Breve Introduzione a PostGIS

Claudio Rocchini
claudio@rockini.name

17 dicembre 2013

Edizione 1
Stampato su Lulu

Claudio Rocchini
Breve Introduzione a PostGIS

Queste note nascono dall'esperienza delle lezioni tenute dall'autore durante il Master di II livello in *Sistemi informativi geografici per la gestione e il monitoraggio del territorio* dell'Universitá di Firenze ed il *Corso di Aggiornamento in DB Topografici* del Centro di GeoTecnologie dell'Universitá di Siena.

Indice

1 Introduzione alle Basi di Dati

Premessa: se hai una vaga idea di cosa siano le basi di dati, salta al prossimo capitolo.

Vediamo brevemente alcuni aspetti che riguardano le *Basi di Dati*, con particolare riferimento al loro utilizzo con i dati geografici. Una introduzione (anche breve) alle Basi di Dati, richiederebbe un intero corso: presenteremo quindi i concetti minimi necessari alla comprensione di questo testo.

1.1 Il Modello Relazionale

Partiamo con un esempio: nasce l'esigenza di realizzare un sistema informatico che memorizzi i dati sulle strade italiane. Ad ogni strada verranno associati alcuni valori (attributi), come ad esempio un eventuale nome, il numero di corsie etc. É importante anche conoscere l'elenco delle regioni d'Italia attraversate da ogni strada (e viceversa). Ogni strada fa parte di una classifica funzionale (es. autostrada, extraurbana principale, urbana di quartiere, etc.). Vediamo di seguito come questa esigenza sia risolta attraverso una Base di Dati nel Modello Relazionale.

Le basi di dati contemporanee sono realizzate nel cosidetto *Modello Relazionale*[1]. Il modello relazionale è caratterizzato dalle seguenti componenti principali:

[1] In passato esistevano altri modelli (es. gerarchico, reticolare)

entità : corrispondono alle classi di oggetti distinguibili della base di dati, nel nostro esempio le intità sono le strade, le regioni e le classifiche funzionali. Le *entità* sono caratterizzate da attributi associati, nel nostro esempio gli attributi delle strade sono il nome, il numero di corsie, etc.

relazioni : le relazioni legano fra di loro le intità. Ad esempio le strade e le regioni sono legate dalle relazione di inclusione (una strada passa per una regione). Come vedremo più avanti, le relazioni possono essere di vario tipo:

- *uno a uno*: ad ogni oggetto di una entità ne corrisponde uno ed uno solo dell'altra;
- *uno a molti*: ad ogni oggetto di una entità ne corrispondono uno o più dell'altra. Ad esempio ad ogni classifica funzionale (autostrada) corrispondono una serie di strade di quel tipo (A1, A13, ...);
- *molti a molti*: ad ogni oggetto di una entità ne corrispondo uno o più dell'altra *e viceversa*. Ad esempio per ogni regione passano molte strade, viceversa una strada può passare per più regioni.

1.1.1 Il Diagramma ER (Entità-Relazioni)

Una base di dati relazionale viene presentata graficamente attraverso il *Diagramma ER* (Entità-Relazioni), di cui potete vedere un esempio nella figura 1.1.

Nel Diagramma ER vengono rappresentate le *entità* (rettangoli), con i relativi *attributi* (ellissi). Le entità sono poi interconnesse attraverso *relazioni* (losanghe). Le relazioni possono riportare le diciture $1-1$, $1-n$, $n-n$, che indicano la cardinalità della realazione (rispettivamente uno a uno, uno a molti, molti a molti.

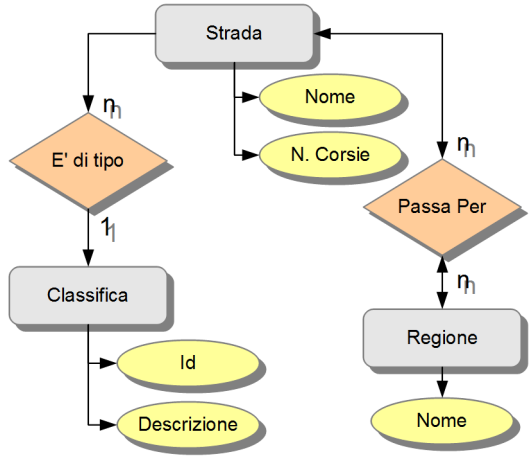


Figura 1.1: Esempio di *Diagramma ER* (Entità-Relazioni).

1.1.2 Realizzazione del Modello Relazionale

In pratica, il modello relazione è realizzato attraverso una serie di *tabelle*. In particolare:

1. ad ogni *entità* (es. strada) corrisponde una *tabella*;
2. ad ogni *attributo* (es. nome) di una entità corrisponde una *colonna*;
3. ad ogni singolo *oggetto* (es. una particolare strada) corrisponde una *riga* della relativa tabella;
4. le relazioni sono realizzate:
   a) implicitamente dagli attributi e con vincoli (se di cardinalità $1-1$ o $1-n$);
   b) attraverso un'ulteriore tabella (se di cardinalità $n-n$).

Nella tavola 1.1 potete vedere un esempio di realizzazione con tabelle, delle entità *strada* (a sinistra) e *classifica* (a destra). Ad esempio l'entità *strada* corrisponde ad una tabella con tre colonne, una per ogni attributo: *nome*, *numero corsie* e *classifica*. Ogni riga della tabella poi corrisponde ad una particolare strada: A1, A22, Aurelia...

La relazione $1 - n$ fra strade e classifiche è memorizzata implicitamente nella coppia di colonne *Class* della tabella *Strada* e *Codice* della tabella *Classifica*: la relazione si ottiene facendo coincidere i valori di queste due colonne. Ad esempio la strada *A1* è classificata come *autostrada* (Codice=01), mentre l'*Emilia* è classificata come *extraurbana principale* (Codice=02).

Una cosa di ricordarsi bene è che in una base di dati le righe di una tabella non hanno un *ordine predefinito*. Sarà poi il modo in cui la tabella viene interrogata che definirà l'ordine delle righe.

Strada

Nome	**N.C.**	**Class**
A1	4	01
A22	4	01
Aurelia	4	02
Emilia	2	02
...	...	...

Classifica

Codice	**Descrizione**
01	autostrada
02	extraurbana principale
03	extraurbana secondaria
...	...

Tabella 1.1: Esempio di realizzazione di entità e relazioni: strada e classifica.

La tavola 1.2 mostra la realizzazione della relazione $n - n$ fra strade e regioni: questo tipo di relazione richiedere la creazione di una tabella aggiuntiva, che memorizza le coppie di valori strada-regione, per ogni strada che passa per una regione.

Strada		
Nome	**N.C.**	**Class**
A1	4	01
A22	4	01
Aurelia	4	02
Emilia	2	02
...	...	...

SR	
S	**R**
A1	Lazio
A1	Campania
Aurelia	Toscana
Aurelia	Lazio
...	...

Regione
Nome
Lazio
Toscana
Campania
...

Tabella 1.2: Esempio di relazione $n - n$ fra strade e regioni.

1.2 Documenti e Basi di Dati

Molte delle applicazioni che siamo abituati ad utilizzare (sistemi di scrittura, editor di immagini, etc.) adottano il principio del *documento di lavoro*: utilizzando una di queste applicazioni possiamo creare un nuovo documento, salvarlo su disco, eventualmente ricaricalo in seguito. Dopo aver fatto delle modifiche ad un documento possiamo decidere se salvarlo oppure no; c'è una distinzione quindi fra documento in *memoria* e documento salvato. Il documento poi, di solito, corrisponde ad un file su disco; questo file può essere spostato, copiato, inviato per posta etc. Nella maggior parte dei casi i documenti possono essere modificati da una persona alla volta (non è possibile lavorare contemporanemante sullo stesso documento).

I dati contenuti in una base di dati, funzionano con una filosofia diversa: non c'è distinzione fra dato in memoria e su disco; si suppone che i dati siamo instataneamente memorizzati su disco. Non c'è quindi il concetto di salvataggio e caricamento del documento. Invece di caricamento si parla piuttosto di *connessione* ai dati. Anche se i sistemi di basi di dati ad "uso casalingo" come Microsoft Access© salvano i database in singoli file, in realtà i sistemi di basi di dati generalmente sono realizzati attraverso servizi remoti, in cui non è possibile sapere quali file contengano effettivamente i dati.

La copia dei dati o la loro spedizione deve avvenire quindi attraverso sistemi dedicati di importazione/esportazione. Infine le basi di dati sono studiate per un utilizzo condiviso e contemporaneo fra più utenti.

1.3 Concetti delle Basi di Dati

Vediamo adesso alcuni concetti minimi riguardanti le basi di dati.

1.3.1 Tipi di Dato

Gli attributti delle entità (e quindi le colonne delle tabelle) non sono tutti uguali. Ad ogni attributo è associato un particolare *tipo di dato*. I principali tipi di dato sono:

- *testi*: dati che contengono parole e frasi, con lunghezza in caratteri prefissata oppure no (es. nomi di strade);
- *numeri*: dati che contengono valori numerici (es. numeri indentificativi, larghezze e lunghezze, etc.); si distinguono poi vari sotto tipi: numeri interi, numeri con la virgola a singola o doppia precisione, etc.
- *date, ore, intervalli di durata, etc.*: dati che riguardano il tempo (es. data di creazione di un oggetto, misure temporali, etc.);
- *boolean*: valori di verità, ovvero dati che possono prendere solo uno dei due valori di *vero* o di *falso* (es. stato di percorribilità di una strada)[2].
- *tipi speciali*: altri tipi di dato speciale, come *GEOMETRY* che permette la memorizzazione di dati geografici.

[2]Curiosamente Oracle© non possiede questo tipo di dato.

Ad ogni colonna di una tabella deve essere assegnato un tipo di dato, dopodichè i valori contentuti devono appartenere a quel tipo. Ad esempio se una colonna è di tipo numerico non potrà in alcun modo contenere valori testuali; il contenuto di una colonna quindi deve essere di tipo omogeneo.

1.3.2 Chiavi

Come abbiamo detto, l'ordine delle righe di una tabella non è definito a priori; ci vuole un modo quindi per poter indentificare una particolare riga. É opportuna che in ogni tabella sia definita la *chiave primaria*: una particolare colonna (oppure un insieme di colonne) che contiene valori unici (tutti diversi tra loro) per ogni riga della tabella. La chiave primaria rappresenta quindi un attributo che permette di identificare in modo univoco un oggetto, e quindi una riga della tabella. Ad esempio, volendo realizzare una tabella contenente dati personali, una buona chiave primaria è rappresentata dal codice fiscale.

Spesso per i dati geografici è difficile definire un'attributo proprio univoco, ad esempio non tutte le strade d'Italia hanno un nome diverso (oppure addirittura non è detto che abbiano un nome). In questo caso si ovvia alla mancanza di una chiave, aggiungendo un opportuno attributo numerico progressivo, che numera i nostri oggetti e ne definisce la chiave primaria.

Oltre alla chiave primaria esistono anche le *chiavi esterne*, particolari colonne che realizzano le relazioni fra tabelle, come ad esempio la colonna *class* della tabella *strada* nella tavola 1.1.

1.3.3 Indici

Gli *indici* rappresentano il cuore del funzionamento di una base di dati, ma spesso la loro presenza rimane nascosta. Gli indici sono strutture associate ad una o più colonne di una tabella e servono per velocizzare la ricerca dei dati su queste colonne. Per il buon

funzionamento di una base di dati, bisogna creare un indice per ogni dato che si prevede sia oggetto di ricerca. Una volta creato, l'indice di un dato viene automaticamente aggiornato quando i dati vengono modificati.

Come vedremo più avanti, esistono anche gli *indici spaziali*, che permettono l'utilizzo efficiente dei dati geografici.

1.3.4 Transazioni

Spesso una base di dati è utilizzata da più utenti in contemporanea; le singole operazioni sui dati sono garantite per definizione, ma talvolta è utile che una serie di aggiornamenti ai dati sia garantita in modo coerente. Supponete di dover prenorare due posti vincini sull'Eurostar; il gestore probabilmente mantiene una base di dati con i posti prenotati. Ovviamente la prenotazione avviene in concorrenza con molti utenti, quindi se i vostri due posti fossero assegnati uno alla volta, potrebbe capitare che fra un posto ed un altro qualcun'altro infili la sua prenotazione. Il meccanismo delle *transazioni* permette di incaspulare in un unico contenitore una serie di modifiche ai dati, che poi vengono eseguite in modo indivisibile (se possibile, altrimenti nessuna modifica viene eseguita).

1.3.5 Schemi

Le basi di dati reali contengono un gran numero di tabelle, la cui gestione può risultare difficoltosa. I sistemi di basi di dati prevedono un meccanismo per suddividere le tabelle: gli *schemi*. Uno schema ha una funzione simile alle cartelle per i file su disco; è possibile creare una serie di schemi e quindi ripartire le tabelle all'interno di questi contenitori. Due tabelle in schemi diversi possono avere lo stesso nome, mentre ovviamente non lo possono avere se si trovano all'interno dello stesso schema.

1.4 Forme Normali

Con i termini *Prima*, *Seconda* e *Terza Forma Normale* di definiscono una serie di regole che definiscono quale sia una buona base di dati. Le tre regole si possono riassumere nel fatto che ogni tabella deve contenere una chiave primaria, gli attributi devono contenere valori *atomici* (cioè non liste di valori) e i dati non devono essere ridondanti o ripetuti.

Ad esempio, se una tabella contiene le date di nascita e i segni zodiacali di alcune persone, non è in forma normale, dato che il segno zodiacale può essere derivato dalla data di nascita.

1.5 Utilizzo delle Basi di Dati

Una volta che possiedo una base di dati, che cosa me ne faccio?

1.5.1 Interrogazioni

Le basi di dati si caratterizzano tra l'altro per essere formate da una grande mole di informazioni. L'operazione tipica che si effettua su una base di dati è l'*interrogazione* (*query* in inglese). Un'interrogazione server per *estrarre* dalla grande mole di dati presente nella nostra base, le sole informazioni che ci interessano, secondo opportuni criteri. Le tipiche interrogazioni su una base di dati prevedono:

- l'estrazione di una parte degli attributi (selezione di alcune colonne);
- l'estrazione di un sottoinsieme di oggetti (selezione di una parte delle righe), specificando alcune condizioni di filtro;
- la combinazione di dati di due o più tabelle (operazione di *join*), sfruttando le relazioni presenti, per l'estrazione di nuovi dati;

- altri tipi di interrogazione, come quelle di tipo spaziale.

1.5.2 Viste

Le interrogazioni più interessanti possono essere *salvate* dentro la base di dati (si salva la struttura dell'interrogazione, non il suo risultato!). Le interrogazioni salvate nella base di dati vengono chiamate *viste* (*view* in inglese), perchè servono per *vedere* in una particolare ottica i dati presenti nella base. Una volta create, le viste si comportano in modo simile alle tabelle, anche se non contengono propriamente nessun dato. Le viste hanno un comportamento dinamico: se i dati delle tabelle su cui sono definite cambiano, cambia instantaneamente anche il contenuto della vista.

1.5.3 SQL

I sistemi di gestione delle basi di dati sono molti (Oracle©, Access©, Microsoft SQL Server©, IBM DB2©, PostgreSQL, MySQL, ...), ogniuno con le sue caratteristiche. Esiste però un linguaggio standard per l'utilizzo di ogni base di dati: *SQL* (=Structured Query Language).

In teoria, conoscendo SQL è possibile utilizzare in modo uniforme ogni sistema di basi di dati. In pratica vedremo che possono esserci alcune differenze fra un produttore di software e l'altro, soprattutto per quanto riguarda la parte spaziale. SQL verrà introdotto nel prossimo capitolo; nel listato 4.1 potete vedere in anteprima un esempio di interrogazione che seleziona i nomi di tutte le strade con classifica autostradale.

```
1 SELECT nome
2 FROM   strada
3 WHERE  class='01';
```

Listato 1.1: Esempio di interrogazione SQL.

1.5.4 Interrogazioni Spaziali

Nelle basi di dati spaziali (argomento di questo libro) è possibile infine realizzare delle interrogazioni con relazioni spaziali, ad esempio è possibile selezionare gli oggetti che sono entro una certa distanza da un dato punto, oppure trovare tutte le coppie di oggetti adiacenti, etc. Una dettagliata introduzione alle interrogazioni spaziale sarà l'oggetto principale dei prossimi capitoli.

2 Breve introduzione a SQL

Se sapete già qualcosa di SQL, passate al prossimo capitolo.

2.1 Introduzione

SQL (sigla che sta per Structured Query Language) è un linguaggio testuale standard per operare con le basi di dati. Standard vuol dire che è (quasi) indipendente la particolare database scelto (*Oracle, Microsoft SQL Server, Postgres, Mysql*, etc.). Il linguaggio è funzionale (un solo costrutto esegue le operazioni specificate), non imperativo (non ci sono variabili o elenchi di operazioni), anche se una sua estensione (il PL-SQL) permette di dichiarare funzioni in modo imperativo.

Un'introduzione al linguaggio richiederebbe un corso universitario annuale: in questa breve nota si vuole dare una breve descrizione alla struttura del linguaggio, in modo che poi sia possibile introdurne la parte propriamente spaziale. Inizieremo col vedere gli elementi di base (tipi di dato: numeri e parole), passeremo quindi alla definizione dei dati (schemi, colonne e tabelle), alle operazioni di inserimento e modifica dei dati, quindi all'interrogazione degli stessi. Per finire faremo un breve accenno agli elementi avanzati: indici, chiavi e relazioni.

2.2 Postgres

Tutte le esercitazioni verranno effettuate sul database *PostgreSQL*[1], un database gratuito che ha un supporto spaziale molto ben sviluppato. Un altro database che ha un ottimo supporto spaziale è *Oracle*.

Non prendiamo in considerazione le procedure di scaricamento od installazione del software, che sono molto variabili da una versione all'altra. Questo documento non vuole essere in alcun modo un manuale di riferimento di PostgreSQL. Il nostro scopo è quello di introdurre SQL nel modo più indipendente possibile dal particolare software (marca e versione) utilizzato.

Per utilizzare la base di dati faremo uso dell'Interfaccia standard *pgAdmin III*, esistono però una serie di interfacce più evolute. Nel nostro esempio il server della base di dati si trova sulla stessa macchina dell'interfaccia; nessuno vieta però l'utilizzo su di una macchina remota.

2.3 Prepararsi al Lavoro

Lanciate Postgres - pgAdmin III: vi apparirà la finestra in figura 2.1. Se non avete la configurazione del server che vi serve, selezionate il menù File-Add Server, altrimenti cliccate su *PostgreSQL X.Y* oppure su *localhost* (è lo stesso server), che è il vostro server locale. Se avete impostato una password durante l'installazione, questa vi verrà richiesta. Nella parte sinistra della finestra viene visualizzato un albero con tutti gli oggetti presenti sul server. Per prima cosa l'elenco dei database presenti (ogni server ne può contenere molti), un elenco di *tablespace* ed un elenco di *Group* e *Login*. I *tablespace* servono per gestire la memorizzazione fisica dei dati; ad esempio grandi moli di dati possono essere partizionate in più tablespace

[1] In particolare PostgreSQL 9.1.2, scaricato da http://www.postgresql.org

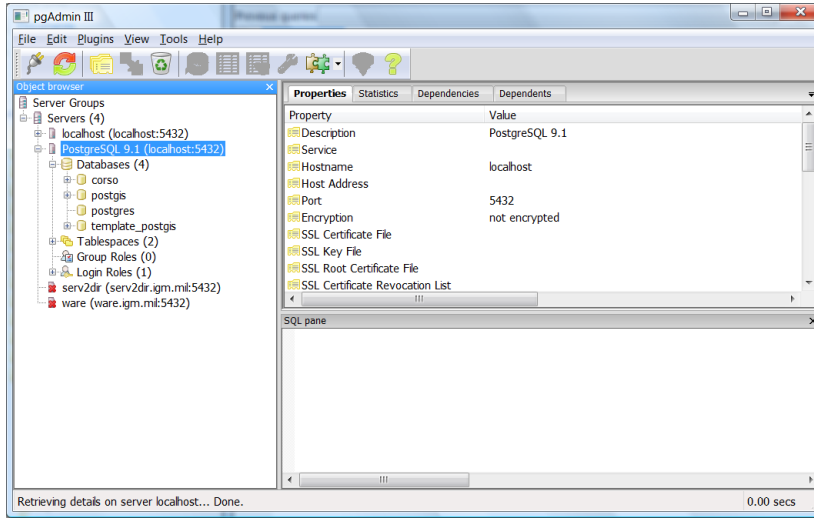


Figura 2.1: La schermata di avvio di pgAdmin III.

per aumerare l'efficienza. Questo argomento esula dai nostri scopi. La gestione dei gruppi e dei login permette di regolare i livelli di accesso ad ogni risorsa, in presenza di molti utilizzatori; anche questo argomento esula dai nostri scopi.

Come abbiamo visto, il nostro server contiene già alcuni database, ma per i nostri esperimenti ne creeremo uno ad hoc. Cliccate col *bottone destro* sulla casella *databases* nell'albero e selezionate il menù *New Database...* . Nel diaologo che si apre, scrivere il nome del database da creare (corso), controllate che l'encoding sia *UTF8*; questo vuol dire che il db memorizzerà i testi utilizzando questa codifica di carattere. Inoltre selezionate nella casella *template* il valore *postgis_template*. Se questa scelta non è presente vuol dire che non avete installato il supporto spaziale. Per ora questa scelta è un poco oscura; in seguito si vedrà che questa opzione abilita il supporto geografico al database appena creato. Una volta creato il nostro database, questo apparirà nell'albero; cliccateci sopra ed aprite gli oggetti contenuti. C'è un sacco di roba... ma nieste

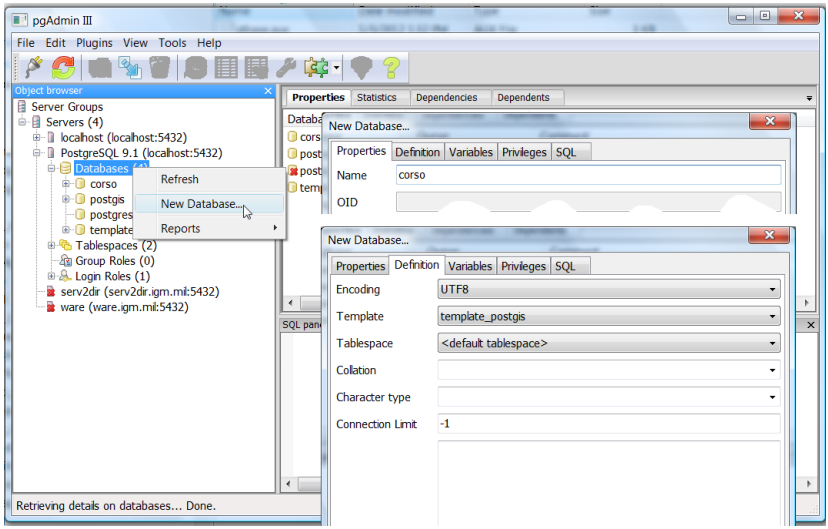


Figura 2.2: Dialogo di creazione di un nuovo db.

paura, a noi interessano solo poche cose. Innazzitutto vedete la casella *Schemas*; questi sono gli schemi in cui un db può essere suddiviso. Corrispondolo alle cartelle di un disco. Il nostro db contiene lo schema di default, che si chiama *public*. Lo schema public a sua volta contiene tra l'altro le *Tables* (le tabelle) e la *Views* (le viste) ovvero le interrogazioni salvate con un nome.

Cliccando col bottone destro sui vari oggetti dell'albero è possibile effettuare delle operazioni tramite interfaccia grafica; ad esempio cliccando su *tables* è possibile creare nuove tabelle (figura 2.3). Noi non utilizzeremo mai questa opzione durante l'esercitazione, ma effettueremo ogni operazione tramite il linguaggio SQL. Questo perchè tale linguaggio è standard per tutti gli altri software di database.

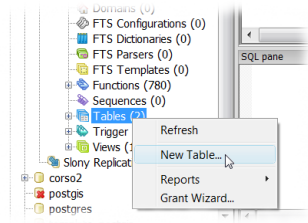


Figura 2.3: Contenuto del db ed interfaccia.

2.4 La finestra di comandi SQL

Una volta che avete selezionato il database *corso* cliccate sullo strumento *SQL*, rappresentato dall'icona la lente di ingrandimento e la scritta SQL (oppure da una matita e il foglio nelle versioni precedenti). Si aprirà la finestra col editor SQL (figura 2.4). Nella

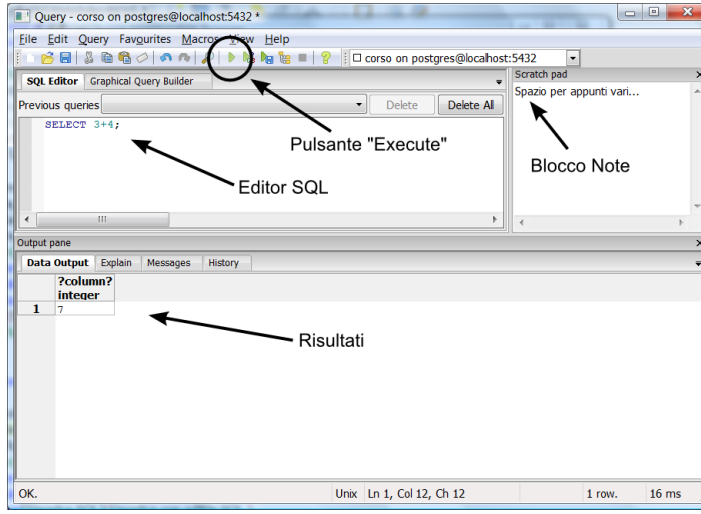


Figura 2.4: Finestra con editor SQL.

parte sinistra della finistra c'è l'editor vero e proprio, nella parte destra c'è una specie di blocco per gli appunti, utile per copiare ed

incollare testi. Nella parte sottostante invece vengono visulizzati i risultati delle operazioni ed altre cose.

L'utilizzo della finestra è il seguente: si scrive la query SQL nella finestra di sinistra, si preme il tasto *Execute* (icona play) e si legge il risultato nella parte sottostate.

Per chi non ha dimestichezza con la rigidità di un linguaggio formale per computer, l'approccio iniziale sarà molto duro. La sintassi SQL deve essere esatta: ricordatevi di non inserire spazi all'interno delle parole dei comandi, di non confondere zero con la lettera o, di non confondere l'apicetto singolo con le doppie virgolette o di non confondere le vigole, punti e punti e virgola. Per fortuna, SQL non è mai *case sensitive*, vale a dire che non si distingue maiuscole e minuscole (potete scrivere indifferentemente SELECT, select o Select).

Nota sullo stile di questa dispensa: i comandi SQL saranno scritti con il font *courier*; negli esempi, per chiarezza, scriveremo sempre i comandi di SQL in maiuscolo, mentre scriveremo in minuscolo i valori ed i nomi definiti dall'utente. Si ricorda che SQL non distingue in genere le maiuscole dalle minuscole[2]. In generale i comandi SQL saranno scritti su più righe e con opportuna indentatura: questa suddivisione viene fatta solo per chiarezza, dato che in SQL la divisione in righe non è significativa ai fini dell'interpretazione della query.

2.5 Pre-Introduzione al comando SELECT

Il comando fondamentale di SQL è *SELECT* e verrà spiegato in dettaglio più avanti. Dobbiamo però introdurlo per effettuare

[2]Sse si vuole specificare un nome (di tabella o di colonna) in maiuscolo/-minuscolo in modo specifico, è necessario racchiudere il nome fra doppie virgolette.

alcune prove sui dati: questo comando infatti ci permetterà di visualizzare le operazioni effettuate sui tipi di dati di base. La sintassi minima del comando *SELECT* è:

```
1 SELECT {valori};
```

dove *valori* è la cosa che ci interessa selezionare.

Ad esempio provate a scrivere nella vostra finestra SQL il seguente comando:

```
1 SELECT 42;
```

quindi premete il pulsante *Execute*. Il risultato visualizzato visualizzato nella finestra in basso sarà *42*. La query che abbiamo scritto richiede infatti al sistema il numero 42; una query non molto intelligente per ora, ma miglioreremo in futuro[3].

Il risultato di una query è sempre (anche in questo caso) una tabella, il numero 1 a sinistra del risultato (vedi figura 2.4 in basso) sta a significare che questa è la prima riga della tabella risultato, mentre la scritta sopra il numero 42 è il nome della prima colonna (in questo caso *?column?*). Per rendere più chiara la differenza fra il valore ed il nome di una colonna dell risultato provate a scrivere il comando:

```
1 SELECT 42 AS valore;
```

Scrivendo dopo il valore desiderato l'istruzine*AS* seguita da un nome, è possibile dare il nome specificato alla colonna del risultato.

2.6 Valori letterali

Per *valori letterali* si intendono valori costanti dati, come numeri o parole. Come in molti linguaggi di programmazione, in SQL è

[3]Nota curiosa: in Oracle è obbligatorio specificare sempre almeno una tabella anche se non serve. A questo scopo esiste sempre una tabella fittizia che si chiama *DUAL* da cui è possibile selezionare tutto, in quanto non contiene niente.

possibile operare con i numeri interi e numeri con la virgola (che si scrive punto, alla moda anglosassone); è inoltre possibile calcolare espressioni o chiamare funzioni matematiche e sulle parole. Provate ad eseguire:

```
1 SELECT 21*2;
```

oppure per i matematici:

```
1 SELECT cos(3.1415926);
```

Dove il simbolo $*$ sta per la moltiplicazione, e *cos* per la chiamata alla funzione coseno. Provate ad indovinare quali saranno i risultati di queste query.

Oltre che con i numeri, è possibile operare con le parole (stringhe di caratteri). Per distiguere le parole intese come valori dai nomi di colonne e tabelle, è necessario racchiudere le parole fra apicetti singoli (non doppie virgolette!). Ad esempio provate ad eseguire:

```
1 SELECT 'Buongiorno';
```

Il risultato sarà la parola *Buongiorno*. Se invece avessi scritto la query nella forma:

```
1 SELECT Buongiorno;
```

avrei ottenuto un errore da Postgres: il sistema infatti non riesce a trovare una colonna che si chiama *Buongiorno*. I numeri non vanno scritti fra virgolette perchè in questo caso non c'è nessuna ambiguità: le colonne di una tabella non possono avere un numero come nome.

Per concludere la descrizione delle parole, bisogna dire che nel caso in cui io voglia inserire nella mia parola un apostrofo, lo devo scrivere due volta di fila dentro la stringa, ad esempio:

```
1 SELECT 'L''area␣dell''edificio';
```

produce il risultato: *L'area dell'edificio*.

Come per i numeri, anche le parole possono avere le loro espressioni e le loro chiamate di funzione, ad esempio la funzione *LENGTH* calcola la lunghezza in caratteri di una parola, provate:

```
1 SELECT LENGTH('casa');
```

produce il risultato di 4 (la lunghezza in caratteri della parola). Un esempio di operazione fra parole molto utile è la concatenazione di due parole, che si ottiene con l'operatore doppia barra ||, provate ad indovinare quale sia il risultato della query:

```
1 SELECT 'pesce' || 'cane';
```

Attenzione a non confondere il numero 1984 (senza apicetti) dalla parola '1984' (fra apicetti). Nel secondo caso il valore è una parola. La confuzione fra parole e numeri può portare a risultati sorprendenti: provate la query

```
1 \lstset{caption={}}
2 SELECT 99 < 100;
```

(si vuole sapere se 99 è minore di 100) la risposta è *t* (che sta per true = vero), cioè 99 è minore di 100. Provate ora la query

```
1 SELECT '99' < '100';
```

la risposta è *f*=falso, in quanto 99 viene DOPO in ordine alfabetico (o come si dice lessicografico) di 100.

Esistono comunque una serie di funzione per convertire un tipo di dato in un altro, ad esempio *TO_NUMBER* trasforma un qualcosa in un numero, la query seguente dà il risultato atteso (il secondo parametro della funzione *TO_NUMBER* specifica la formattazione del numero):

```
1 SELECT TO_NUMBER('99','00') < TO_NUMBER('100','000');
```

Esistono anche altre centinaia di funzioni che operano sui dati, per effettuare tute le operazioni che servono; sarebbe troppo lungho elencarle in questa sede. Quando serve una certa funzione, basta cercarla nel manuale.

2.7 Tipi di dato

I dati memorizzati nelle tabelle di un database appartengono ad un tipo. Il concetto di tipo di dato è alla base di molti concetti dell'informatica. Quando definite un attributo di una feature class di Arcgis ad esempio, dovete sempre specificare il tipo di dato associato. Quindi i valori con cui si opera nelle basi di dati (e in quasi tutti i linguaggi di programmazione) sono classificati in tipi. Tipi di dato sono: numeri interi, numeri con la virgola, parole (stringhe di caratteri), ore e date, valori di verità (vero o falso), BLOB (dati binari generici). Nei database abilitati ai dati geografici ci sono inotre tipi di dato spaziali.

In Postgres ogni tipo di dato ha un nome ben preciso, che andrà specificato nel comando di creazione di una tabella. I principali tipi di dato sono:

INTEGER : numero intero;

REAL oppure *DOUBLE PRECISION*: numero con la virgola in singola o doppia precisione;

CHARACTER(n) : stringhe di lunghezza n;

CHARACTER VARYING : stringhe di lunghezza variabile;

BOOLEAN : valori di verità (vero o falso)[4];

DATE : data e ora.

Spesso i tipi di dato hanno dei parametri numerici, ad esempio il tipo stringa ha bisogno della definizione del massimo numero di caratteri memorizzabili.

I tipi di dato di base sono moltissimi e non abbiamo il tempo di elencarli, ma non solo: nel corso degli anni i sistemi informatici

[4]curiosamente Oracle non ha questo tipo di dato.

hanno seguito un evoluzione: i tipi di dato di base si sono prima trasformati in tipi complessi (strutture) e poi in *oggetti*. Anche se non è questo il luogo per approfondire l'argomento dovremmo introdurre parzialmente i tipi orientati agli oggetti per poter descrivere la componente spaziale di Postgres: infatti il tipo di dato *GEOMETRY* di Postgres, che difinisce la componente spaziale di un'entità, è un *oggetto* vale a dire che è un tipo di dato complesso.

2.8 Definizione dei Dati

Vediamo per prima cosa la serie di comandi che permette di definire la struttura dei dati (vale a dire la forma delle tabelle). I comandi SQL per definire i dati sono 3:

- CREATE TABLE : crea una tabella;
- DROP TABLE: distrugge una tabella;
- ALTER TABLE: modifica la struttura di una tabella.

2.8.1 Creazione di una tabella

Creiamo adesso la nostra prima tabella: Il comando di creazione di una tabella *CREATE TABLE* ha la seguente struttura generale:

```
1 CREATE TABLE nome_tabella
2 (   nome_colonna1 TIPO1,
3     nome_colonna2 TIPO2,
4     ...
5     nome_colonnan TIPOn
6 );
```

All'interno delle parentesi tonde che seguono il nome della tabella bisogna specificare la lista delle colonne della tabella stessa, separate da virgola. Ricordatevi che la virgola separa, quindi l'ultima colonna non è seguita da virgola! Le colonne sono specificate dal

loro nome e dal nome del tipo (attenzione! Tutti i nomi di colonna e tabella devono essere un'unica parola senza spazi: al massimo si può usare la barra di sottolineato _ . Non c'è differenza fra maiuscole e minuscole. Nei nomi si possono usare lettere, cifre e la barra sopra detta, anche se il nome non può iniziare con una cifra. Ad esempio *pippo, codice_corso, pluto42* sono nomi corretti, *12pippo, codice corso, pippo$* sono nomi errati). Provate adesso a creare la nostra prima tabella, con il comando:

```
1 CREATE TABLE strada
2 (
3     nome CHARACTER VARYING,
4     classifica CHARACTER(2),
5     larghezza REAL
6 );
```

Questo comando creerà la tabella *strada*, formata da tre colonne: il nome dello strada (parola a lunghezza viariabile), il codice della sua classificazione (parola di due caratteri) e la larghezza media in metri (numero con la virgola). Notate le 2 virgole che separano le 3 colonne e il fatto che i nomi di colonna non possano contenere spazi ne tanto meno lettere accentate.

Nota: si ricorda che la formattazione (i ritorni a capo e gli spazi) non conta nulla. Scriveremo i comandi in una certo modo solo per renderli più chiari. Potevamo scrivere anche (in modo molto meno chiaro):

```
1 CREATE TABLE strada (nome CHARACTER VARYING, ...
```

2.8.2 Analisi di una tabella

Una volta che abbiamo creato una tabella possiamo analizzare la sua struttura tramite l'interfaccia grafica. Torniamo alla finestra principale, aggiorniamo l'elenco delle tabelle e selezioniamo la tabella *strada*; l'interfaccia ad albero ci mostra gli elementi contenuti nella struttura della tabella (ma non i dati). A destra, dentro *SQL*

Pane è possibile visualizzare il comando SQL che ha generato la tabella stessa (figura 2.5). L'analisi di struttura delle tabelle può

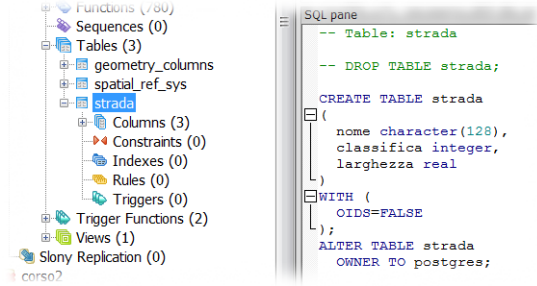


Figura 2.5: Visualizzazione della struttura di una tabella.

essere utile per studiare le tabelle non create da noi ma dal sistema stesso: ad esempio le tabelle *geometry_columns* e *spatial_ref_sys* che fanno parte del sistema geografico.

2.8.3 Distruzione di una tabella

Per distruggere definitivamente una tabella, si utilizza il comando *DROP TABLE*. Adesso l'esercitazione prevede la distruzione della tabella appena creata (la rifaremo meglio dopo), prima di tutto però copiate il testo della query di creazione e incollatelo nello *Scratch Pad*, in modo da poterlo riutilizzare in seguito. Quindi provate adesso a distruggere la tabella *strada*, con il comando:

```
1 DROP TABLE strada;
```

Il comando distrugge per sempre la tabella (attenzione ad usarlo con cognizione di causa), non c'è l'annulla.

Abbiamo usato il verbo italiano (piuttosto desueto) *distruggere* e non *cancellare* per non confondere le due operazioni: diremo cancellare (in inglese DELETE) nel caso in cui vogliamo cancellare i dati di una tabella senza distruggerne la struttura. Mentre

diremo distruggere (in inglese DROP) per eliminare una tabella completamente. Attenzione! Non c'è modo di recuperare una tabella distrutta, a meno che non si sia iniziata una *transazione* (di cui non parleremo in questo capitolo).

2.8.4 Un inciso: commenti al codice

Come in tutti i linguaggi per computer, in SQL è possibile inserire un testo di commento che viene ignorato dal database. La sintassi per inserire commenti nei comandi SQL è di due tipi: commenti a fine riga; tutto quello che segue il simbolo - - (due simboli meno consecutivi) viene ignorato. Commenti multi riga (derivati dal linguaggio C/C++): tutto quello che è compreso fra i simboli /* e */. Ad esempio la creazione della nostra tabella può essere scritta nel seguente modo:

```
1 /* creazione della tabella
2    contenente le strade */
3 CREATE TABLE strada
4 (
5     nome CHARACTER VARYING, -- nome della strada
6     classifica CHARACTER(2), -- autostrada, urbana, ...
7     larghezza REAL  -- media in metri
8 );
```

A cosa servono i commenti? Servono per inserire note e spiegazioni al codice SQL, in modo tale le la documentazione sia compresa nel codice stesso ad uso dell'uomo e non della macchina.

2.8.5 Creazione avanzata di una tabella

Le colonne di una tabella possono avere molte specifiche aggiuntive, oltre il nome ed il tipo di ogni colonna. ne vediamo due:

- chiave primaria;
- specifica di campo obbligatorio.

Ricordiamo che la colonna *chiave primaria* specifica il dato (o i dati) che identificano univocamente ogni riga della tabella, mentre un campo è obbligatorio se il suo valore è definito non nullo. Di norma invece le caselle di una tabella possono essere anche vuote (avere valore nullo). Il nuovo comando di creazione della tabella studenti rivisto è il seguente:

```
1 CREATE TABLE strada
2 (
3     nome CHARACTER VARYING PRIMARY KEY,
4     classifica CHARACTER(2) NOT NULL,
5     larghezza REAL
6 );
```

La specifica *PRIMARY KEY* indica che il campo *nome* è quello che identifica univocamente le righe della tabella. La specifica *NOT NULL* indica che il campo *classifica* è obbligatorio e non può rimanere vuoto durante l'inserimento delle righe della tabella (vale a dire che non può assumere il valore speciale NULL). Provate adesso ad eseguire la nuova query di creazione della tabella (se vi siete dimenticati di distruggerla, il nuovo comando di creazione vi segnalerà un errore). Provate anche a visualizzare la tabella nell'albero grafico; vedrete che adesso vengono riportate tutte le informazioni riguardati i campi, compresa la presenza della chiave primaria e dei campi obbligatori.

2.8.6 Modifica della struttura di una tabella

La struttura delle tabelle può essere modificata dinamicamente. Ad esempio possiamo aggiungere o togliere colonne, oppure modificare le specifiche dei campi (chiavi primarie, campi obbligatori) senza dover distruggere o ricreare la tabella. Una volta che una tabella è stata creata, le modifiche dinamiche alla sua struttura sono possibili tramite il comando *ALTER TABLE*: aggiungiamo la colonna *num_corsie* alla nostra tabella:

```
1 ALTER TABLE strada
2 ADD num_corsie INTEGER;
```

Di solito i comandi SQL sono molto chiari ed auto-esplicativi: questo comando modifica la tabella *strada* aggiungendo il campo *num_corsie*, che è di tipo numero intero. Provate ad eseguire il comando e poi a visualizzare la tabella per controllare l'effetivo cambio di struttura.

Nello stesso modo è possibile cancellare colonne o modificarne il tipo e i vincoli. Le modifiche di struttura ad una tabella possono essere eseguite anche se la tabella contiene già dei dati: le colonne ed i relativi dati non interessati dalle modifiche di struttura verranno conservati.

2.9 Manipolazione dei dati

Abbiamo imparato a creare, distruggere e modificare le nostre tabelle. Adesso vediamo come si manipolano i dati. I principali comandi di manipolazione dei dati sono 3:

- INSERT : inserisce nuove righe in una tabella (quindi inserisce nuovi dati);
- DELETE: cancella righe da una tabella;
- UPDATE: modifica i dati esistenti di una tabella.

2.9.1 Inserimento di dati

Il comando *INSERT* server per inserire valori in una tabella (vale a dire nuove righe). La struttura del comando *INSERT* è la seguente:

```
1 INSERT INTO nome_tabella
2  (nome_colonna1, nome_colonna2, ... , nome_colonnaN)
3  VALUES (valore1, valore2, ... , valoreN);
```

Per inserire righe in una tabella bisogna quindi specificare la tabella, l'elenco dei nomi delle colonne che vogliamo inserire, quindi l'elenco corrispondente dei valori.

Proviamo adesso ad inserire alcune righe nella nostra tabella *strada*; per ogniuna delle strade da inserire bisogna specificare, nome, classica, larghezza e numero di corsie:

```
1 INSERT INTO strada
2 (nome,classifica,larghezza,num_corsie)
3 VALUES
4 ('A1','01',16,4);
```

Notate che il nome (*A1*) e la classifica (*01*)[5] sono parole, quindi vanno fra apici, mentre la larghezza (*16*) ed il numero di corsie (*4*) sono numeri, quindi sono senza apicetti. I termini *nome,classifica,larghezza,...* sono i nomi delle colonne così come *strada* è il nome della tabella: quindi vanno scritti anche loro senza apicetti. Notate anche le virgole, che separano colonne e valori: ovviamente dopo l'ultimo valore (il numero 4) la virgola non ci vuole. Niente panico: la sintassi à una brutta bestia, che si doma con l'esperienza.

Proviamo adesso ad inserire altri valori nella tabella (potete anche provare ad inserire dati di fantasia, basta che la classifica sia del tipo *'01','02','03',...*). In particolare proviamo ad inserire un dato incompleto:

```
1 INSERT INTO strada
2 (nome,classifica,num_corsie)
3 VALUES
4 ('Aurelia','02',4);
```

Notate che in questo caso non abbiamo inserito la *larghezza* della strada, omettendola sia nell'elenco delle colonne che nei valori. Questo attributo comunque non è obbligatorio (non possiede l'opzione *NOT NULL*; il campo *classifica* invece è obbligatorio e va sempre specificato.

[5]Ebbene si, la classifica sembra un numero, ma è una parola di due lettere!

Nel caso in cui si inseriscano valori per tutte le colonne, la sintassi del comando *INSERT* può essere semplificata omettendo la lista dei campi da inserire e specificando solo i valori, nell'ordine con cui devono essere inseriti, ad esempio possiamo eseguire:

```
1 INSERT INTO strada
2 VALUES
3 ('A23','01',12,4);
```

Dove i valori *'A23','01',12,4* sono nell'ordine il contenuto delle colonne da inserire.

2.9.2 Un'altro inciso: il valore NULL

Abbiamo visto che il campo età non è obbligatorio. Quando un dato di una riga non è inserito, la relativa casella nella tabella è vuota. Il valore *vuoto* ha in SQL un nome: *NULL*. Ad esempio potevamo scrivere il comando di inserimento parziale nel seguente modo:

```
1 INSERT INTO strada
2 VALUES
3 ('Emilia','02',NULL,4);
```

Intendendo che il campo *larghezza* (il terzo valore) deve rimanere nullo e quindi vuoto. Il termine *NULL* sarà particolarmente utile nei controlli, che vedremo in seguito.

2.9.3 Test dei vincoli

Nella tabella che abbiamo costruito ci sono vari vincoli, come ad esempio la chiave primaria e l'obbligatorietà del campo *classifca*. Se proviamo ad inserire una nuova riga con un nome di strada duplicato, violiamo il vincolo di chiave primaria ed il database ci comunicherà l'errore; proviamo ad eseguire il comando:

```
1 INSERT INTO strada
```

```
(nome,classifica,larghezza,num_corsie)
VALUES
('A1','03',12,2);
```

Otteniamo un errore del tipo (scritto in inglese): *una chiave duplicata viola il vincolo di unicità*. La chiave primaria infatti deve essere unica per ogni valore in tabella, mentre noi abbiamo tentato di inserire due strade diverse con lo stesso nome (*A1*).

Ricordiamo che il campo *larghezza* non è obbligatorio, mentre è obbligatorio il campo *classifica* (vale a dire che possiede l'opzione *NOT NULL*). Proviamo ad eseguire il seguente comando, per inserire una strada di cui non conosciamo la classifica:

```
INSERT INTO strada
(nome,larghezza,num_corsie)
VALUES
('campestre',12,4);
```

Otteniamo un errore del tipo (in inglese): *un valore nullo nella colonna classifica viola il vincolo not-null*. Inserire dei controlli nelle tabelle è molto importante per controllare a monte la correttezza e la completezza dei dati.

2.9.4 Cancellazione di dati

Il comando *DELETE* permette di cancellare righe da una tabella. La sua forma più semplice sarebbe (**MA NON ESEGUITELO!**):

```
DELETE FROM strada;
```

il comando sopra citato cancella TUTTE le righe della tabella *strada* (ma non distrugge la tabella stessa) senza possibilità di recupero (a meno che non utilizziate le transazioni). La forma del comando *DELETE* che invece di solito si utilizza è la seguente:

```
DELETE FROM strada
WHERE {condizioni};
```

dove le *condizioni* specificate dopo il termine *WHERE* filtrano e selezionano le righe da cancellare. La specifica di una condizione permette di eliminare solo quelle righe che rispettano la condizione specificata, ad esempio se vogliamo eliminare dalla tabella le strade che più larghe di 20 metri scriviamo:

```
1 DELETE FROM strada
2 WHERE larghezza>20;
```

Il comando cancellerà (se ci sono ma non credo) tutte strede che hanno il valore dell'attributo *larghezza* maggiore di 20. Nella condizione è possibile scrivere espressioni aritmetiche, invocare funzioni, controllare le colonne, eseguire confronti di uguaglianza ($a = b$), diversità ($a <> b$), confronti di quantità ($a < b, a <= b, a >= bea > b$), ed usare i connettivi logici AND, OR, NOT (che stanno per e, o, non). Una descrizione accurata di tutte le forme di controllo esula dagli scopi di questo corso, facciamo solo alcuni semplici esemi esempi, il filtro:

```
1 ...
2 WHERE larghezza<10 AND num_corsie=2
```

identifica tutti le strade larghe meno di 20 metri *E* con 2 corsie. Il filtro:

```
1 ...
2 WHERE nome='A1' OR NOT classifica='01'
```

identifica la strada *A1* oppure tutte le altre strade che non sono di tipo autostradale (*'01'*). Per le parole si possono usare i confronti di uguaglianza, ma anche il minore ed il maggiore, intesi come ordine alfabetico (es. 'abaco' < 'zuzzurellone'). L'operatore *LIKE* invece permette di eseguire confronti fra parole facendo utilizzo di caratteri jolly, il filtro:

```
1 ...
2 WHERE nome LIKE 'Em%';
```

identifica tutti le strade il cui nome inizia per *Em*: il simbolo % sta ad indicare qualsiasi sequenza di lettere.

2.9.5 Modifica dei dati

I dati presistenti di una tabella si modificano con il comando *UPDATE*. La struttura generale del comando *UPDATE* è:

```
1 UPDATE nome_tabella
2 SET nome_colonna1 = valore1,
3     nome_colonna2 = valore2,
4     ...
5 WHERE {condizioni};
```

dove la definizione delle condizioni è del tutto uguale a quella del comando *DELETE*. Proviamo adesso a cambiare il numero di corsie di qualche strada, eseguiamo la query:

```
1 UPDATE strada
2 SET num_corsie = 2
3 WHERE nome='A23';
```

; Il valore della colonna specificata viene cambiato per tutte le righe che rispettano la condizione impostata. In questo caso quindi alla riga che contiene lo strada *A23*, verrà cambiato il numero di corsie in 2. Se non si specifica la condizione, il comando *UPDATE* modifica TUTTE le righe della tabella, assegnando un valore costante a tutta colonna indicata.

La dicitura *IS NULL* può essere utilizzato nei controlli per determinare la presenza di valori nulli; se vogliamo ad esempio impostare una valore di default uguale a 8 per ogni strada in cui non abbiamo specificato la larghezza, scriviamo:

```
1 UPDATE strada
2 SET larghezza = 8
3 WHERE larghezza IS NULL;
```

In questo modo, tutte le caselle *larghezz* vuote (con valore NULL) vengono riempite con il valore 8.

2.10 Interludio: una Seconda Tabella e le Relazioni

Fino ad adesso abbiamo operato su di una sola tabella, ma ovviamente le basi di dati possono contenere molte tabelle.

2.10.1 Una Seconda Tabella

Prima di passare all'interrogazione dei dati, per rendere più interessante il nostro database, creiamo una tabella *clas_stradale*, che ci servirà per fare degli esempi di interconnessione fra tabelle, eseguiamo la query:

```
1 CREATE TABLE clas_stradale
2 (
3   codice CHARACTER(2) PRIMARY KEY,
4   descrizione CHARACTER VARYING NOT NULL
5 );
```

Ormai siamo esperti: il codice della classifica è la sua chiave primaria (essere chiave primaria implica anche obbligatorio), segue una descrizione testuale obbligatoria di lunghezza variabile.

Riempiamo adesso la tabella, inserendo i relativi dati. Potete eseguire il codice seguente in un sol colpo, dato che è possibile eseguire più di un comando SQL alla volta, separando i singoli comandi con un punto e virgola[6]:

```
1 INSERT INTO clas_stradale
2         VALUES('01','autostrada');
3 INSERT INTO clas_stradale
4         VALUES('02','extraurbana␣principale');
5 INSERT INTO clas_stradale
6         VALUES('03','extraurbana␣secondaria');
7 INSERT INTO clas_stradale
8         VALUES('04','urbana␣di␣scorrimento');
```

[6] Questi valori non sono a caso, rispecchiano le specifiche ufficiali dei database geografici italiani.

```
9  INSERT INTO clas_stradale
10         VALUES('05','urbana di quartiere');
11 INSERT INTO clas_stradale
12         VALUES('06','strada locale/vicinale');
```

2.10.2 Le relazioni

Un database non è fatto solo di entità (tabelle) ma anche di relazioni. Le relazioni sono importanti tanto quanto lo sono i dati. Due oggetti sono in relazione se esiste un dato che li mette in collegamento. Le strade sono in relazione con la tabella classifica, dato che per ogni strada abbiamo specificato un codice di classifica.

Vedremo adesso che le relazioni possono anche essere specificate esplicitamente con l'aggiunta di un vincolo (constraint) alla tabella.

La tabella *strada* contiene per ora una relazione logica con la tabella *clas_stradale*: infatti il campo *classifica* della prima tabella si riferisce al campo *codice* della seconda tabella. Questa relazione *sottointesa* fra tabelle può essere esplicitata tramite il seguente comando:

```
1 ALTER TABLE strada
2 ADD CONSTRAINT strada_classifica_fk
3 FOREIGN KEY (classifica)
4         REFERENCES clas_stradale(codice);
```

Il comando esplicita la relazione fra strade e classifiche, ed è costituito da un vincolo sulla tabella *strada*. Analizziamo la struttura del comando: la prima riga indica la volontà di modificare la tabella *strada* (come nel caso di aggiunta di una nuova colonna), in questo caso però vogliamo aggiungere un vincolo (*constraint* in inglese): *strada_classifica_fk* è il nome di questo nuovo vincolo (*fk* sta per foreign key = chiave straniera e si aggiunge per convenzione, in realtà potevamo scegliere come nome anche *pippo*). Il vincolo afferma (nell'ultima riga del comando) che la *chiave esterna* formata dalla colonna *classifica* della tabella *strada* DEVE

riferire una valore (vale a dire contenere un numero di codice) della colonna *codice* contenuta nella tabella *clas_stradale*.

Se nella tabella *clas_stradale* non ci sono tutti i codici necessari, la creazione della relazione sarà impossibile, dato che il sistema controlla la congruenza dei dati anche durante la creazione del vincolo. Una volta che il vincolo di relazione è impostato, siamo sicuri che tutti i codici di classifica stradale siano corretti e presenti nella tabella delle classifiche.

Ai lettori più attenti può dar fastidio che la relazione sia rappresentata da un vincolo sulla sola tabella *strada*. Perchè non c'è un vincolo anche nell'altra tabella? Perchè questo tipo di relazione è asimettrica, con cardinalità $1 : n$. E' la strada che appartiene ad una particolare classifica, mentre per ogni classifica ci possono essere innumerevoli strade.

Tentiamo adesso di inserire una strada con classifica stradale inesistente, eseguiamo:

```
1 INSERT INTO strada
2 VALUES
3 ('Canistracci','99',16,4);
```

La classifica con codice *'99'* non esiste: otteniamo un errore che ci informa (in inglese) della violazione del vincolo di integrità che si chiama *strada_classifica_fk*. La congruenza delle relazioni viene controllata dinamicamente in ogni momento, in particolare durante la modifica o all'inserimento dei dati nelle tabelle *strada* e *clas_stradale*. Ad esempio non è più possibile cancellare una classifica stradale se esiste almeno una strada con quella classifica: provate ad inventare una query che prova questo vincolo[7].

[7]DELETE FROM clas_stradale WHERE codice='01';

2.11 Indici

Accenniamo adesso alla gestione degli indici. Una descrizione dettagliata degli indici esula però dagli scopi di questo corso.

Supponiamo di prevedere molte ricerche sulle larghezze delle strade; inoltre supponiamo che le strade della nostra tabella siano tante. Normalmente il sistema deve scorrere l'intera tabella delle strade per eseguire tale ricerca: se le strade sono tante questa ricerca può richiedere del tempo. Per velocizzare una ricerca del genere è possibile creare un *indice*. Gli indici servono per velocizzare le ricerche di valori su una (o più) colonne di una tabella; il loro funzionamento è simile agli indici (o megli agli indici analitici) dei libri. Per creare un indice sulla colonna *larghezza* della tabella *strada*, eseguiamo il semplice comando:

```
1 CREATE INDEX strada_larghezza_idx ON strada(larghezza);
```

Al solito, *strada_larghezza_idx* è il nome dell'indice (idx sta per index), mentre la dicitura *strada(larghezza)* indica che l'indice va creato nella tabella *strada* ed in particolare sulla colonna *larghezza*.

Apparentemente la presenza di un indice non cambia il funzionamento del database: il risultato delle interrogazioni è lo stesso. Quello che cambia è la velocità di funzionamento. In realtà vedremo che nel caso di dati spaziali, l'indice è fondamentale per la ricerca veloce dei dati. Gli indici non vengono mai creati automaticamente (eccetto che per gli indici sulle colonne chiave primaria, che vengono create implicitamente, come ci avverte il messaggio di Postgres): devono essere progettati con cura da chi crea la struttura del database, in funzione del tipo di ricerche da effettuare e dal tipo (e dalla quantità) dei dati presenti: la presenza di un indice su di una colonna velocizza sempre le operazioni di ricerca, mentre ne può rallentare leggermente le operazioni di modifica (dato che in questo caso è necessario aggiornare anche l'indice). Inoltre la creazione di un indice richiede un utilizzo aggiuntivo di spazio disco.

2.12 Le interrogazioni: SELECT

Siamo arrivati (ovvero ritornati) finalmente alla parte finale di SQL: l'interrogazioni dei dati. Sebbene le interrogazioni siano eseguito dall'unico comando *SELECT*, questo è il comando più complesso. Le forme del comando SELECT sono moltissime, quindi ne vedremo alcuni brevissimi esempi. Inoltre l'apparente semplicità di tale comando nasconde la notevole difficoltà di tradurre la richiesta che abbiamo in mente nella dicitura SQL. Creare il comando *SELECT* che ci interessa richiede una notevalo dose di esperienza.

2.12.1 Forma semplice di SELECT

La forma più semplice di *SELECT* è la seguente:

```
1 SELECT colonna1,colonna2,...,colonnaN
2 FROM tabella
3 WHERE {condizioni}
4 ORDER BY colonna1,colonna2;
```

Nella forma semplice di SELECT bisogna specificare: l'elenco delle colonne da visualizzare, la tabella sorgente, eventuali condizioni analoghe a quelle dei comandi UPDATE e DELETE, ed un eventuale ordine[8].

Invece di scrivere un elenco di colonne è possibile scrivere il simbolo * (asterisco) che indica tutte le colonne della tabella.

La condizione *WHERE* e l'ordine *ORDER BY* possono essere anche omessi: ad esempio per visualizzare un'intera tabella *strade* possiamo scrivere il comando:

```
1 SELECT *
2 FROM strada;
```

Si ottiene il seguente risultato:

[8] L'ordine non è mai definito per default! Questo è un concetto fondamentale delle basi di dati, l'ordine delle righe è indefinito, ad esempio non centra niente con l'ordine di inserimento.

```
  nome   | classifica | larghezza | num_corsie
---------+------------+-----------+------------
 A23     | 01         |        12 |          2
 Aurelia | 02         |         8 |          4
 Emilia  | 02         |         8 |          4
 A1      | 01         |        16 |          4
```

Oppure per ordinare il risultato in ordine di nome e selezionare solo nome e larghezza possiamo scrivere:

```
1 SELECT nome, larghezza
2 FROM strada
3 ORDER BY nome;
```

Ottenendo il risultato:

```
  nome   | larghezza
---------+-----------
 A1      |        16
 A23     |        12
 Aurelia |         8
 Emilia  |         8
```

Si ricorda che * sta per tutte le colonne; inoltre, non essendoci filtro, vengono estratte tutte le righe della tabella indicata. Nel secondo caso le strade sono ordinate per nome (ordine alfabetico), mentre nel primo caso l'ordine è casuale.

Se vogliamo vedere un sottoinsieme delle righe di una tabella, possiamo specificare una condizione di filtro, in modo del tutto analogo ai comandi *UPDATE* e *DELETE*:

```
1 SELECT nome, classifica
2 FROM   strada
3 WHERE  larghezza<=12
4 AND    num_corsie=4;
```

che in italiano si legge: selezionare il nome e la classifica dalla (tabella) strade dove la larghezza è minore o uguale a 12 (metri) e il numero di corsie è uguale a 2. Il risultato è qualcosa del tipo:

```
  nome   | classifica
---------+------------
 Aurelia | 02
 Emilia  | 02
```

2.12.2 Aggregazioni di righe

Un secondo tipo di *SELECT*, è quella del tipo *aggregante*, in cui più linee di una tabella possono essere aggregate insieme, da particolari funzioni di aggregazione, lo schema della query diventa:

```
1 SELECT {funzioni_aggreganti}(attributi da aggregare)
2 FROM tabella
3 WHERE {condizione sulle righe}
4 GROUP BY {colonne che discriminano l'aggretazione}
5 HAVING (condizione sul risultato aggregregato)
```

Niente paura, è più complicato da dire che da fare. In generale l'aggregazione produce un dato totale, che riguarda l'intera tabella o insiemi di righe raggruppate (sub-totali). Quali sono le funzioni di aggregazione? Le principali funzioni di aggregazione sono:

- MIN: minimo dei valori;
- MAX: massimo dei valori;
- AVG: media dei valori (*average* in inglese);
- SUM: somma dei valori;
- COUNT: numero di valori;

Vedremo nei prossimi capitoli che ci sono anche funzioni di aggregazione spaziale (es. l'area unione di insiemi di oggetti).

Supponiamo ad esempio di voler sapere la larghezza minima, media e massima delle strade presenti nella nostra tabella, possiamo eseguire:

```
1 SELECT MIN(larghezza), MAX(larghezza), AVG(larghezza)
2 FROM strada;
```

Il risultato saranno i valori minimo, massimo e medio di tutte le larghezze della tabella strada:

```
 min | max | avg
-----+-----+-----
   8 |  16 |  11
```

Si noti che il risultato in questo caso è una sola riga: tutte le righe della tabella *strada* sono state aggregate in una sola. Le funzioni di aggregazione (come le funzioni matematiche) hanno bisogno della specifica dei parametri (nel nostro caso la colonna *larghezza*) su cui operare, i quali vanno specificati fra parentesi tonde.

Vediamo adesso come si possono raggruppare le aggregazioni di valori, ottenendo dei sotto-totali. Vogliamo sapere la larghezza minima e massima ed il numero di strade, suddivise però per classifica. Vogliamo in altre parole *raggruppare* i risultati secondo la colonna *classifica*:

```
1 SELECT classifica,
2        MIN(larghezza),
3        MAX(larghezza),
4        COUNT(larghezza)
5 FROM strada
6 GROUP BY classifica;
```

La query è simile alla precedente: in questo caso però le righe non sono aggregate tutte insieme, ma secondo il codice del corso. Questo raggruppamento è dovuto all'aggiunta della riga *GROUP BY classifica*:

```
 classifica | min | max | count
------------+-----+-----+-------
 02         |   8 |   8 |     2
 01         |  12 |  16 |     2
```

In questo caso il risultato è formato da più righe: una per ogni codice di classifica presente: per ogni valore viene calcolata la minima e massima larghezza, nonchè il numero di strade presenti.

La funzione *COUNT* è particolare, sebbene richieda un parametro in realtà non viene applicata a nessuna colonna in particolare, dato che deve semplicemente contare le righe; per questo di solito si inserisce come parametro di *COUNT* il simbolo $*$ (tutte le colonne). Per sapere quante righe contiene una tabella basta scrivere:

```
1 SELECT COUNT(*)
2 FROM clas_stradale;
```

2.12.3 Join

Nel nostro database abbiamo due tabelle: strade e classifiche. Inoltre le due tabelle sono collegate da una relazione esplicita. Vogliamo adesso visualizzare il nome di ogni strada con associata la descrizione della classifica (e non il suo codice). Per fare questo è necessario utilizzare la relazione che intercorre fra le due tabelle: il termine tecnico di questa operazione è JOIN (unificazione). L'esecuzione di una SELECT con JOIN implica l'utilizzo di più tabelle conteporaneamente, quindi la clausola *FROM* della nostra query avrà una forma del tipo

```
1 ...
2 FROM strada, clas_stradale
3 ...
```

L'utilizzo di più tabelle in una query comporta alcune complicazioni. Ad esempio più tabelle potrebbero contenere colonne con nomi uguali, per cui la specifica del nome di colonna diventerebbe ambigua.

Per togliere ogni ambiguità è possibile specificare il nome di colonna insieme a quello della tabella che la contiene: i due nomi devono essere serapati da un punto. Ad esempio è corretto scrivere:

```
1 SELECT strada.nome
2 FROM   strada
3 WHERE  strada.larghezza<20;
```

Per realizzare una *JOIN* dobbiamo specificare quale sia la regola di unificazione delle due tabelle: nel nostro caso la regole di unificazione è che il codice di classifica di una strada deve essere uguale al codice della tabella delle classifiche: la query è la seguente (l'ultima riga contiene il vincolo di unificazione):

```
1 SELECT strada.nome,
2        clas_stradale.descrizione
3 FROM   strada,
4        clas_stradale
5 WHERE  strada.classifica = clas_stradale.codice;
```

Il risultato è il seguente:

```
  nome    |      descrizione
----------+------------------------
 A23      | autostrada
 Aurelia  | extraurbana principale
 Emilia   | extraurbana principale
 A1       | autostrada
```

Ci sono alcuni particolari da notare: per prima cosa in questa query facciamo utilizzo di DUE tabelle: dopo FROM infatti possiamo utilizzare quante tabelle vogliamo enumerandole e separando i nomi con virgole. In secondo luogo vediamo che le colonne dopo la *SELECT* sono specificate nella forma *NOME_TABELLA. NOME_COLONNA*: questa specifica è necessaria in presenza di più tabelle per chiarire da quale tabella si pesca la colonna. Infine analizziamo la clausola *WHERE*: in questo caso la clausola non ha una funzione di filtro sul risultato, serve invece per esplicitare la relazione presente fra le tabelle. Senza la clausola *WHERE* non ci fosse (provate a cancellarla ed eseguire la query), il database esegue quello che si chiama *prodotto cartesiano* dei valori, vale a

dire produce tutte le combinazioni possibili fra strade e classifiche, senza nessun nesso fra le coppie. La clausola *WHERE* invece, fra tutte le combinazioni, seleziona solo quelle in relazione.

Le join fra tabelle sono molto importanti nel campo spaziale: vedremo che lo stesso meccanismo può essere utilizzato per creare relazioni spaziali, che includono cioè vincoli di posizione relativa.

La join è un'operazione molto importante e complessa; per questo nasconde alcune difficoltà che non verranno trattate in questa sede. In particolare la gestione dei campi vuoti o che non hanno corrispondenza nella join deve essere gestita specificando il tipo di comportamento da tenere: si prendono solo le coppie con oggetti associati (*inner join*), tutte le coppie (*outer join*), tutti gli elementi della prima tabella con gli eventuali elementi della seconda (*left outer join*), etc.

2.13 Viste

E' possibile che una interrogazione (*SELECT*) interessante (come la precedente join) ci serva più volte. Oltre al meccanismo di salvataggio delle query presente nell'interfaccia di Postgres (File-save), è possibile dare un nome ad una query importante, ed in questo modo salvarla permanentemente nella base di dati. Le query salvate con nome prendono il nome di *viste* (view in inglese). E' possibile salvare le query come viste, aggiungendo in testa al codice della query il comando *CREATE VIEW nome_vista AS*, provate ad esempio ad eseguire:

```
1 CREATE VIEW cstrade AS
2 SELECT strada.nome,
3        clas_stradale.descrizione
4 FROM   strada,
5        clas_stradale
6 WHERE  strada.classifica = clas_stradale.codice;
```

Dalla seconda riga in poi la query è identica a quella della sezione precedente. In questo caso però la query non viene eseguita: invece le viene dato il nome *cstrade* e salvata nella base di dati come *vista*. Le viste in pratica sono query con nome: una volta create si utilizzano come se fossero tabelle. Provate adesso ad eseguire:

```
1 SELECT * FROM cstrade;
```

I dati delle viste variano al variare delle tabelle sottostanti (*strada* e *clas_stradale*), vale a dire che il risultato della query non è salvato al momento della creazione della vista, ma varia al variare delle tabelle originali. Se adesso cambiassimo i valori contenuti nella tabella *strada*, cambierebbe di conseguenza il contenuto della vista.

Ripetiamo che le viste si usano esattamente come se fossero tabelle: è quindi possibile aggiungere filtri, ordinamenti, etc. alla SELECT su viste. Provate:

```
1 SELECT *
2 FROM cstrade
3 WHERE nome LIKE 'A%';
```

Il risultato è:

```
  nome   |      descrizione
---------+------------------------
 A23     | autostrada
 Aurelia | extraurbana principale
 A1      | autostrada
```

2.14 Creazione di Dati da Interrogazioni

Il risultato di una interrogazione può essere utilizzato per creare una tabella *al volo*; basta far precedere il comando *SELECT* dal comando *CREATE TABLE nome_tabella AS*:

```
1 CREATE TABLE cstrade_table AS
2 SELECT strada.nome,
3        clas_stradale.descrizione
```

```
4 FROM   strada,
5        clas_stradale
6 WHERE  strada.classifica = clas_stradale.codice;
```

Il listato è simile a quello della creazione di una vista; si ricorda però che in questo caso i dati vengono effettivamente memorizzati nella nuova tabella creata. Se il comando *SELECT* contiene chiamate di funzione o espressioni complesse è utile specificare il nome della nuova colonna attraverso la dicitura *AS*. Ad esempio:

```
1 CREATE TABLE lunghezze AS
2 SELECT LENGTH(strada.nome) AS lunghezza
3 FROM   strada
4 WHERE  strada.larghezza;
```

Se non avessi specificato il nome *AS lunghezza*, il database avrebbe scelto autonomamente il nome della colonna.

Nel caso invece che si voglia inserire i dati calcolati da un interrogazione in una tabella pre-esistente, basta far precedere il comando *SELECT* da un comando *INSERT INTO*:

```
1  CREATE TABLE stat_strade
2  (
3    classifica character(2) NOT NULL,
4    numero INTEGER
5  );
6
7  INSERT INTO stat_strade
8  SELECT classifica, COUNT(*)
9  FROM   strada
10 GROUP  BY classifica;
```

Una volta creata la tabella *stat_strade* possiamo inserirci i dati provenienti dall'interrogazione, facendola precedere dal comando *INSERT INTO stat_strade*. Ovviamente il numero ed il tipo delle colonne prodotte dall'interrogazione deve coincidere con il numero ed il tipo di quelle presenti nella tabella da riempire.

2.15 Schemi

Facciamo adesso qualche esempio di utilizzo di schemi. Supponiamo di voler duplicare la nostra struttura di strade, che però contenga i dati relativi ad un altro foglio cartografico. In assenza di specifica, le tabelle vengono create nello schema di default che in PostgreSQL si chiama *public*. Creiamo adesso un nuovo schema di nome *foglio2*:

```
1 CREATE SCHEMA foglio2;
```

Gli schemi si comportano come le cartelle per i file: possono contenere tabelle (ma anche relazioni, vincoli, indici, etc.). Per creare una nuova tabella *strada* all'interno dello schema appena creato, bisogna far precedere il nome della tabella dal nome dello schema, separato da un punto:

```
1 CREATE TABLE foglio2.strada
2 (
3     nome CHARACTER VARYING PRIMARY KEY,
4     classifica CHARACTER(2) NOT NULL,
5     larghezza REAL
6 );
```

La creazione di questa nuova tabella è possibile perchè verrà inserita nello schema *foglio2*. Per utilizzare le tabelle all'interno di uno schema bisogna specificare il nome completo, ad esempio:

```
1 SELECT * FROM foglio2.strada;
```

estrarrà i dati (nessuno) della nuova tabella appena creata, mentre:

```
1 SELECT * FROM strada;
```

oppure

```
1 SELECT * FROM public.strada;
```

estrarrà i dati dalla tabella *strada* degli esempi precedenti (lo schema public può essere sottointeso).

2.16 Editor grafici di query

SQL è un linguaggio molto elegante, spesso le interrogazioni sono chiare ed autoesplicative. Il modo migliore per progettare una query è quello di scriverla in modo testutale; tuttavia molti sistemi prevedono la possibilità di costruire un'interrogazione (specialmente la parte che riguarda i filtri). In figura 2.6 potete vedere alcuni esempi di interfacce grafiche.

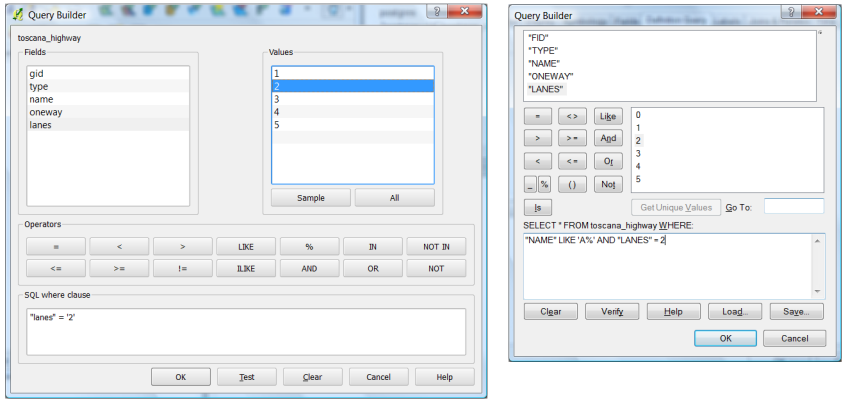


Figura 2.6: Interfacce grafiche per SQL: a sinistra quella di QuantumGIS, a destra quella di ArcGIS 10.

2.17 Conclusioni

Questa è solo una brevissima introduzione a *SQL*. Le basi di dati reali sono formate da centinaia, se non migliaia di tabelle e relazioni. La struttura del comando SELECT ha molte altre possibilità, che richiederebbero almeno molto tempo per essere spiegate. Citiamo a titolo di esempio la possibilità di utilizzare sotto-query, di creare dati o inserire righe a partire da una select. Il linguaggio SQL è di

per sè molto semplice, ma la creazione di un comando SELECT non banale, richiede, in alcuni casi, una certa esperienza.

3 Introduzione ai Dati Vettoriali

Se avete già un idea di cosa sia un dato geografico vettoriale, saltate al prossimo capitolo. Introduciamo i concetti base riguardanti i dati geografici in formato vettoriale; questi sono quei dati rappresentati principalmente da elementri geometrici (punti, linee, aree, etc.), definiti da una serie di coordinate e con alcuni attributi associati.

I dati geografici di tipo *Raster* (foto aeree o satellitari, immagini di cartografia, modelli digitali del terreno, etc.) sono concettualmente più semplici dei dati vettoriali; sono infatti più economici da acquisire e più facili da gestire, anche se di solito richiedono un grande spazio di memoria.

I dati vettoriali invece hanno una struttura molto compatta, ma le procedure di manipolazione risultano più complesse e sono costosi da produrre.

La tipologia del dato dipende soprattutto dal metodo di acquisizione (es. una foto satellitare produrrà ovviamente un dato raster, mentre la cattura di un percorso GPS ne produrrà uno vettoriale).

3.1 Tipi di Geometria

Gli oggetti geografici rappresentati in un dato vettoriale sono formati da *primitive geometriche*. I tipi di geometria sono caratterizzati dal numero di dimensioni:

- **0**: punti (e punti orientati);

- **1**: linee e curve;
- **2**: aree;
- **3**: oggetti tridimensionali.

Un'altra caratterizzazione riguarda i tipi di coordinate utilizzate:

- 2 dimensioni (coordinate piane o geografiche);
- 3 dimensioni (posizione e dati altimetrici);
- 4 dimensioni (si aggiunge un dato scalare associato alla posizione).

Nei dati geografici classici in 3 dimensioni, la dimensionze Z (altimetria) è considerata più un attributo associato che un dato geometrico vero e proprio; le analisi dei Sistemi Informativi Geografici sono spesso bidimensionali. Solo negli ultimi tempi si è iniziato ad operare con veri dati tridimensionali (es. edifici 3D).

3.1.1 Punti

Gli oggetti geometrici più semplici sono quelli puntuali, la geometria è rappresentata da una singola coppia di coordinate[1], in cui può essere presente la terza dimensione (vedi figura 3.1 a sinistra).

Alcuni sistemi prevedono la possibilità di memorizzare un *orientamento* insieme alla posizione del punto, vale a dire associare un angolo ai dati di posizione.

[1] I dettagli su come sono rappresentate le coordinate verranno presentati nella sezione **??**.

3.1.2 Linee

Gli oggetti lineari sono rappresentati come linee spezzate in cui sono definite le coordinate dei vertici (vedi figuta 3.1 a destra. In alcuni casi l'ordine dei vertici è significativo; ad esempio nella definizione di un tracciato idrografico, l'ordine potrebbe rappresentare la direzione di scorrimento dell'acqua.

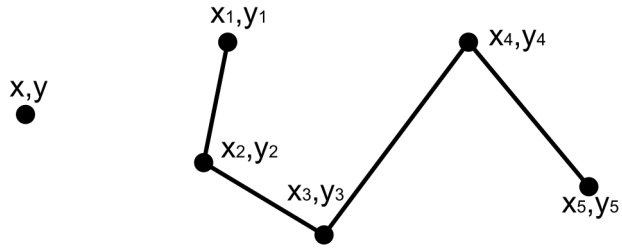


Figura 3.1: Rappresentazione di una geometria puntuale (a sinistra) e lineare (a destra).

3.1.3 Aree

Gli oggetti areali sono rappresentati come poligoni, in cui sono definite le coordinate dei vertici. I poligono possono contenere alcuni buchi interni (es. radure nei boschi, cortili di edifici, etc.); i bordi interni dei buchi sono rappresentati anch'essi attraverso le coordinate dei vertici (vedi figura 3.3).

Il bordo esterno e l'elenco dei bordi interni dei buchi vengono chiamati *anelli* (rings in inglese). L'ordine dei vertici è spesso fissato in modo univoco; ad esempio si richiede che l'ordine sia in senso orario (oppure antiorario) percorrendo l'area dall'esterno (questo vuol dire che l'ordine dei vertici dei buchi è inverso rispetto all'ordine dei vertici del bordo esterno).

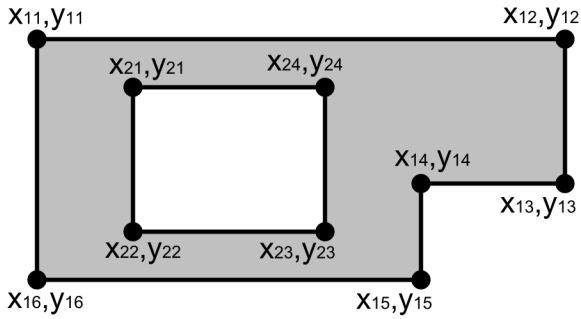


Figura 3.2: Rappresentazione di una geometria areale.

3.1.4 Geometrie Multiple

Oltre agli oggetti geometrici semplici, si definiscono oggetti geometrici multipli (punti multipli, linee multiple, aree multiple); oggetti cioè in cui un'unica geometria è costituita da una collezione di parti semplici (vedi l'esempio di figura **??**).

3.2 Caratteristiche dei Dati Vettoriali

Le forme geometriche non sono l'unica caratteristca dei Dati Vettoriali. Approfondiamo qui di seguito alcuni aspetti salienti di questo tipo di dato geografico.

3.2.1 Le coordinate

Le coordinate di un punto geografico sono rappresentate da una coppia di numeri[2], con l'aggiunta della terza dimensione (quota

[2]Di solito a livello internazionale e nei software si cita prima la x (longitudine) e poi la y (latitudine); storicamente in Italia invece è uso citare prima la y e poi la x. In caso di dubbio è meglio specificare.

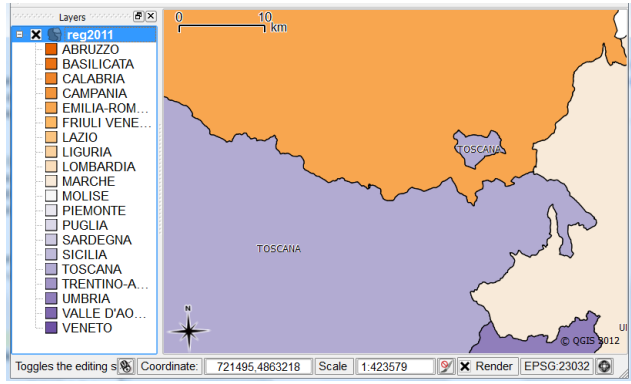


Figura 3.3: L'enclave del comune di Badia Tedalda fa parte della Regione Toscana; l'area della regione deve essere quindi rappresentata da una geometria multipla.

altimetrica).

I numeri da soli non bastano però a specificare una posizione geografica; è fondamentale includere la specifica del *Sistema di Riferimento*. In Italia ad esempio sono in uso svariati sistemi di riferimento geografici: Roma40, ED1950, WGS84, sistemi catastali, eventualmente utilizzati con svariate proiezioni (UTM - Fusi 32,33,34 Nord, Canonica di Lamber per carte in scale a grande denominatore). Una descrizione dell'argomento esula gli scopi di questo libro; ricordiamo solo che esiste una standard internazionale definito da *EPSG* (e a cui PostGIS si riferisce) che enumera i sistemi di riferimento in uso.

3.2.2 Vincoli Geometrici e Topologici

Non tutte le disposizioni di vertici formano geometrie corrette. Perchè una forma geometrica sia considerata corretta deve sottostare ad alcune condizioni. Ad esempio non ci devono essere vertici

identici ripetuti, i bordi delle aree non si devono *annodare* (vale a dire sovrapporre), inoltre i buchi devono essere contenuti nelle loro rispettive aree e così via.

I dati vettoriali moderni sono spesso sottoposti ad alcuni *vincoli topologici*; ad esempio si richiede che le posizioni di inizio e fine dei tratti stradali coincida esattamente con la posizione degli incroci, oppure che le aree amministrative comunali coincidano esattamente (come ci si aspetta) lungo i bordi con le aree dei comuni adiacenti e così via.

Nella vecchia cartografia numerica, in cui l'unico scopo era quella della visualizzazione e stampa, questi vincoli non erano necessari. I dati moderni al contrario devono rispettare alcune caratteristiche di correttezza per poter eseguire le analisi richieste (es. ricerche di percorsi per navigatori, calcolo dell'estensione di superfici, etc.).

3.2.3 Attributi alfanumerici

La cartografia numerica degli anni passati era finalizzata alla stampa, percui conteneve le geometrie nude e crude, eventualmente suddivise in strati od associate ad uno stile di visualizzazione. Oltre ai dati numerici, i dati vettoriali geografici moderni sono corredati da attributi alfanumerici; ad esempio la definizione di un tratto stradale può contenere attributi che descrivono il tipo di manto, il numero di corsie, il nome e la classifica della strada, etc. Gli attributi alfanumerici concorrono all'informazione geografica e vengono utilizzati nelle analisi dei dati.

Nelle basi di dati spaziali come PostGIS questi attributi corrispondono agli attributi classici della base di dati (colonne alfanumeriche delle tabelle).

3.2.4 Struttura Gerarchica

Gli oggetti vettoriali presenti di una collezioni di dati non sono elecanti alla rinfusa, ma raggruppati in classi di oggetti (*Feature*

CLass o in breve *Feature* in inglese). In alcuni sistemi questi raggruppamenti vengono chiamati *Strati Infomativi* (*Layers* in inglese). Le classi di oggetti raggruppano oggetti omologhi; esempi di classi sono: fiumi, strade, edifici, boschi, etc.

In molti sistemi geografici all'interno di una classe gli oggetti devono essere tutti dello stesso geometrico (punto, linea od area), inoltre condividono il numero ed il tipo degli attributi associati (ma ovviamente non i valori). In particolare nei database spaziali come PostGIS, di solito una classe corrisponde ad un tabella con una o più colonne di tipo geometrico.

La suddivisione in classi è sempre presente in ogni sistema geografico. Alcuni sistemi poi introducono altri livelli gerarchici che raggruppano insiemi di classi. Ad esempio il personal-db di ArcGIS può raggruppare le classi in *Feature Dataset*.

3.3 Formati di Memorizzazione e di Scambio

Non esistono molti formati di memorizzazione di dati vettoriali geografici. Esistono una serie di formati *proprietari* basati su software commerciali. Uno dei formati commerciali più diffuso è il cosidetto *shapefile* (che PostGIS può importare direttamente); in questo formato ogni classe di oggetti è memorizzata in una serie di 3,4 o 5 file con nome comune ed estesione diversa.

Un interessante tentativo di realizzare un formato di scambio non commerciale è rappresentate dal *GML*, anche se per adesso non ha avuto la fortuna sperata. PostGIS possiede alcune funzioni di importazione di singole geometrie GML.

Esistono poi delle specifiche standard per l'interconnessione diretta a dati vettoriali tramite rete; in questo caso i dati non vengono scambiati su *file* ma attraverso servizi di rete. Lo standard

per i dati vettoriali si chiama *WFS* (=Web Feature Service) ed è definito da OGC.

3.4 Fattore di scala

Le carte geografiche (su sopporto cartaceo) sono caratterizzate da una *fattore di scala*, che in questo caso è il fattore di riduzione rispetto alla dimensione reale. Ad esempio in una carta in scala 1:200,000 i particolare di 1 cm sono nella reltà lunghi esattamente 2 Km.

Per quanto riguarda i dati geografici *raster* (ad esempio foto satellitari) si può parlare di risoluzione del pixel a terra; questi dati sono caratterizzati quindi dalla dimensione della superficie reale rappresentata da un singolo pixel. Ovviamente una foto satellitare con un pixel a terra di 1 cm sarà più dettagliata di una con una risoluzione di 1 m.

Per i dati vettoriali la questione è più sottile. Si può parlare di scala? A prima vista sembrerebbe di no, dato che posto visualizzare questo tipo di dati a qualsiasi livello di zoom. In realtà la costruzione di un dato vettoriale parte sempre dalla definizione di un fattore di scala nominake (oppure di un intervallo di scala nominale). La scala nominale di un dato vettoriale influisce su alcuni suoi aspetti:

- densità geometrica delle geometrie: il numero di oggetto ed il numero di vertici di cui sono composti deve essere proprorzionale alla scala nominale. Un dato vettoriale se troppo ingrandito rispetto alla sua scala non mi mostrerà nessuna informazione, mentre se troppo rimpicciolito risulterà un guazzabuglio incomprensibile di geometrie;

- definizione delle classi di oggetti da modellare: ogni scala nominale influisce sulla scelta degli oggetti da modellare. Ad esempio un dato in scala 1:10,000 conterrà classi di oggetti

tipo: piste ciclabili, marciapiedi, aree stradali, piscine (intese come oggetti areali). Un dato in scala 1:50,000 conterrà classi del tipo: assi stradali, stadi, isolati urbani, piscine (intese come oggetti puntuali). Un dato in scala 1:250,000 conterrà classi del tipo: centri abitati (come oggetti puntuali), aree urbane, aeroporti, ...

- precisione dei dati: può darsi che i dati in scale nominali a più grande denominatore siano catturati con una precisione minore dei dati a scale con più piccolo denominatore.

4 Introduzione a PostGIS

PostGIS è il nome del supporto spaziale a PostgreSQL, come quest'ultimo è un software Open Source e gratuito. PostGIS offre un supporto spaziale veramente completo, che fa buona concorrenza ai software commericali ben più costosi. Inoltre si basa su standard spaziali affermati, come quelli dell'*Open GIS Consortium*.

Gli esempi di questo capitolo si basano sul fatto che voi abbiate installato correttamente PostgreSQL, compreso la componente spaziale PostGIS, che abbiate create un database spaziale come descritto nella sezione 2.3 e che infine siete anche riusciti a connettervi a questo database.

Il contenuto di questa capitolo è volutamente scarno: il nostro scopo non è l'esposizione di un manuale di riferimento, ma l'introduzione di un utilizzo avanzato attraverso esempi e implementazioni di funzionalità complesse.

4.1 Componenti del supporto spaziale

In che cosa consiste il supporto spaziale fornito da PostGIS? Principalmente nelle seguenti componenti:

- il nuovo tipo di dato *GEOMETRY*, che permette la memorizzazione di geometrie geo-riferite. Esistono anche altri tipi di dato secondari come i *Box* e *Geography*;
- due table di supporto: *spatial_ref_sys* che elenca i sistemi di riferimento disponibili, *geometry_columns* che elenca i meta-dati delle colonne spaziali;

- una serie di circa 700 funzioni spaziali di supporto;
- l'implementazione di *indici spaziali*;
- alcuni strumenti esterni di importazione/esportazione.

Sembra poco, ma è tutto quello che serve. PostGIS ad esempio non ha un proprio visualizzatore grafico; è possibile comunque utilizzare un qualsiasi GIS Open Source, come *Quantum GIS*.

4.1.1 Il tipo di Dato GEOMETRY

Come sappiamo bene, i dati di una colonna di database sono associati ad un tipo (es. INTEGER, CHARACTER, BOOLEAN). Il supporto spaziale introduce un nuovo tipo di dato: GEOMETRY, questo tipo è un tipo di dato ad oggetti (complesso); Il nuovo tipo di dato contiene la geometria di un singolo oggetto geografico e eventualmente il sistema di riferimento associato (codice SRID=System Reference IDentifier). Il tipo è multiforme: può contenere dati a 2, 3 o 4 dimensioni (x,y,z e il campo M) ed in varie forme geometriche: punti, linee, aree, curve, etc.

4.1.2 La Tabella spatial_ref_sys

Questa tabella memorizza l'elenco dei sistemi di riferimento supportati. Si basa principalmente sullo standard EPSG[1], che elenca ed identifica con un codice numerico, i vari sistemi di riferimento del mondo. La tabella *spatial_ref_sys* contiene le seguenti colonne:

- il codice numerico del sistema di riferimento;
- il nome ed il codice dell'autorità che ha definito questo sistema;

[1] http://www.epsg.org.

- la definizione testuale del sistema di riferimento (simile a quella presente nei file *prj* del formato *shapefile*;

- i parametri *proj4*[2] che definiscono la prioiezione.

Ad esempio il sistema EPSG numero 4326 corrisponde a WGS84, coordinate geografiche (quelle utilizzate dai GPS per intenderci). La sua definizione testuale è la seguente:

```
GEOGCS[
  "WGS 84",
  DATUM[
    "WGS_1984",
    SPHEROID[
      "WGS84",
      6378137,298.257223563,
      AUTHORITY["EPSG","7030"]
    ],
    AUTHORITY["EPSG","6326"]
  ],
  PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],
  UNIT["degree",0.01745329251994328,
AUTHORITY["EPSG","9122"]],
  AUTHORITY["EPSG","4326"]
]
```

I i codici SRID 32632 e 32633 corrispondono rispettivamente a UTM WGS84 fusi 32 e 33 Nord, mentre i codici SRID 23032 e 23033 corrispondono rispettivamente a UTM ED50 fusi 32 e 33 Nord.

Se un sistema di riferimento non è previsto può essere aggiunto alla tabella dall'utente, definendo l'opportuna stringa proj4 in modo corretto, ovviamente il sistema non diventerà uno standard! Ad esempio, dato che l'Italia si trova a cavallo di due fusi UTM, è comodo definire un *Fuso Italia*, media del fuso 32 e 33 Nord, con i seguenti parametri:

[2]Proj4 è una libreria Open Source di gestione dei sistemi di riferimento, PostGIS la utilizza per la loro gestione.

```
SRID: 90000;
auth_name: "IGMI";
auth_srid: 90000;
srtext: ...;
proj=tmerc +lat_0=0 +lon_0=12 +k=0.9985 +x_0=7000000.0
    +y_0=0 +ellps=WGS84 +datum=WGS84 +units=m +no_defs
```

Un inciso: alcuni sistemi di riferimento non possono essere trattati in modo analitico. Ad esempio *Roma40*, ha bisogno (per una sua eventuale trasformazione precisa in altro sistema di riferimento), di griglie di trasformazione punto-punto, che memorizzano la differenza di coordinate da un sistema dato (es. WGS84). Proj4 (e quindi Postgres) supportano le griglie di trasformazione nel formato standard *NAD (ntv2)*; un esempio di stringa proj che utilizza un eventuale grigliato di trasformazione è la seguente:

```
+proj=longlat +ellps=WGS84 +to +proj=longlat
  +ellps=WGS84 +nadgrids=./roma40_to_wgs84.gsb
```

4.1.3 La Tabella geometry_columns

La seconda tabella del sistema spaziale è inizialmente vuota; conterrà i meta-dati minimi delle nostre colonne spaziali. Per ogni colonna di tipo GEOMETRY presente nella base di dati, questa tabella memorizza:

- il catalogo, lo schema e la tabella contenente la colonna;
- il nome della colonna[3];
- il numero di dimensioni delle geometrie contenute (2,3 o 4);
- il codice del sistema di riferimento (SRID);
- l'eventuale sotto-tipo geometrico: punti, linee, aree, etc.

[3]Nei database spaziali una tabella può avere anche più di una colonna geometrica, cose che invece nei sistemi GIS standard non accade, come ad esempio negli *shapefile*.

L'aggiornamento di questa tabella non è automatico; è buona cura dell'utilizzatore della base di dati tenere aggiornata questa tabella. Alcuni GIS esterni infatti si basano su di essa per poter utilizzare i dati geografici contenuti.

Il codice del sistema di riferimento (SRID) deve essere uno di quelli presenti nella tabella *spatial_ref_sys*, oppure è possibile utilizzare il valore -1 (sistema di riferimento non definito); in questo caso non sarà possibile utilizzare le funzioni che utilizzano il sistema di riferimento.

Vedremo più avanti che la gestione del sistema di riferimento comporta alcune problematiche: ad esempio lo strumento di importazione degli shapefile non riesce ad interpretare il contenuto del file *prj*, quindi il codice del sistema di riferimento va inserito manualmente.

4.1.4 Le funzioni spaziali

PostGIS definisce circa 800 funzioni spaziali. Questo libro non vuole essere un manuale di riferimento delle funzioni, ma piùttosto una raccolta di esempi di applicazione. Le funzioni verranno introdotte quindi negli esempi di utilizzo.

Le funzioni possono essere raggruppate nelle seguenti classi:

Gestione es. *AddGeometryColumn* aggiunge una colonna geometrica ad una tabella;

Costruzione es. *ST_MakePoint* costruisce una geometria puntuale;

Accesso alla geometria es. *ST_NDims* restituisce il numero di dimensioni di una geometria;

Modifica della geometria es. *ST_Transform* proietta una geometria in un nuovo sisterma di riferimento;

Output es. le funzioni *ST_AsSVG, ST_AsKML, ST_AsGML* trasformano le geometrie rispettivamente nei formati *SVG, KML, GML*;

Misura es. le funzioni *ST_Length* e *ST_Area* misurano lunghezza e superficie di una geometria;

Relazioni es. *ST_Intersects* controlla se due geometrie si intersecano oppure no;

Elaborazione es. *ST_Union* calcola l'unione di una serie di geometrie;

Miscellanea es. *ST_YMin* ritorna la più piccola latitudine di una geometria.

Esistono poi funzioni per la gestione delle *transazioni lunghe* che verranno trattate nel capitolo **??** e le funzioni sul *Linear Referencing* che verranno trattate nel capitolo **??**.

4.2 Utilizzo di PostGIS

Introduciamo adesso i concetti basi per l'utilizzo di PostGIS.

4.2.1 Valori Letterali

Come per i numeri e le parole, anche il tipo *GEOMETRY* ha la possibilità di gestire valori letterali (costanti). Il formato dei valori letterali si basa sullo standard *OGC* detto WKT[4]. I valori geometrici possono essere specificati in modo letterale; alcuni esempi:

- `'POINT(6.1 43.2)'` geometria puntuale bidimensionale;

[4]OGC=Open GIS Consortium, WKT=Well-Known Text.

- `'LINESTRING(0 0 0,1 1 0,1 2 0)'` linea tridimensionale;
- `'POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1, 2 1, 2 2, 1 2, 1 1))'` area poligonale con un buco interno;
- `'MULTIPOINT(0 0,1 2)'` punto multiplo;

PostGIS estente poi lo standard OGC includendo nella geometria la definizione del sistema di riferimento:

- `'SRID=4326;POINT(6.1 43.2)'` punto noto in coordinate geografiche WGS84.

I valori letterali geometrici non possono esser utilizzati in SQL così come sono, perchè verrebbero scambiati per parole. Ci sono due modalità di utilizzo:

1. applicando un *cast*[5], accondando alla definizione la dicitura `::GEOMETRY`, che forza il tipo del dato;
2. utilizzando la funzione *ST_GeomFromEWKT* che trasforma un testo in una vera e propria geometria.

Il seguente listato non funziona:

```
1 SELECT ST_XMax('LINESTRING(6 42,7 43)');
```

mentre le due seguenti SELECT danno il risultato atteso:

```
1 SELECT ST_XMax('LINESTRING(6 42,7 43)'::GEOMETRY);
2 SELECT ST_XMax(ST_GeomFromEWKT('LINESTRING(6 42,7 43)'));
```

Esercizio guidato: vogliamo proiettare una coordinate geografica (longitudine, latitudine) misurata col GPS, nel sistema UTM WGS84, fuso 32 Nord. Il listato 4.10 risolve il nostro problema.

[5]Ovvero una trasformazione esplicita di tipo di dato.

```
1 SELECT
2   ST_AsEWKT(
3     ST_Transform(
4       'SRID=4326;POINT(11.25␣43.75)'::GEOMETRY
5       ,32632
6     )
7   )
```

Listato 4.2: Esempio di riproiezione di dati letterali.

Alcune note al listato 4.10: abbiamo costruito la geometria puntuale digitando la longitudine e la latitudine del punto, definendone anche il sistema di riferimento (SRID=4326). Quindi abbiamo forzato il tipo (::GEOMETRY) e abbiamo passato la geometria alla funzione *ST_Transform* che richiede come secondo parametro, il codice del sistema di riferimento di arrivo (32632=UTM WGS84, fuso 32 Nord). Abbiamo finito? No perchè per default il risultato geometrico viene visualizzato in binario; utilizziamo quindi la funzione *ST_AsEWKT* che ci mostra una geometria *come* fosse testo.

4.2.2 Creazione di una Tabella Geometrica

Vediamo adesso come sia possibile creare una tabella con attributi spaziali; questa tabella rappresenterà la classe (*Feature Class* in inglese) degli edifici di un dato geografico vettoriali. In teoria è possibile creare la tabella con i suoi attributi alfanumerici, più il nostro attributo di tipo *GEOMETRY*, quindi inserire i metadati necessari nella tabella *geometry_columns*.

Invece di fare tutto il lavoro a mano faremo uso della funzione PostGIS *AddGeometryColumn*, che:

1. aggiunge una colonna geometrica ad una tabella pre-esistente;

2. inserisce gli opportuni metadati nella tabella *geometry_columns*;

3. aggiunge alcuni vincoli di controllo sulla tabella (numero di dimensioni geometriche, tipo geometrico e codice del sistema di riferimento).

Per prima cosa creiamo la tabella con i suoi attributi alfa-numerici:

```
1 CREATE TABLE edificio
2 (
3   id INTEGER PRIMARY KEY,
4   descr CHARACTER VARYING
5 );
```

Per ora la nostra tabella *edificio* è una classica tabella alfa-numerica. Adesso utilizziamo la funzione *AddGeometryColumn* per trasformarla in una tabella spaziale. La funzione ha varie forme, ad esempio:

```
1 SELECT AddGeometryColumn(
2   'edificio',
3   'shape',
4   4623,
5   'POLYGON',
6   2
7 );
```

La funzione richiede come parametro:

1. la tabella su cui operare (*edificio*);
2. il nome della colonna geometrica che vogliamo aggiungere (*shape*);
3. lo SRID, cioè il codice del sistema di riferimento (*4326*);
4. il tipo delle geometrie contenuto (*POLYGON*, ma poteva essere anche *POINT, LINESTRING, MULTIPOLYGON,* ...);
5. il numero di dimensioni (*2* = bidimensionale, ma poteva essere *3* = tridimensionale).

4.2.3 Creare un Indice Spaziale

La funzione *AddGeometryColumn* fa quasi tutto il lavoro, eccetto che non si occupa degli *indici*. Perché il funzionamento delle query spaziali sia veloce è fondamentale creare un indice sulla colonna geometrica. Nella sezione 2.11 abbiamo visto la sintassi generale per la creazione di un indici. Gli indici spaziali però sono diversi da quelli su numeri e parole e si basano su algoritmi completamente diversi. Gli indici spaziali di PostGIS si basano su un tipo di indice denominato *GIST*, che va dichiarato durante la creazione dell'indice stesso. Scriviamo quindi:

```
1 CREATE INDEX edificio_shape_idx
2 ON edificio USING gist (shape);
```

Il nostro indice spaziale è in funzione!

4.2.4 Uno sguardo alla tabella spaziale

Analizziamo adesso il lavoro svolto dalla funzione di PostGIS. Possiamo utilizzare l'interfaccia grafica di *pgAdminIII* per scorrere l'albero degli oggetti e visualizzare la tabella *edificio* (vedi figura 4.1). Oltre a contenere la colonna *shape* di tipo *GEOMETRY*, la

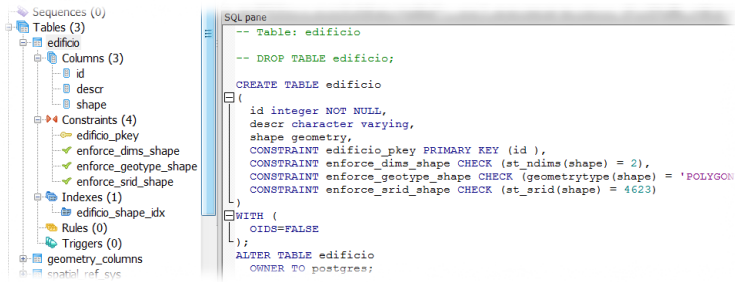


Figura 4.1: Analisi della struttura di una tabella spaziale.

tabella possiede anche tre vincoli di controllo, che verificano il

numero di dimensioni, il tipo geometrico e il codice del sistema di riferimento. Infine alla tabella è associato un indice spaziale.

Adesso visualizziamo la tabella *geometry_columns* che conterrà una riga del tipo:

```
f_table_sch|f_table_name|f_geom_c|c_dim|srid| type
-----------+------------+--------+-----+----+-------
public     |edificio    |shape   |    2|4623|POLYGON
```

La prima colonna memorizza lo schema contenente la tabella spaziale, che per default è *public*. Vedremo in seguito come sia comodo distribuire i dati su vari schemi per tenerli ben in ordine.

4.2.5 Creazione di dati spaziali

La nostra tabella spaziale è bella ma vuota. Adesso bisogna creare dei dati spaziali. Per farlo utilizziamo il comando SQL *INSERT INTO*, definendo la componente spaziale in modo letterale come abbiamo imparato a fare:

```
1 INSERT INTO edificio
2 VALUES (
3   1,
4   'Ospedale',
5   'SRID=4623;POLYGON((6␣42,␣8␣42,␣8␣43,␣6␣43,6␣42))'
6         ::GEOMETRY
7 );
```

Listato 4.6: Creazione di un dato spaziale: poligono semplice.

Il valore *1* è il codice del nostro edificio, il valore *'Ospedale'* è la sua descrizione, segue la definizione della componente geometrica. Notate come sia stato specificato il sistema di riferimento e le coppie di coordinate che definiscono logitudine e latitudine degli spigoli del nostro ospedale quadrato[6]. Come mai un quadrato ha cinque spigoli? Perchè la definizione di un poligono richiede che

[6] A dire la verità è un edificio grandino: un grado per un grado!

questo venga *chiuso* vale a dire che la definizione del bordo si deve concludere con la ripetizione delle coordinate del primo spigolo (*6 42*).

La definizione della geometria termina con il *cast* di tipo, come abbiamo imparato a fare nella sezione 4.2.1.

Divertiamoci adesso a creare un altro oggetto spaziale:

```
1 INSERT INTO edificio
2 VALUES (
3   2,
4   'Industria',
5   'SRID=4623;POLYGON(
6 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(10␣43,␣13␣43,␣13␣46,␣10␣46,␣10␣43),
7 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(11␣44,␣12␣44,␣12␣45,␣11␣45,␣11␣44)
8 ␣␣␣␣␣␣␣␣)'::GEOMETRY
9 );
```

Listato 4.7: Creazione di un dato spaziale: poligono con buchi.

In questo caso la geometria della nostra frabbrica corrisponde ad un poligono con un buco (cortile) interno; la seconda serie di coordinate definisce infatti la geometria del cortile. La figura 4.2 mostra il nostro dato geospaziale visto da QGis (vedremo più avanti come si fa).

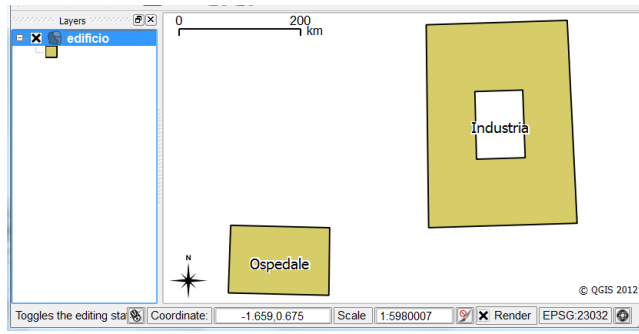


Figura 4.2: Visualizzazione delle geometrie spaziali.

4.3 Introduzione alle analisi spaziali

Una volta che possiedo un dato geo-spaziale cosa ci faccio? Vediamo qui di seguito alcune interrogazioni SQL di base che è possibile effettuare sulla tabella spaziale.

4.3.1 Visualizzazione testuale delle geometrie

La prima cosa che vogliamo fare è quella di visualizzare l'intera tabella. Siamo tentati di scrivere:

```
1 SELECT *
2 FROM   edificio;
```

Il fatto è che gli attributi geometrici vegono per default visualizzati in binario. Per visualizzare una geometria in modo *leggibile* possiamo far uso della funzione *ST_AsEWKT*:

```
1 SELECT id, descr, ST_AsEWKT(shape)
2 FROM   edificio;
```

ottenendo il seguente risultato:

```
 id |   descr   |                st_asewkt
----+-----------+-------------------------------------
  1 | Ospedale  | SRID=4623;POLYGON((6 42,8 42,8 43 ...
  2 | Industria | SRID=4623;POLYGON((10 43,13 43,13 ...
```

Ovviamente il modo migliore per *vedere* una geometria è quello di farlo attraverso un software grafico come QGIS, cosa che faremo più avanti.

4.3.2 Semplici Misure

Supponiamo di voler conoscere l'estensione dei nostri edifici. Possiamo utilizzare la funzione *ST_Area* nel seguente modo:

```
1 SELECT id, descr, ST_Area(shape)
2 FROM   edificio;
```

Questa funzione però calcola il risultato nella stessa unità di misura della geometria, quindi in gradi sessadecimali quadri, una ben strana unità di misura di superficie.

Per ottenere una misura in metri possiamo *proiettare* i nostri nati in un sistema che contenga coordinate piane in metri (es. UTM-WGS84 fuso 32 Nord); per questo utilizziamo la funzione di trasformazione di sistema *St_Transform*:

```
1  SELECT id, descr,
2         ST_Area(St_Transform(shape,32632))
3  FROM   edificio;
```

Otteniamo così la superficie in metri quadrati.

4.3.3 Funzioni spaziali aggreganti

Come per i dati alfa-numerici, anche la componente spaziale contiene funzioni *aggreganti*. Ad esempio se si vuole conoscere l'estensione totale dell'insieme degli oggetti geografici contenuti in un tabella è possibile utilizzare la funzione *ST_Extent*:

```
1  SELECT ST_Extent(shape)
2  FROM   edificio;
```

Il risultato sarà nel nostro caso:

```
"BOX(6 42,13 46)"
```

Si noti come il risultato è unico per i due edifici e rappresente l'estensione massima dell'unione di tutte le geometrie.

5 Import/Export

I dati geografici devono vaggiare. Vediamo adesso alcuni meccanismi di interscambio di dati da e verso PostGIS. Vedremo in questo capitolo alcuni meccanismi per importare dati dentro PostGIS, per esportarli e per collegarsi ad alcuni dei software GIS più diffusi.

5.1 Importazione di shapefile

Il formato di dato geografico vettoriale più diffuso è (ancora) senza dubbio lo *shapefile*. L'installazione standard di PostGIS prevede uno strumento specifico per l'importazione di dati in questo formato, l'applicazione *Shape File to PostGIS Importer*.

Una volta lanciata l'applicazione, apparirà l'interfaccia di Figura 5.1. La struttura dell'interfaccia cambia frequentemente da una versione all'altra, quindi la versione che state utilizzando potrebbe essere leggermente diversa: l'importante è compredere i concetti, che sono sempre gli stessi. I passi necessari per importare uno shapefile sono i seguenti:

1. selezionate il file da importare (ad esempio reg2011.shp);

2. impostate i parametri di connessione alla base di dati: utente e password, il nome del server che contiene la base di dati (nel nostro caso *localhost*) ed infine il nome della base di dati (nel nostro caso *corso*). Una volta impostato tutti i dati è possibile provare la connessione premendo il relativo pulsante;

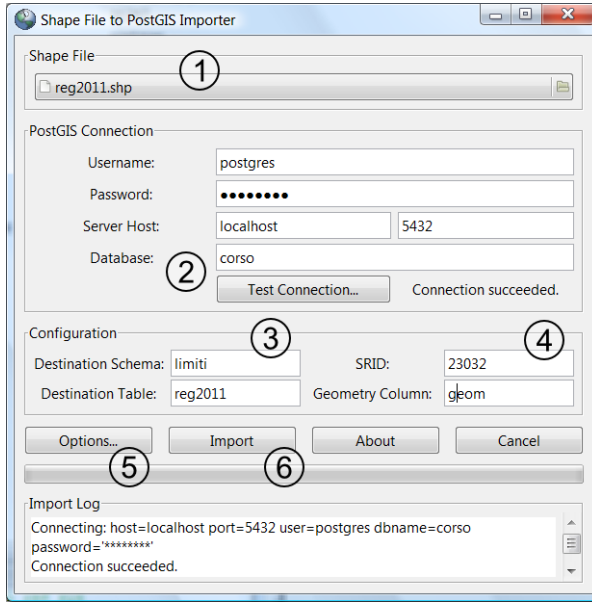


Figura 5.1: Interfaccia dello strumento *Shape File to PostGIS Importer*.

3. scegliete la destinazione dei dati: il nome dello schema (che deve essere stato preventivamente creato) o *public* se non si vuole usare nessun schema particolare, ed il nome della tabella di destinazione (che invece non deve esistere);

4. scegliete i dettagli geometrici: lo SRID, vale a dire il codice del sistema di riferimento, che nel caso dei dati catastali è 23032 (ED50 UTM Fuso32N) ed il nome della colonna che conterrà i dati geometrici (nel nostro caso *geom*). La specifica del sistema di riferimento è necessaria anche in presenza del file *prj*, dato che PostGIS non è in grado di determinarlo automaticamente;

5. impostate le opzioni particolari che vi interessano (di cui parleremo dopo);

6. premete il pulsante *import*: se tutto va bene i dati verranno importati in PostGIS.

Premendo il tasto *Options...* appare il dialogo di Figura 5.2, che permette di impostare alcuni parametri aggiuntivi; il più importante è la codifica carattere dei dati (*character encoding*). Gli shapefile devono essere tutti codificati come *LATIN1*.

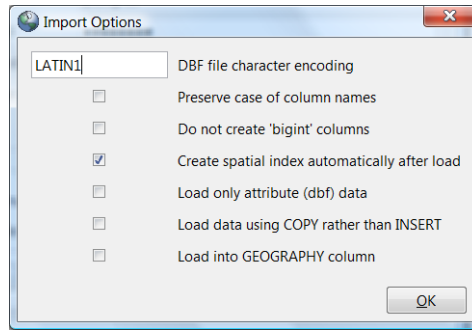


Figura 5.2: Opzioni aggiuntive di importazione.

5.2 Esportazione di shapefile

Curiosamente, mentre lo strumento per importare i dati ha un'interfaccia grafica, quello per esportare è invece un comando *DOS* (forse per scoraggiare le esportazioni). Per esportare uno shapefile quindi aprite una finestra DOS (o prompt dei comandi). Se non sapete come si fa chiedere al vostro amico appassionato di informatica.

Lo strumento per esportare gli shapefile è il comando DOS *pgsql2shp.exe*; di seguito riportiamo la schemata di aiuto dello strumento:

```
USAGE: pgsql2shp.exe [<options>] <database>
[<schema>.]<table>
       pgsql2shp.exe [<options>] <database> <query>

OPTIONS:
  -f <filename>  Use this option to specify the name of
     the file to create.
  -h <host>  Allows you to specify connection to a
     database on a machine other than the default.
  -p <port>  Allows you to specify a database port other
     than the default.
  -P <password>  Connect to the database with the
     specified password.
  -u <user>  Connect to the database as the specified
     user.
  -g <geometry_column> Specify the geometry column to
     be exported.
  -b Use a binary cursor.
  -r Raw mode. Do not assume table has been created by
     the loader. This would not unescape attribute
 names and will not skip the 'gid' attribute.
  -k Keep postgresql identifiers case.
  -? Display this help screen.
```

Per esportare una tabella in forma di shapefile bisogna indicare il nome del file di uscita (opzione -f), il nome del server della base di dati (opzione -h), il nome dell'utente (opzione -u), il nome della colonna che contiene la geometria (opzione -g), quindi di seguito bisogna elencare il nome della base di dati (nel nostro esempio *corso*) ed il nome della tabella, eventualmente preceduta dal nome dello schema (nel nostro esempio *limiti.reg2011*).

Ad esempio il comando (scritto tutto di seguito su di una sola riga), esporta la tabella dei limiti regionali:

```
pgsql2shp.exe -f reg2011.shp -h localhost -u
   postgres -g geom corso limiti.reg2011
```

Se tutto procede correttamente appaiono le seguenti diciture:

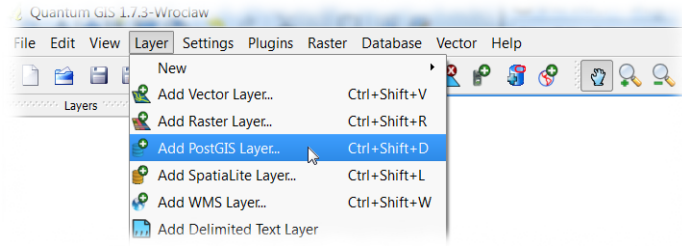
```
Initializing... Done (postgis major version: 1).
```

```
Output shape: Polygon
Dumping: XX [20 rows].
```

Notate che lo strumento di esportazione genera anche (e correttamente) il file *prj* con la definizione del sistema di riferimento.

5.3 Connessione con QGIS

QGIS è uno dei più notevoli software GIS Open Source e gratuiti. Per visualizzare i dati di PostGIS utilizzando QGIS si procede nel seguente modo: lanciate QGIS e selezionate il menù *Layer / Add PostGIS Layer...*. Se è la prima volte che vi connette al



vostro server dovete per prima cosa creare una nuova connessione, cliccate quindi sul pulsante *New* (Figura 5.3). Apparirà il dialogo

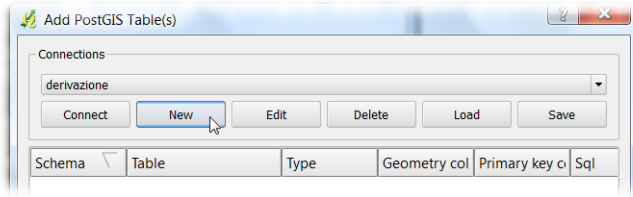


Figura 5.3: Pulsante per la creazione di una nuova connessione.

di Figura **??**: ormai siete esperti di parametri di connessione e non diremo niente più. Anche in questo caso è possibile testare la correttezza dei parametri. Una volta creata la nostra connessione,

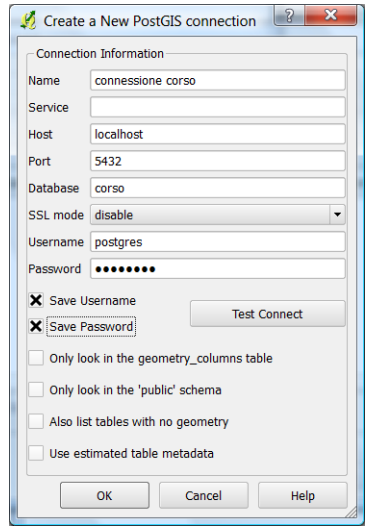


Figura 5.4: Parametri per la creazione della connessione.

selezionatela (un futuro ve la troverete già fatta) e premete il pulsante *Connect*: apparirà una lista di feature presenti nella base di dati (Figura 5.5). Scegliete le feature che vi interessano e quindi premete il tasto *Add*. Le feature selezionate appariranno nella mappa di QGIS.

QGIS riconosce automaticamente il sistema di riferimento dei dati PostGIS. Nel caso in cui il sistema di riferimento non coincida con quello utilizzato in visualizzazione, QGIS può riproiettare i dati al volo.

Il fatto veramente interessante è che QGIS si connette a PostGIS in lettura/scrittura, vale a dire che i dati connessi possono essere editati, sia per quanto riguarda la geometria che gli attributi

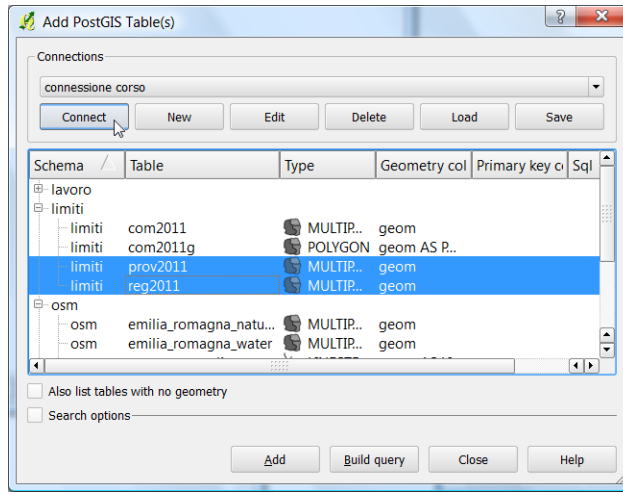


Figura 5.5: Connessione e scelta delle feature da inserire.

alfanumerici associati: è quindi possibile creare nuovo oggetti, cancellarli, spostarli, o modificarne la forma.

Un'altra interessate opportunità di QGIS è quella di poter facilmente esportare i dati in formato *shapefile*: basta selezionare il dato nella legenda, cliccare col bottone destro per far apparire il menù contestuale, selezionare la voce *Save as* e salvare i dati in formato *shapefile*.

6 Operazioni

In questo capitolo vedremo come si possono realizzare alcune operazioni di processamento tipiche. Gli esempi sono costruiti supponendo di aver importato nel nostro database i limiti amministrativi Istat [1] (nello schema *limiti*) e i dati OSM[2] nello schema *OSM*. Questo dati sono di pubblico dominio. Inoltre supponiamo di aver creato uno schema di lavoro denominato appunto *lavoro*.

I nomi delle operazioni corrispondenti alle realtive sezioni, sono quelli usati nei GIS commerciali di riferimento.

6.1 Operazioni elementari

Nella prima sezione, tanto per scaldarci un po', si mostra come realizzare alcune operazioni di base tipiche.

6.1.1 Append

Volendo aggiungere una serie di oggetti ad una feature esistente basta far seguire il comando *INSERT INTO* da una query che genera i dati da inserire. Ovviamente gli attributi devono concordare in numero e tipo. Un primo abbozzo di esempio è rappresentato dal listato 6.1.

```
1 INSERT INTO osm.toscana_water
2 SELECT *
3 FROM   osm.emilia_romagna_water;
```

[1] http://www.istat.it/

[2] http://www.openstreetmap.org/

Listato 6.1: Esempio di *Append* dei laghi emiliani a quelli toscani (attenzione: non viceversa!). Versione NON funzionante.

In realtà il listato 6.1 non funziona: questo perchè il campo *gid*, che è la chiave primaria, conterrebbe valori duplicati (esiste un lago 1 elimiliano ed un lago 1 toscano, etc.). Per ovviare alla duplicazione del campo *gid* è necessario ignorarlo durante la duplicazione, lasciando che il database generi nuovi codici di chiave primaria automaticamente. Per far questo bisogna eliminare la copia del campo *gid*, specificando esplicitamente tutti gli altri campi (natural, name e geom) sia nel comando *INSERT* che nel comando *SELECT*. La versione funzionante della query è visibile nel listato 6.2.

```
1 INSERT INTO osm.toscana_water ("natural",name,geom)
2 SELECT "natural",name,geom
3 FROM   osm.emilia_romagna_water;
```

Listato 6.2: Esempio di *Append* dei laghi emiliani a quelli toscani: versione funzionante.

6.1.2 Add e Calculate Field

Aggiunta di attributi e calcolo di valori possono essere realizzati direttamente con comandi SQL standard. Ad esempio se vogliamo aggiungere un attributo *superficie* alla feature *osm.toscana_natural* e calcolarne il valore, possiamo usare il codice del listato 6.3. Per il calcolo della superficie abbiamo fatto uso della funzione *ST_Area*.

```
1 ALTER TABLE osm.toscana_natural
2 ADD superficie FLOAT;
3
4 UPDATE osm.toscana_natural
5 SET    superficie = ST_Area(geom);
```

Listato 6.3: Aggiunta e calcolo di un attributo: superficie delle riserve naturali.

6.1.3 Add XY(Z) Coordinates

Aggiunta di attributi ed estrazioni geometriche sono molto semplici: a titolo di esempio riportiamo nel listato 6.4 la creazione dei campi *point_x, point_y* e *point_z* che contengono le coordiante numeriche del punto geometrico. L'estrazione delle coordiante avviene attraverso le funzioni *ST_X*, *ST_Y* e *ST_Z*.

```
1 CREATE TABLE lavoro.toscana_poi_ext AS
2 SELECT gid,
3        category,
4        name,
5        ST_X(geom) AS point_x,
6        ST_Y(geom) AS point_y,
7        ST_Z(geom) AS point_z,
8        geom
9 FROM   osm.toscana_poi;
```

Listato 6.4: Aggiunta e calcolo di attributi da dati geometrici.

6.1.4 Check Geometry

PostGIS fornisce una serie di funzioni per la validazione e l'analisi delle geometrie: il listato 6.5 presenta un esempio di applicazione di queste funzioni (in questo caso non si è creata una tabella dei risultati). La figura 6.2 mostra il risultato del controllo (nelle colonne *boolean* il valore *t* sta per vero (true), il valore *f* per falso (false).

```
1 SELECT ST_GeometryType(geom),
2        ST_CoordDim(geom),
3        ST_IsClosed(geom),
```

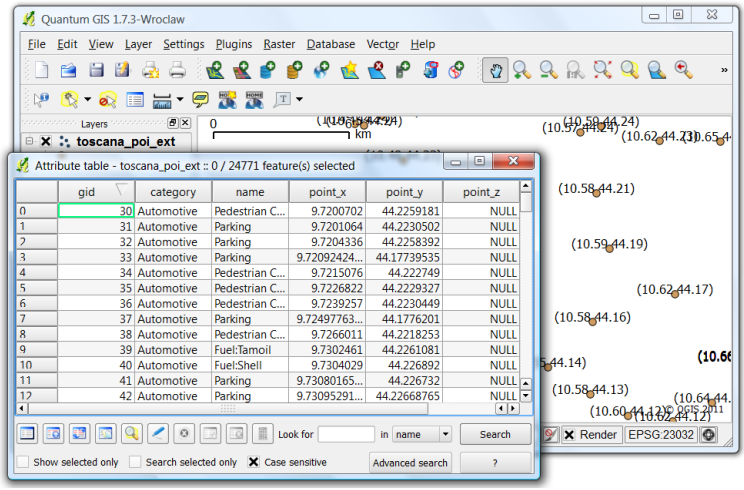


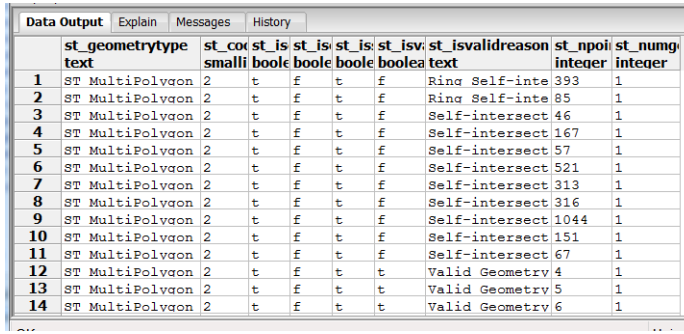
Figura 6.1: Esempio di operazione *Add XY(Z) Coordinates*.

```
        ST_IsEmpty(geom),
        ST_IsSimple(geom),
        ST_IsValid(geom),
        ST_IsValidReason(geom),
        ST_NPoints(geom),
        ST_NumGeometries(geom)
FROM    osm.toscana_natural;
```

Listato 6.5: Aggiunta e calcolo di attributi da dati geometrici.

6.2 Operazioni geometriche di base

Nella prossima sezione si realizzano le operazioni di base per la manipolazione della forma geometrica.



Data Output | Explain | Messages | History

	st_geometrytype text	st_coo smalli	st_is boole	st_is boole	st_is boole	st_isv boolea	st_isvalidreason text	st_npoi integer	st_numg integer
1	ST_MultiPolygon	2	t	f	t	f	Ring Self-inte	393	1
2	ST_MultiPolygon	2	t	f	t	f	Ring Self-inte	85	1
3	ST_MultiPolygon	2	t	f	t	f	Self-intersect	46	1
4	ST_MultiPolygon	2	t	f	t	f	Self-intersect	167	1
5	ST_MultiPolygon	2	t	f	t	f	Self-intersect	57	1
6	ST_MultiPolygon	2	t	f	t	f	Self-intersect	521	1
7	ST_MultiPolygon	2	t	f	t	f	Self-intersect	313	1
8	ST_MultiPolygon	2	t	f	t	f	Self-intersect	316	1
9	ST_MultiPolygon	2	t	f	t	f	Self-intersect	1044	1
10	ST_MultiPolygon	2	t	f	t	f	Self-intersect	151	1
11	ST_MultiPolygon	2	t	f	t	f	Self-intersect	67	1
12	ST_MultiPolygon	2	t	f	t	t	Valid Geometry	4	1
13	ST_MultiPolygon	2	t	f	t	t	Valid Geometry	5	1
14	ST_MultiPolygon	2	t	f	t	t	Valid Geometry	6	1

Figura 6.2: Esempio di risultato di controlli e validazioni.

6.2.1 Cambio di sistema di riferimento (Project)

Per cambiare sistema di riferimento ai dati geografici di una tabella, basta utilizzare la funzione *ST_Transform*, che proietta una qualsiasi geometria in un nuovo sistema di riferimento, specificandone il codice EPSG. Supponiamo il caso che vogliate mostrare questi dati su un sito web utilizzando *MapServer* o *GeoServer* e che il sistema di riferimento della mappa visualizzata sia diverso da quello del dato originale: per impedire che il sistema di visualizzazione sia costretto e riproiettare i dati al volo e migliore l'efficienza di visualizzazione è utile memorizzare i dati nel sistema di riferimento utilizzato per la visualizzazione.

Il listato 6.6 mostra un esempio di proiezione dei dati della costa toscana (che in origine sono in coordinate geografiche) nel sistema EPSG:32632, corrispondente a UTM-WGS84 Fuso 32 Nord. Gli attributi non geografici sono copiati come sono, mentre il campo *geom* è riproiettato attraverso la funzione *ST_Transform*. Alcuni dettagli: l'attributo *natural* va scritto fra virgolette perchè è una parola chiave di Postgres, la dicitura *AS geom* dopo la funzione *ST_Transform* server per assegnare correttamente il nome del campo nella nuova tabella che stiamo creando.

```
1 CREATE TABLE lavoro.toscana_coastline_p AS
2 SELECT gid,
3        "natural",
4        name,
5        ST_Transform(geom,32632) AS geom
6 FROM   osm.toscana_coastline;
```

Listato 6.6: Cambio di sistema di riferimento (Project) della costa toscana.

Un inciso: attenti a non confondere la funzione *ST_Transform* con la funzione *ST_SetSRID*, la prima server per riproiettare il coordinate in un nuovo sistema di riferimento, la seconda forza artificialemente il codice ad un particolare sistema di riferimento, ignorando quale sia quello in origine. La *ST_SetSRID* è utile in alcuni casi, come quello in cui siamo in presenza di un dato senza sistema di riferimento (SRID=-1) perchè in un primo momento sconosciuto, che in seguito vogliamo invece assegnare ad un certo codice dato.

6.2.2 Feature Envelope to Polygon

Con questa operazione si vuole costruire il rettangolo di ingombro minimo di una geometria. Questa funzione si realizza applicando la funzione *ST_Envelope* (vedi listato 6.7). La figura 6.3 ne mostra il risultato[3].

```
1 CREATE TABLE lavoro.water_env AS
2 SELECT gid,
3        "natural",
4        name,
5        ST_Envelope(geom) AS geom
6 FROM   osm.toscana_water;
```

[3]I rettangoli degli ingombri sono leggermente ruotati perchè il sistema di riferimento dei dati non corrisponde a quello di visualizzazione.

Listato 6.7: Esempio di calcolo di *envelope*.

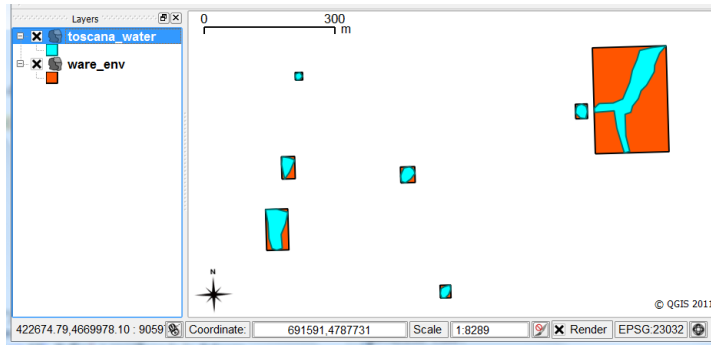


Figura 6.3: Risultato del calcolo degli *envelope* sui laghi toscani.

6.2.3 Buffer zone

Con *Buffer zone* (in italiano Zona "cuscinetto") si intede il poligono corrispondente alla zona che si trova entro una certa distanza dalla geometria data. L'operazione viene effettuata utilzzando la funzione *ST_Buffer* e specificando la distanza nell'unità di misura del sistema di riferimento utilizzato (es. metri per i sistemi in proiezione, radianti per i sistemi in coordiante geografiche). Avremmo potuto applicare la funzione in modo del tutto analogo al listato 6.6, vogliamo invece sfruttare la situazione per costruire il nuovo dato come una *VISTA* e non come una *TABELLA*. L'utilizzo di una vista fa sí che il dato prodotto sia "dinamico": ad ogni modifica del dato originale la buffer zone relativa si adatterà automaticamente. La contropartita è costituita dal fatto che avremo un rallentamento del sistema, dato che Postgres è costretto a calcolare la Buffer zone ogni volta che viene richiesta.

Il listato 6.8 mostra un esempio di creazione della Buffer zone come vista. Alcuni particolari: per fare in modo che QGIS non

si arrabbi troppo dobbiamo dichiarare la *chiave primaria* della tabella di partenza, se questa non esiste già (prime due righe del codice); di seguito la creazione della vista procede in modo del tutto analogo alla tabella. Nel nostro esempio i dati di partenza sono in proiezione e la distanza (1000) è espressa in metri. La figura 6.4 mostra il risultato dell'operazione.

```
1 ALTER TABLE lavoro.toscana_coastline_p
2 ADD CONSTRAINT tcp_pk PRIMARY KEY(gid);
3
4 CREATE VIEW lavoro.zona_costiera_p AS
5 SELECT gid,
6        "natural",
7        name,
8        ST_Buffer(geom,1000) as geom
9 FROM   lavoro.toscana_coastline_p;
```

Listato 6.8: Buffer zone intorno alla costa, realizzata come vista.

6.2.4 Feature To Point

Con l'operazione *Feature To Point* si intende l'estrazione da ogni oggetto geometrico (linea, area, ma anche multipoint) di un punto rappresentativo dell'intero oggetto, di solito il centro di massa (baricentro). Supponiamo di voler "generalizzare" i lagni della toscana per una visualizzazione ad una scala con grande denominatore; per questo vogliamo trasformare i laghi da areali a puntuali. Il listato 6.9 mostra il codice corrispondente, che fa semplicemente utilizzo della funzione *ST_Centroid*. Il filtro serve per selezionare solo i laghi (escludendo linea di costa ed altre feature).

```
1 CREATE TABLE lavoro.laghi_p AS
2 SELECT gid,
3        name,
4        ST_Centroid(geom) AS geom
5 FROM   osm.toscana_water
6 WHERE  "natural"='water';
```

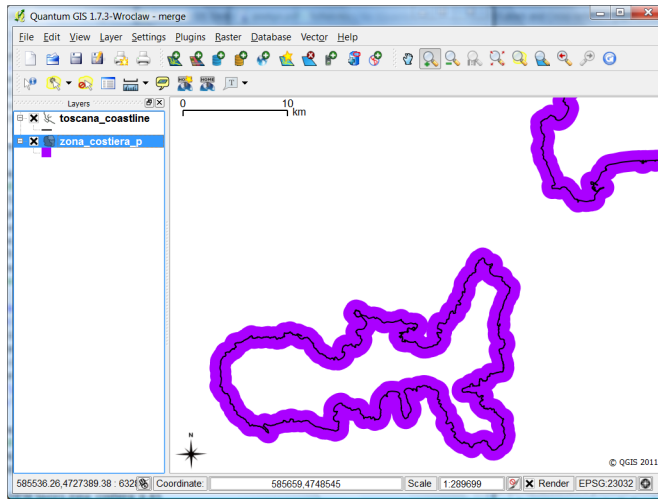


Figura 6.4: La costa della toscana con una Buffer zone di un chilometro.

Listato 6.9: *Feature To Point*: creazione di laghi puntuali.

6.2.5 Dissolve

Con *dissolve* si intende l'operazione che aggregare una serie di feature con attributi comuni. Supponiamo di voler generare la feature *regioni* a partire dai limiti amministrativi provinciali (dati Istat). La tabella delle province contiene l'attributo *cod_reg* che specifica la regione di appartenenza; raggruppando le province al variare di questo attributo è possibile ottenere le aree corrispondenti alle regioni. Il listato 6.10 mostra un esempio di *dissolve*.

```
1 CREATE TABLE lavoro.mie_regioni AS
2 SELECT cod_reg,
3        ST_Union(geom) AS geom
4 FROM   limiti.prov2011
```

```
5 GROUP BY cod_reg;
```

Listato 6.10: Operazione *dissolve* sulle aree provinciali.

Per prima cosa notiano l'inizio della query: il risultato dell'interrogazione che segue verrà salvato nella tabella *lavoro.mie_regioni*. L'interrogazione fa uso della funzione *ST_Union*; questa funzione è aggregante (come min, max, avg, etc.), vale a dire che produce un risultato aggregando i valori di più elementi di una tabella. La funzione *ST_Union* genera un'unica area ottenuta dall'unione di aree a cui è applicata; nel nostro caso la funsione è applicata all'attributo *geom* della tabella *limiti.prov2011*.

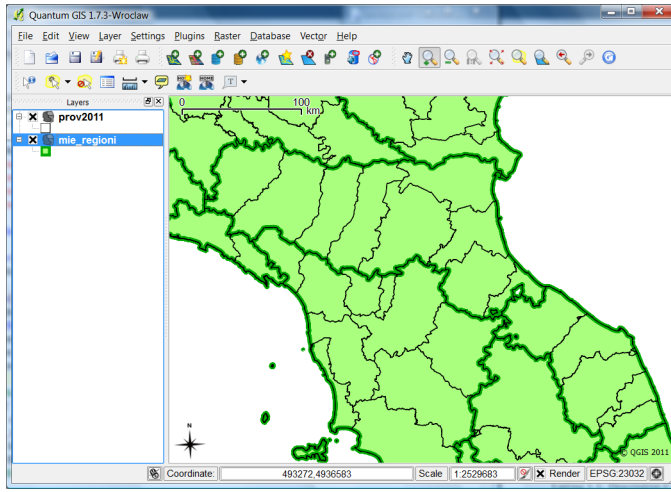


Figura 6.5: Risultato dell'operazione di *dissolve*.

6.2.6 Merge

Con *Merge* di intende l'operazione che unisce più tabelle (feature) con struttura simile in un unica tabella. Supponiamo nel nostro

caso di aver caricato le aree naturali dell'Emilia-Romagna, insieme a quelle della Toscana; l'idea è quella di unificare le due feature in una sola, che chiameremo *lavoro.tosco_emiliano_nat*. Il listato 6.11 mostra il codice SQL corrispondente.

```
1 CREATE TABLE lavoro.tosco_emiliano_nat AS
2   SELECT name,type,geom
3   FROM   osm.toscana_natural
4 UNION
5   SELECT name,type,geom
6   FROM   osm.emilia_romagna_natural;
```

Listato 6.11: Operazione *merge* sulle aree tosco-emiliane.

L'operazione di *merge* è ottenuta semplicemente utilizzando il costrutto *UNION* di SQL che giustappone due query. L'operazione può essere eseguita anche su tre o più tabelle; l'unica cosa importante è che la struttura delle tabelle in questione corrisponda esattamente (numero e tipo delle colonne).

6.2.7 Clip

Con *Clip* di intende l'operazione che "taglia" una geometria con un altra. Supponiamo di dover estrarre da tutte le strade della Toscana, quelle del comune di Firenze. Le eventuali strade a cavallo del territorio comunale vanno inoltre tagliate al limite dell'area di Firenze.

Per comodità cambiamo il sistema di riferimento dei limiti comunali Istat in WGS84, attraverso il codice presente nel listato 6.12; creiamo quindi la tabella *limiti.com2011g* con tanto di chiave primaria.

```
1 CREATE TABLE limiti.com2011g AS
2 SELECT gid,cod_reg,cod_pro,pro_com,nome_com,nome_ted,
3        ST_Transform(geom,4326) AS geom
4 FROM   limiti.com2011;
5
6 ALTER TABLE limiti.com2011g
```

```
7 ADD CONSTRAINT com2011g_pk PRIMARY KEY (gid);
```

Listato 6.12: Cambio di sistema di riferimento dei comuni italiani (4326=WGS84 geografiche).

A questo punto l'operazione di *Clip* è realizzata dalla funzione *ST_Intersection*, che calcola l'intersezione di due geometrie. Il codice del listato 6.13 crea la tabella *lavoro.thighway_clip* che contiene le strade toscane "clippate" sul comune di Firenze. Alcune note sul codice:

- Per prima cosa ci sono due tabelle in ballo: le strade (*osm.toscana_highway*) ed i comuni (*limiti.com2011g*), la query infatti è tecnicamente una *JOIN* spaziale. Per comodità mettiamo dei soprannomi alle tabelle, rispettivamente *h* e *c*.

- Come al solito, gli attributi non geometrici delle strade (gid, type, ...) sono presi così come sono.

- L'attributo *geom* è ottenuto dalla funzione *ST_Intersection* fra le geometrie delle strade e dei comuni.

- Ci sono poi due filtri: il primo filtro seleziona il solo comune di Firenze (nome del comune = 'Firenze'), dato che vogliamo prendere in considerazione solo quel comune.

- Il secondo filtro è più sottile: fra tutte le coppie comune/strada, seleziona solo quelle che si intersecano[4]. Senza questo filtro la tabella del risultato conterrebbe in realtà TUTTE le strade della Toscana: le strade non contenute nel comune di Firenze sarebbero presenti ma con GEOMETRIA VUOTA, dato che l'intersezione con l'area comunale è nulla. Questo

[4]non bisogna confondere la funzione *ST_Intersects* che ci dice se due geometrie si intersecano oppure no, dalla funzione *ST_Intersection* che calcola effettivamente la "forma" dell'intersezione di due geometrie.

risultato poco intuitivo è dovuto al fatto che il database applica la funzione intersezione a tutte le strade, che tocchino il comune di Firenze oppure no. In generale bisogno sempre tenere sotto controllo la possibilità di generare geometrie nulle, che risultano poi fastidiose nell'utilizzo dei dati.

```
1 CREATE TABLE lavoro.thighway_clip AS
2 SELECT h.gid, h.type, h.name, h.oneway, h.lanes,
3        ST_Intersection(h.geom,c.geom) AS geom
4 FROM   osm.toscana_highway AS h,
5        limiti.com2011g AS c
6 WHERE  c.nome_com = 'Firenze'
7 AND    ST_Intersects(h.geom,c.geom)
```

Listato 6.13: *Clip* delle strade toscane sul comune di Firenze.

La figura 6.6 mostra il risultato dell'operazione di *Clip*.

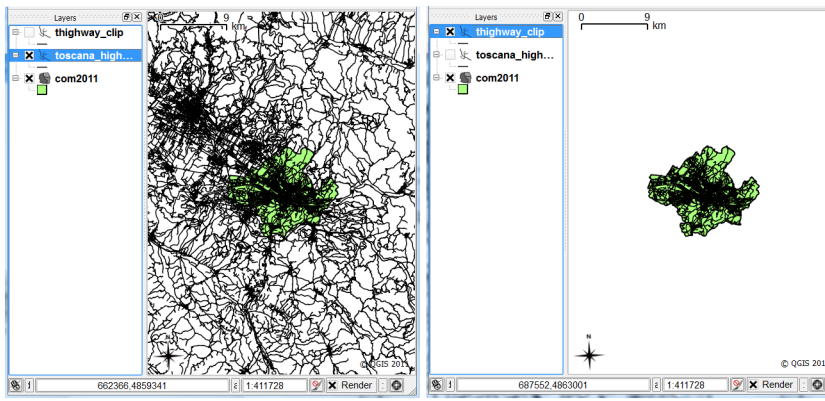


Figura 6.6: Clip: a sinistra le strade della Toscana, a destra le strade "clippate" sul comune di Firenze.

Un inciso sugli indici spaziali e l'efficienza: la tabella *osm. toscana_highway* possiede un indice spaziale e quindi l'interrogazione del listato 6.13 è molto veloce. La figura 6.7 mostra i piani di

esecuzione di Postgres nei casi in cui non è presente l'indice spaziale (a sinistra) oppure è presente (a destra): il costo massimo previsto (il tempo di esecuzione stimato) è dieci volte superiore nel caso in cui non sia presente un indice spaziale.

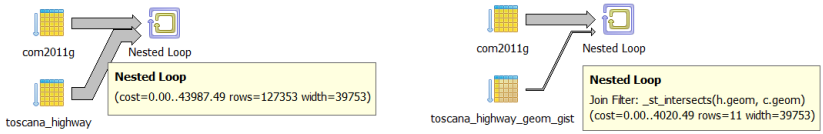


Figura 6.7: Clip, indici e efficienza. A sinistra il piano di esecuzione della query 6.13 senza indici spaziali, a sinistra con indice spaziale: il costo temporale è ridotto ad un decimo.

6.2.8 Intersect

L'operazione di *Intersect* è del tutto simile al *Clip*: la differenza stà nell'avere più di una feature utilizzata nell'intersezione e nel copiare nel risultato anche gli attributi della seconda feature. In pratica il risultato contiene le intersezioni degli oggetti in questione con i loro attributi incrociati. Il listato 6.14 mostra un esempio di *Intersect* limitata ai soli comuni nella provincia di Firenze (la numero 48). Il codice è uguale a Clip, eccetto il fatto che vengono copiati in uscita anche gli attributi dei comuni (c.cod_reg, c.cod_pro, ...).

```
1 CREATE TABLE lavoro.thighway_int AS
2 SELECT h.gid, h.type, h.name, h.oneway, h.lanes,
3        c.cod_reg, c.cod_pro, c.pro_com, c.nome_com,
4        ST_Intersection(h.geom,c.geom)
5 FROM   osm.toscana_highway AS h,
6        limiti.com2011g AS c
7 WHERE  c.cod_pro = 48
8 AND    ST_Intersects(h.geom,c.geom)
```

Listato 6.14: *Clip* delle strade toscane sul comune di Firenze.

La figura 6.8 mostra il risultato dell'intersezione: eventuali strade a cavallo di due comuni sono tagliate in pezzi ed a ogni pezzo vengono assegnati gli attributi dell'area comunale.

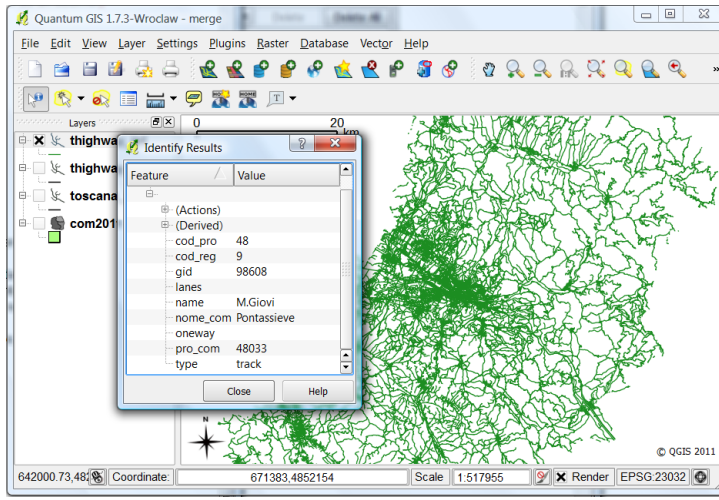


Figura 6.8: Intersection fra comuni e strade: il risultato contiene gli attributi di entrambe le feature di partenza.

6.2.9 Erase

L'operazione di *Erase* è l'inverso del *Clip*, nel senso che da una prima feature viene "tagliata" la parte contenuta nella seconda. Il listato 6.15 cancella le porzioni di strada che si trovano sul comune di Pisa. Il codice è simile a quello di Clip (listato 6.13): invece di *ST_Intersect* abbiamo utilizzato la funzione *ST_Difference*, che sottrae una geometria da un'altra.

Alcune note al codice: attenzione che la funzione *ST_Difference* non è simmetrica come *ST_Intersect*, vale a dire che *ST_Difference(a,b)* è diverso da *ST_Difference(b,a)*: dobbiamo togliere dalle strade il comune e non viceversa. Attenzione anche al secondo filtro (*NOT ST_Contains ...*): differisce da quello di *Clip* perché in questo caso le differenze che generano geometrie nulle sono date dalle strade contenute interamente nel comune. Attraverso la funzione *ST_Contains* controlliamo quindi di prendere in considerazione solo le strade non completamente contenute del comune di Pisa.

```
1 CREATE TABLE lavoro.thighway_erase AS
2 SELECT h.gid, h.type, h.name, h.oneway, h.lanes,
3        ST_Difference(h.geom,c.geom) AS geom
4 FROM   osm.toscana_highway AS h,
5        limiti.com2011g AS c
6 WHERE  c.nome_com = 'Pisa'
7 AND    NOT ST_Contains(c.geom,h.geom);
```

Listato 6.15: *Erase*: eliminazione delle strade sul comune di Pisa.

Nella figura 6.9 il risultato dell'operazione di *Erase*.

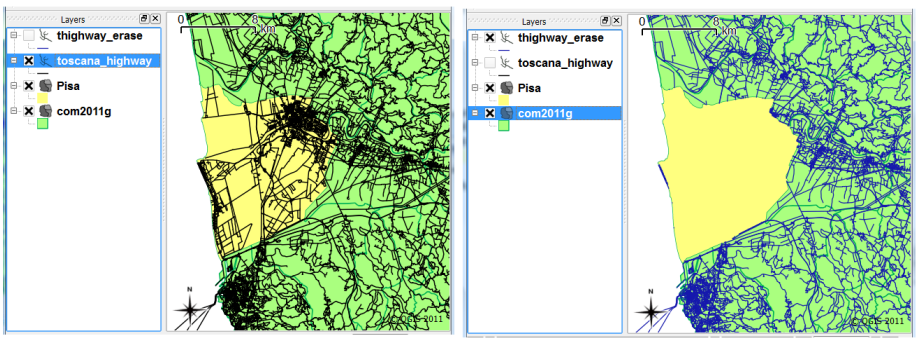


Figura 6.9: Erase: a sinistra tutte le strade della Toscana, a destra le strade rimanenti dopo l'eliminazione del comune di Pisa.

6.2.10 Simplify Line or Polygon

Capita che ad una certa scala o a un certo livello di zoom, il dettaglio degli oggetti geometrici sia troppo elevato. Un modo per semplificare le descrizione geometriche è quello di ridurre il numero di vertici che le compongono; il discorso si applica sia alle linee che alle aree.

PostGIS possiede la funzione *ST_Simplify* che riduce il numero di vertici, mantenendo una massima distanza dalla geometria inziale. Esiste anche la funzione *ST_SimplifyPreserveTopology* che tiene sotto controllo gli eventuali errori introdotti dalla semplificazione.

Il listato 6.16 applica la funzione di semplificazione ai limiti regionali semplificandoli con una tolleranza di 500 metri. La figura 6.10 mostra un esempio del risultato.

```
1 CREATE TABLE lavoro.reg_sempl AS
2 SELECT gid,
3        ST_SimplifyPreserveTopology(geom,500) AS geom
4 FROM   limiti.reg2011;
```

Listato 6.16: *Simplify*: creazione di una versione semplificata dei limiti amministrativi regionali.

6.2.11 Symmetrical Difference

Si vuole eliminare da una coppia di feature, le varie parti comuni ad entrambe. Es: dalle province della Toscana e dell'Emilia si vogliono eliminare le (eventuali) parti sovrapposte, perché considerate errore.

Postgis prevede la funzione *ST_SymDifference*, per cui un primo tentativo potrebbe essere:

```
1 SELECT ST_SymDifference(a.geom,b.geom)
2 FROM   limiti.province AS a,
3        limiti.province AS b
4 WHERE  a.cod_reg=8
5 AND    b.cod_reg=9
```

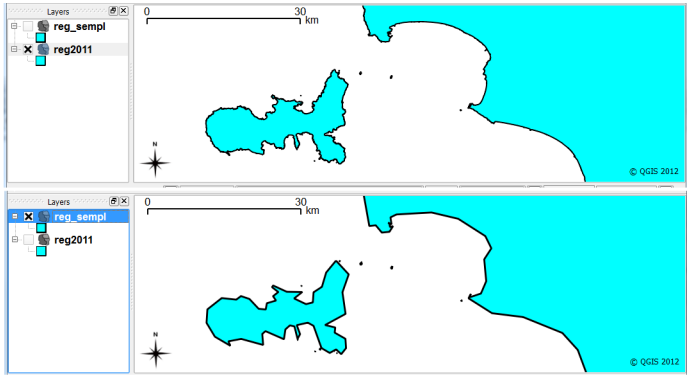


Figura 6.10: *Simplification*: in alto il dato originale, in basso quello semplificato.

Listato 6.17: Codice ERRATO per *Symmetrical Difference*.

In realtá questo primo tentativo è errato: viene eseguita l'operazione di differenza a tappeto, senza considerare se due geometrie si intersecano veramente oppure no. Questo prodoce come risultato il prodotto cartesiano delle due regioni; vale a dire che ogni provincia viene duplicata un sacco di volte.

La soluzione corretta é come al solito un po' piu' complicata. Bisogna unire tre parti: le geometrie che veramente si intersecano, e quelle di partenza che non intersecano l'altra regione.

```
1 SELECT ST_SymDifference(a.geom,b.geom)
2 FROM   limiti.province AS a,
3        limiti.province AS b
4 WHERE  ST_Intersects(a.geom,b.geom)
5 AND    a.cod_reg=8
6 AND    b.cod_reg=9
7
8 UNION
9
```

```
10 SELECT a.geom
11 FROM   limiti.province AS a
12 WHERE  NOT EXISTS (
13   SELECT *
14   FROM   limiti.province AS b
15   WHERE  ST_Intersects(a.geom,b.geom)
16   AND    b.cod_reg=9
17 )
18 AND    a.cod_reg=8
19
20 SELECT b.geom
21 FROM   limiti.province AS b
22 WHERE  NOT EXISTS (
23   SELECT *
24   FROM limiti.province AS a
25   WHERE  ST\_Intersects(a.geom,b.geom)
26   AND    a.cod_reg=8
27 )
28 AND    b.cod_reg=9
```

Listato 6.18: Codice corretto per una *Symmetrical Difference*.

6.3 Operazioni intermedie

Adesso arrivano una serie di funzioni concettualmente un po' piú complicate, ma sempre realizzabili con una sola query.

6.3.1 Spatial Join

Con il termine *Spatial Join* si intende l'operazione che mette in relazione una coppia di feature secondo un criteri spaziale. Il listato 6.19 mostra la struttura generica di una *Spatial Join* , dove *ST_XXX* è una delle seguenti funzioni di relazione: *ST_Contains* (contiene), *ST_ContainsProperly* (contiene strettamente), *ST_Covers* (copre), *ST_CoveredBy* (è coperto da), *ST_Crosses* (attraversa), *ST_Disjoint* (è disgiunto da), *ST_Equals* (è uguale a), *ST_Intersects*

(interseca), *ST_Overlaps* (si sovrappone a), *ST_Touches* (tocca), *ST_Within* (è all'interno di).

```
SELECT a.*, b.*
FROM   tabella1 AS a,
       tabella2 AS b
WHERE  ST_XXX(a,b);
```

Listato 6.19: Struttura generica di una *Spatial Join*.

Ad esempio se vogliamo trovare tutte le strade che attraversano (il bordo di) il comune di Firenze, possiamo scrivere una *Spatial Join* nella forma del listato 6.20. La figura 6.11 ne mostra il risultato.

```
CREATE TABLE lavoro.thighway_cross AS
SELECT h.*
FROM   osm.toscana_highway AS h,
       limiti.com2011g AS c
WHERE  c.nome_com = 'Firenze'
AND    ST_Crosses(h.geom,c.geom);
```

Listato 6.20: Esempio di *Spatial Join*: Crosses.

6.3.2 Spatial Join con distanze

Una *Spatial Join* classica è quella che prevede di trovare gli oggetti che si trovano "entro una certa distanza data". Ad esempio, se vogliamo trovare tutte le strade che si trovano entro 5 chilometri (che corrispondono a circa 0.05 radianti) dal comune di Firenze, dovremmo scrivere qualcosa tipo il listato 6.21.

```
CREATE TABLE lavoro.thighway_dist AS
SELECT h.*
FROM   osm.toscana_highway AS h,
       limiti.com2011g AS c
WHERE  c.nome_com = 'Firenze'
AND    ST_Distance(h.geom,c.geom)<0.05
```

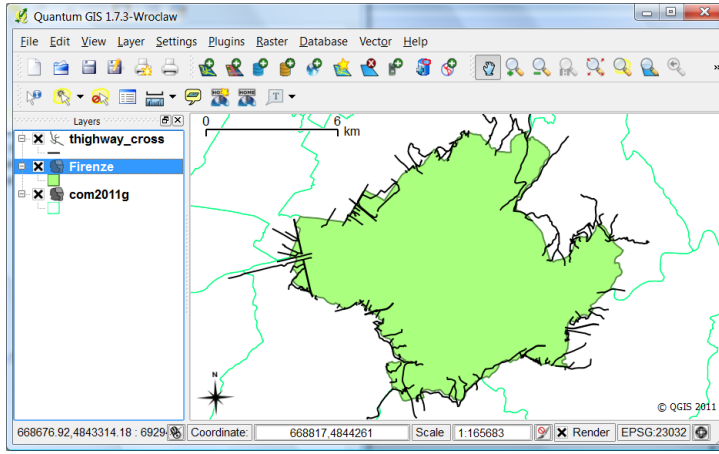


Figura 6.11: Esempio di *Spatial Join*: strade che attraversano il bordo del comune di Firenze.

Listato 6.21: Esempio di *Spatial Join* con distanze, versione inefficiente.

In realtà il listato 6.21 è del tutto inefficiente: il database tenta di calcolare la distanza di TUTTE le coppie strada/comune per poi valutarne il valore: gli indici spaziali non vengono usati. Dobbiamo dare un consiglio al database: introduciamo con un piccolo trucchetto un vincolo in più alla query, ed otteniamo il listato 6.22.

```
1 CREATE TABLE lavoro.thighway_dist AS
2 SELECT h.*
3 FROM   osm.toscana_highway AS h,
4        limiti.com2011g AS c
5 WHERE  c.nome_com = 'Firenze'
6 AND    ST_Distance(h.geom,c.geom)<0.05
7 AND    ST_Intersects( ST_Expand(ST_Envelope(c.geom),
8                  0.05+0.01), h.geom )
```

Listato 6.22: Esempio di *Spatial Join* con distanze, versione efficiente.

Per prima cosa calcoliamo il minimo rettangolo che include la geometria del comune, attraverso la funzione *ST_Envelope*, quindi "gonfiamo" il rettangolo con la funzione *ST_Expand* della distanza da cercare più un "tot" di sicurezza, infine imponiamo che il risultato intersechi (funzione *ST_Intersects*) la geometria della strada.

In altre parole intruduciamo come vincolo aggiuntivo il fatto che la geometria della strada debba intersecare il rettangolo che contiene il comune espanso di un valore opportuno. Questo vincolo è logicamente del tutto inutile, dato che è incluso nel vincolo di distanza (le interrogazioni 6.21 e 6.22 quindi producono lo stesso esatto risultato). Il database però è in grado di utilizzare il secondo vincolo per accedere all'indice spaziale e la seconda query risulta notevolmente più veloce della prima: provate a visualizzare il piano di esecuzione delle due query per confrontarne i costi di esecuzione. L'immagine 6.12 mostra il risultato della *Spatial Join* con distanze.

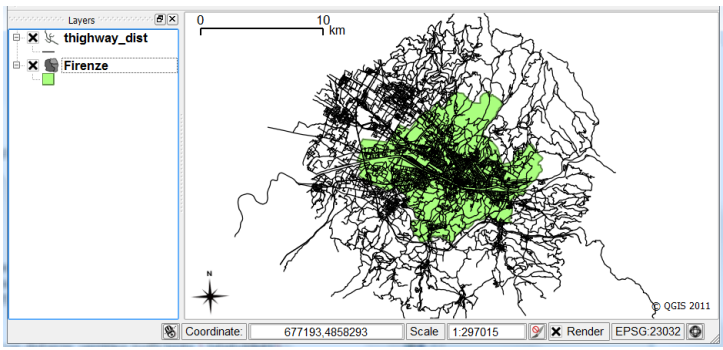


Figura 6.12: Esempio di *Spatial Join* con distanze: strade entro 5 chilometri dal comune di Firenze.

6.3.3 Feature Vertices To Points

Con l'operazione *Feature Vertices To Points* di intende la trasformazione di ogni vertici di una geometria in feature puntuale. Supponiamo di voler trasformare ogni vertice geometrico componente le strade della toscana in una feature puntuale (che eredita gli attributi della strada da cui ha origine). Il listato 6.23 mostra un esempio di codice che fa utilizzo della funzione *ST_DumpPoints*.

La sintassi di utilizzo della funzione è particolare perchè *ST_DumpPoints* è una funzione molto speciale: fa parte della classe di funzioni con risultato "multi-riga". In altre parole questa funzione genera più righe di risultato per ogni singola geometria in ingresso: per ogni singola strada della tabella di partenza, la funzione *ST_DumpPoints* genera una riga di risultato per ogni vertice contenuto nella strada stessa. Le partentesi intorno al nome della funzione, che sono obbligatorie, indicano che la funzione è multi-riga. Rimane da spiegare ancora la sintassi *.geom* dopo le parentesi: per complicare le cose la nostra funzione *ST_DumpPoints* non è solo multi-riga, ma ritorna per ogni risultato un oggetto composto da due parti: la prima componente (*geom*) contiene la geometria del vertice, la seconda componente (*path*) contiene le indicazioni della posizione del vertice all'interno della geometria originale.

Bisogna notare che l'attributo *gid* (indice della strada originale) non è più univoco nel risultato, dato che viene ripetuto per ogni vertice di una singola strada. Per questo motivo, ad esempio, non potrebbe più essere utilizzato come chiave primaria.

La figura 6.13 mostra graficamente il risultato dell'operazione.

```
1 CREATE TABLE lavoro.t_nodi AS
2 SELECT gid,
3        type,
4        (ST_DumpPoints(geom)).geom AS geom
5 FROM   osm.toscana_highway
```

Listato 6.23: *Feature Vertices To Points*: estrazione dei vertici delle geometrie.

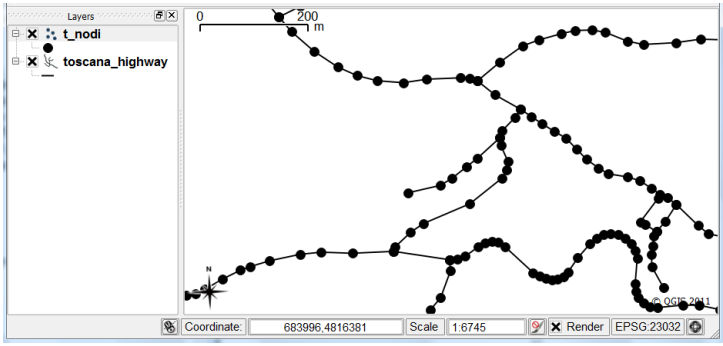


Figura 6.13: Esempio di *Feature Vertices To Points*: i vertici componenti le strade diventano feature puntuali.

6.4 Funzioni avanzate

Finiamo il capitolo con una serie di funzioni avanzate. Queste funzioni sono spesso realizzare con una serie di query.

6.4.1 Align Marker To Stroke

Con questa operazione si vuole allineare la simbologia di un oggetto puntuale a quella del più prossimo oggetto lineare di una determiante feature. Supponiamo di voler visualizzare i *Point of Interserts* della tipologia *Automotive* con un simbolo direzionale allineato alla più vicina strada. *QGIS* ha la possibilità di ruotare un simbolo al variare del valore di un attributo numerico, che

esprime l'angolo di rotazione in gradi sessadecimali (Style > Advanced > Rotation Field): il nostro scopo è quello di creare un nuovo attributo *angle* nella tabella *toscana_poi*, per poi calcolarci l'angolo di orientamento. Il codice del listato 6.24 riesce in questo intento! Analizziamo la query interna (fra parentesi) che calcola effettivamente l'angolo: l'idea generale è:

- preso in considerazione il *poi* (=Point of Interest) dato, cercare la strada più vicina, limitandosi ad una massima distanza di ricerca di 0.005 radianti;
- trovata la strada più vicina, estrarne il punto più vicino al *poi* attraverso la funzione *ST_ClosestPoint*;
- calcolare l'orientamento relativo del segmento che congiunge il *poi* in esame con il punto più vicino della strada, utilizzando l'apposita funzione *ST_Azimuth*;
- il risultato di *ST_Azimuth* è in radianti, quindi bisogna convertirlo in gradi sessadecimali moltiplicandolo per 180 e divindendolo per π.

Il filtro *ST_Expand(ST_Envelope ...* è inserito per motivi di efficienza (vedi i commenti al listato 6.22). La combinazione di *ORDER BY ST_Distance ...* con la dicitura *SELECT DISTINCT ON(osm.toscana_poi.gid)* serve per estrarre da tutte le strade vicine al punto dato, quella di minore distanza: infatti una volta ordinate le strade, fra tutte le righe del risultato ne viene estratta una sola per ogni punto, selezionando quindi la strada più vicina[5]. Attenzione ai parametri di *ST_ClosestPoint* che devono essere nell'ordine corretto: prima la geometria da cui si vuole estrarre il punto più vicino, poi il punto di riferimento.

La figura 6.14 mostra il risultato del calcolo.

[5]Ma perchè invece di ripetere tutte le volte il nome della tabella *osm.toscana_poi* non abbiamo definito un soprannome come facciamo sempre? Perchè il comando *UPDATE* non permette di definire soprannomi.

```
1  ALTER TABLE osm.toscana_poi
2  ADD angle FLOAT;
3
4  UPDATE osm.toscana_poi
5  SET ANGLE =
6  ( SELECT DISTINCT ON(osm.toscana_poi.gid)
7          ST_Azimuth(
8              osm.toscana_poi.geom,
9              ST_ClosestPoint(h.geom,
10                                 osm.toscana_poi.geom)
11         ) * 180 / pi()
12   FROM   osm.toscana_highway h
13   WHERE  ST_Distance(h.geom,osm.toscana_poi.geom)<0.0005
14   AND    ST_Intersects(
15             ST_Expand(ST_Envelope(osm.toscana_poi.geom)
16                       ,0.0005+0.0001),
17             h.geom
18         )
19   ORDER  BY osm.toscana_poi.gid,
20         ST_Distance(h.geom,osm.toscana_poi.geom)
21 )
22 WHERE  category = 'Automotive';
```

Listato 6.24: Allinemanento della simbologia alla linea più vicina.

Un inciso: il codice 6.24 è bello, funzionale e efficiente! La figura 6.16 mostra il piano di esecuzione che utilizza correttamente l'indice spaziale *toscana_highway_geom_gist*.

6.4.2 Unsplit Line

Con *Unsplit Line* si intende l'operazione di collegamento di oggetti lineare che condividono i nodi di partenza o di fine. Supponiamo di voler semplificare il grafo stradale elinimando le interruzioni stradali inutili, quelle cioe' in cui incidono solo due strade, senza cambio di attributi.

L'operazione non è affatto banale e richiede alcuni passi intermedi. Per prima cosa costruiamo una tabella ausiliaria *nodi* che

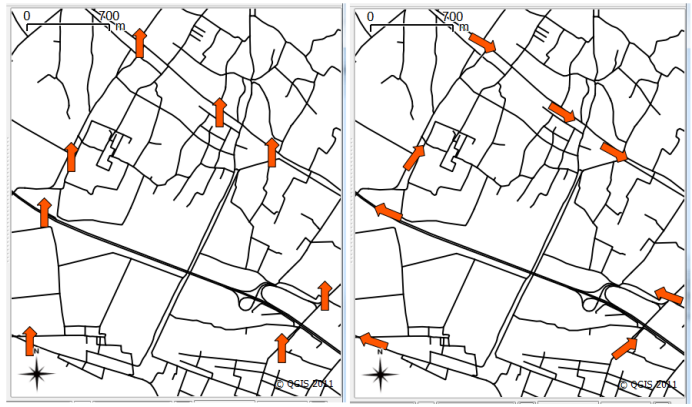


Figura 6.14: A sinistra i simboli stradali senza allinemento, a destra con allinemento automatico con la strada più vicina.

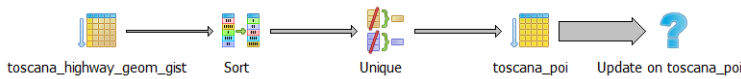


Figura 6.15: Calcolo dell'allinemento: piano di esecuzione della query che fa uso dell'indice spaziale.

conterrà i punti di inizio e fine di tutte le strade, nella forma di feature puntuale. Questa tabella memorizzerà inoltre il codice identificativo *gid* della strada di appartenenza. Il listato 6.25 realizza questa operazione.

```
1 CREATE TABLE lavoro.nodi
2 (
3   gid integer NOT NULL
4 );
5
6 SELECT AddGeometryColumn('lavoro','nodi','geom',
7         4326,'POINT',2);
```

```
8
9  INSERT INTO lavoro.nodi
10 SELECT gid,ST_StartPoint(geom)
11 FROM   osm.toscana_highway
12 UNION
13 SELECT gid,ST_EndPoint(geom)
14 FROM   osm.toscana_highway;
```

Listato 6.25: *Unsplit Line* fase 1: costruzione della tabella dei nodi (punti di inizio/fine).

Dopo aver creato la tabella *lavoro.nodi*, abbiamo estratto gli estremi dei tratti stradali attraverso le funzioni *ST_StartPoint* e *ST_EndPoint*. Il comando *UNION* unisce in una sola tabella i punti iniziali e quelli finali.

A questo punto raggruppiamo i vari nodi di inizio fine coincidenti geometricamente, stando attenti a selezionare solo quelli in cui incidono due e due sole strade; gli incroci di tre o più strade infatti non vanno unificati! Il listato 6.26 attua l'operazione creando la nuova tabella *coppie*. Questa tabella memorizza gli identicativi delle due strade incidenti (*min(gid), max(gid)*), raggruppando i nodi coincidenti (*GROUP BY geom*) e selezionando gli incroci con sole due strade (*HAVING count(*)=2*).

```
1  CREATE TABLE lavoro.coppie AS
2  SELECT min(gid) AS a,
3         max(gid) AS b
4  FROM   lavoro.nodi
5  GROUP  BY geom
6  HAVING count(*)=2;
```

Listato 6.26: *Unsplit Line* fase 2: creazione dei nodi e lista delle coppie da unificare.

La tabella *coppie* contiene già i codici delle strade da unificare. Abbiamo finito? No perchè in realtà una strada potrebbe essere interrotta in più di un punto, cioè potrebbe essere formata da tre o più segmenti da unificare. Quello che dobbiamo ottenere a questo

punto è la cosiddetta *chiusura transitiva* dell'operazione di unione; vale a dire che se dobbiamo unire il segmento A al segemento B e il segmento B al segmento C, allora dobbiamo anche unire il segmento A al segmento C, anche se questi ultimi due segmenti non sono adiacenti.

Per ottenere la chiusura transitiva della realzione definita in *coppie* dobbiamo utilizzare un costrutto avanzato di *SQL*, che corrisponde a *WITH RECURSIVE*; con questo costrutto possiamo definire la tabella virtuale *chiusura* in termini si sé stessa! Analizziamo il listato 6.27:

```
1  CREATE TABLE lavoro.gruppi_nodi AS
2  WITH RECURSIVE chiusura(a,b) AS (
3      SELECT DISTINCT a, a FROM   lavoro.coppie
4    UNION
5      SELECT DISTINCT b, b FROM   lavoro.coppie
6    UNION
7      SELECT a, b FROM   lavoro.coppie
8    UNION
9      SELECT c.a, k.b
10     FROM   lavoro.coppie AS c,
11            chiusura AS k
12     WHERE  c.b=k.a
13 )
14 SELECT n.n, max(k.b) as grp_id
15 FROM   chiusura AS k,
16        ( SELECT DISTINCT a AS n FROM   lavoro.coppie
17          UNION
18          SELECT DISTINCT b AS n FROM   lavoro.coppie
19        ) AS n
20 WHERE  n.n = k.a
21 GROUP  BY n.n;
```

Listato 6.27: *Unsplit Line* fase 3: *chiusura transitiva* dei sottogruppi di segmenti da unificare, calcolo dell'identificativo di gruppo come codice massimo fra le strade presenti.

Lo scopo di questo listato è quello di raggruppare i vari segmenti di strada da unire insieme. Per prima cosa calcoliamo la chiusura

transitiva (e simmetrica) della relazione *coppie*. Le prime due componenti della definizione (*SELECT DISTICT a,a ... SELECT DISTINCT b,b ...*) realizzano la chiusura simmetrica, vale a dire che ogni segmento deve essere per prima cosa unito con sé stesso. La terza componente copia l'intera tabella *lavoro.coppie*, mentre l'ultima realizza la vera e propria chiusura transitiva, essendo definita ricorsivamente sulla stessa tabella *chiusura*.

La parte finale del listato infine, basandosi sulla relazione *chiusura* che ora è completa, calcola per ogni codice di strada da unificare, il massimo fra i codici dei segmenti dello stesso gruppo (*max(k,b)...*), elegendolo a rappresentante unico del gruppo (il quale viene battezzato come *grpd_id*).

Siamo arrivati alla conclusione del percorso. Il listato 6.28 realizza finalmente la tanto agognata unificazione. Raggruppando fra loro i segmenti da unire secondo l'identificatore del gruppo (*GROUP BY g.grp_id*), le geometrie vegono per prima unite con la funzione *ST_Union*, in seguito applichiamo la funzione *ST_LineMerge* che esegue l'unificazione vera e propria.

```
1 CREATE TABLE lavoro.strade_intere AS
2 SELECT g.grp_id,
3        ST_LineMerge( ST_Union(h.geom) ) AS geom
4 FROM   osm.toscana_highway AS h,
5        lavoro.gruppi_nodi  AS g
6 WHERE  h.gid = g.n
7 GROUP  BY g.grp_id;
```

Listato 6.28: *Unsplit Line* fase 4: unificazione geometrica

La figura 6.16 mostra un esempio del risultato.

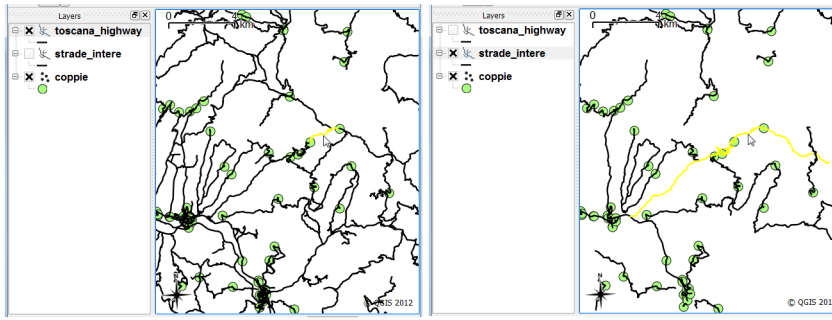


Figura 6.16: *Unsplit Line*: a sinistra l'operazione di selezione sulle strade originali (segmentate). A destra la stessa operazione di selezione sulle strade riunite. I cerchi indicano le posizione delle operazioni di congiunzione.

7 Linear Referencing

Con il termine *Linear Referencing* si intende quella serie di tecniche di posizionamento geografico che non fanno uso di coordinate assolute, ma di posizioni relative lungo un percorso lineare. Si pensi ad esempio al posizionamento di uscite ed aree di sosta lungo un percorso autostradale; dal punto di vista dell'automobilista non è importante sapere la posizione in latitudine e longitudine dell'uscita o dell'autogrill, quanto piuttosto a quanti chilometri di distanza si trovano questi luoghi.

Nel *Linear Referencing* la posizione dei vari oggetti (numeri civici, cippi chilometrici, etc.) viene memorizzata come distanza relativa lungo un percorso lineare (di solito curvilineo). Di seguito introdurremo alcune funzionalità di PostGIS sul *Linear Referencing*, applicando le funzioni a casi reali.

7.1 Preparazione dei Dati di Esempio

Dato che lavoreremo su distanze chilometriche, rimane comodo avere i dati memorizzati in coordinate piane (piuttosto che giografiche). Per questo creiamo la tabella *lavoro.strade_utm* che conterrà le strade in coordinate piane UTM - WGS84, fuso 32 Nord (listato 7.1).

```
1 CREATE TABLE lavoro.strade_utm AS
2 SELECT gid,type,name,oneway,lanes,
3        ST_Transform(geom,32632) as geom
4 FROM   osm.toscana_highway;
```

Listato 7.1: Preparazione dei dati in coordinate piane.

Per i nostri esempi prenderemo in esame il tratto di autostrada corrispondente al GID numero 106933[1].

7.2 Posizionamento di Distanza Relative

Per il tratto di autostrada in questione, supponene di aver memorizzato i dati di alcuni punti di interesse, i dati vengono memorizzati nella tabella *lavoro.autostrada*, come mostrato nel listato 7.2[2].

```
1  CREATE TABLE lavoro.autostrada
2  (
3    id INTEGER,
4    posizione FLOAT,
5    descr CHARACTER VARYING
6  );
7  INSERT INTO lavoro.autostrada
8    VALUES(1, 5100,'AutoGrill␣Canistracci');
9  INSERT INTO lavoro.autostrada
10   VALUES(2,20200,'Area␣Sosta␣I␣Lupi');
11 INSERT INTO lavoro.autostrada
12   VALUES(3,30000,'Uscita␣Ristonchi');
13 INSERT INTO lavoro.autostrada
14   VALUES(4,35000,'Uscita␣Troghi');
```

Listato 7.2: Dati chilometrici di esempio.

La posizione è rappresentata come distanza (in metri) lungo il tratto autostradale in questione. Non si conosce invece la posizione geografica assoluta.

Per poter visualizzare i punti di interesse come posizioni geografiche facciamo uso della funzione *ST_Line_Interpolate_Point*; questa funzione, data una geometria lineare ed una distanza lungo

[1] Per il vostro esempio cercate il tratto di autostrada (type='motorway') più lungo.

[2] I dati sono puramente di fantasia!

la linea, ne calcola la posizione corrispondente. Il listato 7.3 mostra un esempio di creazione dei punto associati alle distanze.

```
1  CREATE TABLE lavoro.autostrada_pt AS
2  SELECT a.id,
3         a.descr,
4         ST_Line_Interpolate_Point(
5           ST_GeometryN(s.geom,1),
6           a.posizione/ST_Length(s.geom)
7         )
8  FROM   lavoro.autostrada AS a,
9         lavoro.strade_utm AS s
10 WHERE  s.gid=106933;
```

Listato 7.3: Posizionamento di oggetti tramite la distanza.

Vediamo alcuni dettagli: la query è scritta come una *join* fra la tabella delle strade e quella dei punti autostradali; fra tutte le strade selezioniamo la numero 106933 (quella che ci interessa). Il secondo parametro della funzione di interpolazione in realtà non accetta distanze metriche, ma un numero da zero ad uno, che corrisponde in proporzione ad un punto lungo la strada. Per otterene il valore giusto bisogna quindi dividere il valore *posizione* per la lughezza dell'intera strada (*ST_Length(s.geom)*).

Con le impostazioni di default, l'importatore dei file shape carica i dati lineari come *oggetti multipli* anche se sono semplici linee; per questo applichiamo la funzione *ST_GeometryN* con parametro 1, che estrae la prima componente geometrica da una geometria complessa. Se nel vostro caso le geometrie delle strade sono già semplici, è possibile utilizzare direttamente il valore *s.geom*.

A questo punto il gioco è fatto: copiamo gli attributi *a.id* e *a.descr* della tabella *lavoro.autostrada* e calcoliamo la posizione geografica al variare dell'attributo *posizione*. La figura 7.1 mostra il risultato dell'operazione.

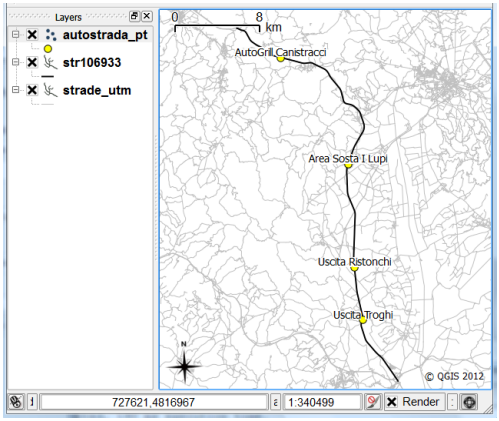


Figura 7.1: Posizionamento di Distanza Relative.

7.3 Localizzazione di un Punto

Supponiamo adesso di essere nel caso inverso: abbiamo la posizione geografica di una serie di oggetti interessanti (*osm.toscana_poi*) e vogliamo trovare il loro chilometraggio relativo sul nostro tratto di autostrada. Il listato 7.4 fa al nostro caso:

```
1  SELECT  p.category,
2          p.name,
3          ST_Line_Locate_Point(
4           ST_GeometryN(s.geom,1),
5           ST_Transform(p.geom,32632)
6          )*ST_Length(s.geom) AS distanza
7  FROM    osm.toscana_poi AS p,
8          lavoro.strade_utm AS s
9  WHERE   s.gid=106933
10 AND     ST_Distance(s.geom,ST_Transform(p.geom,32632))
11                 <500
12 ORDER   BY distanza;
```

Listato 7.4: Recupero della distanza lineare di oggetti.

Questa volta facciamo uso della funzione *ST_Line_Locate_Point* che data una geometria lineare ed un punto di coordinate note, calcola il chilometraggio relativo del punto sulla linea. I punti presi in considerazione sono solo quelli che si trovano entro 500 metri dal nostro tratto di autostrada (penultima riga del listato). La tabella *osm.toscana_poi* è in coordinate geografiche quindi va trasformata in piane. Come la funzione precedente il risultato non è in metri ma è costitutito da un numero da zero a uno, proporzionale alla lunghezza del tratto stradale; quindi questa volta dobbiamo moltiplicare il valore per *ST_Length(s.geom)*.

Infine, il risultato della query (di seguito) è ordinato per distanza (ultima riga del listato): la distanza è in metri[3].

```
        category        |       name       |  distanza
------------------------+------------------+------------
Government and Public S|Railway Laterina  |2993.0645806
Automotive             |Parking           |3207.7980622
Automotive             |Parking           |7488.7834689
Automotive             |Parking           |7510.8367238
Government and Public S|Railway Ponticino|7852.1104979
Automotive             |Fuel:Erg          | 19044.22109
Automotive             |Fuel:Total        | 19268.48095
Automotive             |Fuel              |37249.187914
Automotive             |Fuel:Shell        |37407.848525
Leisure                |Fountain          |40490.573766
Automotive             |Parking           |40512.975472
Eating&Drinking        |Restaurant:Apogeo|41837.618914
Automotive             |Parking           |41895.312153
```

[3]Accidenti! La prossima pompa di benzina è a quasi 20 chilometri! 19044 metri e 22 centimetri per la precisione.

Elenco delle tabelle

Elenco delle figure

Listati