Breve Introduzione a SQL (con Postgres)

Claudio Rocchini
Istituto Geografico Militare
rockini@tele2.it

Gennaio 2010

Indice

1 Introduzione

SQL (sigla che sta per Structured Query Language) é un linguaggio testuale standard per operare con le basi di dati. Standard vuol dire che é (quasi) indipendente la particolare database scelto (*Oracle, Microsoft SQL Server, Postgres, Mysql*, etc.). Il linguaggio é 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 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 Postgres

Tutte le esercitazioni verranno effettuate sul database *Postgres*, un database gratuito che ha un supporto spaziale molto ben sviluppato. Un altro database che ha un ottimo supporto spaziale é *Oracle*, che peró é un prodotto commerciale ed anche piuttosto caro.

Saltiamo la fase di installazione del software che non ci interessa e passiamo direttamente al suo utilizzo. Per accedere al database utilizzeremo l'interfaccia base, denominata *pgAdmin III*, mentre per visualizzare i dati geografici in modo grafico occorrerá utilizzare un qualche GIS (es. QGis, UDig, Grass, etc.).

Nota: Per scopi dididattici, é opportuno che ciascuno abbia installato il proprio server Postgres personale. In realtá l'utilizzo tipico di Postgres come database enterprise é quello di installare un server unico, in modo tale che tutti gli utenti operino su dati condivisi ed in modo concorrente. Si é deciso un'installazione separata per ciascuno per evitare di dover fare una serie di utenti diversi, questo per evitare conflitti con la creazione delle tabelle (per ogni database deve esistere una sola tabella per ogni nome).

3 Prepararsi al Lavoro

Lanciate Postgres - pgAdmin III: vi apparirá l a finestra in Fig. 1. Se non avete la configurazione del server che vi serve, selezionate il menú File-Add Server, altrimenti cliccate su *PostgreSQL 8.4*, che é il vostro server locale. Se avete impostato una password durante l'installazione, questa vi verrá richiesta. Nella parte sinistra della finestra viene visualizzato un

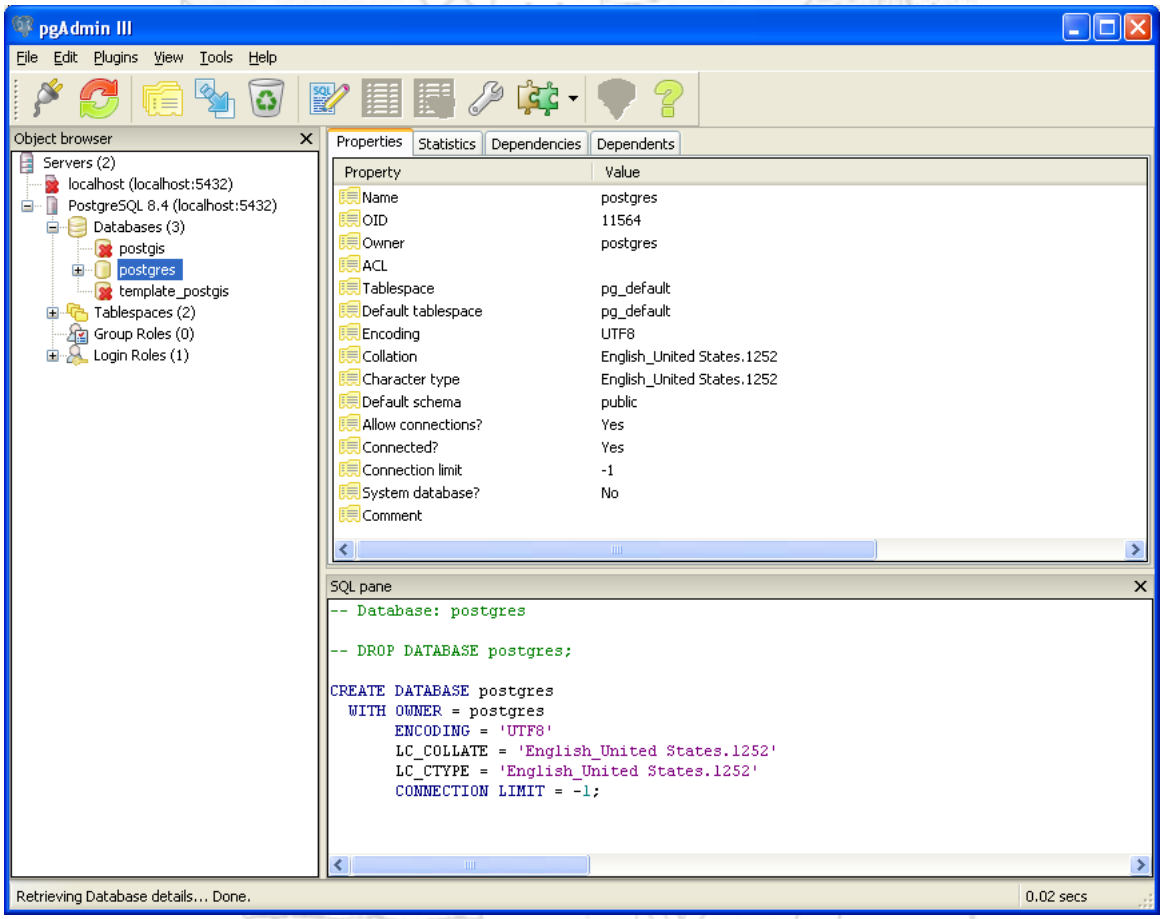

Figura 1: La schermata di avvio di pgAdmin III.

albero con tutti gli oggetti presenti sul server. Per prima cosa l'elenco dei database presenti (ogni server ne puó 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 piú tablespace per aumerare l'efficienza. Questo argomento esula dagli scopi di questa esercitazione. 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 giá alcuni database, ma per i nostri scopi ne creiamo uno ad hoc. Cliccate col *bottone destro* sulla casella databases nell'albero e selezionate

il menú *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 nostro db memorizzerá i testi utilizzando questa codifica di carattere (non abbiamo tempo per soffermarci su questo argomento). Inoltre selezionate nella casella *template* il valore *postgis_template*. Se questa scelta non é presente vuol dire che non avete installato il supporto spaziale. Per ora questa scelta é un poco oscura; in seguito si vedrá che questa opzione abilita il supporto geografico al database appena creato. Una volta creato il nostro database, questo apparirá nell'albero;

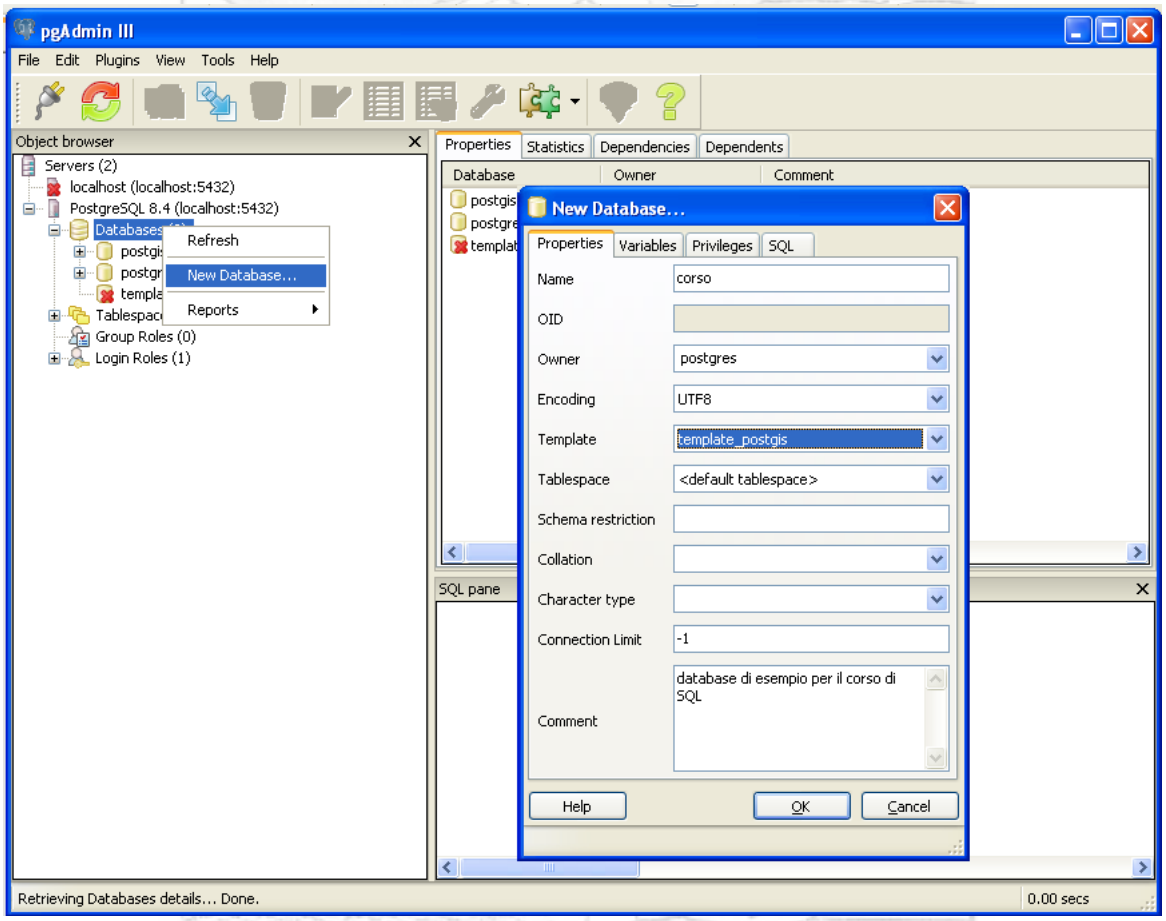

Figura 2: Dialogo di creazione di un nuovo db.

cliccateci sopra ed aprite gli oggetti contenuti. C'é un sacco di roba... ma nieste paura, a noi interessano solo poche cose. Innazzitutto vedete la casella *Schemas*; questi sono gli schemi in cui un db puó 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 é possibile effettuare delle operazioni tramite interfaccia grafica; ad esempio cliccando su *tables* é possibile creare nuove tabelle (Fig. 3). Noi non utilizzeremo mai questa opzione durante l'esercitazione, ma effettueremo

ogni operazione tramite il linguaggio SQL. Questo perché tale linguaggio é standard per tutti gli altri software di database.

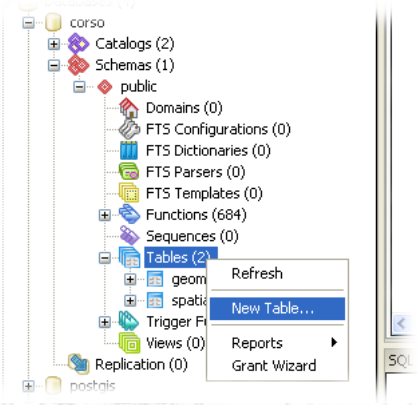

Figura 3: Contenuto del db ed interfaccia.

4 La finestra di comandi SQL

Una volta che avete selezionato il database *corso* cliccate sullo strumento *SQL*, rappresentato dall'icona con la matita e il foglio con su scritto SQL. Si aprirá la finestra col editor SQL (Fig. 4) Nella parte sinistra della finistra c'é l'editor vero e proprio, nella parte destra c'é una

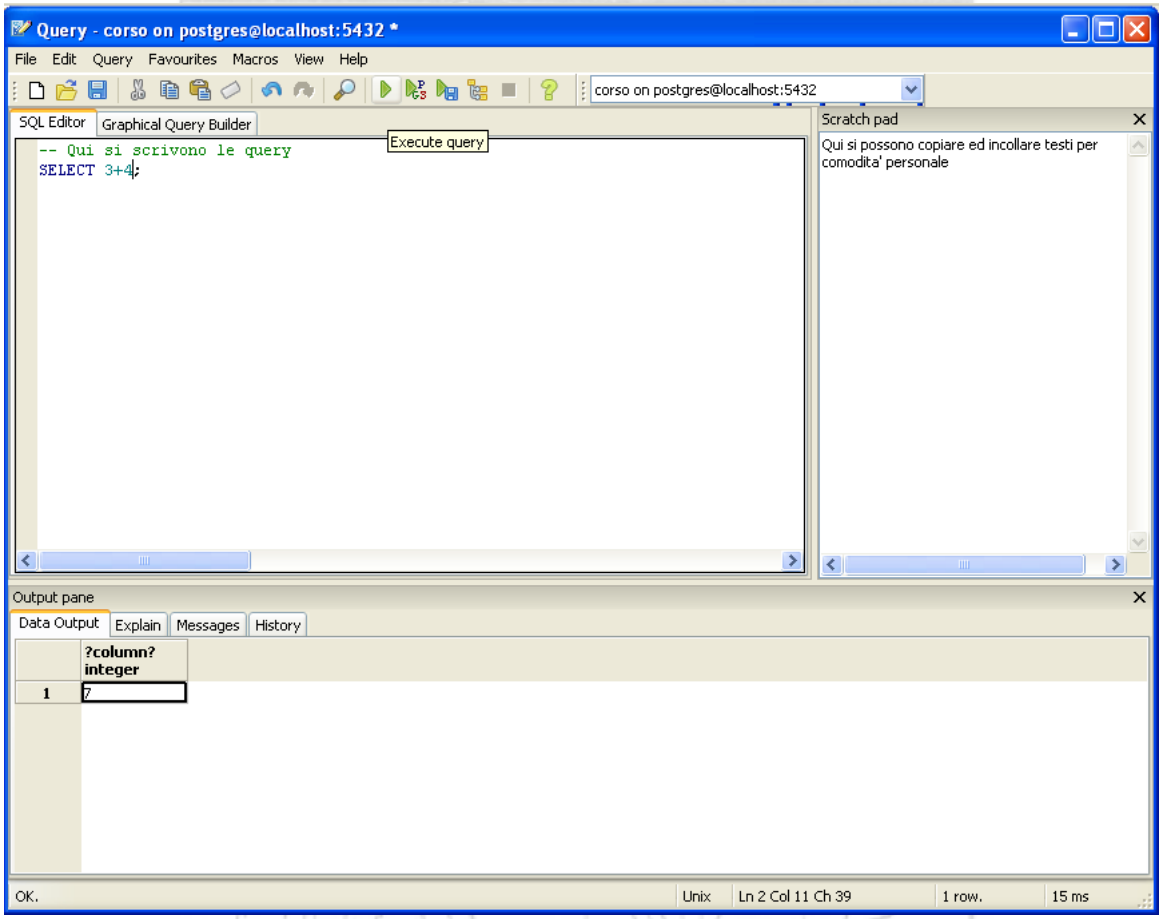

Figura 4: Finestra con editor SQL.

specie di notepad, utile per copiare ed incollare testi. Nella parte sottostante invece vengono visulizzati i risultati delle operazioni ed altre cose.

L'utilizzo della finestra é 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 rigiditá di un linguaggio formale per computer, l'approccio iniziale sará molto duro. La sintassi SQL deve essere esatta: ricordatevi di non inserire spazi all'interno 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 é 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: se si vuole specificare un nome (di tabella o di colonna) in maiuscolo/minuscolo in modo specifico, é necessario racchiudere il nome fra doppie virgolette. In generale i comandi SQL saranno scritti su piú righe e con opportuna indentatura: questa suddivisione viene fatta solo per chiarezza, dato che in SQL la divisione in righe non é significativa ai fini dell'interpretazione della query.

5 Pre-Introduzione al comando SELECT

Il comando fondamentale di SQL é *SELECT* e verrá spiegato in dettaglio piú avanti. Dobbiamo peró introdurlo per effettuare alcune prove sui dati: questo comando infatti ci permetterá di visualizzare le operazioni effettuate sui tipi di dati di base. La sintassi minima del comando *SELECT* é:

```
SELECT {valori};
```

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

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

```
SELECT 42;
```

quindi premete il pulsante *Execute* Il risultato visualizzato in basso sará 42. La query che abbiamo scritto richiede infatti al sistema il numero 42; una query non molto intelligente per ora, ma miglioreremo in futuro (nota curiosa: in Oracle é obbligatorio specificare sempre una tabella anche se non serve, a questo scopo esiste sempre una tabella fittizia che si chiama *DUAL* da cui é possibile selezionare tutto).

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

```
SELECT 42 AS valore;
```

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

6 Valori letterali

Per valori *letterali* si intendono valori costanti dati, come numeri o parole. Come in molti linguaggi di programmazione, in SQL é possibile operare con i numeri interi e con la virgola (che si scrive punto); é inoltre possibile calcolare espressioni o chiamare funzioni. Provate ad eseguire:

```
SELECT 21*2;
```

Oppure per i matematici:

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

Provate ad indovinare quali sono i risultati di queste query.

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

```
SELECT 'Buongiorno';
```

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

```
SELECT Buongiorno;
```

avrei ottenuto un errore da Postgres: il sistema infatti non riesce a trovare una colonna che si chiama *Buongiorno*. 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:

```
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:

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

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

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

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

```
SELECT 99 < 100;
```

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

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

la risposta é falso questo perché 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 dá il risultato atteso (il secondo parametro della funzione specifica la formattazione del numero):

```
SELECT SELECT TO_NUMBER('42','9999')+1;
```

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.

7 Tipi di dato

I dati memorizzati nelle tabelle di un database appartengono ad un tipo. Il concetto di tipo di dato é 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 veritá (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 andrá 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 veritá (vero o falso), curiosamente Oracle non ha questo tipo di dato;
- 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 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 é questo il luogo per approfondire l'argomento dovremmo introdurre parzialmente questo argomento per poter descrivere la componente spaziale di Postgres: infatti il tipo di dato *GEOMETRY* di Postgres, che difinisce la componente spaziale di un'entitá, é un *oggetto* vale a dire che é un tipo di dato complesso (nel caso di Oracle invece é addirittura un *oggetto*.

8 Definizione dei Dati

Iniziamo adesso il corso vero e proprio di SQL. 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 4:

- CREATE TABLE : crea una tabella
- DROP TABLE: distrugge una tabella
- ALTER TABLE: modifica una tabella

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:

```
CREATE TABLE nome_tabella
(
    nome_colonna1 TIPO1,
    nome_colonna2 TIPO2,
    ...
    nome_colonnan TIPOn
);
```

All'interno delle parentesi tonde che seguono il nome della tabella bisogna specificare la lista delle colonne della tabella stessa, separate da virgola (l'ultima colonna non é 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 puó usare la barra di sottolineato _ . Non c'é differenza fra maiuscole e minuscole. Nei nomi si possono usare lettere, cifre e la barra sopra detta, anche se il nome non puó iniziare con una cifra). Provate adesso a creare la nostra prima tabella, con il comando:

```
CREATE TABLE studenti
(
    nome CHARACTER(128),
    eta  INTEGER,
    codice_corso INTEGER
);
```

Questo comando creerá la tabella studenti, formata da tre colonne: il nome dello studente (parola), la sua etá (numero intero) e il codice numerico del corso che lo studente segue (numero intero). 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. A questo punto salvate la query di creazione (se non l'avete giá fatto), tramite il bottone *SAVE*, per poterla recuperare in seguito.

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

```
CREATE TABLE studenti(nome CHARACTER(128), eta
INTEGER, codice_corso INTEGER);
```

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 *studenti*; l'interfaccia ad albero ci mostra gli elementi contenuti nella

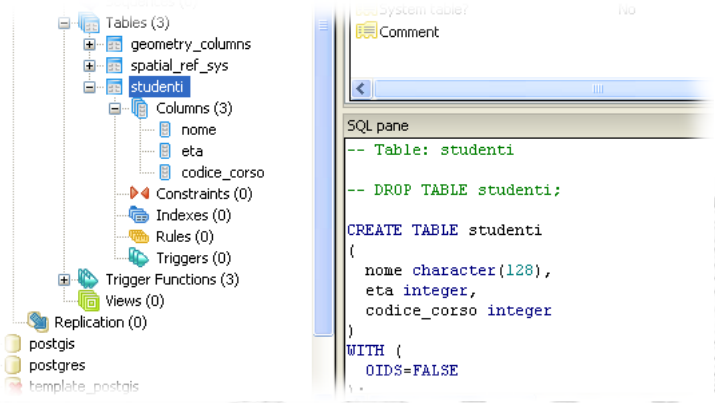

Figura 5: Visualizzazione della struttura di una tabella.

struttura della tabella (ma non i dati). A destra, dentro *SQL Pane* é possibile visualizzare il comando SQL che ha generato la tabella stessa (Fig. 7). L'analisi di struttura delle tabelle puó 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.

8.3 Distruzione di una tabella

Per distruggere definitivamente una tabella, si utilizza il comando *DROP*. Adesso l'esercitazione prevede la distruzione della tabella appena creata (la rifaremo meglio dopo), prima di tutto peró 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 *studenti*, con il comando:

```
DROP TABLE studenti;
```

Il comando distrugge per sempre la tabella (attenzione ad usarlo con cognizione di causa), non c'é 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'é modo di recuperare una tabella distrutta, a meno che non si sia iniziata una *TRANSAZIONE* (di cui non parleremo in questo corso).

8.4 Un inciso: commenti al codice

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

```
CREATE TABLE studenti
(
    nome CHARACTER(128),  -- Nome dipendente (questo testo viene ignorato)
    /* Altri dati dello
       Studente (questo testo viene ignorato) */
    eta INTEGER,
codice_corso INTEGER
);
```

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.

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;
- campo obbligatorio

Ricordiamo che la colonna *chiave primaria* specifica il dato (o i dati) che identificano univocamente ogni riga della tabella, mentre un campo é obbligatorio se il suo valore é sempre 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 é il seguente:

```
CREATE TABLE studenti
(
    nome CHARACTER (128) PRIMARY KEY,
    eta INTEGER,
    codice_corso INTEGER NOT NULL
);
```

La specifica *PRIMARY KEY* indica che il campo *nome* é quello che identifica univocamente le righe della tabella. La specifica *NOT NULL* indica il campo *codice_corso* é obbligatorio e non puó rimanere vuoto durante l'inserimento delle righe della tabella (vale a dire che non puó 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 segnalerá 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.

8.6 Modifica della struttura di una tabella

La struttura delle tabelle puó 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 é stata

creata, le modifiche dinamiche alla sua struttura sono possibili tramite il comando *ALTER TABLE*, ad esempio aggiungiamo la colonna *professione* alla nostra tabella:

```
ALTER TABLE studenti
    ADD professione CHARACTER(128);
```

Di solito i comandi SQL sono molto chiari (sono auto-esplicativi): questo comando modifica la tabella *studenti* aggiungendo il campo *professione*, che é una parola di (al massimo) 128 caratteri. Provate a visualizzare la tabella per controllare l'effetivo cambio di struttura.

Allo stesso modo possiamo cancellare una colonna con il comando *DROP COLUMN* (non necessariamente l'ultima), provate ad eseguire:

```
ALTER TABLE studenti
    DROP COLUMN professione;
```

il comando avrá l'ovvio effetto che vi aspettate. Se la tabella contenesse giá dei dati, le operazioni di modifica della struttura possono essere eseguite comunque. I dati delle colonne non interessati dalle modifiche di struttura verranno conservati.

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 di una tabella;
- UPDATE: modifica i dati esistenti delle righe di una tabella.

9.1 Inserimento di dati

Il comando *INSERT* server per inserire nuove righe in una tabella. La struttura del comando *INSERT* é la seguente:

```
INSERT INTO nome_tabella
 (nome_colonna1, nome_colonna2, ... , nome_colonnaN)
 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 studenti; come dati dobbiamo specificare per ogni studente il nome, l'etá e il codice numerico del corso:

```
INSERT INTO studenti
    (nome,eta,codice_corso)
    VALUES ('Claudio Rocchini', 43, 1);
```

Notate che il nome é una parola, quindi va fra apici, mentre l'etá (43) e il codice del corso (1) sono numeri, quindi sono senza apicetti. I termini *studenti,nome,eta,...* sono i nomi delle tabelle e delle colonne SQL e quindi vanno scritti anche loro senza apicetti. Notate anche le virgole, che separano colonne e valori: ovviamente dopo l'ultimo valore (il numero 1) la virgola non ci vuole. Niente panico: la sintassi á una brutta bestia, che si doma con l'esperienza.

Proviamo adesso ad inserire altri valori nella tabella (potete anche provare ad inserire i vostri dati, mettendo dei codici di corso fittizzi). In particolare proviamo ad inserire un dato incompleto:

```
INSERT INTO studenti
    (nome, codice_corso)
    VALUES ('Margherita Azzariti', 1);
```

Notate che in questo caso non abbiamo inserito l'etá (per cavalleria), che comunque non é obbligatoria (non possiede l'opzione *NOT NULL*; il campo codice corso invece é obbligatorio e va sempre specificato. Nel caso in cui invece inseriamo dati per tutte le colonne, la sintassi del comando *INSERT* puó essere semplificata omettendo la lista dei campi da inserire e specificando solo i valori, nell'ordine con cui devono essere inseriti, ad esempio:

```
INSERT INTO studenti
    VALUES ('Gianfranco Amadio', 86, 2);
```

9.2 Un'altro inciso: il valore NULL

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

```
INSERT INTO studenti
    VALUES ('Salvatore Arca', NULL, 2);
```

Intendendo che il campo etá (il secondo valore) deve rimanere nullo e quindi vuoto. Il valore *NULL* sará particolarmente utile nei controlli, che vedremo in seguito.

9.3 Test dei vincoli

Nella tabella che abbiamo inserito ci sono due vincoli: la chiave primaria e l'obbligatorietá del campo *codice_corso*. Se proviamo ad inserire una nuova riga con un nome di studente duplicato, violiamo il vincolo di chiave primaria ed il database ci comunicherá l'errore; proviamo ad eseguire il comando:

```
INSERT INTO studenti
(nome,eta,codice_corso)
VALUES ('Claudio Rocchini', 16, 1);
```

Otteniamo un errore del tipo (scritto in inglese): una chiave duplicata viola il vincolo di unicitá. La chiave primaria infatti deve essere unica, mentre noi abbiamo tentato di inserire due studenti diversi con lo stesso nome.

Ricordiamo che il campo *eta* non é obbligatorio, mentre é obbligatorio il campo *codice_corso* (vale a dire che possiede l'opzione *NOT NULL*). Proviamo ad eseguire il seguente comando, per inserire uno studente di cui non conosciamo il codice del corso:

```
INSERT INTO studenti
    (nome, eta)
    VALUES ('Fantomas', 42);
```

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

9.4 Cancellazione di dati

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

```
DELETE FROM studenti;
```

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

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

dove le *condizioni* specificate dopo il termine WHERE filtrano le righe da cancellare effettivamente. La specifica di una condizione permette di eliminare solo quelle righe che rispettano la condizione specificata, ad esempio se vogliamo eliminare dalla tabella gli studenti che hanno 20 anni scriviamo:

```
DELETE FROM studenti
WHERE eta=20;
```

Il comando cancellerá (se ci sono) tutte gli studenti che hanno l'etá uguale a 20. Nella condizione é possibile scrivere espressioni, controllare le colonne, eseguire confronti di uguaglianza (=), diversitá (<>), confronti di quantitá (< e >), 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 esempi, il filtro:

```
...
WHERE eta<40 AND codice_corso=1
```

identifica tutti gli studenti che hanno meno di 40 anni *E* seguono il corso numero 1. Il filtro:

```
...
WHERE eta>40 OR NOT codice_corso=2
```

identifica tutti gli studenti che hanno piú di 40 anni OPPURE NON seguono il corso numero due. Per le parole si possono usare i confronti di uguaglianza, ma anche < e >, intesi come ordine alfabetico (es. 'abaco' < 'zuzzurellone'). L'operatore *LIKE* invece permette di eseguire confronti fra parole facendo utilizzo di caratteri jolly, il filtro:

```
...
    WHERE nome LIKE 'Cla%';
```

identifica tutti gli studenti il cui nome inizia per *Cla*: il simbolo % sta ad indicare qualsiasi sequenza di lettere.

9.5 Modifica dei dati

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

```
UPDATE nome_tabella
SET nome_colonna = valore
WHERE {condizioni};
```

dove la definizione delle condizioni é del tutto uguale a quella del comando *DELETE*. Proviamo adesso a cambiare il codice corso di qualcuno, eseguiamo la query:

```
UPDATE studenti
SET codice_corso = 2
WHERE nome='Claudio Rocchini';
```

; 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 studente indicato, verrá cambiato dil codice del corso da 1 a 2. Se non si specifica la condizione, il comando *UPDATE* modifica TUTTE le righe della tabella, settando un valore costante sulla colonna indicata.

La dicitura *IS NULL* puó essere utilizzato nei controlli per determinare la presenza di valori nulli; se vogliamo ad esempio impostare l'etá di 60 anni a tutti gli studenti che non hanno indicazione di etá, scriviamo:


```
UPDATE studenti
SET eta=60
WHERE eta IS NULL;
```

In questo modo, tutte le caselle etá vuote (con valore NULL) vengono riempite con 60.

Fino ad adesso abbiamo operato su di una sola tabella. Ovviamente le basi di dati possono contenere molte tabelle. Prima di passare all'interrogazione dei dati, per rendere piú interessante il nostro database, creiamo una tabella corsi, che ci servirá per fare degli esempi di interconnessione fra tabelle, eseguiamo la query:

```
CREATE TABLE corsi
(
    codice_corso INTEGER PRIMARY KEY,
    descrizione CHARACTER(128) NOT NULL
);
```

Ormai siamo esperti: il codice del corso é la sua chiave primaria, segue una descrizione testuale obbligatoria (testo di al massimo 128 caratteri). Popoliamo adesso la tabella dei corsi:

```
INSERT INTO corsi
(codice_corso,descrizione)
VALUES (1,'Basi di Dati');
```

Ed infine un altro corso:

```
INSERT INTO corsi
(codice_corso,descrizione)
VALUES (2,'Sistemi Informativi Territ.');
```

Provate ad inserire altri corsi di fantasia. Ovviamente, dato che il codice numerico é la chiave primaria, questo deve essere unico. Perché gli esempi successivi funzionino, é importante che siano presenti tutti i codici di corso che abbiamo inserito nella tabella studenti studenti.

10 Le relazioni

Un database non é fatto solo di entitá (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. Gli studenti sono in relazione con i corsi, dato che per ogni studente abbiamo specificato un codice di corso. Le relazioni possono anche essere specificate esplicitamente con l'aggiunta di un vincolo (constraint) alla tabella.

10.1 Definizione di una relazione

La tabella *studenti* contiene logicamente una relazione con la tabella *corsi*: infatti il campo *codice_corso* della prima tabella riferisce lo stesso campo della seconda tabella. Questa relazione *logica* fra tabelle puó essere esplicitata tramite il seguente comando:

```
ALTER TABLE studenti
    ADD CONSTRAINT studenti_corso_fk
    FOREIGN KEY (codice_corso) REFERENCES corsi(codice_corso);
```

Il comando esplicita la relazione fra studenti e corsi, ed é formato da un vincolo sulla tabella studenti. Analizziamo la struttura del comando: la prima riga indica che vogliamo modificare la tabella studenti (come nel caso di aggiunta di una nuova colonna), in quest caso peró vogliamo aggiungere un vincolo (constraint in inglese): *studenti_corso_fk* é il nome di questo vincolo (fk sta per foreign key = chiave straniera e si aggiunge per convenzione). Il

vincolo afferma (nell'ultima riga del comando) che la *chiave straniera* formata dalla colonna *codice_corso* della tabella *studenti* DEVE riferire una valore (vale a dire contenere un numero di codice) della colonna *codice_corso* nella tabella *corsi*.

Se nella tabella corsi non ci sono tutti i codici necessari, la creazione della relazione sará impossibile, dato che il sistema controlla la congruenza dei dati anche durante la creazione del vincolo. Una volta che il vincolo di relazione é impostato, siamo sicuri che tutti i codici di corso seguiti dagli utenti sono presenti nella tabella dei corsi.

Ai lettori piú attenti puó dar fastidio che la relazione studenti-corsi sia rappresentata da un vincolo sulla sola tabella studenti. Perché non c'é un vincolo anche in corsi? Perché questo tipo di relazione é asimettrica (1:n). E' lo studente che segue un particolare corso, mentre ad un corso possono partecipare ovviamente molti studenti.

Tentiamo adesso di inserire uno studente che segue un corso inesistente, eseguiamo:

```
INSERT INTO studenti
(nome,codice_corso)
VALUES ('Fantomas',999);
```

Se il corso numero 999 non esiste, otteniamo un errore del tipo (in inglese) che ci informa della violazione del vincolo che si chiama *studenti_corso_fk*. La congruenza delle relazioni viene controllata dinamicamente in ogni momento, in particolare durante la modifica dei dati delle tabelle *studenti* e *corsi*. Ad esempio non é piú possibile cancellare un corso se é seguito da almeno uno studente.

11 Indici

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

Supponiamo di prevedere molte ricerche sull'etá degli studenti; inoltre supponiamo che gli studenti siano tanti (non é il caso di questo esempio). Normalmente il sistema deve scorrere l'intera tabella degli studenti per eseguire tale ricerca: se gli studenti sono tanti questa ricerca puó richiedere del tempo. Per velocizzare una ricerca del genere é possibile creare un indice. Gli indici servono per velocizzare le ricerche di valori su una (o piú) colonne di una tabella; il loro funzionamento é simile agli indici (o megli agli indici analitici) dei libri. Per creare un indice sulla colonna etá della tabella studenti, eseguiamo il semplice comando:

```
CREATE INDEX studenti_eta_idx ON studenti(eta);
```

Al solito, *studenti_eta_idx* é il nome dell'indice (idx sta per index), mentre la dicitura *studenti(eta)* indica che l'indice va creato nella tabella *studenti* ed in particolare sulla colonna *eta*.

Apparentemente la presenza di un indice non cambia il funzionamento del database: il risultato delle interrogazioni é lo stesso. Quello che cambia é la velocitá di funzionamento. In realtá vedremo che nel caso di dati spaziali, l'indice é 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 quantitá) dei dati presenti: la presenza di un indice su di una colonna velocizza sempre le operazioni di ricerca, mentre ne puó rallentare leggermente le operazioni di modifica (dato che in questo caso é necessario aggiornare anche l'indice). Inoltre la creazione di un indice richiede un utilizzo aggiuntivo di spazio disco.

Nota: non é mai necessario creare indici per le colonne chiave primaria: in questo caso un indice é creato automaticamente.

Uno sguardo all'albero nella finestra principale ci mostra la struttura corrente del db, con le nostre tabelle, colonne, vincoli e indici (Fig. 6).

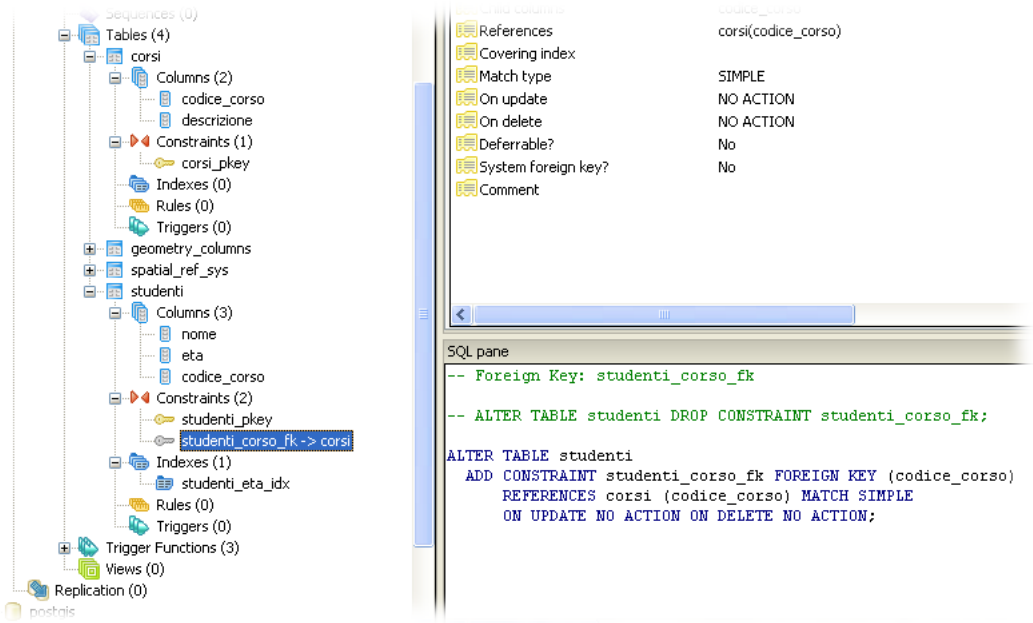

Figura 6: pgAdmin III; struttura del db con vincoli ed indici.

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 é il comando piú complesso. Le forme del comando SELECT sono moltissime, quindi ne vedremo alcuni brevissimi esempi. Inoltre l'apparente semplicitá di tale comando nasconde la notevole difficoltá di tradurre la richiesta che abbiamo in mente nella dicitura SQL; questo richiede un notevole livello di esperienza.

12.1 Forma semplice di SELECT

La forma piú semplice di *SELECT* é la seguente:

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

Nella forma semplice di SELECT bisogna specificare: l'elenco delle colonne da visualizzare, la tabella sorgente, eventuali condizioni uguali a quelle dei comandi UPDATE e DELETE, ed un eventuale ordine (l'ordine non é mai definito per default! Questo é un concetto fondamentale delle basi di dati, l'ordine delle righe é indefinito, ad esempio non centra niente con l'ordine di inserimento).

Invece di scrivere un elenco di colonne é 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 possiamo scrivere il comando:

```
SELECT *
FROM studenti;
```

Oppure:

```
SELECT *
FROM corsi
ORDER BY descrizione;
```

Si ricorda che * sta per tutte le colonne; inoltre, non essendoci filtro, vengono estratte tutte le righe della tabella indicata. Nel secondo caso i corsi saranno ordinati per descrizione (ordine alfabetico), mentre nel primo caso l'ordine e casuale. Il risultato della prima query é mostrato in Fig. 6.

Data Output | Explain | Messages | History

	nome character(128)	eta integer	codice_corso integer
1	Gianfranco Amadio	86	2
2	Claudio Rocchini	43	2
3	Margherita Azzariti	60	1
4	Salvatore Arca	60	2

Figura 7: Visualizzazione del risultato di una query.

Per visualizzare un sottoinsieme di colonne di una tabella, basta indicarne la lista dopo la parola *SELECT*; ad esempio possiamo scrivere:

```
SELECT nome,eta
FROM studenti
ORDER BY eta;
```

In questo caso si visualizza solo il nome e l'etá degli studenti, mentre l'ordine é per etá crescente.

Se invece 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*:

```
SELECT nome
FROM  studenti
WHERE eta < 60 AND codice_corso=2;
```

che in italiano si legge: selezionare il nome dalla degli studenti dove l'etá é minore di 60 (anni) e il codice del corso seguito é uguale ad 2.

12.2 Aggregazioni di righe

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

```
SELECT {funzioni_aggreganti}
FROM tabella
WHERE {condizione sulle righe}
GROUP BY {colonne che discriminano l'aggretazione}
HAVING (condizione sul risultato aggregregato)
```

Niente paura, é piú complicato da dire che da fare. In generale l'aggregazione produce un dato totale, che riguarda l'intera tabella o insiemi di righe raggruppate. Quali sono le funzioni di aggregazione? Le principali funzioni di aggregazione sono:

- Min: minimo dei valori
- Max: massimo dei valori
- Avg: media dei valori
- Sum: somma dei valori
- Count: numero di valori

Vedremo che ci sono anche funzioni di aggregazione spaziale (es. il baricentro di insieme di oggetti).

Supponiamo ad esempio di voler sapere l'etá minima, media e massima degli studenti presenti nella nostra tabella. Possiamo scrivere:

```
SELECT min(eta),max(eta),avg(eta)
FROM studenti;
```


Il risultato saranno i valori minimo, massimo e medio di tutte le etá della tabella studenti. Si noti che il risultato in questo caso é una sola riga: tutte le righe della tabella studenti 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 etá) su cui operare, i quali vanno specificati fra parentesi tonde.

Vediamo adesso come si possono raggruppare le aggregazioni di valori. Vogliamo sapere l'etá minima e massima degli studenti, ma suddivisa secondo il corso seguito; proviamo ad eseguire:

```
SELECT codice_corso, min(eta),max(eta),count(eta)
FROM studenti
GROUP BY codice_corso;
```

La query é simile alla precedente: in questo caso peró le righe non sono aggregate tutte insieme, ma secondo il codice del corso. Questo raggruppamento é dovuto all'aggiunta della riga *GROUP BY codice_corso*. Il risultato é l'analisi dell'etá degli studenti al variare del corso a cui appartengono. In questo caso il risultato é formato da piú righe: una per ogni corso presente: per ogni codice viene stampata la minima e massima etá degli studenti, limitatamente al corso in questione. La funzione *count(...)* conta semplicemente il numero di righe corrispondenti, vale a dire il numero di partecipanti ad un corso.

Per sapere quante righe contiene una tabella basta scrivere:

```
SELECT count(*)
FROM corsi;
```

Dato che alla funzione *count* non interessa la particolare colonna (conta solo il numero di righe), é possibile scrivere il carattere * (che sta per tutte le colonne) al posto del nome della colonna.

12.3 Join

Nel nostro database abbiamo due tabelle: studenti e corsi. Nella tabella studenti é presente il nome dello stesso e il codice del corso. Nella tabella corsi é presente la descrizione. Inoltre le due tabelle sono collegate da una relazione esplicita (FOREIGN KEY). Vogliamo adesso visualizzare il nome di ogni studente con associata la descrizione del corso seguito. Per fare questo é necessario utilizzare la relazione che intercorre fra le due tabelle: il termine tecnico di questa operazione é JOIN (unificazione). L'esecuzione di una SELECT con JOIN implicata l'utilizzo di piú tabelle conteporaneamente, quindi la clausola *FROM* della nostra query avrá una forma del tipo

```
...
FROM studenti, corsi
...
```

L'utilizzo di piú tabelle in una query comporta alcune complicazioni. Ad esempio dato che entrambe le tabelle in gioco hanno la colonna *codice_corso*, indicandone solo il nome il sistema non saprebbe a quale tabella ci vogliamo riferire, se quella degli studenti o quella dei corsi. Per togliere ogni ambiguitá bisogna specificare il nome di colonna completa del nome della tabella: i due nomi devono essere serapati da un punto. Inoltre dobbiamo specificare quale sia la regola di unificazione delle due tabelle: nel nostro caso la regole di unificazione é che il codice del corso seguito da uno studente (colonna *studenti.codice_corso*) deve essere uguale al codice del corso della tabella corsi (colonna *corsi.codice_corso*). Colleghiamo le due tabelle con la query:

```
SELECT studenti.nome, corsi.descrizione
FROM   studenti,corsi
WHERE  studenti.codice_corso = corsi.codice_corso;
```

Il risultato é il seguente:

```
"Margherita Azzariti ";"Basi di Dati               "
"Salvatore Arca      ";"Sistemi Informativi Territ."
"Claudio Rocchini    ";"Sistemi Informativi Territ."
"Gianfranco Amadio   ";"Sistemi Informativi Territ."
```

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. In secondo luogo vediamo che le colonne dopo la *SELECT* sono specificate nella forma *NOME_TABELLA. NOME_COLONNA*: questa specifica é necessaria in presenza di piú tabelle per chiarire da quale tabella si pesca la colonna. Codice_corso ad esempio é presente in entrambe le tabelle ed Postgres non sa decidere di quale tabella fa parte. Infine analizziamo la clausola *WHERE*: in questo caso la clausola non ha una funzione di filtro sul risultato, é invece questa che mette in relazione concretamente le due tabelle. Senza la clausola *WHERE* (provate a cancellarla ed eseguire la query), Postgres esegue quello che si chiama *prodotto cartesiano* dei valori, vale a dire tutte le combinazioni possibili fra studenti e corsi, senza nessun nesso fra le coppie studente-corsi. 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 puó essere utilizzato per creare relazioni spaziali (es. relazionare gli edifici con le strade a seconda della minima distanza relativa).

La join é un'operazione molto importante e complessa; per questo nasconde alcune difficoltá. In particolare la gestione dei campi vuoti o che non hanno corrispondenza nella join deve essere gestita specificando il tipo di comportamento da tenere.

13 Viste

Una volta che abbiamo creato una *SELECT* interessante (come quella fra studenti e corsi), é possibile che ci serva piú volte. Oltre al meccanismo di salvataggio delle query presente nell'interfaccia di Postgres (File- save), é 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*. É possibile salvare le query come viste, aggiungendo al codice della query il comando *CREATE VIEW nome_vista AS*, provate ad esempio ad eseguire:

```
CREATE VIEW stud_corsi AS
SELECT studenti.nome,
       corsi.descrizione
FROM studenti,corsi
WHERE studenti.codice_corso = corsi.codice_corso;
```

Dalla seconda riga in poi la query é identica a quella della sezione precedente. In questo caso peró la query non viene eseguita: invece le viene dato il nome *stud_corsi* 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:

```
SELECT * FROM stud_corsi;
```

I dati delle viste variano al variare delle tabelle sottostanti (studenti e corsi), vale a dire che il risultato della query non é salvato al momento della creazione della vista, ma varia al variare delle tabelle originali. Se i dati della tabella studenti o corsi vengono cambiati, il risultato dell'interrogazione della vista cambia in modo conforme.

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

14 Editor grafici di query

SQL é (a nostro parere) un linguaggio molto elegante, inoltre spesso é chiaro di per sé ed é autoesplicativo. Molti sistemi (anche Postgres) prevedono peró un ausilio grafico alla costruzione della query.

Vediamo ad esempio il Query Builder di ArcGis in Fig. 8. Questa interfaccia permette di

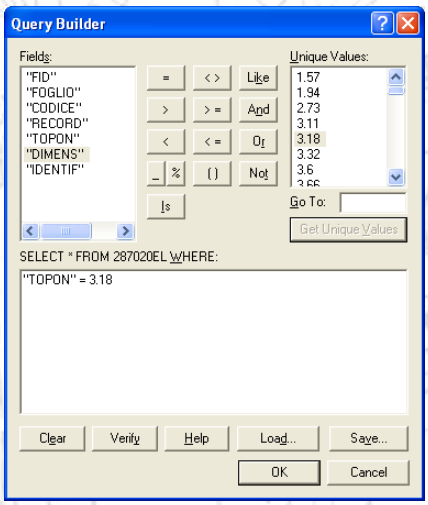

Figura 8: Schermata di creazione guidata di un filtro SQL (Query Builder di ArcGIS).

costruire in modo guidato una clausola WHERE per la nostra query SQL, anche se rimane possibile digitarla manualmente. Altri sistemi di basi di dati (es. Access) hanno strumenti simili di costruzione.

15 Basi di dati reali

I database reali (ovviamente) possono essere molto complessi o contenere grandi moli di dati. Il database geografico del catalogo IGM on line contiene 728,625 Toponimi (ricerca spaziale-testuale); 213,906 Fotogrammi Aerei georeferenziati (raster+scheda monografica); 44,582 Punti Trigonometrici, IGM95 e Capisaldi di Livellazione; 27,414 Schede foto storiche (immagine + monografia), 384040 attributi; 22,051 Carte storiche (immagine + monografia), 72,834 relazioni spaziali; 8,274 Limiti Amministrativi Geografici; 6,825 Prodotti in catalogo con stato produzione; 5,000 Raster, preview dei prodotti cartografici; Per avere un'idea una basi di dati reale, vediamo lo schema grafico del programma per la gestione del personale IGM.

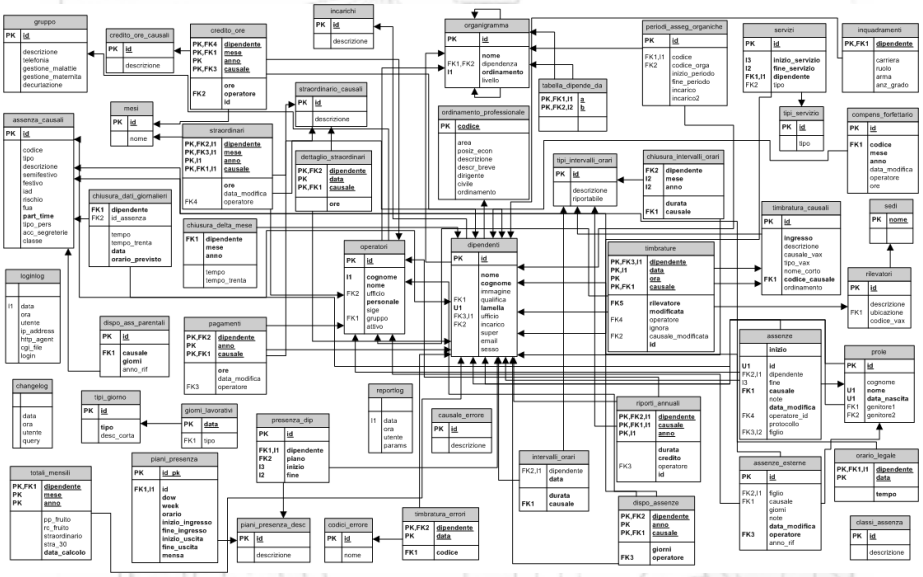

Figura 9: Gestione personale IGM (milioni di fatti registrati).

16 Conclusioni

Questa é solo una brevissima introduzione a *SQL*. La struttura del comando SELECT ha molte altre possibilitá, che richiederebbero almeno un anno di corso. Il linguaggio SQL é di per sé molto semplice, ma la creazione di un comando SELECT non banale, richiede, in alcuni casi, una certa esperienza. Es. banale: come si scrive una SELECT che mostra il nome dello studente piú giovane?

Riferimenti bibliografici

[1] Renzo Sprugnoli, *Libri di base: le basi di dati*, Editori Riuniti (*www.editoririuniti.it*), 1987.

[2] PostGIS , *PostGIS Online Documentatio*, (*postgis.refractions.net/documentation*), December 2005.

[3] PostgreSQL Global Development Group, *PostgreSQL 8.4.1 Documentation*, (*www.postgresql.org/docs/manuals*), 2010.

Elenco delle figure