONS found ??????\n");
    goto stop;
}
else
{
    for (i = 1; i <= rows; i++)
    {
        cnt = atoi (results[(i * columns) + 0]);
        type = results[(i * columns) + 1];
        srid = atoi (results[(i * columns) + 2]);
        fprintf (stderr, "Inserted %d entities of type %s SRID=%d\n",
            cnt, type, srid);
    }
}
sqlite3_free_table (results);

/* closing the DB connection */
stop:
    sqlite3_close (db_handle);
    return 0;
}

```

Compilazione e linkaggio [Collegamento]

SpatiaLite è disponibile in diverse forme:

- **libspatialite** è la libreria standard prevista per essere caricata come una estensione
- **libspatialite-amalgamation** è un *motore SQL* completo e autosufficiente che fornisce una *copia privata interna* di SQLite.

Se il vostro interesse principale è quello di sviluppare applicazioni C/C++ pronte all'uso e monolitiche, allora l'uso della *libreria statica libspatialite-amalgamation* è certamente l'opzione preferibile.

Comunque niente vi impedisce di usare uno schema diverso basato sulle librerie condivise *collegate dinamicamente*. E c'è anche una terza opzione: potete completamente evitare le librerie SQLite e SpatiaLite.

Entrambi sono dei semplici file *sorgente monolitici*: così potete compilare direttamente ogni sorgente necessario in una singola volta (*questo ovviamente genera un eseguibile linkato staticamente*).

Diamo uno sguardo veloce al codice sorgente **`spatialite_sample.c`**; potete facilmente notare che sono massicciamente usate due ***macro condizionali***:

- **`SPATIALITE_AMALGAMATION`** è usata per verificare se state usando separatamente **`libssqlite`** e **`libspatialite`** o se state usando la onnicomprensiva **`libspatialite-amalgamation`**
 - come potete notare l'unica differenza riguarda l'inclusione degli header (file di intestazione) quando usate **`amalgamation`** non potete usare **`#include <sqlite3.h>`** (*perchè questo è l'header file standard di sqlite*)
 - dovete usare **`#include <spatialite/sqlite3.h>`** (*una versione appositamente modificata per funzionare con amalgamation*)
- **`SPATIALITE_EXTENSION`** è usata per verificare se state usando la versione **`hard-linked`** di **`libspatialite`** o se intendete usare questa libreria come un'estensione:
 - **attenzione:** il caricamento di **`libspatialite`** come estensione esclude l'uso di amalgamation, in quanto **`libssqlite`** è già *linkata* all'eseguibile in questo caso.

Ora potete costruire (build) **`spatialite_sample.c`** in numerosi modi diversi, semplicemente definendo in modo appropriato queste due macro, nel modo più flessibile.

I seguenti sono degli esempi pratici basati sul compilatore standard per l'ambiente Linux: GNU gcc.

Caricamento di SpatiaLite come estensione

```
gcc -Wall -Wextra -Wunused -DSPATIALITE_EXTENSION \
    spatialite_sample.c -o spatialite_sample -lsqlite3
```

Occorre notare alcuni punti interessanti:

- non dovete linkare esplicitamente libspatialite; *questo perchè questa libreria verrà caricata dinamicamente* in fase di esecuzione
- in questo caso verrà usata la libreria di sistema **`libssqlite`**
- **nota 1:** questo metodo semplifica enormemente le fasi di compilazione e di linking, ma semplicemente sposta il problema all'esecuzione.
E chiaramente si verificherà un fallimento se (*per qualsiasi ragione*) la libreria di estensione non potrà essere caricata.
- **nota 2:** usando questo metodo il vostro eseguibile non potrà accedere direttamente alle funzioni proprie di SpatiaLite, poiché **`libspatialite`** non è collegato
di conseguenza qualsiasi *simbolo definito* in questa libreria è in realtà *invisibile* al codice eseguibile.
Comunque, potete chiamare qualsiasi funzione disponibile con il linguaggio SQL, una volta che **`libssqlite`** ha caricato con successo l'estensione.

Questo approccio elementare è esattamente lo stesso fornito da ogni altro linguaggio (Java, Python, PHP ...). Ma il C/C++ vi dà opzioni di configurazione più flessibili: guardate gli esempi che seguono, per favore.

Collegamento dinamico alle librerie libspatialite + libsqlite

```
gcc -Wall -Wextra -Wunused -I/usr/local/include \
    spatialite_sample.c -o spatialite_sample
    -L/usr/local/lib -lspatialite -lsqlite3
```

Alcuni punti interessanti da notare:

- questa volta **libspatialite** è collegata in modo statico al vostro eseguibile
- non useremo **amalgamation**, quindi non dobbiamo dichiarare né **-DSPATIALITE_AMALGAMATION** né **-DSPATIALITE_EXTENSION**
- normalmente **libspatialite** è installata nella cartella **/usr/local** quindi definiremo le opzioni **-I** e **-L** in modo di estenderela ricerca dei file di intestazione e le librerie.
- notate per favore: questa volta linkiamo sia **libspatialite** che **libsqlite**.

Collegamento dinamico con libspatialite-amalgamation

```
gcc -Wall -Wextra -Wunused -DSPATIALITE_AMALGAMATION \
    -I/usr/local/include spatialite_sample.c \
    -o spatialite_sample \
    -L/usr/local/lib -lspatialite
```

Qualche punto interessante da notare:

- questa volta useremo **libspatialite-amalgamation**
- e di conseguenza definiremo l'opzione **-DSPATIALITE_AMALGAMATION**
- come nel precedente esempio attiveremo le opzioni **-I** e **-L** per estendere la ricerca dei file di intestazione e le librerie
- **notate per favore:** questa volta linkiamo solo **libspatialite**
non c'è assolutamente bisogno di linkare anche **libsqlite**, perchè useremo la copia privata all'interno di **libspatialite-amalgamation**.

Collegamento dinamico con libspatialite-libsqlite

```
gcc -Wall -Wextra -Wunused -I/usr/local/include \
    spatialite_sample.c -o spatialite_sample \
    /usr/local/lib/libspatialite.a \
    /usr/lib/libsqlite.a \
    /usr/lib/libgeos_c.a \
    /usr/lib/libgeos.a \
    /usr/lib/libproj.a \
    -lstdc++ -ldl -lpthread -lm
```

Punti interessanti da notare:

- l'adozione di una strategia di *collegamento statica* obbliga a risolvere ogni simbolo nella fase di linking quindi siamo obbligati a fare esplicito riferimento a parecchie altre librerie
 - **libproj**, **libgeos_c** e **libgeos** sono sempre necessarie
 - anche la libreria **-lstdc++** è necessaria poichè **libgeos** è una libreria scritta in C++, quindi richiede il supporto C++ in fase di esecuzione [run- time]
 - sui sistemi Linux le opzioni **-ldl** e **-lpthread** sono strettamente necessarie
 - su alcuni sistemi Linux può essere necessaria anche la dichiarazione **-lm** (ed è sempre **harmless**, anche quando non strettamente necessaria).

Collegamento statico con libspatialite-amalgamation

```
gcc -Wall -Wextra -Wunused -DSPATIALITE_AMALGAMATION \
-I/usr/local/include spatialite_sample.c \
-o spatialite_sample \
/usr/local/lib/libspatialite.a \
/usr/lib/libgeos_c.a \
/usr/lib/libgeos.a \
/usr/lib/libproj.a \
-lstdc++ -ldl -lpthread -lm
```

Esattamente lo stesso di prima; semplicemente questa volta **libsqllite** non è più necessaria, perchè fornita dalla *copia privata interna* ad **amalgamation**.

Note per favore: differenti piattaforme, differenti librerie di sistema, differenti schemi di file system. Il compilatore gcc è disponibile su quasi ogni piattaforma (*incluso Microsoft Windows, ovviamente*). Sfortunatamente ogni piattaforma ha le sue specifiche idiosincrasie.

MacOsX:

```
gcc -Wall -Wextra -Wunused -DSPATIALITE_AMALGAMATION \
-I/usr/local/include -I/opt/local/include
spatialite_sample.c -o spatialite_sample \
/usr/local/lib/libspatialite.a \
/opt/local/lib/libgeos_c.a \
/opt/local/lib/libgeos.a \
/opt/local/lib/libproj.a \
/opt/local/lib/libiconv.a \
/opt/local/lib/libcharset.a \
-lstdc++ -ldl -lpthread -lm
```

MacOsX è fondamentalmente un derivato di Unix (*più precisamente un derivato di FreeBSD*). Il fantastico **MacPorts** consente l'uso delle librerie *standard open source* (nel nostro caso **libgeos**, **libproj** e **libiconv**):

- i file di intestazione [header files] di **MacPorts** saranno installate nella cartella **/opt/local/include**
- le librerie **MacPorts** saranno installate nella cartella **/opt/local/lib**
- sia **libiconv** che **libcharset** sono integrate nel C run-time di Linux
- ma su MacOsX è necessario linkare esplicitamente queste librerie.

Windows [MinGW + MSYS]:

```
gcc -Wall -Wextra -Wunused -DSPATIALITE_AMALGAMATION \
-I/usr/local/include spatialite_sample.c \
-o spatialite_sample.exe \
/usr/local/lib/libspatialite.a \
/usr/lib/libgeos_c.a \
/usr/lib/libgeos.a \
/usr/lib/libproj.a \
/usr/local/lib/libiconv.a \
-lstdc++ -lm
```

Windows e Unix sono molto diversi:

- come abbiamo appena visto per MacOsX anche per Windows è necessario linkare esplicitamente **libiconv** (ma non **libcharset**)
- **-ldl** e **-lpthread** non sono per niente richiesti (*semplicemente perché il caricatore dinamico e multithreading di Windows è differente dalla implementazione Unix (e sono gestiti direttamente dal WIN32 run-time)*).

Il metodo più semplice: nessuna libreria

```
gcc -Wall -Wextra -Wunused -DSPATIALITE_AMALGAMATION \
-DOMIT_GEOS=1 -DOMIT_PROJ=1 -DOMIT_ICONV=1 \
-DOMIT_GEOCALLBACKS=1 \
-I./headers -I/usr/local/include \
spatialite.c sqlite3.c spatialite_sample.c \
-o spatialite_sample.exe
```

Infine esiste una *terza via*: quella di incorporare un motore Spatial SQL completo nel modo più semplice ed assolutamente meno faticoso.

[*Penso che seguendo questo ultimo metodo sarete anche capaci di avere un SpatiaLite minimale per PDA o smart-phones*].

Solo qualche spiegazione:

- dovete creare una nuova cartella
- copiarvi il sorgente di **spatialite_sample.c**
- copiare nella stessa cartella i sorgenti di **sqlite3.c** e **spatialite.c** (*sono inclusi in ogni distribuzione di libspatialite-amalgamation*)
- ed infine dovete copiare l'intera cartella **headers** dei sorgenti di **libspatialite-amalgamation**.

Okay, ora siete pronti a costruire i vostri eseguibili, *linkati staticamente*.

Notate per favore: non è necessaria alcuna libreria esterna:

- questo build è basato su **libspatialite-amalgamation**, quindi definiremo l'opzione **-DSPATIALITE_AMALGAMATION**
- tramite l'istruzione **-DOMIT_GEOS=1** (*e così via*) disabiliteremo completamente GEOS, PROJ.4 e ICONV
- ma questo eviterà del tutto ogni ulteriore collegamento ad altre librerie
- certo, è vero, ora SpatiaLite è pesantemente gilded: ma il **nucleo principale** di Spatial SQL è rimasto intatto.

E questo è più che sufficiente per tanti e tanti obiettivi pratici.



Febbraio 2011

Linguaggi di collegamento: Java/JDBC

Verifica dell'ambiente

Linux Debian:

Per assicurarmi di testare la situazione aggiornata allo stato dell'arte ho usato **Debian Squeeze** (32 bit). In questo modo sono ragionevolmente sicuro che tutti i pacchetti necessari sono aggiornati alla versione più recente.

Java:

Ero molto curioso su **OpenJava** (non l'avevo mai usato finora), così ho deciso di non usare Sun Java. Dopo le prime difficoltà iniziali ho scoperto che dovevo installare i seguenti pacchetti:

- **openjdk-6-jre** (Java JDK)
- **gcj** (compilatore Java).

SQLite's JDBC connector:

Ho usato quello disponibile su <http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC>. Ho effettivamente scaricato la versione disponibile più recente:

<http://www.xerial.org/maven/repository/artifact/org/xerial/sqlite-jdbc/3.7.2/sqlite-jdbc-3.7.2.jar>.

In effetti questa non è la versione più recente fornita con SQLite, perché l'8 ottobre 2010 è stata rilasciata la v.3.7.3 e dall'8 dicembre 2010 è disponibile la v.3.7.4: quindi sembra che Xerial sia un po' datata.

Non è un grande problema, comunque.

Il mio primo test

Proprio per verificare se il mio ambiente era valido, ho dapprima compilato e lanciato la demo standard di Xerial.

Trovate facilmente il codice sulla loro pagina principale HTML su JDBC. Ho semplicemente copiato il codice Java dalla loro pagina HTML in un file di testo chiamato **Sample.java**.

Avviso importante: seguendo le istruzioni Xerial ho copiato il file **sqlite-jdbc-3.7.2.jar** direttamente nella stessa cartella contenente il **sorgente Java**, in modo da evitare il mal di testa con la CLASSPATH.

```
$ javac Sample.java
$ java -classpath ".:sqlite-jdbc-3.7.2.jar" Sample
```

Perfecto: tutto ha funzionato come mi aspettavo. Comunque questo prima verifica ha provato semplicemente le capacità SQLite di base.

Adesso andiamo avanti, cercando di testare se il connettore Xerial JDBC effettivamente funziona con SpatiaLite.

Il primo test con SpatiaLite (fallimento)

Sul mio sistema di prova (in realtà questo è uno dei destrieri Linux che uso correntemente per lo sviluppo e la prova) era già installata **libspatialite-2.4.0-RC4**.

Ho costruito questo pacchetto da solo, pertanto la libreria condivisa corrispondente era **/usr/local/lib/libspatialite.so**.

La prima cosa ovvia da fare era caricare la libreria condivisa SpatiaLite, così da consentire al connettore JDBC di collegarsi a SpatiaLite.

Quindi ho implicitamente aggiunto la seguente linea sorgente (immediatamente dopo aver stabilito la connessione):

```
stmt.execute("SELECT load_extension('/usr/local/lib/libspatialite.so');
```

Troppo facile: questa volta ho ricevuto uno scoraggiante messaggio di errore: **NOT AUTHORIZED** (non autorizzato).

Dopo un pò ho trovato un utile consiglio guardando in rete: per abilitare il caricamento dinamico delle estensioni è assolutamente necessario un passo preliminare:

```
import org.sqlite.SQLiteConfig;
...
SQLiteConfig config = new SQLiteConfig();
config.enableLoadExtension(true);
Connection conn = DriverManager.getConnection("path", config.toProperties());
Statement stmt = conn.createStatement();
stmt.execute("SELECT load_extension('/usr/local/lib/libspatialite.so')");
```

Perfetto: ora la libreria condivisa SpatiaLite è stata correttamente caricata.

E questa è stata l'unica disavventura che ho provato durante il mio test di JDBC: una volta che ho risolto questo problema, tutto ha funzionato fluidamente e senza ulteriori incidenti.

Eccetto che per una stranezza di JDBC che ho notato: ma vi racconterò di questo alla fine della storia.

Programma di esempio Java

SpatialiteSample.java

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.PreparedStatement;
import org.sqlite.SQLiteConfig;

public class SpatialiteSample
{
    public static void main(String[] args) throws ClassNotFoundException
    {
        // load the sqlite-JDBC driver using the current class loader
        Class.forName("org.sqlite.JDBC");

        Connection conn = null;
        try
        {
            // enabling dynamic extension loading
            // absolutely required by Spatialite
            SQLiteConfig config = new SQLiteConfig();
            config.enableLoadExtension(true);

            // create a database connection
            conn = DriverManager.getConnection("jdbc:sqlite:spatialite.sample",
                config.toProperties());
            Statement stmt = conn.createStatement();
            stmt.setQueryTimeout(30); // set timeout to 30 sec.

            // loading Spatialite
            stmt.execute("SELECT load_extension('/usr/local/lib/libspatialite.so')");

            // enabling Spatial Metadata
            // using v.2.4.0 this automatically initializes SPATIAL_REF_SYS and GEOMETRY_COLUMNS
            String sql = "SELECT InitSpatialMetadata();";
            stmt.execute(sql);

            // creating a POINT table
            sql = "CREATE TABLE test_pt (" +
                "id INTEGER NOT NULL PRIMARY KEY," +
                "name TEXT NOT NULL)";
            stmt.execute(sql);
            // creating a POINT Geometry column
            sql = "SELECT AddGeometryColumn('test_pt', 'geom', 4326, 'POINT', 'XY')";
            stmt.execute(sql);

            // creating a LINESTRING table
            sql = "CREATE TABLE test_ln (" +
                "id INTEGER NOT NULL PRIMARY KEY," +
                "name TEXT NOT NULL)";
            stmt.execute(sql);
            // creating a LINESTRING Geometry column
            sql = "SELECT AddGeometryColumn('test_ln', 'geom', 4326, 'LINESTRING', 'XY')";
            stmt.execute(sql);

            // creating a POLYGON table
            sql = "CREATE TABLE test_pg (" +
                "id INTEGER NOT NULL PRIMARY KEY," +
                "name TEXT NOT NULL)";
            stmt.execute(sql);
            // creating a POLYGON Geometry column
            sql = "SELECT AddGeometryColumn('test_pg', 'geom', 4326, 'POLYGON', 'XY')";
            stmt.execute(sql);
        }
    }
}

```

```

stmt.execute(sql);

// inserting some POINTS
// please note well: SQLite is ACID and Transactional,
// so (to get best performance) the whole insert cycle
// will be handled as a single TRANSACTION
conn.setAutoCommit(false);
int i;
for (i = 0; i < 100000; i++)
{
    // for POINTs we'll use full text sql statements
    sql = "INSERT INTO test_pt (id, name, geom) VALUES (" +
    sql += i + 1;
    sql += ", 'test POINT #'";
    sql += i + 1;
    sql += ", GeomFromText('POINT(";
    sql += i / 1000.0;
    sql += " ";
    sql += i / 1000.0;
    sql += ")', 4326))";
    stmt.executeUpdate(sql);
}
conn.commit();

// checking POINTS
sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), ";
sql += "ST_Srid(geom) FROM test_pt";
ResultSet rs = stmt.executeQuery(sql);
while(rs.next())
{
    // read the result set
    String msg = "> Inserted ";
    msg += rs.getInt(1);
    msg += " entities of type ";
    msg += rs.getString(2);
    msg += " SRID=";
    msg += rs.getInt(3);
    System.out.println(msg);
}

// inserting some LINESTRINGS
// this time we'll use a Prepared Statement
sql = "INSERT INTO test_ln (id, name, geom) ";
sql += "VALUES (?, ?, GeomFromText(?, 4326))";
PreparedStatement ins_stmt = conn.prepareStatement(sql);
conn.setAutoCommit(false);
for (i = 0; i < 100000; i++)
{
    // setting up values / binding
    String name = "test LINESTRING #";
    name += i + 1;
    String geom = "LINESTRING (";
    if ((i%2) == 1)
    {
        // odd row: five points
        geom += "-180.0 -90.0, ";
        geom += "-10.0 - (i / 1000.0);
        geom += " ";
        geom += "-10.0 - (i / 1000.0);
        geom += " ";
        geom += "-10.0 - (i / 1000.0);
        geom += " ";
        geom += "10.0 + (i / 1000.0);
        geom += " ";
        geom += "10.0 + (i / 1000.0);
        geom += " ";
        geom += "10.0 + (i / 1000.0);
        geom += ", 180.0 90.0";
    }
    else
    {
        // even row: two points
        geom += "-10.0 - (i / 1000.0);
        geom += " ";
        geom += "-10.0 - (i / 1000.0);
        geom += " ";
        geom += "10.0 + (i / 1000.0);
        geom += " ";
        geom += "10.0 + (i / 1000.0);
    }
    geom += ")";
    ins_stmt.setInt(1, i+1);
    ins_stmt.setString(2, name);
    ins_stmt.setString(3, geom);
    ins_stmt.executeUpdate();
}
conn.commit();

// checking LINESTRINGS
sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), ";
sql += "ST_Srid(geom) FROM test_ln";
rs = stmt.executeQuery(sql);
while(rs.next())
{
    // read the result set
    String msg = "> Inserted ";
    msg += rs.getInt(1);
    msg += " entities of type ";
    msg += rs.getString(2);
    msg += " SRID=";
    msg += rs.getInt(3);
    System.out.println(msg);
}

// inserting some POLYGONS
// this time too we'll use a Prepared Statement
sql = "INSERT INTO test_pg (id, name, geom) ";
sql += "VALUES (?, ?, GeomFromText(?, 4326))";
ins_stmt = conn.prepareStatement(sql);
conn.setAutoCommit(false);
for (i = 0; i < 100000; i++)
{
    // setting up values / binding
    String name = "test POLYGON #";
    name += i + 1;
}

```

```

        ins_stmt.setInt(1, i+1);
        ins_stmt.setString(2, name);
        String geom = "POLYGON(";
        geom += -10.0 - (i / 1000.0);
        geom += " ";
        geom += -10.0 - (i / 1000.0);
        geom += " ";
        geom += 10.0 + (i / 1000.0);
        geom += " ";
        geom += -10.0 - (i / 1000.0);
        geom += " ";
        geom += 10.0 + (i / 1000.0);
        geom += " ";
        geom += 10.0 + (i / 1000.0);
        geom += " ";
        geom += -10.0 - (i / 1000.0);
        geom += " ";
        geom += 10.0 + (i / 1000.0);
        geom += " ";
        geom += -10.0 - (i / 1000.0);
        geom += " ");
        ins_stmt.setInt(1, i+1);
        ins_stmt.setString(2, name);
        ins_stmt.setString(3, geom);
        ins_stmt.executeUpdate();
    }
    conn.commit();

    // checking POLYGONS
    sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), ";
    sql += "ST_Srid(geom) FROM test_pg";
    rs = stmt.executeQuery(sql);
    while(rs.next())
    {
        // read the result set
        String msg = "> Inserted ";
        msg += rs.getInt(1);
        msg += " entities of type ";
        msg += rs.getString(2);
        msg += " SRID=";
        msg += rs.getInt(3);
        System.out.println(msg);
    }
}
catch(SQLException e)
{
    // if the error message is "out of memory",
    // it probably means no database file is found
    System.err.println(e.getMessage());
}
finally
{
    try
    {
        if(conn != null)
            conn.close();
    }
    catch(SQLException e)
    {
        // connection close failed.
        System.err.println(e);
    }
}
}
}

```

```
$ javac -classpath ".:sqlite-jdbc.3.7.2.jar" SpatialiteSample.java  
$ java -classpath ".:sqlite-jdbc.3.7.2.jar" SpatialiteSample
```

Raccontare tutta la storia è noioso e molto poco interessante: potete studiare, provare il codice di esempio voi stessi, e questo è assolutamente tutto.

Conclusioni

- Il connettore JDBC Xerial sembra essere perfettamente a suo agio con SpatiaLite
 - le prestazioni sono veramente buone: più o meno le stesse che avreste usando processi basati sul C.
Ma non c'è molto da sorprendersi dopo tutto: usando il connettore JDBC Xerial state in realtà usando librerie in C, e Java agisce prettamente come un guscio avvolgente.

Attenzione

Per quanto il connettore JDBC Xerial sembra molto buono, ho notato alcuni difetti potenziali. Detto molto brevemente, le vostre istruzioni SQL devono essere assolutamente pulite e ben testate: perchè quando il connettore JDBC incontra qualche istruzione SQL non valida (*situazione non eccezionale durante il periodo di sviluppo*) è molto più probabile ricevere un crash fatale di JVM che un dolce messaggio di errore.

Sono diventato quasi pazzo cercando di identificare la causa di così frequenti crash durante le mie prove: finchè ho capito finalmente che il problema era semplicemente dovuto a stupide dimenticanze di parentesi o segni di apostrofo in istruzioni SQL complesse.

Il C può tranquillamente sopravvivere a tutte questi problemi senza danni, riportando un dolce e pulito messaggio di errore.

D'altra canto JDBC / JVM sono inesorabilmente unforgiving (e instabili) quando trattano questi banali errori.

Stranezze JDBC

Come avevo detto in precedenza, ho notato una effettiva stranezza in JDBC. E' ora dispiegare questo stupido problema.

Spezzone di codice in linguaggio C / SQLite API

```
strcpy(sql, "INSERT INTO xxx (id, geometry) VALUES (?, '')";
strcat(sql, "GeomFromText('POINT(? ?, ? ?)', 4326))";
sqlite3_prepare_v2 (db_handle, sql, strlen (sql), &stmt, NULL);
sqlite3_bind_int (stmt, 0, 1);
sqlite3_bind_double (stmt, 1, 10.01)
sqlite3_bind_double (stmt, 2, 20.02);
sqlite3_bind_double (stmt, 3, 30.03);
sqlite3_bind_double (stmt, 4, 40.04);
sqlite3_step (stmt);
```

- Useremo un Prepared Statement (dichiarazione preparata) per eseguire un INSERT INTO
- il primo argomento corrisponde alla colonna ID
- ogni altro argomento corrisponde a coordinate POINT
- tutto questo funziona assolutamente fluido, poichè SQL applica semplicemente la sostituzione dei valori in accordo con le istruzioni impartite.

Se voi (come sviluppatori) definite qualche tipo di dato contraddittorio, avrete qualche errore SQL. Questo suona curato e raffinato: dopo tutto lo sviluppatore ha la completa autorità (e responsabilità)

Spezzone di codice in linguaggio Java / JDBC

```
sql = "INSERT INTO xxx (id, geometria) VALUES (,?";  
sql + "GeomFromText ('POINT (,)???' , 4326))" =;  
stmt = conn.prepareStatement (SQL);  
stmt.setInt (1, 1);  
stmt.setDouble (2, 10.01);  
stmt.setDouble (3, 20.02);  
stmt.setDouble (4, 30.03);  
stmt.setDouble (5, 40.04);  
stmt.executeUpdate ();
```

- oops questo fallisce in Java: avrete un misterioso errore fatale **array-out-of-bounds** (eccezione irreversibile).

Post-scriptum: JDBC cerca di essere più furbo di voi. Durante il controllo dell'istruzione JDBC Prepared Statement (istruzione preparata) scopre il vostro dirty trick (sporco trucco): gli ultimi quattro argomenti sono racchiusi da apici singoli, così JDBC ignora semplicemente tutto perchè considera la stringa di caratteri un'entità *assolutamente intoccabile*.

Potete provare da soli usando **ParameterMetaData.getParameterCount()** ; questa dichiarazione preparata si aspetta semplicemente un argomento unico.



Febbraio 2011

Linguaggi di collegamento: Python

Verifica dell'ambiente

Linux Debian:

per essere sicuro di testare una situazione aggiornata allo stato dell'arte ho usato **Debian Squeeze** (32 bit). Così sono praticamente sicuro che tutti i pacchetti necessari stanno ragionevolmente usando le versioni più recenti.

Python:

Sul mio sistema di prova è già stato installato **python-2.6.6**, così immediatamente pronto a partire con il mio test.

pyspatialite connector:

Il sorgente del connettore è disponibile per essere scaricato dal sito <http://code.google.com/p/pyspatialite/>. Ho effettivamente scaricato l'ultima versione disponibile

<http://pyspatialite.googlecode.com/files/pyspatialite-2.6.1.tar.gz>. Per costruire ed installare il connettore **pyspatialite** ho usato i canonici scripts Python:

```
$ python setup.py build
... verbose output follows [suppressed] ...
$ su
# python setup.py install
```

Avviso molto importante: ho scoperto che l'uso degli scripts standard non è sufficiente: in questo modo ho ottenuto una versione obsoleta **v.2.3.1** di SpatiaLite e questa non è una bella cosa.

Attenzione

Le più recenti versioni di QGIS (1.6 / 1.7 trunk) in realtà usano SpatiaLite v.2.4.0 pertanto ogni plugin che usa la precedente versione v.2.3.1 può facilmente creare conflitti a causa della incompatibilità delle versioni. E questo rende completamente conto dei problemi segnalati dagli utenti di QGIS.

Sistemare setup.py

Ho velocemente scoperto che per disporre della più recente SpatiaLite v.2.4.0-RC4 basta poco: occorre semplicemente cambiare due linee nello script **setup.py** (**linea 87** e seguenti).

```

def get_amalgamation():
    """Download the Spatialite amalgamation if it isn't there, already."""
    if os.path.exists(AMALGAMATION_ROOT):
        return
    os.mkdir(AMALGAMATION_ROOT)
    print "Downloading amalgamation."
    # find out what's current amalgamation ZIP file
    download_page = urllib.urlopen("http://www.gaia-gis.it/spatialite-2.4.0-4/sources.html").read()
    pattern = re.compile("(libspatialite-amalgamation.*?\.zip)")
    download_file = pattern.findall(download_page)[0]
    amalgamation_url = "http://www.gaia-gis.it/spatialite-2.4.0-4/" + download_file
    zip_dir = string.replace(download_file,'.zip','')
    # and download it
    urllib.urlretrieve(amalgamation_url, "tmp.zip")

```

Una volta applicata questa banale correzione tutto ha funzionato nel modo più dolce e piacevole.

Programma di esempio in Python

spatialite_sample.py

```

# importing pyspatialite
from pyspatialite import dbapi2 as db

# creating/connecting the test_db
conn = db.connect('test_db.sqlite')

# creating a Cursor
cur = conn.cursor()

# testing library versions
rs = cur.execute('SELECT sqlite_version(), spatialite_version()')
for row in rs:
    msg = "> SQLite v%s Spatialite v%s" % (row[0], row[1])
    print msg

# initializing Spatial MetaData
# using v.2.4.0 this will automatically create
# GEOMETRY_COLUMNS and SPATIAL_REF_SYS
sql = 'SELECT InitSpatialMetadata()'
cur.execute(sql)

# creating a POINT table
sql = 'CREATE TABLE test_pt (' \
    'id INTEGER NOT NULL PRIMARY KEY,' \
    'name TEXT NOT NULL)'
cur.execute(sql)

# creating a POINT Geometry column
sql = "SELECT AddGeometryColumn('test_pt',"
" sql += "'geom', 4326, 'POINT', 'XY')"
cur.execute(sql)

# creating a LINESTRING table
sql = 'CREATE TABLE test_ln (' \
    'id INTEGER NOT NULL PRIMARY KEY,' \
    'name TEXT NOT NULL)'

```

```

cur.execute(sql)

# creating a LINESTRING Geometry column
sql = "SELECT AddGeometryColumn('test_ln', "
sql += "'geom', 4326, 'LINESTRING', 'XY')"
cur.execute(sql)

# creating a POLYGON table
sql = 'CREATE TABLE test_pg ('
sql += 'id INTEGER NOT NULL PRIMARY KEY, '
sql += 'name TEXT NOT NULL)'
cur.execute(sql)

# creating a POLYGON Geometry column
sql = "SELECT AddGeometryColumn('test_pg', "
sql += "'geom', 4326, 'POLYGON', 'XY')"
cur.execute(sql)

# inserting some POINTs
# please note well: SQLite is ACID and Transactional
# so (to get best performance) the whole insert cycle
# will be handled as a single TRANSACTION
for i in range(100000):
    name = "test POINT #%d" % (i+1)
    geom = "GeomFromText('POINT("
    geom += "%f " % (i / 1000.0)
    geom += "%f" % (i / 1000.0)
    geom += ")', 4326)"
    sql = "INSERT INTO test_pt (id, name, geom) "
    sql += "VALUES (%d, '%s', %s)" % (i+1, name, geom)
    cur.execute(sql)
conn.commit()

# checking POINTs
sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), "
sql += "ST_Srid(geom) FROM test_pt"
rs = cur.execute(sql)
for row in rs:
    msg = "> Inserted %d entities of type " % (row[0])
    msg += "%s SRID=%d" % (row[1], row[2])
    print msg

# inserting some LINESTRINGS
for i in range(100000):
    name = "test LINESTRING #%d" % (i+1)
    geom = "GeomFromText('LINESTRING("
    if (i%2) == 1:
        # odd row: five points
        geom += "-180.0 -90.0, "
        geom += "%f " % (-10.0 - (i / 1000.0))
        geom += "%f, " % (-10.0 - (i / 1000.0))
        geom += "%f " % (10.0 + (i / 1000.0))
        geom += "%f" % (10.0 + (i / 1000.0))
        geom += ", 180.0 90.0"
    else:
        # even row: two points
        geom += "%f " % (-10.0 - (i / 1000.0))
        geom += "%f, " % (-10.0 - (i / 1000.0))
        geom += "%f " % (10.0 + (i / 1000.0))

```

```

    geom += "%f" % (10.0 + (i / 1000.0))
    geom += ")", 4326)"
    sql = "INSERT INTO test_ln (id, name, geom) "
    sql += "VALUES (%d, '%s', %s)" % (i+1, name, geom)
    cur.execute(sql)
conn.commit()

# checking LINESTRINGS
sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), "
sql += "ST_Srid(geom) FROM test_ln"
rs = cur.execute(sql)
for row in rs:
    msg = "> Inserted %d entities of type " % (row[0])
    msg += "%s SRID=%d" % (row[1], row[2])
    print msg

# inserting some POLYGONS
for i in range(100000):
    name = "test POLYGON #%" % (i+1)
    geom = "GeomFromText('POLYGON(("
    geom += "%f " % (-10.0 - (i / 1000.0))
    geom += "%f, " % (-10.0 - (i / 1000.0)))
    geom += "%f " % (10.0 + (i / 1000.0))
    geom += "%f, " % (-10.0 - (i / 1000.0)))
    geom += "%f " % (10.0 + (i / 1000.0))
    geom += "%f, " % (10.0 + (i / 1000.0)))
    geom += "%f " % (-10.0 - (i / 1000.0)))
    geom += "%f, " % (10.0 + (i / 1000.0)))
    geom += "%f " % (-10.0 - (i / 1000.0)))
    geom += "%f" % (-10.0 - (i / 1000.0)))
    geom += "))', 4326)"
    sql = "INSERT INTO test_pg (id, name, geom) "
    sql += "VALUES (%d, '%s', %s)" % (i+1, name, geom)
    cur.execute(sql)
conn.commit()

# checking POLYGONS
sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), "
sql += "ST_Srid(geom) FROM test_pg"
rs = cur.execute(sql)
for row in rs:
    msg = "> Inserted %d entities of type " % (row[0])
    msg += "%s SRID=%d" % (row[1], row[2])
    print msg

rs.close()
conn.close()
quit()

```

\$ python spatialite_sample.py

Raccontare la storia intera è noioso e poco interessante: potete analizzare e provare il codice di esempio da soli, e questo è proprio tutto.



Febbraio 2011

Linguaggi di collegamento: PHP

Verifica dell'ambiente

Linux Debian:

Per essere sicuro di usare un ambiente aggiornato ho usato la distribuzione **Debian Squeeze** (32 bit). Sono quindi ragionevolmente sicuro che i pacchetti necessari sono aggiornati alla versione più recente.

Connettore PHP e SQLite:

La mia macchina virtuale Debian non ha tutto l'armamentario Apache e PHP già installato. Così ho iniziato la mia prova installando i seguenti pacchetti:

- **apache2**
- **php5-cli**
- **php5-sqlite**.

Avvertenza

Sembra che per operare con SpatiaLite sia necessario il più recente **PHP 5.3**.

Ho eseguito qualche ulteriore test con la vecchia **Debian Lenny**, scoprendo subito che PHP (e sqlite) erano così vecchi da rendere impossibile l'uso di SpatiaLite.

Penso che se siete fortemente interessati ad usare SpatiaLite, l'aggiornamento a **PHP 5.3** sia assolutamente necessario prima di fare qualsiasi prova.

Configurazione di PHP:

Ho subito scoperto che, usando la configurazione prestabilita di PHP, non si può caricare dinamicamente SpatiaLite come estensione del connettore SQLite. Occorre prima applicare almeno le seguenti modifiche nello script di configurazione **/etc/php5/apache2/php.ini**.

php.ini *prestabilito*

```
[sqlite3]
;sqlite3.extension_dir =
```

php.ini *modificato:*

```
[sqlite3]
sqlite3.extension_dir = /var/www/sqlite3_ext
```

Il connettore SQLite per PHP ha effettivamente capacità di caricare dinamicamente le estensioni. In ogni caso dovete abilitare esplicitamente una cartella per contenere tutte le estensioni da caricare dinamicamente.

```
# /etc/init.d/apache2 restart
```

Dopo la modifica dello script **php.ini** è obbligatorio riavviare il server Web Apache, per rendere efficace la nuova configurazione.

```
# mkdir /var/www/sqlite3_ext
# cp /usr/local/lib/libspatialite.so /var/www/sqlite3_ext
```

Ho semplicemente creato la cartella **/var/www/sqlite3_ext**.

E quindi ho copiato la libreria condivisa **libspatialite.so** dalla cartella **/usr/local/lib** in questa.

Si prega di notare bene: al fine di eseguire tutte le operazioni di cui sopra è necessario accedere come **root**

Esempio di programma in PHP

SpatialiteSample.php

```
<html>
  <head>
    <title>Testing SpatiaLite on PHP</title>
  </head>
  <body>
    <h1>testing SpatiaLite on PHP</h1>
<?php
# connecting some SQLite DB
# we'll actually use an IN-MEMORY DB
# so to avoid any further complexity;
# an IN-MEMORY DB simply is a temp-DB
$db = new SQLite3(':memory:');

# loading Spatialite as an extension
$db->loadExtension('libspatialite.so');
# enabling Spatial Metadata
# using v.2.4.0 this automatically initializes SPATIAL_REF_SYS
# and GEOMETRY_COLUMNS
$db->exec("SELECT InitSpatialMetadata()");

# reporting some version info
$rs = $db->query('SELECT sqlite_version()');
while ($row = $rs->fetchArray())
{
  print "<h3>SQLite version: $row[0]</h3>";
}
$rs = $db->query('SELECT spatialite_version()');
while ($row = $rs->fetchArray())
{
  print "<h3>Spatialite version: $row[0]</h3>";
}
```

```

# creating a POINT table
$sql = "CREATE TABLE test_pt (";
$sql .= "id INTEGER NOT NULL PRIMARY KEY,";
$sql .= "name TEXT NOT NULL)";
$db->exec($sql);

# creating a POINT Geometry column
$sql = "SELECT AddGeometryColumn('test_pt', ";
$sql .= "'geom', 4326, 'POINT', 'XY')";
$db->exec($sql);

# creating a LINESTRING table
$sql = "CREATE TABLE test_ln (";
$sql .= "id INTEGER NOT NULL PRIMARY KEY,";
$sql .= "name TEXT NOT NULL)";
$db->exec($sql);

# creating a LINESTRING Geometry column
$sql = "SELECT AddGeometryColumn('test_ln', ";
$sql .= "'geom', 4326, 'LINESTRING', 'XY')";
$db->exec($sql);

# creating a POLYGON table
$sql = "CREATE TABLE test_pg (";
$sql .= "id INTEGER NOT NULL PRIMARY KEY,";
$sql .= "name TEXT NOT NULL)";
$db->exec($sql);

# creating a POLYGON Geometry column
$sql = "SELECT AddGeometryColumn('test_pg', ";
$sql .= "'geom', 4326, 'POLYGON', 'XY')";
$db->exec($sql);

# inserting some POINTs
# please note well: SQLite is ACID and Transactional
# so (to get best performance) the whole insert cycle
# will be handled as a single TRANSACTION
$db->exec("BEGIN");

for ($i = 0; $i < 10000; $i++)
{
    # for POINTs we'll use full text sql statements
    $sql = "INSERT INTO test_pt (id, name, geom) VALUES (";
    $sql .= $i + 1;
    $sql .= ", 'test POINT #'";
    $sql .= $i + 1;
    $sql .= "', GeomFromText('POINT(";
    $sql .= $i / 1000.0;
    $sql .= " ";
    $sql .= $i / 1000.0;
    $sql .= ")', 4326))";
    $db->exec($sql);
}

$db->exec("COMMIT");

```

```

# checking POINTs
$sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom) , ";
$sql .= "ST_Srid(geom) FROM test_pt";
$rs = $db->query($sql);
while ($row = $rs->fetchArray())
{
    # read the result set
    $msg = "Inserted ";
    $msg .= $row[0];
    $msg .= " entities of type ";
    $msg .= $row[1];
    $msg .= " SRID=";
    $msg .= $row[2];
    print "<h3>$msg</h3>";
}

# inserting some LINESTRINGS
# this time we'll use a Prepared Statement
$sql = "INSERT INTO test_ln (id, name, geom) ";
$sql .= "VALUES (?, ?, GeomFromText(?, 4326))";
$stmt = $db->prepare($sql);
$db->exec("BEGIN");
for ($i = 0; $i < 10000; $i++)
{
    # setting up values / binding
    $name = "test LINESTRING #";
    $name .= $i + 1;
    $geom = "LINESTRING(";
    if (($i%2) == 1)
    {
        # odd row: five points
        $geom .= "-180.0 -90.0, ";
        $geom .= -10.0 - ($i / 1000.0);
        $geom .= " ";
        $geom .= -10.0 - ($i / 1000.0);
        $geom .= ", ";
        $geom .= -10.0 - ($i / 1000.0);
        $geom .= " ";
        $geom .= 10.0 + ($i / 1000.0);
        $geom .= ", ";
        $geom .= 10.0 + ($i / 1000.0);
        $geom .= " ";
        $geom .= 10.0 + ($i / 1000.0);
        $geom .= ", 180.0 90.0";
    }
    else
    {
        # even row: two points
        $geom .= -10.0 - ($i / 1000.0);
        $geom .= " ";
    }
}

```

```

$geom .= -10.0 - ($i / 1000.0);
$geom .= ", ";
$geom .= 10.0 + ($i / 1000.0);
$geom .= " ";
$geom .= 10.0 + ($i / 1000.0);

}

$geom .= ")";

$stmt->reset();
$stmt->clear();
$stmt->bindValue(1, $i+1, SQLITE3_INTEGER);
$stmt->bindValue(2, $name, SQLITE3_TEXT);
$stmt->bindValue(3, $geom, SQLITE3_TEXT);
$stmt->execute();

}

$db->exec("COMMIT");

# checking LINESTRINGS
$sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), ";
$sql .= "ST_Srid(geom) FROM test_ln";
$rs = $db->query($sql);
while ($row = $rs->fetchArray())
{
    # read the result set
    $msg = "Inserted ";
    $msg .= $row[0];
    $msg .= " entities of type ";
    $msg .= $row[1];
    $msg .= " SRID=";
    $msg .= $row[2];
    print "<h3>$msg</h3>";
}

# inserting some POLYGONS
# this time too we'll use a Prepared Statement
$sql = "INSERT INTO test_pg (id, name, geom) ";
$sql .= "VALUES (?, ?, GeomFromText(?, 4326))";
$stmt = $db->prepare($sql);
$db->exec("BEGIN");
for ($i = 0; $i < 10000; $i++)
{
    # setting up values / binding
    $name = "test POLYGON #";
    $name .= $i + 1;
    $geom = "POLYGON((";
    $geom .= -10.0 - ($i / 1000.0);
    $geom .= " ";
    $geom .= -10.0 - ($i / 1000.0);
    $geom .= ", ";
}

```

```

$geom .= 10.0 + ($i / 1000.0);
$geom .= " ";
$geom .= -10.0 - ($i / 1000.0);
$geom .= ", ";
$geom .= 10.0 + ($i / 1000.0);
$geom .= " ";
$geom .= 10.0 + ($i / 1000.0);
$geom .= ", ";
$geom .= -10.0 - ($i / 1000.0);
$geom .= " ";
$geom .= 10.0 + ($i / 1000.0);
$geom .= ", ";
$geom .= -10.0 - ($i / 1000.0);
$geom .= " ";
$geom .= -10.0 - ($i / 1000.0);
$geom .= ") )";

$stmt->reset();
$stmt->clear();
$stmt->bindValue(1, $i+1, SQLITE3_INTEGER);
$stmt->bindValue(2, $name, SQLITE3_TEXT);
$stmt->bindValue(3, $geom, SQLITE3_TEXT);
$stmt->execute();

}
$db->exec("COMMIT");

# checking POLYGONS
$sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), ";
$sql .= "ST_Srid(geom) FROM test_pg";
$rs = $db->query($sql);
while ($row = $rs->fetchArray())
{
    # read the result set
    $msg = "Inserted ";
    $msg .= $row[0];
    $msg .= " entities of type ";
    $msg .= $row[1];
    $msg .= " SRID=";
    $msg .= $row[2];
    print "<h3>$msg</h3>";
}

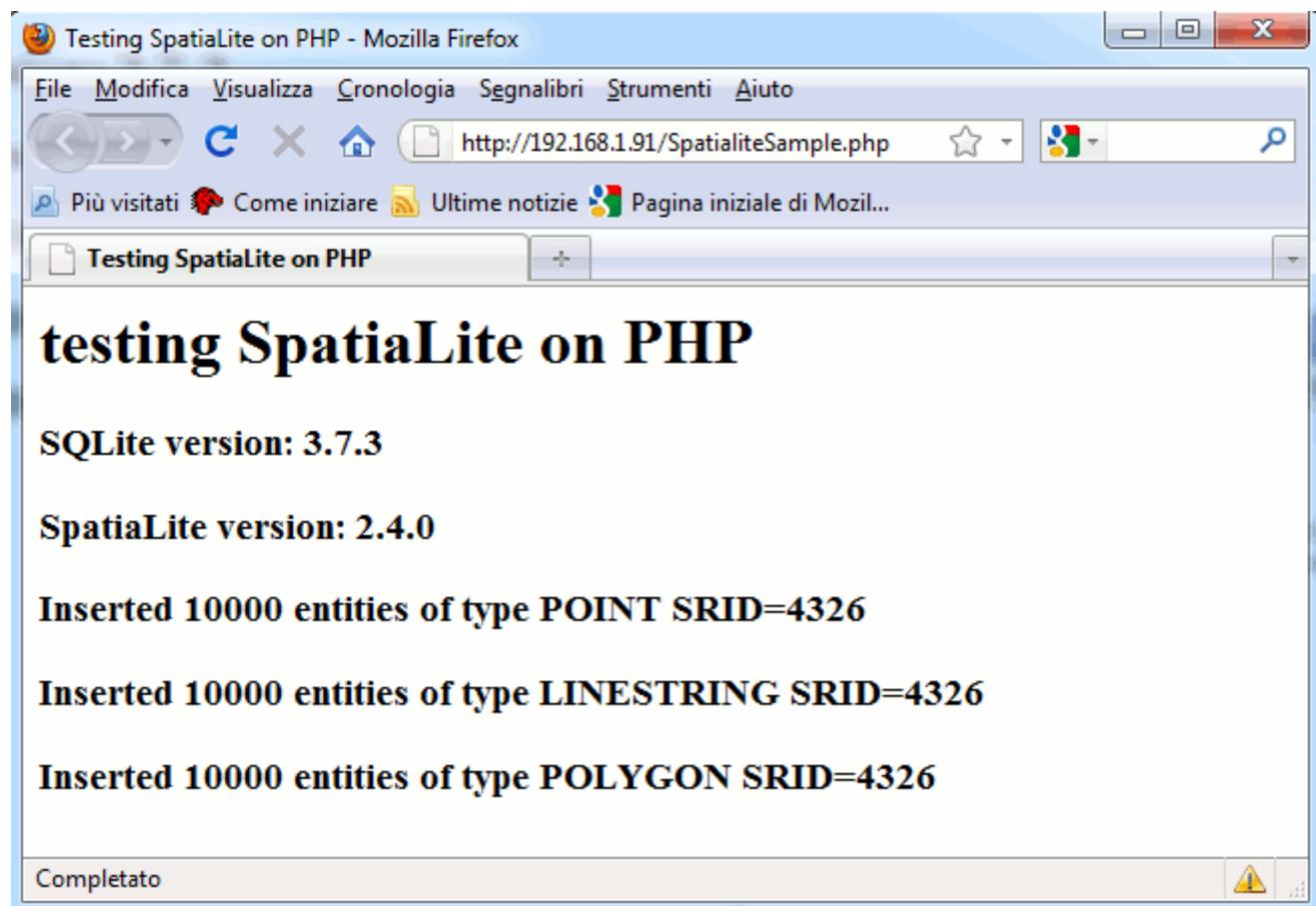
# closing the DB connection
$db->close();
?>

</body>
</html>

```

Ho salvato questo script di esempio PHP come: **/var/www/SpatialiteSample.php**.

Poi ho semplicemente iniziato il mio browser WEB Firefox richiedendo l'URL corrispondente:

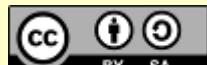


E' tutto.

Prendete nota: può essere che usando altre distribuzioni Linux (o Windows) occorra sistemare i percorsi dei file come richiesto dalla vostra *specifica piattaforma*

Tradotto in "italiano" da:

Giuliano Curti giulianc@tiscali.it
Filippo Racioppi rfilippor@gmail.com



Author: Alessandro Furieri a.furieri@lqt.it

Traduced in italiano by Giuliano Curti giulianc@tiscali.it and Filippo Racioppi rfilippor@gmail.com
This work is licensed under the [Attribution-ShareAlike 3.0 Unported \(CC BY-SA 3.0\)](#) license.



Permission is granted to copy, distribute and/or modify this document under the terms of the [GNU Free Documentation License](#), Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.