Copyright (c) 2008, 2009, 2010 Fabio Proietti

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Authors and contributors
Fabio Proietti

Feedback
Please direct any comments or suggestions about this document to
fabio.proietti AT istruzione DOT it

Publication date and version
2008-11-15, v.0.4
2009-11-15, v.0.5
2010-04-21, v.0.6

last modified 2014-05-04

Appunti di lavoro sui database

Indice

Il DBMS
Creare utenti e database
SQL
Creare una tabella
Una tabella non è una relazione
Vincoli intrarelazionali
Creare indici
Vincoli extrarelazionali

Interrogazioni del database
Il comando SELECT
selezione e proiezione
Il manuale
Congiunzione (FROM)
Selezione (WHERE)
Funzioni di aggregazione
Clausola di aggregazione (GROUP BY)
Selezione in sottogruppi di righe (HAVING)
Ordinamenti (ORDER BY)
LIMIT e OFFSET

Il DBMS

Il Database Management System (DBMS) è un software che permette la gestione di database.
I DBMS sono spesso basati su livello logico relazionale e sono chiamati ancheR-DBMS, ma esistono anche DBMS basati sulla programmazione orientata ad oggetti.

I DBMS sono software che hanno già in sé molte delle funzioni necessarie alla gestione del database e possono essere facilmente programmati dal programmatore per il caso particolare in cui verranno applicati.
Grazie al DBMS, il lavoro del programmatore risulta "alleggerito"; il programmatore che lavora con i DBMS non utilizza però solo i classici linguaggi di programmazione procedurali (es: Pascal), basati sui diagrammi di flusso, ma anche linguaggi di programmazione dichiarativi (es: SQL).
La spiegazione delle differenze tra linguaggi procedurali e dichiarativi verrà spiegata nei prossimi capitoli.

Nella seguente tabella vengono citati alcuni noti DBMS distinguendo quali sono Open Source e quali no, e quali offrono anche un servizio in rete (server) e quali no.

Nome	OpenSource	Server	sist. operativo
SQLite	sì	no	cross-platform
LibreOffice Base	sì	no	cross-platform
HSQLDB	sì	sì	cross-platform
PostgreSQL	sì	sì	cross-platform
MySQL	sì	sì	cross-platform
Oracle 9	no	sì	cross-platform
Informix	no	sì	cross-platform
Microsoft Access	no	no	Microsoft Windows
Microsoft SQL server	no	sì	Microsoft Windows

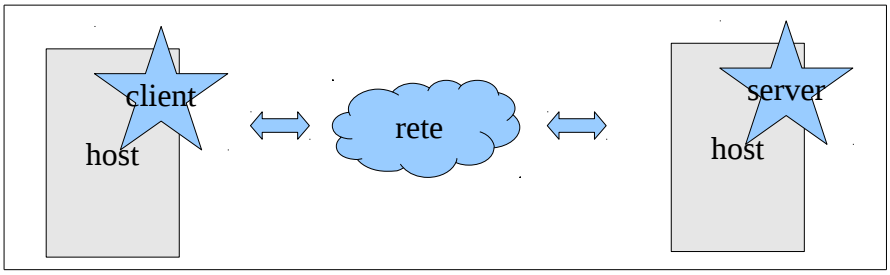
Alcune delle funzioni fornite, svolte o gestite da un DBMS possono essere:

- **integrità dei dati**: controllo del rispetto dei vincoli del modello logico dei dati (rispetto delle regole imposte dal progettista)
- i**ntegrazione degli archivi**: gli archivi che costituiscono un database sono integrati, cioè collegati. I tipi di collegamento dipendono dal tipo di modello logico gestisce usato dal DBMS.
- **database accentrati e distribuiti**: gli archivi che costituiscono un database possono trovarsi in un'unica sede fisica (accentrati) o essere collocati su diverse macchine (distribuiti).
-

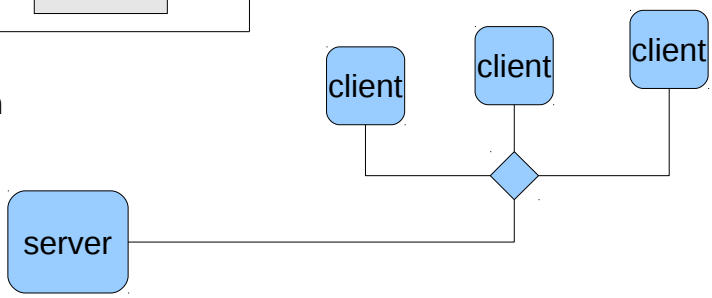
- **multiutenza**: il DBMS può essere usato da più utenti, che possono accedere usare anche lo stesso database
- **accesso concorrente ai dati**: assicura la funzionalità del sistema anche quando più utenti si collegano contemporaneamente alla stessa tabella dello stesso database (ovviamente in tal caso è consentito l'accesso in sola lettura)
- **sicurezza**: il DBMS consente l'accesso ai dati solo agli utenti autorizzati. Ogni utente può avere diversi tipi di privilegi o autorizzazioni. Ad esempio: l'utente con ruolo di amministratore, possiede tutti i diritti sui dati e può concedere o rimuovere diritti ad altri utenti.
- **transazioni**: sono macro-operazioni costituite da un gruppo di operazioni. L'operazione complessiva (la transazione) è considerata eseguita solo quando sono state eseguite correttamente tutte le parti che la costituiscono. Se anche una sola parte della transazione non è eseguita correttamente, vengono annullate tutte le parti che la costituiscono (fase di roll-back). Un esempio di transazione è un movimento di denaro tra due conti correnti: il movimento viene annullato sia nel caso che non ci sia sufficiente disponibilità nel conto di partenza, sia nel caso in cui il conto di destinazione sia inesistente, e la situazione finale deve essere uguale a quella iniziale.
- linguaggi di programmazione: il programmatore può creare delle procedure e/o funzioni per la gestione dei dati, sia in linguaggio SQL che in altri linguaggi. L'utente finale usa i programmi creati dal programmatore

Modello di funzionamento Client/Server

Si tratta di un modello di comunicazione utilizzato nelle reti di computer.
Prima di introdurre qualsiasi altro concetto, si assumerà la seguente convenzione grafica: ogni computer (o host) avrà un simbolo rettangolare, il software che vi viene eseguito avrà il simbolo di una stella.
Per la comunicazione tra due o più computer (o host) è possibile utilizzare il modello Client/Server in cui un host richiede un servizio e un altro host offre il servizio.



Quando si usa un DBMS multiutente, è possibile la connessione contemporanea di più utenti. La multiutenza è possibile utilizzando più software client in una rete di computer. L'autenticazione degli utenti avviene con username e password.

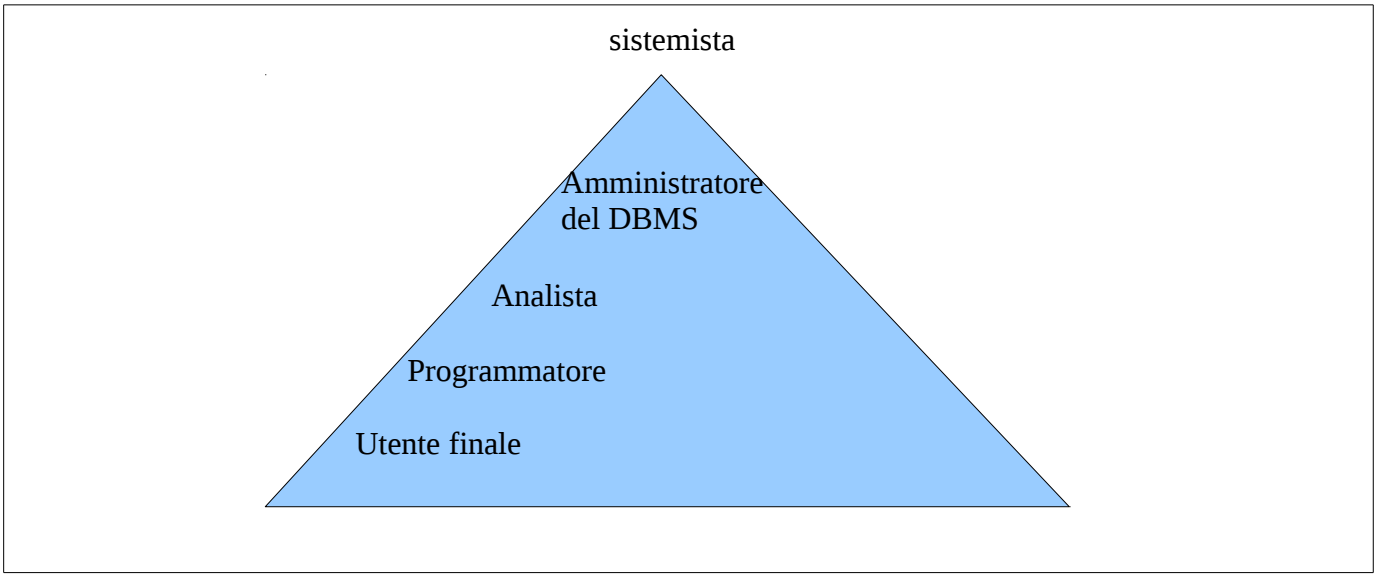


Creare utenti e database

Di solito gli utenti sono aggiunti dall'amministratore del DBMS.
Ogni utente ha uno username e una password e l'amministratore stabilisce quali privilegi può avere un utente e quali operazioni può eseguire un utente su un certo database.
Spesso in un database c'è un solo utente che ha il permesso di crearvi tabelle, mentre altri utenti possono avere solo il diritto di interrogare il database.

In generale si possono distinguere i seguenti tipi di utenti:

- **il sistemista**: installa e configura il sistema operativo e il DBMS ad es. si può chiamare root, ma di solito non è un utente del DBMS
- **l'amministratore** del DBMS: aggiunge e gestisce gli utenti e le loro autorizzazioni, crea e gestisce i database, effettua e ripristina backup dei dati
- **l'analista** programmatore: può progettare lo schema logico di un DB
- **il programmatore** del DBMS: realizza le idee dell'analista, crea le strutture delle tabelle e scrive applicazioni in un qualche linguaggio di programmazione, ad esempio, usando un linguaggio testuale.
- **gli utenti finali** o operatori: usano i programmi per inserire dati, cercare dati, modificare dati, usando un'interfaccia grafica o testuale.



PostgreSQL

http://www.postgresql.org

PostgreSQL (si pronuncia post-gri-siquel) è un DBMS Server, cioè può fornire un servizio di database in una rete di computer a cui gli utenti possono accedere (da un altro computer) utilizzando un software client (ad esempio PGadmin III). È disponibile con licenza BSD e altre licenze libere.

PGAdmin III

http://www.pgadmin.org

Uno dei programmi client che si può utilizzare per connettersi al DBMS server PostgreSQL si chiama **PGadmin III**. La comunicazione tra server e client avviene solo dopo aver stabilito una connessione (fornendo username e password). Poiché PostGreSQL è un programma libero, chiunque può ideare un nuovo software client, diverso da quello "ufficiale". Esiste anche un client web: PHPPgAdmin.

MySQL

MySQL (si pronuncia mai-siquel) è un database open source, acquistato da Sun (ora di Oracle). Dopo l'acquisizione da parte di Sun lo sviluppatore è passato allo sviluppo di un nuovo DBMS derivato dai sorgenti di MySQL che ha chiamato MariaDB.
MySQL è uno dei database liberi più noti sul web perché utilizzato spesso per la gestione i contenuti dei siti web (ad esempio Wordpress). La sua gestione avviene con un client web (browser) PHPMyAdmin. Oggi MySQL è disponibile con licenza GPL, ma anche con licenza proprietaria (versioni commerciali).

MariaDB

Il DBMS ottenuto come fork da MySQL. È disponibile con licenza GPL e LGPL.

Il linguaggio SQL

Per operare sui database esiste un linguaggio che è diventato oggi uno standard internazionale (ISO è l'organizzazione per gli standard internazionali, la stessa che controlla lo standard ISO 9001:2000). Tutti i DBMS che usano questo linguaggio dovrebbero rispettare le regole dello standard, ma purtroppo questo non è sempre vero. Quindi **ogni diverso DBMS può avere un comportamento diverso** pur usando lo stesso comando SQL. Il linguaggio SQL ha subito una evoluzione nel tempo, passando dalla versione SQL-86 per SQL-92, fino a SQL:2008.
Il DBMS PostgreSQL è abbastanza aderente allo standard, ma introduce comunque anche comandi non standard.
SQL significa Structured Query Language (linguaggio strutturato per le interrogazioni) e la sintassi dei comandi SQL di PostgreSQL possono essere trovati nel manuale elettronico all'indirizzo:
`http://www.postgresql.org/docs/8.0/static/sql-commands.html`

Il manuale si legge seguendo le seguenti regole:
Le parentesi quadrate e le parentesi graffe NON devono essere digitate sulla tastiera, mentre le parentesi tonde sono obbligatorie.
Le parentesi quadrate e graffe hanno un significato speciale:

- le parentesi quadrate racchiudono una opzione che può essere usata oppure no:
```
COMANDO  [OPZIONE]
```
- le parentesi graffe racchiudono un gruppo di opzioni a scelta, separate dalla barra verticale. Se la parentesi sono graffe è obbligatorio scegliere almeno una delle opzioni tra quelle proposte:
```
COMANDO  {OPZIONE A SCELTA | OPZIONE A SCELTA}
```

Se l'opzione può essere ripetuta più volte, usando la virgola come separatore, allora ci sono i tre punti di sospensione:
```
COMANDO  [OPZIONE RIPETIBILE] [, ...]
```
Quindi le parentesi quadre, graffe e i puntini di sospensione non fanno parte del comando e non vanno digitati. Gli altri segni, come le parentesi tonde, invece, non hanno questo significato speciale e vanno digitate normalmente.

Il linguaggio SQL è composto da diversi tipi di istruzioni che si occupano di aspetti leggermente diversi del DBMS, per questo motivo, all'interno di questo linguaggio è possibile distinguere i seguenti tipi di sotto-linguaggi:

- **Data Definition Language** (DDL): comandi per la creazione e la modifica delle strutture dati (le tabelle) come, ad esempio, il comando CREATE TABLE.
- **Data Manipulation Language** (DML): comandi per l'inserimento e la modifica dei dati (nelle tabelle) come, ad esempio, il comando INSERT.
- **Query Language** (QL): comandi per le interrogazioni (cercare dati) come, ad esempio, il comando SELECT.
- **Data Control Language** (DCL): comandi per il controllo dei privilegi di accesso degli utenti come, ad esempio, i comandi GRANT e REVOKE.
- **Transactional Control Language** (TCL): comandi per la gestione di transazioni, ma che potrebbero anche rientrare nel DML. Il concetto di transazione è stato spiegato nelle funzioni del DBMS.

- altri tipi di istruzioni servono per eseguire un backup dei dati (o ripristinare un database a partire dal suo backup), per creare funzioni in un linguaggio procedurale
- I linguaggi "visivi" che permettono di interagire con il DBMS usando solo il mouse e le icone non appartengono al linguaggio SQL (ad esempio esistono linguaggi che costruiscono interrogazioni in modo grafico invece che testuale)

Il linguaggio SQL è diverso dai tradizionali linguaggi di programmazione **procedurali**, come il Pascal o il C, basati sui diagrammi di flusso. Nei linguaggi procedurali ogni istruzione descrive l'operazione che l'elaboratore deve effettuare. Ad esempio, in Pascal, l'istruzione:

```
writeln ('Ciao ');
```

indica all'elaboratore di eseguire l'operazione di scrittura (output su video) e poi di andare a capo.
Il linguaggio SQL è detto "**dichiarativo**" perché il programmatore non indica quali azioni vuole eseguire, ma "dichiara" l'aspetto del risultato che vuole ottenere.
Ad esempio, per effettuare una ricerca di dati non è necessario specificare il metodo di ricerca.
In SQL non è importante se il comando è in maiuscolo o in minuscolo, ma è abitudine usare il maiuscolo per i comandi e le parole chiave (come la parola 'UPDATE') e il minuscolo per i dati (come la parola 'telefono'). Non è importante andare a capo, ma farlo in modo regolare può agevolare la lettura del codice SQL.
Ogni comando in SQL deve terminare con il simbolo "punto e virgola" (;).

MySQL non è case sensitive?

Per i tipi CHAR, VARCHAR e TEXT viene ignorata la differenza tra maiuscolo e minuscolo.

```
The default character set and collation are latin1 and
latin1_swedish_ci, so nonbinary string comparisons are case
insensitive by default.
```

Modificando la collation e il charset delle tabelle si può modificare tale comportamento.

Esercizio

Scrivere a fianco dei seguenti comandi a quale gruppo appartiene, basandosi sulla sua descrizione

Comando	descrizione	gruppo
ABORT	abort the current transaction	TCL
ALTER TABLE	change the definition of a table	DDL
BEGIN	start a transaction block	TCL
COMMIT	commit the current transaction	TCL
CREATE SCHEMA	define a new schema	DDL
CREATE SEQUENCE	define a new sequence generator	DDL
CREATE TABLE	define a new table	DDL
CREATE TRIGGER	define a new trigger	
CREATE VIEW	define a new view	DDL
DELETE	delete rows of a table	DML
DROP SCHEMA	remove a schema	DDL
DROP SEQUENCE	remove a sequence	DDL
DROP TABLE	remove a table	DDL
DROP TRIGGER	remove a trigger	
DROP VIEW	remove a view	DDL
END	commit the current transaction	TCL
GRANT	define access privileges	DCL
INSERT	create new rows in a table	DML
NOTIFY	generate a notification	
REVOKE	remove access privileges	DCL
ROLLBACK	abort the current transaction	TCL
SELECT	retrieve rows from a table or view	QL
SET TRANSACTION	set the characteristics of the current transaction	TCL
START TRANSACTION	start a transaction block	TCL
UPDATE	update rows of a table	DML

Creare una tabella

Ogni colonna di una tabella può contenere solo dati di un certo tipo, ad esempio, solo numeri oppure solo parole, ecc.
Nel linguaggio SQL esistono numerosi tipi di dato standard. Nella seguente tabella sono riportati solo alcuni dei più utilizzati.

tipo di dato	esempio o valore	descrizione
`varchar(n)`	`varchar(8)`	Per inserire un testo di lunghezza variabile, di cui si conosce la lunghezza massima, ad esempio di 8 caratteri
`char(n)`	`char(8)`	Per inserire un testo di lunghezza fissa, ad esempio 8 caratteri. Rispetto al precedente tipo di dato la lunghezza del testo inserito è fissa.
`text`	`text`	Per inserire un testo (teoricamente) illimitato
`numeric(t,d)`	`numeric(6,2)` 9000.12	Per inserire un numero decimale specificando il numero complessivo di cifre(t) e di cifre decimali(d).
`real`	1.00E-009	Per inserire un numero a virgola mobile con 6 cifre decimali da 1E-30 a 1E+30
`double`		come `real` ma numeri da 1E-300 a 1E+300
`date`	2009-12-31	una data espressa nello standard ISO. Lo standard SQL non ammette altre forne.
`time`	23:39:22.12	Per inserire un'ora espressa (con i millesimi di secondo)
`timestamp`	2004-10-19 10:23:58	Per inserire una data insieme con un'ora
`boolean`	true	Per inserire un valore che può essere vero o falso

Il tempo coordinato universale UTC (Coordinated Universal Time) è il fuso orario di riferimento da cui sono calcolati tutti gli altri fusi orari del mondo. L'ISO 8601 è uno standard internazionale per la rappresentazione di date ed orari.

Esempio: data ed ora UTC 2011-03-28T13:59Z

Tipo NULL

Oltre ai tipi di dato appena visti, esiste un valore speciale chiamato NULL, che può essere usato per ogni tipo di dato. NULL può essere considerato un segno di marcatura perché non è un dato e non possiede un vero valore.

Quando l'utente non inserisce nessun valore in una cella della tabella, il valore è considerato "ignoto" e gli viene assegnato automaticamente NULL.

Il NULL di solito non è visibile e nella cella che contiene NULL viene visualizzato uno spazio bianco.

Fare attenzione a non confondere il valore ignoto NULL con il valore numerico 0 (zero) o con lo spazio bianco (" ").

Tipo contatore

A volte è utile poter creare un contatore numerico (intero) che si incrementa automaticamente quando si inserisce una nuova riga, senza necessità di specificare nessun valore. Questo tipo di dato non appartiene allo standard SQL.

Chi usa **PostGreSQL** può usare SERIAL, ma SERIAL non è un tipo di dato standard SQL. Quando la chiave primaria è di tipo SERIAL, allora è obbligatorio che la chiave esterna sia di tipo INTEGER.

Chi usa **LibreOffice** o HDBSQL può usare IDENTITY, ma non è un tipo di dato standard SQL.

Chi usa **SQLite** può aggiungere il vincolo AUTOINCREMENT, ma non è un tipo di dato standard SQL.

Chi usa **MySQL** può aggiungere il vincolo AUTO_INCREMENT, ma non è un tipo di dato standard SQL.

Esercizio

Rispondere alla seguente domanda: che tipo di dati si può utilizzare per registrare i seguenti valori?

il costo per la locazione di un immobile	NUMERIC
la data di inizio di una locazione	DATE
il numero di metri quadrati di una superficie di un immobile	NUMERIC
il giorno del mese in cui deve essere pagato uno stipendio	NUMERIC
la percentuale di un tasso di interesse	NUMERIC
il nome di una città	VARCHAR
il numero di telefono di un'azienda	VARCHAR
il numero di una partita IVA	CHAR
il codice avviamento postale (CAP) di una città	CHAR
il numero civico di un immobile	VARCHAR

Esempio

Come esempio, verrà creata la struttura della seguente tabella "classi" (senza ancora inserire nessun valore):

tabella classi

codice	anno	sezione
100	3	A
101	4	A
109	4	B

```
CREATE TABLE classi (
     codice NUMERIC(4,0) PRIMARY KEY,
     anno NUMERIC(1,0),
     sezione CHAR(2)
     );
```

Supponendo di voler lavorare all'interno dello schema "scuola" il codice SQL per creare la tabella inizia in modo leggermente diverso:

```
CREATE TABLE scuola.classi (
     codice NUMERIC(4,0) PRIMARY KEY,
     anno NUMERIC(1,0),
     sezione CHAR(2)
     );
```

Dopo aver visto un esempio è possibile analizzare la sintassi del comando CREATE TABLE dal manuale facendo attenzione il significato speciale delle parentesi quadrate e delle parentesi graffe:

```
CREATE TABLE table_name (
  { column_name data_type [ DEFAULT default_expr ] [
column_constraint [ ... ] ]
    | table_constraint
 }  [, ... ]
)
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
```

dove *column_constraint* è:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY |
  CHECK (expression) |
 }
```

e dove *table_constraint* è:

```
[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) |
  PRIMARY KEY ( column_name [, ... ] ) |
  CHECK ( expression ) |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ (
refcolumn [, ... ] ) ]
     [ ON DELETE action ] [ ON UPDATE action ] }
```

Prima di andare avanti è interessante confrontare, nel manuale di CREATE TABLE, quali sono i vincoli disponibili come vincoli di colonna e quelli disponibili come vincoli di tabella (parte rossa e la parte blu)

tipo di vincolo	***column_constraint***	***table_constraint***
obbligo	NOT NULL	*- non disponibile -*
unicità	UNIQUE	UNIQUE(...)
chiave primaria	PRIMARY KEY	PRIMARY KEY(...)
chiave esterna	**REFERENCES ...(...)*	FOREIGN KEY(...) REFERENCES...(...)
controllo	CHECK	CHECK

*sintassi *non disponibile in LIBREOFFICE e HSQLDB*

Una Tabella non è una Relazione

Chi ha studiato la progettazione dei database attraverso l'utilizzo del modello logico *relazionale*, potrebbero avere inizialmente delle difficoltà a realizzare le tabelle in un DBMS, questo perché il concetto di tabella non coincide esattamente con quello di una *relazione*.

Per comprendere cosa sia una *relazione matematica*, è necessaria una piccola premessa teorica:

Dati gli insiemi:
`A = { Rossi,Verdi }` e `B = { 1980, 1990, 1991 }`

il loro Prodotto Cartesiano C = A x B vale:
`C = A x B = { (Rossi,1980), (Rossi,1990), (Rossi,1991), (Verdi,1980), (Verdi,1990), (Verdi,1991) }`

L'insieme C può essere rappresentato anche con una tabella:

Rossi	1980
Rossi	1990
Rossi	1991
Verdi	1980
Verdi	1990
Verdi	1991

Una Relazione matematica è un sottoinsieme del Prodotto cartesiano, quindi, usando solo alcuni elementi di C si può stabilire una Relazione Matematica su A e B (in breve R(A,B))
La seguente tabella rappresenta perciò una possibile relazione R(A,B):

Rossi	1990
Rossi	1991
Verdi	1980

Si sottolinea che, né l'insieme prodotto cartesiano né la relazione matematica ammettono l'esistenza di due elementi identici, o, in altre parole, in una tabella non ci dovrebbero essere due righe identiche.

Quindi, l'espressione "*una tabella non è una relazione*" significa che, quando in una tabella vengono inserite delle righe identiche, in tal caso la tabella non rappresenta più una relazione.

Nei DBMS è teoricamente possibile inserire due volte la stessa riga. Il progettista deve quindi impegnarsi affinché questo non accada. Lo strumento che può utilizzare per impedirlo è il linguaggio SQL. Attraverso il linguaggio SQL si possono infatti specificare delle regole, dette anche vincoli intrarelazionali e vincoli extrarelazionali. Questi vincoli aiutano il progettista, ma non sono sufficienti a risolvere tutti i problemi che nascono durante la realizzazione di un database.

Vincoli intrarelazionali

Tramite il linguaggio SQL è possibile definire su una tabella i seguenti vincoli intrarelazionali, usando le seguenti parole chiave:

vincolo di unicità	UNIQUE
vincolo di obbligo	NOT NULL
vincolo di chiave primaria	PRIMARY KEY
vincolo di dominio	CHECK (...)
vincolo di tupla	CHECK (...)

Il vincolo UNIQUE impedisce che l'utente inserisca due volte lo stesso valore in una colonna della tabella.

Il vincolo NOT NULL impedisce all'utente di lasciare vuoto un valore nella colonna.

Il vincolo PRIMARY KEY impedisce entrambe le cose, ma ha anche un'altra caratteristica che verrà vista nel paragrafo "Creare indici".

vincolo	**unicità**	**obbligo**
PRIMARY KEY	sì	sì
UNIQUE	sì	no
NOT NULL	no	sì

Vincoli intrarelazionali semplici

Nel caso in cui il vincolo intrarelazionale riguardi una sola colonna, il manuale di PostgreSQL utilizza il termine "column constraint" (vincolo di colonna). In questo caso il vincolo può essere inserito subito dopo il tipo di dato, come in questo esempio, dove viene creata una tabella con tre colonne, di cui la prima è chiave primaria:

Schema logico: classi (<u>codice</u>, numero, sezione)

```
CREATE TABLE scuola.classi (
     codice NUMERIC(4,0) PRIMARY KEY,
     anno NUMERIC(1,0),
     sezione CHAR(2)
);
```

tabella classi

codice	anno	sezione
100	3	A
101	4	A
109	4	B

In realtà, la tabella creata è vuota. I dati sono inseriti solo a scopo illustrativo.

Se si vuole evitare che esistano nella tabelle righe senza numero si può usare il vincolo NOT NULL, come in questo esempio:

```
CREATE TABLE scuola.classi (
     codice NUMERIC(4,0) PRIMARY KEY,
     anno NUMERIC(1,0) NOT NULL,
     sezione CHAR(2)
);
```

Nello schema logico esiste la convenzione di sottolineare un vincolo di chiave primaria, ma non esiste una convenzione comune per evidenziare un vincolo NOT NULL o un vincolo UNIQUE. In questi casi può essere utile aggiungere una descrizione a parole del tipo di vincolo nella documentazione del progetto.

Vincoli intrarelazionali composti

Un vincolo intrarelazionale, secondo il manuale del comando CREATE TABLE, può essere inserito anche dopo tutti i tipi di dato, e in questo caso viene chiamato "table constraint" (vincolo di tabella). Questa forma si usa, di solito, quando il vincolo riguarda contemporaneamente più di una colonna e la sintassi è leggermente diversa da quella dei vincoli semplici (confrontare le due tabelle)

UNIQUE(...)	vincolo di unicità
NON PUO' ESSERE COMPOSTO	vincolo di obbligo
PRIMARY KEY(...)	vincolo di chiave primaria
CHECK (...)	vincolo di dominio
CHECK (...)	vincolo di tupla

Il precedente esempio può essere modificato, definendo una chiave primaria composta contemporaneamente da due colonne:

Schema logico: classi (numero, sezione)

NOTA: la sottolineatura non si deve interrompere!

```
CREATE TABLE classi (
     anno NUMERIC(1,0),
     sezione CHAR(2),
     CONSTRAINT pk PRIMARY KEY (numero,sezione)
);
```

L'ultimo modo descritto per definire il vincolo PRIMARY KEY permette di realizzare quindi chiavi primarie composte (derivate da attributo composto).

tabella classi

anno	sezione
3	A
4	A
4	B

Usando i vincoli PRIMARY KEY e UNIQUE il progettista può impedire che due righe siano identiche o che su una colonna si ripeta due volte lo stesso valore.

Che cosa accade se uso UNIQUE come attributo composto?
Se si usa UNIQUE composto su due colonne, sarà possibile ripetere il valore su una colonna, ma non sarà possibile ripetere una coppia di valori su entrambe le colonne.

Che cosa accade se uso NOT NULL come attributo composto?
Il linguaggio SQL non consente di applicare il vincolo di obbligo composto.

Creare indici

Quando i dati vengono cercati in una tabella, può capitare che il valore desiderato venga cercato tra migliaia di righe. Quando questa operazione deve essere ripetuta molto spesso può essere adottata una soluzione più efficiente della semplice ricerca esaustiva: per evitare di perdere tempo in lunghe ricerche è possibile creare un indice (index) sulla colonna desiderata.

L'indice su una colonna funziona come l'indice analitico di un libro e consente di conoscere velocemente la posizione dell'elemento cercato (se presente). Il DBMS crea automaticamente un indice (di tipo B-Tree) su tutte le colonne con vincolo PRIMARY KEY. È questo il motivo che rende il vincolo di chiave primaria (PRIMARY KEY) diverso dal vincolo di obbligatorietà (NOT NULL) e dal vincolo di unicità (UNIQUE). La creazione di indici sulle colonne ha comunque un piccolo svantaggio: l'indice, una volta creato, deve essere mantenuto costantemente aggiornato e ogni volta che si inserisce o si elimina un elemento in una colonna con indice, l'operazione richiede un tempo leggermente più lungo del normale (ma comunque quasi trascurabile).

Vincoli extrarelazionali

Un vincolo extrarelazionale riguarda di solito più di una tabella.
Il vincolo di chiave esterna, o meglio di "integrità referenziale", è di tipo extrarelazionale .
Come si è già visto per i vincoli intrarelazionali, anche il vincolo di integrità referenziale può essere realizzato in SQL, sia come vincolo di colonna (column constraint) che come vincolo di tabella (table constraint). Di solito questo dipende dal fatto che la chiave esterna sia stata derivata da un attributo composto, cioè sia composta contemporaneamente da più colonne. Nel caso di vincolo composto la sintassi è leggermente diversa:

vincolo semplice	- non disponibile in HSQLDB...

vincolo composto	FOREIGN KEY (...) REFERENCES ... (...)

Nel seguente esempio, nello schema "scuola", viene creata la tabella "alunni" e viene collegata con una chiave esterna alla tabella "classi", precedentemente creata:

alunni (matricola, nome, cognome, codiceclasse)

```
CREATE TABLE alunni (
     matricola NUMERIC(4,0) PRIMARY KEY,
     nome CHAR(32),
     cognome CHAR(32),
     codiceclasse NUMERIC(4,0) REFERENCES scuola.classi(codice)
);
```

tabella classi

codice	numero	sezione
100	3	A
101	4	A
109	4	B

tabella alunni

matricola	nome	cognome	classe
1398	Mario	Rossi	101
1399	Anna	Verdi	101
1400	Giulio	Bianchi	100

MySQL non rispetta il vincolo di integrità referenziale?

MySQL può usare diversi motori (engine), tra cui InnoDB e MyISAM.
il motore MyISAM è più veloce ma non rispetta tutti i vincoli imposti dal programmatore.
Il motore InnoDB è più rigoroso, e possiede delle funzionalità come le transazioni e l'integrità referenziale, ma quest'ultima non funziona se viene applicata come vincolo semplice (su una colonna).
Per avere il vincolo di integrità referenziale, questo deve essere imposto come vincolo composto (di tabella) usando il comando FOREIGN KEY.

Riferimento al manuale
http://dev.mysql.com/doc/refman/5.0/en/innodb-foreign-key-constraints.html

Altri comandi per le tabelle

Altri comandi per operare sulla struttura delle tabelle sono:
`ALTER TABLE, DROP TABLE`

Comandi per modificare i dati

Dopo aver creato lo *schema* dei dati (la struttura della tabella) è possibile anche inserire, modificare, cancellare e leggere i dati.
Per le prime tre operazioni esistono tre diversi comandi SQL appartenenti al sotto-linguaggio DML: `INSERT, UPDATE, DELETE.`
Si tratta di operazioni abbastanza semplici rispetto alla creazione di una tabella.
Esempio

```
INSERT INTO alunni
    VALUES(1398,'Mario','Rossi',101);
```

```
INSERT INTO table [ ( column [, ...] ) ]
    { VALUES ( { expression | DEFAULT } [, ...] ) }
```

Tutti i tipi di dato, ad eccezione di `NUMERIC`, `REAL` e `FLOAT`, devono essere racchiusi tra apici (il carattere usato anche per l'apostrofo).

Per poter comprendere pienamente i comandi di eliminazione ed aggiornamento si consiglia di affrontare prima il comando SELECT

```
DELETE FROM [ ONLY ] table [ WHERE condition ]
```

```
UPDATE [ ONLY ] table SET column = { expression | DEFAULT } [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

Interrogazioni del database

Per affrontare questo paragrafo è necessario aver già compreso il concetto di *relazione* e di *operazioni relazionali*. Poiché le relazioni sono degli insiemi si possono utilizzare le note operazioni insiemistiche di unione, intersezione e differenza (ma queste operazioni si possono effettuare solo tra relazioni simili). Le operazioni di interrogazione sono usate solo per leggere dati nelle tabelle e non modificano il database.

Il comando SELECT

Dal punto di vista teorico, per le relazioni esistono operazioni come: proiezione (π) selezione(σ) e congiunzione. Si tratta di operazioni che possono eventualmente essere combinate tra di loro, a seconda delle necessità. Di solito si esegue prima la selezione e poi la proiezione. Per ragioni di efficienza di calcolo la congiunzione dovrebbe essere eseguita dopo la selezione, ma non sempre è possibile.

Dal punto di vista pratico, in SQL, si utilizza un solo comando per eseguire tutte i comandi di interrogazione in un'unica istruzione: SELECT. Per poter esprimere tutte le possibili condizioni il comando possiede un certo numero di clausole (opzionali), come FROM, WHERE, ecc.

Selezione e proiezione

Nei seguenti esempi il comando SELECT viene applicato su una sola tabella.

Per visualizzare tutte le colonne di tutte le righe di una tabella si può scrivere:

```
SELECT *
     FROM tabella;
```

Per visualizzare solo alcune colonne di una tabella (operazione di proiezione) si sostituisce l'asterisco con i nomi delle colonne:

```
SELECT nomecolonna1,nomecolonna2
     FROM tabella;
```

Per visualizzare solo alcune righe di una tabella (operazione di selezione) si deve aggiungere la clausola WHERE e specificare una condizione:

```
SELECT *
     FROM tabella
     WHERE nomecolonna='valore';
```

che visualizzerà solo le righe della tabella in cui la condizione è verificata.

La selezione e la proiezione possono essere eseguite anche in un unico comando SELECT, come nel seguente esempio:

```
SELECT nomecolonna1,nomecolonna2
     FROM tabella
     WHERE nomecolonna='valore';
```

Tutte queste operazioni non modificano la tabella di partenza, infatti SELECT è un comando che viene usato per le interrogazioni, cioè per leggere i dati nelle tabelle. Il risultato sarà a sua volta una tabella, ma solo una tabella temporanea che non è salvata nel database.

Il manuale di SELECT

Dopo aver visto alcuni esempi è possibile analizzare anche la sintassi del comando SELECT dal manuale di PostgreSQL facendo attenzione al significato speciale delle parentesi quadrate e delle parentesi graffe, di cui si è già parlato.

(il termine *expression* indica una espressione matematica con un nome di una variabile, un nome di colonna o un valore)

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
    [ FOR UPDATE [ OF table_name [, ...] ] ]
```

where *from_item* can be one of:

```
    table_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]

    from_item join_type from_item
          [ ON join_condition | USING ( join_column [, ...] ) ]

    ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
```

where *join_type* can be one of:

```
[ NATURAL ] { [ INNER ] JOIN | { LEFT | RIGHT | FULL }
     [ OUTER ] JOIN | CROSS JOIN }
CROSS JOIN is similar to the Cartesian product
```

select: A sub-SELECT can appear in the FROM clause. This acts as though its output were created as a temporary table for the duration of this single SELECT command. Note that the sub-SELECT **must be** surrounded by parentheses, and an alias ***must* be** provided for it.

Per comprendere le istruzioni di tipo SELECT è necessario conoscere l'ordine di esecuzione in cui vengono elaborate le diverse clausole. Vicino ad ogni clausola del comando SELECT è stato aggiunto un numero, iniziando a partire da quella con il numero 1:

```
[5]SELECT [8]DISTINCT(...)
     [1] FROM
     [2] WHERE
     [3] GROUP BY
     [4] HAVING
     [6] UNION ...
     [7] ORDER BY
     [9] LIMIT
     [10]OFFSET
```

Quindi, nel precedente esempio:

```
SELECT nomecolonna1,nomecolonna2
     FROM tabella
     WHERE nomecolonna='valore';
```

Prima viene individuata la tabella (FROM), poi viene verificata la clausola per la selezione delle righe (WHERE) e infine vengono visualizzate le colonne desiderate (SELECT).

Contrariamente ad altri linguaggi, le stringhe e le date devono essere racchiuse tra due apostrofi (apici) e non tra le virgolette (doppie). Le doppie virgolette possono essere usate per racchiudere i nomi degli oggetti, come nomi di tabelle o nomi di colonne colonne, che contengono caratteri speciali, come lo spazio.

Ad esempio:

```
SELECT nomecolonna1,nomecolonna2
     FROM tabella
     WHERE "codice fiscale"='valore';
```

Esercizi svolti...

Congiunzione [1] FROM

La congiunzione è necessaria per rispondere alle interrogazioni che chiedono contemporaneamente accesso a dati contenuti in più tabelle. L'operazione permette di collegare tabelle legate da un vincolo di integrità referenziale. Il comando SELECT prevedere diverse possibilità tra loro equivalenti per formulare la richiesta di una stessa operazione di congiunzione tra due (o più) tabelle: il programmatore è libero di scegliere la forma che preferisce, ma per semplicità, in questo breve paragrafo, verrà descritto un solo modo.
Come esempio si utilizzerà un database costituito dalle seguenti tabelle:

cittadini

cf	cognome	nome	citta	pr
BNCNTN52	Bianchi	Antonia	Sorbolo	PR
RSSNDR60	Rossi	Andrea	Parma	PR
RSSNTN58	Rossi	Antonio	Parma	PR
SLVPTR54	Salvi	Pietro	Fidenza	PR

auto

targa	marca	modello	colore
AZ158WR	Fiat	Panda	bianco
BC404BB	Fiat	500	nero
EV373FB	Audi	A3	bianco

proprieta

cfcittadino	targaauto	data
BNCNTN52	BC404BB	2008-08-20
RSSNTN58	EV373FB	2003-01-30
RSSNDR60	EV373FB	1999-02-05

La richiesta è la seguente:
Visualizzare cognome, nome e data di possesso per ogni cittadino:

```
SELECT cognome,nome,data,targa
     FROM cittadini JOIN proprieta ON cf=cfcittadino ;
```

Il risultato visualizzato è il seguente:

cognome	nome	data	targa
Bianchi	Antonia	2008-08-20	BC404BB
Rossi	Antonio	2003-01-30	EV373FB
Rossi	Andrea	1999-02-05	EV373FB

Semplificando le cose, si può dire che:
la clausola SELECT è usata per la proiezione,
la clausola FORM è usata per la congiunzione,
la clausola WHERE è usata per la selezione.

Nell'ultima tabella manca il cittadino "Salvi" perché il suo codice fiscale non ha corrispondenza nella tabella "proprieta". Nel caso in cui si vogliano visualizzare anche i dati senza corrispondenza, si deve effettuare un JOIN ESTERNO al posto del JOIN INTERNO modificando il comando nel modo seguente:

```
SELECT cognome,nome,data,targa
     FROM cittadini LEFT JOIN proprieta ON cf=cfcittadino ;
```

Il LEFT JOIN conserva tutti gli elementi della tabella di sinistra (cittadini), anche quelli che non hanno una corrispondenza nella tabella di destra, completando la riga con eventuali valori NULL.
Il risultato visualizzato è il seguente:

cognome	nome	data	targa
Bianchi	Antonia	2008-08-20	BC404BB
Rossi	Antonio	2003-01-30	EV373FB
Rossi	Andrea	1999-02-05	EV373FB
Salvi	Pietro		

Il RIGHT JOIN svolge una funzione analoga con la tabella di destra (proprieta).
Il FULL OUTER JOIN unisce le funzioni dei due precedenti tipi di JOIN.

Nell'ipotetico caso in cui le colonne usate per unire in un JOIN le due tabelle abbiano lo **stesso nome** si può scrivere:

```
SELECT cognome,nome,data,targa
     FROM cittadini JOIN proprieta
          ON cittadini.cfcittadino=proprieta.cfcittadino ;
```

oppure più semplicemente si può usare il NATURAL JOIN:

```
SELECT cognome,nome,data,targa
     FROM cittadini NATURAL JOIN proprieta ;
```

Infine, la congiunzione può essere realizzata anche su più di due tabelle. Siccome la congiunzione è un operatore binario, però, per unire più di due tabelle, l'operatore JOIN deve essere usato più volte, avendo cura di usare le parentesi tonde per controllare le precedenze.

```
SELECT cognome,nome,data,targa,marca,modello
     FROM (cittadini JOIN proprieta ON cf=cfcittadino)
               JOIN auto ON targaauto=targa ;
```

cognome	nome	data	targa	marca	modello
Bianchi	Antonia	2008-08-20	BC404BB	Fiat	500
Rossi	Antonio	2003-01-30	EV373FB	Audi	A3
Rossi	Andrea	1999-02-05	EV373FB	Audi	A3

Selezione [2] WHERE

Operatori logici AND e OR

Poiché le condizioni della clausola WHERE sono condizioni booleane, è possibile usare i tradizionali operatori logici AND e OR per costruire condizioni complesse:

```
SELECT *
     FROM tabella
     WHERE cognome='Rossi' AND nome='Andrea' ;
```

- AND è un operatore che impone siano verificate **entrambe** le condizioni
- OR è un operatore che impone sia verificata **almeno una** delle due condizioni

Perciò, è impossibile che AND sia applicato tra due condizioni sulla stessa colonna

Esempio errato:

```
WHERE cognome='Rossi' AND cognome='Verdi' ;  -- errato
```

Operatore LIKE

Per le condizioni sulle stringhe di caratteri esiste l'operatore LIKE che consente di usare dei caratteri "jolly". Questa è la condizione senza caratteri jolly:

```
SELECT *
     FROM tabella
     WHERE cognome='Rossi' ;
```

Corrispondenza esatta di tutta la parola, mostra solo i cittadini di cognome 'Rossi'

```
SELECT *
     FROM tabella
     WHERE cognome LIKE 'R%'  ;
```

Il simbolo di percentuale funziona come un asterisco. Esempio di corrispondenza della sola lettera iniziale, che mostra sia i cittadini di cognome 'Rossi', che i cittadini 'Ricchi'

```
SELECT *
     FROM tabella
     WHERE nome LIKE '_ario'  ;
```

Il trattino basso (underscore) funziona come un qualsiasi carattere. Esempio di corrispondenza che mostra sia i cittadini di nome 'Mario', che 'Dario'.

Funzioni per la data e l'ora

Permettono di ottenere la data attuale e l'ora del server.

DBMS	data attuale	ora attuale	data e ora attuali
PostGreSQL	`current_date`	`current_time`	`current_timestamp`
LibreOffice	CURRENT_DATE	CURRENT_TIME	CURRENT_TIMESTAMP
SQLite	date('now')	time('now')	datetime('now')
MySQL	`CURDATE()`	`CURTIME()`	`NOW()`

Cosa accade se utilizzo questo?

```
SELECT 5-2;
```

Per ottenere la differenza tra due date espressa in giorni:

DBMS	differenza tra date	intervalli di giorni traformati in anni
PostGreSQL		
LibreOffice		
SQLite		
MySQL	`DATEDIFF()`	`FROM_DAYS()`

Queste funzioni permettono, ad esempio, di calcolare l'età ATTUALE di una persona a partire dalla data di nascita, facendo una differenza tra date.
La differenza tra due valori di tipo DATE dovrebbe produrre un risultato in formato DATE ma può produrre un risultato espresso in giorni a seconda della funzione utilizzata.
Ad esempio, ipotizzando di usare MYSQL e che la tabella cittadini contenga anche la data di nascita, se desideriamo l'età nella forma (giorno, mese, anno) oppure (anno, mese, giorno) si deve usare:

```
SELECT * FROM_DAYS(DATEDIFF(CURDATE(),datadinascita)) AS eta
FROM cittadini;
```

Funzioni di aggregazione [5] SELECT

A livello del comando SELECT, possono essere utilizzate delle funzioni che elaborano i dati poco prima della visualizzazione. Le funzioni si riconoscono dal fatto che terminano con le parentesi tonde...
Alcune di queste funzioni sono: COUNT(), MAX(), MIN(), AVG() e SUM().

- COUNT() visualizza il numero delle righe di una determinata colonna
- MAX() e MIN() visualizza il valore massimo/minimo contenuto in una colonna
- AVG() e SUM() visualizza il valore medio/la somma dei valori contenuti in una colonna

Nota: AVG() e SUM() possono essere applicate solo su dati numerici.

Le funzioni di aggregazione sono operatori che vengono eseguite dopo SELECT. Agiscono su tutte le righe ottenute complessivamente dal comando SELECT, e producono una sola riga.

Esempio:

Questo comando può visualizzare più di una riga:

```
SELECT cognome,nome
     FROM cittadini
     WHERE cognome='Rossi' ;
```

e produce il seguente risultato:

cognome	nome
Rossi	Andrea
Rossi	Antonio

Questo comando può visualizzare una sola riga:

```
SELECT COUNT(cognome)
     FROM cittadini
     WHERE cognome='Rossi' ;
```

e produce il seguente risultato:

COUNT(nome)
2

Questo comando, invece, produce un **errore**:

```
SELECT cognome,COUNT(cognome)
     FROM cittadini
     WHERE cognome='Rossi' ; -- errore
```

Nella colonna `cognome` ci sarebbero due righe e nella colonna `COUNT(nome)` ce ne sarebbe una sola, mentre in una tabella tutte le colonne devono avere lo stesso numero di righe!

IMPORTANTE

1. Ricordare che SELECT viene eseguito per ultimo rispetto a FROM e WHERE
2. Le funzioni di aggregazione si utilizzano su SELECT (non su FROM o WHERE).
3. Quando si utilizzano le funzioni di aggregazione, non si devono specificare altre colonne senza "aggregazione" (tuttavia esiste un'eccezione a questa regola).

Clausola di aggregazione [3] GROUP BY

Il suo **primo effetto** è quello di "raggruppare" o "aggregare" gruppi di righe. Ogni riga di ogni gruppo ha lo stesso valore su una certa colonna (si veda l'esempio).
Il suo **secondo effetto** è di ripetere le espressioni indicate a livello di SELECT, tante volte quanti sono i gruppi di righe, questo significa che si può usare solo inseme alle funzioni di aggregazione.

Si è già visto che le funzioni di aggregazione hanno l'effetto di produrre una tabella con una sola riga, ma ora, aggiungendo la clausola GROUP BY, la funzione di aggregazione viene ripetuta per ogni gruppo di righe e il risultato dell'interrogazione può produrre anche più righe (una per ogni gruppo).
Esempio in cui, scrivendo `GROUP BY cognome`, i gruppi di righe saranno formati in base ad un valore comune sulla colonna `cognome`.

```
SELECT cognome,COUNT(cognome)
     FROM cittadini
     GROUP BY cognome ;
```

Bisogna ricordare di eseguire prima FROM, poi GROUP BY (che crea i gruppi di righe) e infine SELECT:

cognome	COUNT(nome)
Bianchi	1
Rossi	2
Salvi	1

Nelle precedenti pagine si è detto che "quando si utilizzano le funzioni di aggregazione non si devono specificare altre colonne senza funzione di aggregazione". L'ultimo esempio è un'eccezione a questa regola: la clausola `GROUP BY cognome` ha come effetto quello di "aggregare" la colonna `cognome` e di poterla affiancare a `COUNT(nome)` senza problemi.

IMPORTANTE

Si può quindi dire che:

4. In un'interrogazione SELECT che contiene funzioni di aggregazione possono essere visualizzate anche altre colonne (senza funzione di aggregazione), a condizione che siano tutte contenute dentro la clausola `GROUP BY`
5. Come detto, GROUP BY si può usare solo insieme alle funzioni di aggregazione

Per concludere, si osserva che GROUP BY può essere usata insieme a WHERE. Nello stesso esempio viene aggiunta anche una selezione delle righe tramite la clausola `WHERE`:

```
SELECT cognome,COUNT(cognome)
     FROM cittadini
     WHERE cognome='Rossi'
     GROUP BY cognome  ;
```

Che produce il seguente risultato:

cognome	**COUNT(cognome)**
Rossi	2

Selezione nei sottogruppi di righe [4] HAVING

HAVING è una clausola il cui funzionamento ricorda quello di WHERE, con la seguente differenza:

- La clausola WHERE indica una condizione di selezione che viene eseguita PRIMA di GROUP BY.
- La clausola HAVING indica una condizione di selezione che viene eseguita DOPO la clausola GROUP BY, quando cioè le funzioni di aggregazione sono state già applicate ad ogni gruppo di righe.

HAVING può contenere una condizione su *qualsiasi* colonna indicata in SELECT.

In questo esempio, HAVING può contenere una funzione di aggregazione (COUNT) perché tale funzione è presente in SELECT.

Esempio:

```
SELECT cognome,COUNT(cognome)
     FROM cittadini
     GROUP BY cognome
     HAVING  COUNT(cognome)>1 ;
```

IMPORTANTE

6. Anche per la clausola HAVING come GROUP BY, può essere usata solo nel caso in cui dentro SELECT si utilizzi qualche funzioni di aggregazione.
7. Dopo SELECT, la clausola HAVING è l'unica che può contenere funzioni di aggregazione.

Ordinamenti [7]ORDER BY

I dati nel database vengono inseriti in modo casuale (non si segue nessun ordine). Per presentare meglio il risultato di una tabella può essere utile ordinare le righe in base ad una o più colonne, in modo crescente o decrescente.
La clausola ORDER BY può contenere il nome di *qualsiasi* colonna indicata in SELECT.

```
SELECT cognome,COUNT(cognome)
     FROM cittadini
     GROUP BY cognome
     ORDER BY COUNT(cognome),cognome;
```

Il comando ordina i risultati di una delle precedenti interrogazioni in modo crescente per la colonna COUNT(nome) e, a parità di valore in questa colonna, per valore crescente del cognome.

cognome	COUNT(cognome)
Bianchi	1
Salvi	1
Rossi	2

Per ordinare in modo inverso si aggiunge l'opzione DESC dopo il nome della colonna o delle colonne:

```
     ORDER BY COUNT(cognome) DESC, cognome DESC;
```

Limit e offset [9]LIMIT

La clausola LIMIT consente di vedere solo le prime righe della tabella ottenuta come risultato dell'interrogazione. La seguente interrogazione visualizza solo le prime 2 righe.

Per questo motivo LIMIT ha senso solo se è utilizzata insieme a ORDER BY.

```
SELECT cognome,nome
     FROM cittadini
     ORDER BY cognome
     LIMIT 2 ;
```

[10] OFFSET

OFFSET consente di escludere le prime righe della tabella ottenuta come risultato dell'interrogazione. La seguente interrogazione visualizza solo le prime 2 righe, a partire dalla riga dopo la numero 3.

```
SELECT cognome,nome
     FROM cittadini
     ORDER BY cognome
     LIMIT 2
     OFFSET 3;
```

Interrogazioni parametriche

Nel seguente esempio si cercano i cittadini di cognome 'Rossi'

```
SELECT *
     FROM cittadini
     WHERE cognome='Rossi';
```

Nel caso in cui l'utente voglia ogni volta cercare un cognome diverso, il programmatore può scrivere un comando più "flessibile" con un cognome "variabile", che l'utente deve specificare prima di eseguire l'interrogazione.

Purtroppo il linguaggio SQL non prevede l'uso di variabili.

Il linguaggio SQL può essere usato all'interno di altri linguaggi di programmazione e la forma usata per indicare la variabile dipende dal linguaggio usato dal programmatore.

Nel caso in cui il comando SELECT venga utilizzato all'interno di una pagina dinamica PHP, ogni variabile deve iniziare con il simbolo "dollaro". Per ulteriori chiarimenti si rimanda ad un corso di programmazione in linguaggio PHP.

L'interrogazione sarà contenuta in una string

```
$var="SELECT * FROM cittadini WHERE cognome='".$varcognome."';";
```

Le pagine dinamiche possono essere create anche in molti altri linguaggi di programmazione, ed avere come estensioni: PHP, CGI, JSP, ASP, PY

Approfondimento su PostGreSQL

Gestione delle prestazioni del DBMS

EXPLAIN comando-sql;

Usando EXPLAIN davanti ad un comando SQL si visualizza il piano di esecuzione del *comando-sql*, senza eseguirlo effettivamente:

Mostra come le tabelle verranno lette, quali algoritmi verranno usati.
L'esecuzione di ogni comando ha un "costo di esecuzione" diverso da un altro comando.
Il costo di esecuzione può essere stimato e mostrato, mostrando la stima dei seguenti valori:
-numero di pagine del disco lette (dipende dalla dimensione della tabella)
-tempo di preparazione (start-up time) (prima della prima riga del risultato)
-tempo totale (total time) (dopo tutte le righe del risultato)

```
"Index Scan using auto_pkey on tabella  (cost=0.00..8.27 rows=1 width=246)"
"  Index Cond: (telefono='099999'::bpchar)"
```

Il pianificatore delle istruzioni decide in che modo eseguire il comando e quale algoritmo usare, in base a quello che ha stimato avere un minore "total time".

EXPLAIN ANALYZE comando-sql;

Visualizza il piano di esecuzione del *comando-sql* e lo esegue effettivamente, ma SENZA mostrare il risultato del comando.

Oltre ai tempi stimati dal pianificatore vengono mostrati anche i tempi che sono stati effettivamente impiegati.

ANALYZE;

Raccoglie informazioni di tipo statistico sulla distribuzione dei dati nelle tabelle.

Per poter scegliere il miglior metodo per eseguire un comando, il pianificatore deve conoscere come sono distribuiti i dati nelle tabelle.
Per avere queste informazioni statistiche bisogna periodicamente eseguire il comando ANALYZE .
Questo comando dovrebbe essere esguito anche quando si cancellano o si inseriscono molte righe in una tabella, per aggiornare le informazioni statistiche sui dati.

VACUUM;

Rimuove dal disco i dati eliminati nelle tabelle, ancora presenti nel cestino.

Anche questo comando permette di migliorare la velocità di esecuzione delle interrogazioni. Dopo un comando VACUUM è molto utile eseguire anche ANALIZE per aggiornare le statistiche dei dati effettivamente presenti nelle tabelle.