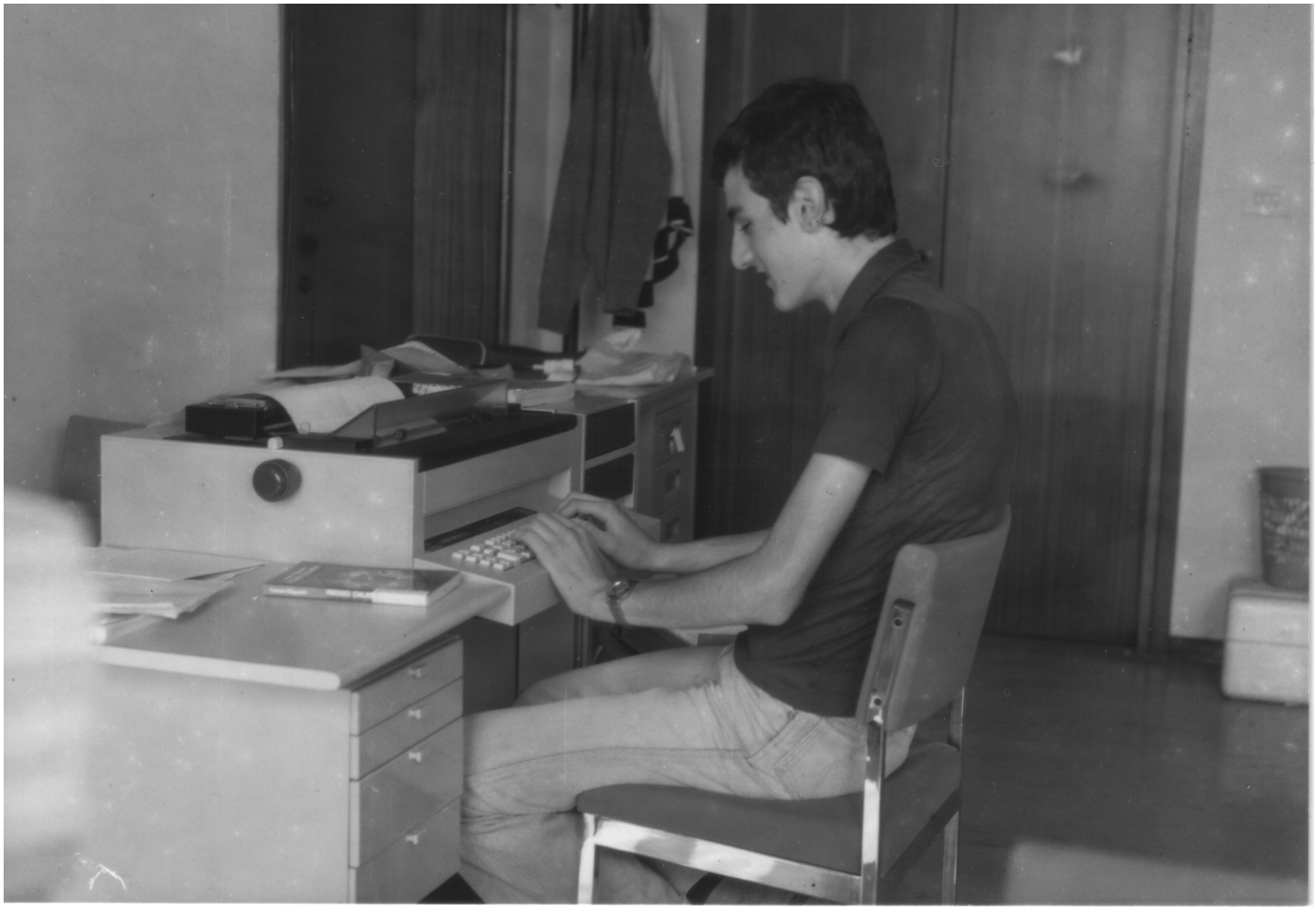
Parte v

Basi di dati

Figura v.1. Uno studente all'opera con una macchina TES-501 Olivetti, nel 1979, per la catalogazione dei libri della biblioteca scolastica.



1862

Capitolo 74

DBMS e SQL

74.1 Introduzione ai DBMS

«

Un DBMS (*Data base management system*) è, letteralmente, un sistema di gestione di ***basi di dati***, che per attuare questa gestione utilizza il software. Queste «basi di dati» sono dei contenitori atti a immagazzinare una grande quantità di dati, per i quali il «sistema di gestione» è necessario a permetterne la fruizione (diverso è il problema della semplice archiviazione dei dati).

74.1.1 Caratteristiche fondamentali

Un DBMS, per essere considerato tale, deve avere caratteristiche determinate. Le più importanti che permettono di comprenderne il significato sono elencate di seguito.

- Un DBMS è fatto per gestire grandi quantità di dati.

  Convenzionalmente si può intendere che un gruppo di informazioni sia di grandi dimensioni quando questo non possa essere contenuto tutto simultaneamente nella memoria centrale dell'elaboratore. In generale un DBMS non dovrebbe porre limiti alle dimensioni, tranne quelle imposte dai supporti fisici in cui devono essere memorizzate le informazioni.

- I dati devono poter essere condivisibili.

  L'idea che sta alla base dei sistemi di gestione dei dati è quella di accentrare le informazioni in un sistema di amministrazione unico. In tal senso è poi necessario che questi dati siano condivisibili da diverse applicazioni e da diversi utenti.

- I dati devono essere persistenti e affidabili.

  I dati sono persistenti quando continuano a esistere dopo lo spegnimento della macchina con cui vengono elaborati; sono affidabili quando gli eventi per cui si possono produrre alterazioni accidentali sono estremamente limitati.

- L'accesso ai dati deve essere controllabile.

  Dovendo trattare una grande mole di dati in modo condiviso, è indispensabile che esistano dei sistemi di controllo degli accessi, per evitare che determinate informazioni possano essere ottenute

da chi non è autorizzato, oppure che vengano modificate da chi non ne è il responsabile.

74.1.1.1 Livelli di astrazione dei dati

I dati gestiti da un DBMS devono essere organizzati a diversi livelli di astrazione. Generalmente si distinguono tre livelli: esterno, logico e interno.

- Il livello interno è quello usato effettivamente per la memorizzazione dei dati. In pratica è rappresentato dai file che contengono effettivamente le informazioni e dal modo con cui questi file vengono utilizzati. Il livello interno non è importante per la progettazione di una base di dati e nemmeno per la scrittura di programmi che devono interagire con il DBMS.
- Il livello logico, o concettuale, è quello che descrive i dati secondo la filosofia del DBMS particolare con cui si ha a che fare.
- Lo schema esterno è un'astrazione aggiuntiva che permette di definire dei punti di vista differenti dei dati descritti a livello logico. L'accesso ai dati avviene solo a questo livello, anche se di fatto può coincidere con il livello logico.

74.1.1.2 Ruoli e sigle standard

Lo studio sui DBMS ha generato degli acronimi che rappresentano persone o componenti essenziali di un sistema del genere. Questi acronimi devono essere conosciuti perché se ne fa uso abitualmente nella documentazione riferita ai DBMS.

L'organizzazione di una base di dati è compito del suo amministratore, definito DBA (*Data base administrator*). Eventualmente può trattarsi anche di più persone; in ogni caso, chi ha la responsabilità dell'amministrazione di uno o più basi di dati è un DBA.

Il concetto di DBA non è molto preciso, perché include assieme ruoli che possono essere differenti. Si va dall'amministratore dell'intero DBMS, fino a coloro che hanno la responsabilità di una base di dati singola.

La definizione della struttura dei dati (sia a livello logico, sia a livello esterno) viene fatta attraverso un linguaggio di programmazione definito DDL (*Data definition language*), la gestione dei dati avviene attraverso un altro linguaggio, detto DML (*Data manipulation language*), infine, la gestione della sicurezza viene fatta attraverso un linguaggio DCL (*Data control language*). Nella pratica, DDL, DML e DCL possono coincidere, come nel caso del linguaggio SQL.

74.1.2 Modello relazionale

Una base di dati può essere impostata secondo diversi tipi di modelli (logici) di rappresentazione. Quello più comune, che è anche il più semplice dal punto di vista umano, è il modello relazionale. In tal senso, un DBMS relazionale viene anche definito semplicemente come RDBMS.

Nel modello relazionale, i dati sono raccolti all'interno di ***relazioni***. Ogni relazione è una raccolta di nessuna o più ***tuple*** di tipo omogeneo. La tupla rappresenta una singola informazione completa, in rapporto alla relazione a cui appartiene, suddivisa in ***attributi***. Le in-

formazioni che possono essere inserite in ogni singolo attributo, dipendono da un ***dominio***. Una relazione, nella sua definizione, non ha una «forma» particolare, tuttavia questo concetto si presta a una rappresentazione tabellare: gli attributi sono rappresentati dalle colonne e le tuple dalle righe. Si osservi l'esempio della figura 74.1.

Figura 74.1. Relazione '**`Indirizzi (Cognome, Nome, Indirizzo, Telefono)`**'.

Indirizzi			
Cognome	**Nome**	**Indirizzo**	**Telefono**
Pallino	Pinco	Via Biglie 1	0222,222222
Tizi	Tizio	Via Tazi 5	0555,555555
Cai	Caio	Via Caini 1	0888,888888
Semproni	Sempronio	Via Sempi 7	0999,999999

In una relazione, le tuple non hanno una posizione particolare, sono semplicemente contenute nell'insieme della relazione stessa. Se l'ordine ha una rilevanza per le informazioni contenute, questo elemento dovrebbe essere aggiunto tra gli attributi, senza essere determinato da un'ipotetica collocazione fisica. Osservando l'esempio, si intende che l'ordine delle righe non ha importanza per le informazioni che si vogliono trarre; al massimo, un elenco ordinato può facilitare la lettura umana, quando si è alla ricerca di un nome particolare, ma ciò non ha rilevanza nella struttura che deve avere la relazione corrispondente.

Il fatto che la posizione delle tuple all'interno della relazione non

sia importante, significa che non è necessario poterle identificare: le tuple si distinguono in base al loro contenuto. In questo senso, una relazione non può contenere due tuple uguali: la presenza di doppioni non avrebbe alcun significato.

A differenza delle tuple, gli attributi devono essere identificati attraverso un nome. Infatti, il semplice contenuto delle tuple non è sufficiente a stabilire di quale attributo si tratti. Osservando la prima riga dell'esempio, diventa difficile distinguere quale sia il nome e quale il cognome:

Pallino	Pinco	Via Biglie 1	0222,222222

Assegnando agli attributi (cioè alle colonne) un nome, diventa indifferente la disposizione fisica di questi all'interno delle tuple.

74.1.2.1 Relazioni collegate

Generalmente, una relazione da sola non è sufficiente a rappresentare tutti i dati riferiti a un problema o a un interesse della vita reale. Quando una relazione contiene tante volte le stesse informazioni, è opportuno scinderla in due o più relazioni più piccole, collegate in qualche modo attraverso dei riferimenti. Si osservi il caso delle relazioni rappresentate dalle tabelle che si vedono nella figura 74.3.

Figura 74.3. Relazioni di un'ipotetica gestione del magazzino.

Articoli

Codice	Descrizione	Fornitore1	Fornitore2
vite30	Vite 3 mm	123	126
vite40	Vite 4 mm	126	127
dado30	Dado 3 mm	122	123
dado40	Dado 4 mm	126	127
rond50	Rondella 5 mm	123	126

Movimenti

Codice	Data	Carico	Scarico	CodFor	CodCli
vite40	01/01/2012	1200		124	
vite30	01/01/2012		800		825
vite30	02/01/2012	1000		954	
vite30	03/01/2012	2000		127	
rond50	03/01/2012		500		954

Fornitori

CodFor	Ditta	Indirizzo	Telefono
127	Vitoni spa	Via Ferri 2	0123,45678
122	Ferroni spa	Via Metalli 34	0234,5678
126	Nuova Metal	Via Industrie	0345,6789
123	Viti e Bulloni	Via di sopra 7	0567,9875

Clienti

CodCli	Ditta	Indirizzo	Telefono
925	Tendoni Max	Via di sotto 2	0113,44578
825	Arti Plus	Via di lato 45	0765,23456

La prima relazione, '**`Articoli`**', rappresenta l'anagrafica del magazzino di un grossista di ferramenta. Ogni articolo di magazzino viene codificato e descritto, inoltre vengono annotati i riferimenti ai codici di possibili fornitori. La seconda relazione, '**`Movimenti`**', elenca le operazioni di carico e di scarico degli articoli di magazzino, specificando solo il codice dell'articolo, la data, la quantità caricata o scaricata e il codice del fornitore o del cliente da cui è stato acquistato o a cui è stato venduto l'articolo. Infine seguono le relazioni che descrivono i codici dei fornitori e quelli dei clienti.

Si può intendere che una sola relazione non avrebbe potuto essere utilizzata utilmente per esprimere tutte queste informazioni.

È importante stabilire che, nel modello relazionale, il collegamento tra le tuple delle varie relazioni avviene attraverso dei valori e non

attraverso dei puntatori. Infatti, nella relazione '**Articoli**' l'attributo '**Fornitore1**' contiene il valore 123 e questo significa solo che i dati di quel fornitore sono rappresentati da quel valore. Nella relazione '**Fornitori**', la tupla il cui attributo '**CodFor**' contiene il valore 123 è quella che contiene i dati di quel particolare fornitore. Quindi, «123» non rappresenta un puntatore, ma solo una tupla che contiene quel valore nell'attributo «giusto». In questo senso si ribadisce l'indifferenza della posizione delle tuple all'interno delle relazioni.

74.1.2.2 Tipi di dati, domini e informazioni mancanti

Nelle relazioni, ogni attributo contiene una singola informazione elementare di un certo tipo, per il quale esiste un dominio determinato di valori possibili. Ogni attributo di ogni tupla deve contenere un valore ammissibile, nell'ambito del proprio dominio.

Spesso capitano situazioni in cui i valori di uno o più attributi di una tupla non sono disponibili per qualche motivo. In tal caso si pone il problema di assegnare a questi attributi un valore che definisca in modo non ambiguo questo stato di indeterminatezza. Questo valore viene definito come '**NULL**' ed è ammissibile per tutti i tipi di attributi possibili.

74.1.2.3 Vincoli di validità

I dati contenuti in una o più relazioni sono utili in quanto «sensati» in base al contesto a cui si riferiscono. Per esempio, considerando la relazione '**Movimenti**', vista precedentemente, questa deve contenere sempre un codice valido nell'attributo '**Codice**'. Se così non fosse, la registrazione data da quella tupla che dovesse avere un riferimen-

to a un codice di articolo non valido, non avrebbe alcun senso, perché mancherebbe proprio l'informazione più importante: l'articolo caricato o scaricato.

Il controllo sulla validità dei dati può avvenire a diversi livelli, a seconda della circostanza. Si possono distinguere vincoli che riguardano:

1. il dominio dell'attributo stesso -- quando si tratta di definire se l'attributo può assumere il valore '`NULL`' o meno e quando si stabilisce l'intervallo dei valori ammissibili;
2. gli altri attributi della stessa tupla -- quando dal valore contenuto in altri attributi della stessa tupla dipende l'intervallo dei valori ammissibili per l'attributo in questione;
3. gli attributi di altre tuple -- quando dal valore contenuto negli attributi delle altre tuple della stessa relazione dipende l'intervallo dei valori ammissibili per l'attributo in questione;
4. gli attributi di tuple di altre relazioni -- quando altre relazioni condizionano la validità di un attributo determinato.

I vincoli di tupla, ovvero quelli che riguardano i primi due punti dell'elenco appena indicato, sono i più semplici da esprimere perché non occorre conoscere altre informazioni esterne alla tupla stessa. Per esempio, un attributo che esprime un prezzo potrebbe essere definito in modo tale che non sia ammissibile un valore negativo; nello stesso modo, un attributo che esprime uno sconto su un prezzo potrebbe ammettere un valore positivo diverso da zero solo se il prezzo a cui si riferisce, contenuto nella stessa tupla, supera un valore determinato.

Il caso più importante di un vincolo interno alla relazione, che coinvolge più tuple, è quello che riguarda le ***chiavi***. In certe situazioni, un attributo, o un gruppo particolare di attributi di una relazione, deve essere unico per ogni tupla. Quindi, questo attributo, o questo gruppo di attributi, è valido solo quando non è già presente in un'altra tupla della stessa relazione.

Quando le informazioni sono distribuite fra più relazioni, i dati sono validi solo quando tutti i riferimenti sono validi. Volendo riprendere l'esempio della gestione di magazzino, visto precedentemente, una tupla della relazione '**`Movimenti`**' che dovesse contenere un codice di un fornitore o di un cliente inesistente, si troverebbe, in pratica, senza questa informazione.

74.1.2.4 Chiavi

Nella sezione precedente si è accennato alle ***chiavi***. Questo concetto merita un po' di attenzione. In precedenza è stato affermato che una relazione contiene una raccolta di tuple che contano per il loro contenuto e non per la loro posizione. In questo senso non è ammissibile una relazione contenente due tuple identiche. Una ***chiave*** di una relazione è un gruppo di attributi che permette di identificare univocamente le tuple in essa contenute; per questo, tali attributi devono contenere dati differenti per ogni tupla.

Stabilendo quali attributi devono costituire una chiave per una certa relazione, si comprende intuitivamente che questi attributi non possono mai contenere un valore indeterminato.

Nella definizione di relazioni collegate attraverso dei riferimenti, l'oggetto di questi riferimenti deve essere una chiave per la relazio-

ne di destinazione. Diversamente non si otterrebbe un riferimento univoco a una tupla particolare.

74.1.3 Gestione delle relazioni

«

Prima di affrontare l'utilizzo pratico di una base di dati relazionale, attraverso un linguaggio di manipolazione dei dati, è opportuno considerare a livello teorico alcuni tipi di operazioni che si possono eseguire con le relazioni.

Inizialmente è stato affermato che una relazione è un insieme di tuple... Dalla teoria degli insiemi derivano molte delle operazioni che riguardano le relazioni.

74.1.3.1 Unione, intersezione e differenza

«

Quando si maneggiano relazioni contenenti gli stessi attributi, hanno senso le operazioni fondamentali sugli insiemi: unione, intersezione e differenza. Il significato è evidente: l'unione genera una relazione composta da tutte le tuple distinte delle relazioni di origine; l'intersezione genera una relazione composta dalle tuple presenti simultaneamente in tutte le relazioni di origine; la differenza genera una relazione contenente le tuple che compaiono esclusivamente nella prima delle relazioni di origine.

L'esempio rappresentato dalle relazioni della figura 74.4 dovrebbe chiarire il senso di queste affermazioni.

Figura 74.4. Unione, intersezione e differenza tra relazioni.

Laureati

Codice	Nominativo	...
1245	Tizi Tizio	...
1745	Cai Caio	...
1655	Semproni Sempronio	...

Magazzinieri

Codice	Nominativo	...
1745	Cai Caio	...
1986	Pallino Pinco	...
1245	Tizi Tizio	...

Laureati UNITO Magazzinieri

Codice	Nominativo	...
1245	Tizi Tizio	...
1745	Cai Caio	...
1655	Semproni Sempronio	...
1986	Pallino Pinco	...

Laureati INTERSECATO Magazzinieri

Codice	Nominativo	...
1245	Tizi Tizio	...
1745	Cai Caio	...

Laureati MENO Magazzinieri

Codice	Nominativo	...
1655	Semproni Sempronio	...

74.1.3.2 Ridenominazione degli attributi

«

L'elaborazione dei dati contenuti in una relazione può avvenire previa modifica dei nomi di alcuni attributi. La modifica dei nomi genera di fatto una nuova relazione temporanea, per il tempo necessario a eseguire l'elaborazione conclusiva.

Le situazioni in cui la ridenominazione degli attributi può essere conveniente possono essere varie. Nel caso delle operazioni sugli insiemi visti nella sezione precedente, la ridenominazione può rendere compatibili relazioni i cui attributi, pur essendo compatibili, hanno nomi differenti.

74.1.3.3 Selezione, proiezione e congiunzione

«

La ***selezione*** e la ***proiezione*** sono operazioni che si eseguono su una sola relazione e generano una relazione che contiene una porzione dei dati di quella di origine. La selezione permette di estrarre alcune tuple dalla relazione, mentre la proiezione estrae parte degli attributi di tutte le tuple. Il primo caso, quello della selezione, non richiede considerazioni particolari, mentre la proiezione ha delle implicazioni importanti.

Attraverso la proiezione, utilizzando solo parte degli attributi, si genera una relazione in cui si potrebbero perdere delle tuple, a causa della possibilità che risultino dei doppioni. Per esempio, si consideri la relazione mostrata nella figura 74.5.

Figura 74.5. Relazione '**`Utenti (UID, Nominativo, Cognome, Nome, Ufficio)`**'.

Utenti				
UID	**Nominativo**	**Cognome**	**Nome**	**Ufficio**
0	root	Pallino	Pinco	CED
515	rmario	Rossi	Mario	Contabilità
501	bbianco	Bianchi	Bianco	Magazzino
502	rrosso	Rossi	Rosso	Contabilità

La figura mostra una relazione contenente le informazioni sugli utenti di un centro di elaborazione dati. Si può osservare che sia l'attributo '**`UID`**', sia l'attributo '**`Nominativo`**', possono essere da soli una chiave per la relazione. Se da questa relazione si vuole ottenere una proiezione contenente solo gli attributi '**`Cognome`**' e '**`Ufficio`**', non essendo questi due una chiave della relazione, si perdono delle tuple.

Figura 74.6. Proiezione degli attributi '**`Cognome`**' e '**`Ufficio`**' della relazione '**`Utenti`**'.

Cognome	**Ufficio**
Pallino	CED
Rossi	Contabilità
Bianchi	Magazzino

La figura 74.6 mostra la proiezione della relazione '**`Utenti`**', in cui

sono stati estratti solo gli attributi '**`Cognome`**' e '**`Ufficio`**'. In tal modo, le tuple che prima corrispondevano al numero '**`UID`**' 515 e 502 si sono ridotte a una sola, quella contenente il cognome «Rossi» e l'ufficio «Contabilità».

Da questo esempio si dovrebbe intendere che la proiezione ha senso, prevalentemente, quando gli attributi estratti costituiscono una chiave della relazione originaria

La ***congiunzione*** di relazioni, o *join*, è un'operazione in cui due o più relazioni vengono unite a formare una nuova relazione. Questo congiungimento implica la creazione di tuple formate dall'unione di tuple provenienti dalle relazioni di origine. Se per semplicità si pensa solo alla congiunzione di due relazioni: si va da una congiunzione minima in cui nessuna tupla della prima relazione risulta abbinata ad altre tuple della seconda, fino a un massimo in cui si ottengono tutti gli abbinamenti possibili delle tuple della prima relazione con quelle della seconda. Tra questi estremi si pone la situazione tipica, quella in cui ogni tupla della prima relazione viene collegata solo a una tupla corrispondente della seconda.

La ***congiunzione naturale*** si ottiene quando le relazioni oggetto di tale operazione vengono collegate in base ad attributi aventi lo stesso nome. Per osservare di cosa si tratta, vale la pena di riprendere l'esempio della gestione di magazzino già descritto in precedenza. Nella figura 74.7 ne viene mostrata nuovamente solo una parte.

Figura 74.7. Le relazioni '**`Articoli`**' e '**`Movimenti`**' dell'esempio sulla gestione del magazzino.

Articoli		
Codice	**Descrizione**	**...**
vite30	Vite 3 mm	...
vite40	Vite 4 mm	...
dado30	Dado 3 mm	...
dado40	Dado 4 mm	...
rond50	Rondella 5 mm	...

Movimenti				
Codice	**Data**	**Carico**	**Scarico**	**...**
vite40	01/01/2012	1200		...
vite30	01/01/2012		800	...
vite30	02/01/2012	1000		...
vite30	03/01/2012	2000		...
rond50	03/01/2012		500	...

La congiunzione naturale delle relazioni '**`Movimenti`**' e '**`Articoli`**', basata sulla coincidenza del contenuto dell'attributo '**`Codice`**', genera una relazione in cui appaiono tutti gli attributi delle due relazioni di origine, con l'eccezione dell'attributo '**`Codice`**' che appare una volta sola (figura 74.8).

Tabella 74.8. Il join naturale tra le relazioni '**Movimenti**' e '**Articoli**'.

Codice	**Data**	**Carico**	**Scarico**	**...**	**Descrizione**	**...**
vite40	01/01/2012	1200		...	Vite 4 mm	...
vite30	01/01/2012		800	...	Vite 3 mm	...
vite30	02/01/2012	1000		...	Vite 3 mm	...
vite30	03/01/2012	2000		...	Vite 3 mm	...
rond50	03/01/2012		500	...	Rondella 5 mm	...

Nel caso migliore, ogni tupla di una relazione trova una tupla corrispondente dell'altra; nel caso peggiore, nessuna tupla ha una corrispondente nell'altra relazione. L'esempio mostra che tutte le tuple della relazione '**Movimenti**' hanno trovato una corrispondenza nella relazione '**Articoli**', mentre solo alcune tuple della relazione '**Articoli**' hanno una corrispondenza dall'altra parte. Queste tuple, quelle che non hanno una corrispondenza, sono dette «penzolanti», o *dangling*, e di fatto vengono perdute dopo la congiunzione.

Quando una congiunzione genera corrispondenze per tutte le tuple delle relazioni coinvolte, si parla di congiunzione completa. La dimensione (espressa in quantità di tuple) della relazione risultante in presenza di una congiunzione completa è pari alla dimensione massima delle varie relazioni.

Quando si vuole eseguire la congiunzione di relazioni che non hanno attributi in comune si ottiene il collegamento di ogni tupla di una relazione con ogni tupla delle altre. Si può osservare l'esempio della

figura 74.9 che riprende il solito problema del magazzino, con delle semplificazioni opportune.

Figura 74.9. Le relazioni '**`Articoli`**' e '**`Fornitori`**' dell'esempio sulla gestione del magazzino.

Articoli			
Codice	**Descrizione**	**Fornitore1**	**...**
vite30	Vite 3 mm	127	...
vite40	Vite 4 mm	127	...
dado30	Dado 3 mm	122	...

Fornitori		
CodFor	**Ditta**	**...**
127	Vitoni spa	...
122	Ferroni spa	...

Nessuno degli attributi delle due relazioni coincide, quindi si ottiene un «prodotto» tra le due relazioni, in pratica, una relazione che contiene il prodotto delle tuple contenute nelle relazioni originali (figura 74.10).

Figura 74.10. Il prodotto tra le relazioni '**Articoli**' e '**Fornitori**'.

Codice	Descrizione	Fornitore1	...	CodFor	Ditta	...
vite30	Vite 3 mm	127	...	127	Vitoni spa	...
vite40	Vite 4 mm	127	...	127	Vitoni spa	...
dado30	Dado 3 mm	122	...	127	Vitoni spa	...
vite30	Vite 3 mm	127	...	122	Ferroni spa	...
vite40	Vite 4 mm	127	...	122	Ferroni spa	...
dado30	Dado 3 mm	122	...	122	Ferroni spa	...

Quando si esegue un'operazione del genere, è normale che molte delle tuple risultanti siano prive di significato per gli scopi che ci si prefigge. Di conseguenza, quasi sempre, si applica poi una selezione attraverso delle condizioni. Nel caso dell'esempio, sarebbe ragionevole porre come condizione di selezione l'uguaglianza tra i valori dell'attributo '**Fornitore1**' e '**CodFor**'.

Figura 74.11. La selezione delle tuple che rispettano la condizione di uguaglianza tra gli attributi '**Fornitore1**' e '**CodFor**'.

Codice	Descrizione	Fornitore1	...	CodFor	Ditta	...
vite30	Vite 3 mm	127	...	127	Vitoni spa	...
vite40	Vite 4 mm	127	...	127	Vitoni spa	...
dado30	Dado 3 mm	122	...	122	Ferroni spa	...

Generalmente, nella pratica, non esiste la possibilità di definire una congiunzione basato sull'uguaglianza dei nomi degli attributi. Di solito si esegue una congiunzione che genera un prodotto tra le relazioni, quindi si applicano delle condizioni di selezione come nell'esempio mostrato. Quando la selezione in questione è del tipo visto nell'esempio, cioè basata sull'uguaglianza del contenuto di attributi delle diverse relazioni (anche se il nome di questi attributi è differente), si parla di ***equi-giunzione*** (*equi-join*).

74.1.3.4 Gestione dei valori nulli

Si è accennato in precedenza alla possibilità che gli attributi di una relazione possano contenere anche il valore indeterminato, o '**`NULL`**'. Con questo valore indeterminato non si possono fare comparazioni con valori determinati e di solito nemmeno con altri valori indeterminati. Per esempio, non si può dire che '**`NULL`**' sia maggiore o minore di qualcosa; una comparazione di questo tipo genera solo un risultato indeterminato. '**`NULL`**' è solo uguale a se stesso ed è diverso da ogni altro valore, compreso un altro valore '**`NULL`**'.

Per verificare la presenza o l'assenza di un valore indeterminato si utilizzano generalmente operatori specifici, come in SQL:

- '**`IS NULL`**' -- che si avvera quando il valore controllato è indeterminato;
- '**`IS NOT NULL`**' -- che si avvera quando il valore controllato è determinato, quindi diverso da indeterminato.

Nel momento in cui si eseguono delle espressioni logiche, utilizzando i soliti operatori AND, OR e NOT, si pone il problema di stabilire cosa accade quando si presentano valori indeterminati. La soluzione

è intuitiva: quando non si può fare a meno di conoscere il valore che si presenta come indeterminato, il risultato è indeterminato. Questo concetto deriva dalla cosiddetta logica *fuzzy*.

Figura 74.12. Tabella della verità degli operatori AND e OR quando sono coinvolti valori indefiniti.

```
? AND ? = ?      ? OR ? = ?
? AND F = F      ? OR F = ?
? AND T = ?      ? OR T = T
F AND ? = F      F OR ? = ?
F AND F = F      F OR F = F
F AND T = F      F OR T = T
T AND ? = ?      T OR ? = T
T AND F = F      T OR F = T
T AND T = T      T OR T = T
```

74.1.3.5 Relazioni derivate e viste

«

Precedentemente si è accennato al fatto che la rappresentazione finale dei dati può essere diversa da quella logica. Nel modello relazionale è possibile ottenere delle relazioni derivate da altre, attraverso una funzione determinata che stabilisce il modo con cui ottenere queste derivazioni. Si distingue fondamentalmente tra:

- relazioni derivate virtuali, o ***viste***, che non generano nuove relazioni memorizzate nella base di dati, il cui contenuto viene generato al volo al momento della necessità;
- relazioni derivate materializzate, che generano una nuova relazione nella base di dati.

Il primo dei due casi è semplice da gestire, perché i dati sono sempre allineati correttamente, ma è pesante dal punto di vista elaborativo;

il secondo ha invece i pregi e i difetti opposti. Con il termine «vista» si intende fare riferimento alle relazioni derivate virtuali.

74.2 Questioni organizzative generali dei DBMS comuni

Dopo la teoria astratta, l'organizzazione di un DBMS richiede una realizzazione pratica, che implica delle scelte. In questa sezione si descrivono alcune questioni «pratiche» che è bene conoscere inizialmente, prima di affrontare lo studio di un DBMS specifico.

74.2.1 Configurazione e basi di dati amministrative

Un DBMS, per funzionare, deve poter annotare l'esistenza e i contenuti delle proprie basi di dati, così come l'esistenza e i privilegi dei propri utenti. Per conservare queste informazioni, può gestire dei file di configurazione, oppure, più spesso, impiegare una o più basi di dati amministrative, la cui gestione avviene in modo quasi trasparente.

La presenza di queste basi di dati amministrative implica il fatto che non possano esserne create delle altre con gli stessi nomi, ma ci sono naturalmente anche altre implicazioni più importanti.

Quando si installa il software di gestione del DBMS per la prima volta, è necessario provvedere a costruire le basi di dati amministrative, oltre che, eventualmente, a sistemare altri file di configurazione. Per questo, di solito il software del DBMS include un programma che predispone tali basi di dati speciali con una configurazione iniziale predefinita.

Quando si aggiorna il software di gestione del DBMS, si ha generalmente la necessità di conservare i dati preesistenti. Questo signi-

fica preservare le basi di dati che sono state create e le informazioni sulle utenze, con i privilegi rispettivi. Il problema sta nel fatto che il software aggiornato potrebbe avere un'organizzazione differente nel modo di gestire i file che contengono le informazioni sulle basi di dati, pertanto è poi necessario provvedere a una conversione, con l'ausilio di strumenti realizzati appositamente per quel DBMS. Naturalmente, per evitare circoli viziosi, un software aggiornato dovrebbe essere in grado di accedere a basi di dati di qualche versione precedente.

Generalmente, prima di un aggiornamento del software di gestione del DBMS, si consiglia di eseguire una copia dei dati in una forma indipendente dalla versione, che può essere acquisita successivamente, dopo l'aggiornamento; tuttavia, rimane il problema delle basi di dati amministrative: se dovesse fallire la procedura di aggiornamento automatica, si renderebbe necessaria, nuovamente, la creazione delle basi di dati (vuote) e delle utenze.

74.2.2 Copie di sicurezza delle basi di dati

«

Generalmente, le copie di sicurezza del contenuto di una basi di dati, si fanno in modo di generare del codice SQL, contenente le istruzioni per la creazione delle relazioni e per l'inserimento delle tuple. Questo viene ottenuto tramite programmi o script del software del DBMS e il codice che si ottiene è specifico di quel tipo di DBMS, perciò non è universale.

Ci possono essere dei DBMS che consentono l'acquisizione di dati molto complessi in un solo attributo (dove per «attributo» si intende una cella di una riga di una tabella), ma poi, questi dati non possono essere rappresentati in modo testuale in un codice SQL. In tali casi,

l'archiviazione in forma di codice SQL si deve limitare ai dati consueti, ignorando il resto; pertanto si devono usare formati differenti per l'archiviazione completa dei dati.

74.2.3 Comunicazione con il DBMS e accesso alle basi di dati

I programmi accedono alle basi di dati attraverso un protocollo di comunicazione con il DBMS. Il protocollo in questione dipende dal DBMS, ma generalmente consente di trasportare delle istruzioni SQL.

Nei sistemi Unix, la comunicazione con il DBMS avviene tipicamente attraverso socket di dominio Unix, per le comunicazioni locali, e socket di dominio Internet per quelle remote. Pertanto, il DBMS deve disporre di un demone in attesa di dati da un file di tipo socket e in ascolto di una porta TCP o UDP (di solito si tratta del protocollo TCP).

Naturalmente, per consentire l'accesso alle basi di dati, il DBMS deve avere un modo per «riconoscere» chi vuole accedere.

Un DBMS che consente esclusivamente collegamenti di tipo locale, avrebbe la possibilità di individuare gli accessi in base al numero UID associato al processo elaborativo del programma che tenta il contatto. In questo caso particolare, il riconoscimento delle utenze può essere demandata al sistema operativo.

74.2.4 Utenze e privilegi

Le utenze di un DBMS servono a distinguere le competenze al suo interno. Generalmente si distingue la presenza di un amministratore con poteri illimitati nell'ambito della gestione complessiva del DBMS e di conseguenza di ogni base di dati. Il nome, dal punto del DBMS, di questo amministratore, non è standardizzato. A titolo di esempio, nel caso di PostgreSQL si tratta normalmente dell'utente '**`postgres`**', mentre con MySQL si usa il nome '**`root`**'.

Generalmente, il DBMS riconosce gli utenti attraverso una parola d'ordine, che deve essere fornita all'inizio di ogni collegamento; tuttavia, dovrebbe esistere anche la possibilità di definire utenze senza parola d'ordine (sulla fiducia), oppure dovrebbe essere possibile definire dei contesti per cui l'accesso non debba richiedere questa formalità.

Il problema di evitare l'obbligo di inserire la parola d'ordine si sente in particolare proprio per l'accesso in qualità di amministratore, quando si vogliono realizzare degli script per svolgere certe funzioni amministrative. Le forme di raggiro dipendono dalle caratteristiche del DBMS.

Quando il DBMS è in grado di riconoscere un accesso locale, in quanto proveniente da un utente che ha lo stesso nome usato nell'ambito del sistema operativo, potrebbe accettarlo senza richiesta di una parola d'ordine, perché in sostanza l'autenticazione è già avvenuta. Questo meccanismo viene usato in particolare con PostgreSQL, dove l'utente '**`postgres`**' viene aggiunto anche nel file '`/etc/passwd`'; in tal modo, ammesso che la configurazione di PostgreSQL lo consenta, l'utente '**`root`**' del sistema operativo

può impersonare facilmente l'utente '**postgres**', attraverso il comando '**su**', quindi può accedere localmente al DBMS venendo riconosciuto implicitamente.

Quando non c'è la possibilità di sfruttare il sistema operativo per il riconoscimento dell'utente in modo implicito, si può arrivare ad annotare la parola d'ordine (in chiaro) in un file che solo l'utente '**root**' del sistema operativo può leggere, così che uno script con i privilegi necessari possa leggere questa parola d'ordine, usarla per collegarsi al DBMS e svolgere il suo compito.

Generalmente, le utenze vengono considerate nel DBMS soltanto come nominativi puri e semplici, senza distinguerne la provenienza. Il DBMS potrebbe disporre di una configurazione ulteriore in cui si specifica il metodo di riconoscimento richiesto, in base alla provenienza degli accessi (PostgreSQL), oppure potrebbe arrivare a considerare le utenze come l'unione del nome al nodo di origine, come se si trattasse di utenti distinti. Questo secondo caso riguarda in particolare MySQL e vale la pena di considerarlo con attenzione, perché si possono creare degli equivoci; infatti, se un'utenza è abbinata all'origine '**localhost**' e tale utente accede sì dall'elaboratore locale (come indica convenzionalmente il nome '**localhost**'), ma il file '`/etc/hosts`' non abbina correttamente l'indirizzo locale a tale nome, l'accesso fallisce; inoltre, se l'utenza fosse abbinata all'origine '**127.0.0.1**' e l'utente cercasse di accedere localmente, potrebbe succedergli di non essere riconosciuto se il sistema operativo traduce poi l'indirizzo nel nome.

Per quanto riguarda la gestione delle utenze, c'è da considerare che esistono anche dei DBMS semplificati che, concedendo esclusivamente accessi locali, invece di gestire le utenze in proprio si affi-

dano alla gestione del sistema operativo. In questi casi, i permessi di accesso alle basi di dati vengono regolati tramite la gestione dei permessi corrispondenti ai file che rappresentano le basi di dati stesse.

74.2.5 ODBC

«

ODBC, ovvero *Open database connectivity* è un metodo standardizzato per l'accesso ai DBMS. In pratica, si inserisce un servizio intermedio, tra i DBMS e le applicazioni che devono accedere ai dati: le applicazioni comunicano con il servizio ODBC; il servizio ODBC comunica con i DBMS sottostanti, preoccupandosi di adattarsi alle loro particolarità. In questo modo, invece di scrivere applicazioni che comunicano solo con un certo DBMS, le applicazioni fatte per ODBC, possono utilizzare qualsiasi DBMS che il servizio ODBC è in grado di gestire.

74.3 Introduzione a SQL

«

SQL è l'acronimo di *Structured query language* e identifica un linguaggio di interrogazione (gestione) per basi di dati relazionali. Le sue origini risalgono alla fine degli anni 1970 e questo giustifica la sua sintassi prolissa e verbale tipica dei linguaggi dell'epoca, come il COBOL.

Allo stato attuale, data la sua evoluzione e standardizzazione, l'SQL rappresenta un riferimento fondamentale per la gestione di una base di dati relazionale.

A parte il significato originale dell'acronimo, SQL è un linguaggio completo per la gestione di una base di dati relazionale, includendo

le funzionalità di un DDL (*Data definition language*), di un DML (*Data manipulation language*) e di un DCL (*Data control language*).

Data l'età e la conseguente evoluzione di questo linguaggio, si sono definiti nel tempo diversi livelli di standard. I più importanti sono SQL89, SQL92 e SQL99, noti anche come SQL1, SQL2 e SQL3 rispettivamente.

L'aderenza dei vari sistemi DBMS allo standard SQL92 non è mai completa e perfetta, per questo sono stati definiti dei sottolivelli di questo standard per definire il grado di compatibilità di un DBMS. Si tratta di: *entry SQL*, *intermediate SQL* e *full SQL*. Si può intendere che il primo sia il livello di compatibilità minima e l'ultimo rappresenti la compatibilità totale. Lo standard di fatto è rappresentato prevalentemente dal primo livello, che coincide fondamentalmente con lo standard precedente, SQL89. Da questo si comprende che lo stato di assimilazione di SQL99 è ancora più arretrato.

74.3.1 Concetti fondamentali

Convenzionalmente, le istruzioni di questo linguaggio sono scritte con tutte le lettere maiuscole. Si tratta solo di una tradizione di quell'epoca. SQL non distingue tra lettere minuscole e maiuscole nelle parole chiave delle istruzioni e nemmeno nei nomi di relazioni, attributi e altri oggetti. Solo quando si tratta di definire il contenuto di una variabile, allora le differenze contano.

In questo capitolo e nel resto del documento, quando si fa riferimento a istruzioni SQL, queste vengono indicate utilizzando solo lettere maiuscole, come richiede la tradizione.

I nomi degli oggetti (relazioni e altro) possono essere composti utilizzando lettere, numeri e il trattino basso; il primo carattere deve

essere una lettera oppure il trattino basso.

Le istruzioni SQL possono essere distribuite su più righe, senza una regola precisa. Si distingue la fine di un'istruzione dall'inizio di un'altra attraverso la presenza di almeno una riga vuota. Generalmente, i sistemi SQL richiedono l'uso di un simbolo di terminazione delle righe, che di norma è costituito dal punto e virgola.

L'SQL standard prevede la possibilità di inserire commenti; per questo si può usare un trattino doppio ('--') seguito dal commento desiderato, fino alla fine della riga. Tuttavia, si osservi che per ottenere la massima compatibilità con i DBMS esistenti, conviene lasciare almeno uno spazio dopo il trattino doppio, prima di inserire il commento vero e proprio.

74.3.2 Terminologia

«

Il linguaggio SQL utilizza una propria terminologia per distinguere gli «oggetti» che manipola. Per la precisione, si utilizzano normalmente i termini «tabella», «riga» e «colonna», al posto di «relazione», «tupla» e «attributo».

Esistono delle buone ragioni per utilizzare una terminologia differente nel linguaggio SQL, soprattutto in considerazione del fatto che in questo caso sono ammissibili situazioni che nella teoria generale delle basi di dati relazionali non lo sarebbero (per esempio la possibilità di avere tuple doppie). Tuttavia, si osservi che in questo documento si cerca di mantenere una certa uniformità nei termini, seguendo la tradizione della teoria delle basi di dati, anche a costo di rischiare una contraddizione con questa.

Specchietto 74.13. Associazione tra i termini relativi alla gestione delle basi di dati. Ogni riga dello specchietto, rappresenta un contesto differente, mentre le colonne individuano la traduzione dei termini in base al contesto.

classi	istanze	attributi	tipi di dati contenibili negli attributi
relazioni	tuple	attributi	domini
tabelle	righe	colonne	tipi di dati contenibili nelle colonne

74.4 DDL

DDL, ovvero *Data definition language*, è il linguaggio usato per definire la struttura dei dati (in una base di dati). In questa sezione viene trattato il linguaggio SQL per ciò che riguarda specificatamente i dati, la loro creazione e la loro distruzione.

74.4.1 Tipi di dati

I tipi di dati gestibili con il linguaggio SQL sono molti. Fondamentalmente si possono distinguere tipi contenenti: valori numerici, stringhe e informazioni data-orario. Nelle sezioni seguenti vengono descritti solo alcuni dei tipi definiti dallo standard.

74.4.1.1 Stringhe di caratteri

Si distinguono due tipi di stringhe di caratteri in SQL: quelle a dimensione fissa, completate a destra dal carattere spazio, e quelle a dimensione variabile.

Sintassi	Descrizione

`CHARACTER`
 	
`CHARACTER(`***dimensione***`)`
 	
`CHAR`
 	
`CHAR(`***dimensione***`)`
	Queste sono le varie sintassi alternative che possono essere utilizzate per definire una stringa di dimensione fissa. Se non viene indicata la dimensione tra parentesi, si intende una stringa di un solo carattere.

`CHARACTER VARYING(`***dimensione***`)`
 	
`CHAR VARYING(`***dimensione***`)`
 	
`VARCHAR(`***dimensione***`)`
	Una stringa di dimensione variabile può essere definita attraverso uno dei tre modi appena elencati. È necessario specificare la dimensione massima che questa stringa può avere. Il minimo è rappresentato dalla stringa nulla.

Le costanti stringa si esprimono delimitandole attraverso apici singoli, oppure apici doppi, come nell'esempio seguente:

```
'Questa è una stringa letterale per SQL'
"Anche questa è una stringa letterale per SQL"
```

Non tutti i sistemi SQL accettano entrambi i tipi di delimitatori di stringa. In caso di dubbio è bene limitarsi all'uso degli apici singoli; eventualmente, per inserire un apice singolo in una stringa delimitata con apici singoli, dovrebbe essere sufficiente il suo raddoppio. In pratica, per scrivere una stringa del tipo «l'albero», dovrebbe essere possibile scrivere:

```
'l''albero'
```

74.4.1.2 Valori numerici

I tipi numerici si distinguono in ***esatti*** e ***approssimati***, intendendo con la prima definizione quelli di cui si conosce il numero massimo di cifre numeriche intere e decimali, mentre con la seconda si fa riferimento ai tipi a virgola mobile. In ogni caso, le dimensioni massime o la precisione massima che possono avere tali valori dipende dal sistema in cui vengono utilizzati.

Sintassi	Descrizione

`NUMERIC`
 	
`NUMERIC(`***precisione*** [`,` ***scala***] `)`
	Il tipo '**`NUMERIC`**' permette di definire un valore numerico composto da un massimo di tante cifre numeriche quante indicate dalla precisione, cioè il primo argomento tra parentesi. Se viene specificata anche la scala, si intende riservare quella parte di cifre per quanto appare dopo la virgola. Per esempio, con '**`NUMERIC(5,2)`**' si possono rappresentare valori da +999,99 a −999,99.
	Se non viene specificata la scala, si intende che si tratti solo di valori interi; se non viene specificata nemmeno la precisione, viene usata la definizione predefinita per questo tipo di dati, che dipende dalle caratteristiche del DBMS.

Sintassi	Descrizione

`DECIMAL`
`DECIMAL(`***precisione*** [`,` ***scala***] `)`
`DEC`
`DEC(`***precisione*** [`,` ***scala***] `)`
	Il tipo '**`DECIMAL`**' è simile al tipo '**`NUMERIC`**', con la differenza che le caratteristiche della precisione e della scala rappresentano le esigenze minime, mentre il sistema può fornire una rappresentazione con precisione o scala maggiore.

`INTEGER`
`INT`
`SMALLINT`
	I tipi '**`INTEGER`**' e '**`SMALLINT`**' rappresentano tipi interi la cui dimensione dipende generalmente dalle caratteristiche del sistema operativo e dall'hardware utilizzato. L'unico riferimento sicuro è che il tipo '**`SMALLINT`**' permette di rappresentare interi con una dimensione inferiore o uguale al tipo '**`INTEGER`**'.

`FLOAT` `FLOAT(`***precisione***`)`	Il tipo '**`FLOAT`**' definisce un tipo numerico approssimato (a virgola mobile) con una precisione binaria pari o superiore di quella indicata tra parentesi (se non viene indicata, dipende dal sistema).
`REAL` `DOUBLE PRECISION`	Il tipo '**`REAL`**' e il tipo '**`DOUBLE PRECISION`**' sono due tipi a virgola mobile con una precisione prestabilita. Questa precisione dipende dal sistema, ma in generale, il secondo dei due tipi deve essere più preciso dell'altro.

I valori numerici costanti vengono espressi attraverso la semplice indicazione del numero senza delimitatori. La virgola di separazione della parte intera da quella decimale si esprime normalmente attraverso il punto (‘.’), a meno che sia prevista una forma di adattamento alle caratteristiche di configurazione locale.

74.4.1.3 Valori Data-orario

I valori data-orario sono di tre tipi e servono rispettivamente a memorizzare un giorno particolare, un orario normale e un’informazione data-ora completa.

Sintassi	Descrizione
`DATE`	Il tipo ‘**`DATE`**’ permette di rappresentare delle date composte dall’informazione anno-mese-giorno.

`TIME`
`TIME(`***precisione***`)`
`TIME WITH TIME ZONE`
`TIME(`***precisione***`) WITH TIME ZONE`
	Il tipo ‘**`TIME`**’ permette di rappresentare un orario particolare, composto da ore-minuti-secondi ed eventualmente frazioni di secondo. Se viene specificata la precisione, si intende definire un numero di cifre per la parte frazionaria dei secondi, altrimenti si intende che non debbano essere memorizzate le frazioni di secondo.

Sintassi	Descrizione

`TIMESTAMP`
`TIMESTAMP(`***precisione***`)`
`TIMESTAMP WITH TIME ZONE`
`TIMESTAMP(`***precisione***`) WITH TIME ZONE`
	Il tipo '**`TIMESTAMP`**' è un'informazione oraria più completa del tipo '**`TIME`**' in quanto prevede tutte le informazioni, dall'anno ai secondi, oltre alle eventuali frazioni di secondo.

Se viene specificata la precisione, si intende definire un numero di cifre per la parte frazionaria dei secondi, altrimenti si intende che non debbano essere memorizzate le frazioni di secondo.

L'aggiunta dell'opzione '**`WITH TIME ZONE`**' serve a specificare un tipo orario differente, che assieme all'informazione oraria aggiunge lo scostamento, espresso in ore e minuti, dell'ora locale dal tempo universale (UTC). Per esempio, 22:05:10+1:00 rappresenta le 22.05 e 10 secondi dell'ora locale italiana (durante l'inverno), mentre il tempo universale corrispondente sarebbe invece 21:05:10+0:00.

Le costanti che rappresentano informazioni data-orario sono espresse come le stringhe, delimitate tra apici. Il sistema DBMS potrebbe ammettere più forme differenti per l'inserimento di queste, ma i modi più comuni dovrebbero essere quelli espressi dagli esempi seguenti.

```
'2012-12-31'
'12/31/2012'
'31.12.2012'
```

Questi tre esempi rappresentano la stessa data: il 31 dicembre 1999. Per una questione di uniformità, dovrebbe essere preferibile il primo

di questi formati, corrispondente allo stile ISO 8601. Anche gli orari che si vedono sotto, sono aderenti allo stile ISO 8601; in particolare per il fatto che il fuso orario viene indicato attraverso lo scostamento dal tempo universale, invece che attraverso una parola chiave che definisca il fuso dell'ora locale.

```
'12:30:50+1.00'
'12:30:50.10'
'12:30:50'
'12:30'
```

Il primo di questa serie di esempi rappresenta un orario composto da ore, minuti e secondi, oltre all'indicazione dello scostamento dal tempo universale (per ottenere il tempo universale deve essere sottratta un'ora). Il secondo esempio mostra un orario composto da ore, minuti, secondi e centesimi di secondo. Il terzo e il quarto sono rappresentazioni normali, in particolare nell'ultimo è stata omessa l'indicazione dei secondi.

```
'2012-12-31 12:30:50+1.00'
'2012-12-31 12:30:50.10'
'2012-12-31 12:30:50'
'2012-12-31 12:30'
```

Gli esempi mostrano la rappresentazione di informazioni data-orario complete per il tipo '**`TIMESTAMP`**'. La data è separata dall'ora da uno spazio.

74.4.1.4 Intervalli di tempo

Quanto mostrato nella sezione precedente rappresenta un valore che indica un momento preciso nel tempo: una data o un orario, o entrambe le cose. Per rappresentare una durata, si parla di intervalli. Per l'SQL si possono gestire gli intervalli a due livelli di precisione:

anni e mesi; oppure giorni, ore, minuti, secondi ed eventualmente anche le frazioni di secondo. L'intervallo si indica con la parola chiave '**INTERVAL**', seguita eventualmente dalla precisione con cui questo deve essere rappresentato:

```
INTERVAL [unità_di_misura_data_orario [TO unità_di_misura_data_orario]]
```

In pratica, si può indicare che si tratta di un intervallo, senza specificare altro, oppure si possono definire una o due unità di misura che limitano la precisione di questo (pur restando nei limiti a cui si è già accennato). Tanto per fare un esempio concreto, volendo definire un intervallo che possa esprimere solo ore e minuti, si potrebbe dichiarare con: '**INTERVAL HOUR TO MINUTE**'. La tabella 74.22 elenca le parole chiave che rappresentano queste unità di misura.

Tabella 74.22. Elenco delle parole chiave che esprimono unità di misura data-orario.

Parola chiave	Significato
YEAR	Anni
MONTH	Mesi
DAY	Giorni
HOUR	Ore
MINUTE	Minuti
SECOND	Secondi

Si osservino i due esempi seguenti:

```
INTERVAL '12 HOUR 30 MINUTE 50 SECOND'
INTERVAL '12:30:50'
```

Queste due forme rappresentano entrambe la stessa cosa: una durata di 12 ore, 30 minuti e 50 secondi. In generale, dovrebbe essere preferibile la seconda delle due forme di rappresentazione.

```
INTERVAL '10 DAY 12 HOUR 30 MINUTE 50 SECOND'
INTERVAL '10 DAY 12:30:50'
```

Come prima, i due esempi che si vedono sopra sono equivalenti. Intuitivamente, si può osservare che non ci può essere un altro modo di esprimere una durata in giorni, senza specificarlo attraverso la parola chiave '**DAY**'.

Per completare la serie di esempi, si aggiungono anche i casi in cui si rappresentano esplicitamente quantità molto grandi, che di conseguenza sono approssimate al mese (come richiede lo standard SQL92):

```
INTERVAL '10 YEAR 11 MONTH'
INTERVAL '10 YEAR'
```

Gli intervalli di tempo possono servire per indicare un tempo trascorso rispetto al momento attuale. Per specificare espressamente questo fatto, si indica l'intervallo come un valore negativo, aggiungendo all'inizio un trattino (il segno meno).

```
INTERVAL '- 10 YEAR 11 MONTH'
```

L'esempio che si vede sopra, esprime precisamente 10 anni e 11 mesi fa.

74.4.2 Operatori, funzioni ed espressioni

«

SQL, pur non essendo un linguaggio di programmazione completo, mette a disposizione una serie di operatori e di funzioni utili per la realizzazione di espressioni di vario tipo.

74.4.2.1 Operatori aritmetici

«

Gli operatori che intervengono su valori numerici sono elencati nella tabella 74.27.

Tabella 74.27. Elenco degli operatori aritmetici.

Operatore e operandi	Descrizione
−op	Inverte il segno dell'operando.
op1 + *op2*	Somma i due operandi.
op1 − *op2*	Sottrae dal primo il secondo operando.
op1 * *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo.
op1 % *op2*	Modulo: il resto della divisione tra il primo e il secondo operando.

Nelle espressioni, tutti i tipi numerici esatti e approssimati possono essere usati senza limitazioni. Dove necessario, il sistema provvede a eseguire le conversioni di tipo.

74.4.2.2 Operazioni con i valori data-orario e con intervalli di tempo

Le operazioni che si possono compiere utilizzando valori data-orario e valori che esprimono intervalli di tempo, hanno significato solo in alcune circostanze. La tabella 74.28 elenca le operazioni possibili e il tipo di risultato che si ottiene in base al tipo di operatori utilizzato.

Tabella 74.28. Operatori e operandi validi quando si utilizzano valori data-orario e valori che esprimono intervalli di tempo.

Operatore e operandi	Risultato
data_orario − ***data_orario***	Intervallo
data_orario + ***intervallo***	Data-orario
data_orario − ***intervallo***	Data-orario
intervallo + ***data_orario***	Data-orario
intervallo + ***intervallo***	Intervallo
intervallo − ***intervallo***	Intervallo
intervallo * ***numerico***	Intervallo
intervallo / ***numerico***	Intervallo
numerico * ***intervallo***	Intervallo

74.4.2.3 Operatori di confronto e operatori logici

«

Gli operatori di confronto determinano la relazione tra due operandi. Il risultato dell'espressione composta da due operandi posti a confronto è di tipo booleano: *Vero* o *Falso*. Gli operatori di confronto sono elencati nella tabella 74.29.

Tabella 74.29. Elenco degli operatori di confronto.

Operatore e operandi	Descrizione
op1 = ***op2***	*Vero* se gli operandi si equivalgono.
op1 <> ***op2***	*Vero* se gli operandi sono differenti.
op1 < ***op2***	*Vero* se il primo operando è minore del secondo.
op1 > ***op2***	*Vero* se il primo operando è maggiore del secondo.
op1 <= ***op2***	*Vero* se il primo operando è minore o uguale al secondo.
op1 >= ***op2***	*Vero* se il primo operando è maggiore o uguale al secondo.

Quando si vogliono combinare assieme diverse espressioni logiche si utilizzano gli operatori logici. Come in tutti i linguaggi di programmazione, si possono usare le parentesi tonde per raggruppare le espressioni logiche in modo da chiarire l'ordine di risoluzione. Gli operatori logici sono elencati nella tabella 74.30.

Tabella 74.30. Elenco degli operatori logici.

Operatore e operandi	Descrizione
`NOT` ***op***	Inverte il risultato logico dell'operando.
op1 `AND` ***op2***	*Vero* se entrambi gli operandi restituiscono il valore *Vero*.
op1 `OR` ***op2***	*Vero* se almeno uno degli operandi restituisce il valore *Vero*.

Il meccanismo di confronto tra due operandi numerici è evidente, mentre può essere meno evidente con le stringhe di caratteri. Per la precisione, il confronto tra due stringhe avviene senza tenere conto degli spazi finali, per cui, le stringhe '**`'ciao'`**' e '**`'ciao '`**' dovrebbero risultare uguali attraverso il confronto di uguaglianza con l'operatore '**`=`**'.

Con le stringhe, tuttavia, si possono eseguire dei confronti basati su modelli, attraverso gli operatori '**`IS LIKE`**' e '**`IS NOT LIKE`**'. Il modello può contenere dei metacaratteri rappresentati dal trattino basso ('_'), che rappresenta un carattere qualsiasi, e dal simbolo di percentuale ('`%`'), che rappresenta una sequenza qualsiasi di caratteri. La tabella 74.31 riassume quanto affermato.

Tabella 74.31. Espressioni sulle stringhe di caratteri.

Espressioni e modelli	Descrizione
stringa `IS LIKE` ***modello***	Restituisce *Vero* se il modello corrisponde alla stringa.
stringa `IS NOT LIKE` ***modello***	Restituisce *Vero* se il modello non corrisponde alla stringa.
_	Rappresenta un carattere qualsiasi.

Espressioni e modelli	Descrizione
`%`	Rappresenta una sequenza indeterminata di caratteri.

La presenza di valori indeterminati richiede la disponibilità di operatori di confronto in grado di determinarne l'esistenza. La tabella 74.32 riassume gli operatori ammissibili in questi casi.

Tabella 74.32. Espressioni di verifica dei valori indeterminati.

Operatori	Descrizione
espressione `IS NULL`	Restituisce *Vero* se l'espressione genera un risultato indeterminato.
espressione `IS NOT NULL`	Restituisce *Vero* se l'espressione non genera un risultato indeterminato.

Infine, occorre considerare una categoria particolare di espressioni che permettono di verificare l'appartenenza di un valore a un intervallo o a un elenco di valori. La tabella 74.33 riassume gli operatori utilizzabili.

Tabella 74.33. Espressioni per la verifica dell'appartenenza di un valore a un intervallo o a un elenco.

Operatori e operandi	Descrizione
op1 `IN (`***elenco***`)`	*Vero* se il primo operando è contenuto nell'elenco.
op1 `NOT IN (`***elenco***`)`	*Vero* se il primo operando non è contenuto nell'elenco.
op1 `BETWEEN` ***op2*** `AND` ***op3***	*Vero* se il primo operando è compreso tra il secondo e il terzo.
op1 `NOT BETWEEN` ***op2*** `AND` ***op3***	*Vero* se il primo operando non è compreso nell'intervallo.

74.4.3 Le relazioni dal punto di vista di SQL

SQL tratta le relazioni attraverso il modello tabellare; di conseguenza si adegua tutta la sua filosofia e il modo di esprimere i concetti nella sua documentazione. Pertanto, le relazioni per SQL sono delle «tabelle», che vengono definite nel modo seguente dalla documentazione standard.

- La tabella è un insieme di più righe. Una riga è una sequenza non vuota di valori. Ogni riga della stessa tabella ha la stessa cardinalità e contiene un valore per ogni colonna di quella tabella. L'i-esimo valore di ogni riga di una tabella è un valore dell'i-esima colonna di quella tabella. La riga è l'elemento che costituisce la più piccola unità di dati che può essere inserita in una tabella e cancellata da una tabella.
- Il grado di una tabella è il numero di colonne della stessa. In ogni momento, il grado della tabella è lo stesso della cardinalità di ognuna delle sue righe; la cardinalità della tabella (cioè il numero delle righe contenute) è la stessa della cardinalità di ognuna delle sue colonne. Una tabella la cui cardinalità sia zero viene definita come vuota.

In pratica, la tabella è un contenitore di informazioni organizzato in righe e colonne. La tabella viene identificata per nome, così anche le colonne, mentre le righe vengono identificate attraverso il loro contenuto.

Nel modello di SQL, le colonne sono ordinate, anche se ciò non è sempre un elemento indispensabile, dal momento che si possono identificare per nome. Inoltre sono ammissibili tabelle contenenti righe duplicate.

Si osservi, comunque, che nel resto di questo documento si preferisce la terminologia generale delle basi di dati, dove, al posto di tabelle, righe e colonne, si parla piuttosto di relazioni, tuple e attributi.

74.4.3.1 Creazione di una relazione

La creazione di una relazione avviene attraverso un'istruzione che può assumere un'articolazione molto complessa, a seconda delle caratteristiche particolari che da questa relazione si vogliono ottenere. La sintassi più semplice è quella seguente:

```
CREATE TABLE nome_relazione ( specifiche )
```

Tuttavia, sono proprio le specifiche indicate tra le parentesi tonde che possono tradursi in un sistema molto confuso. La creazione di una relazione elementare può essere espressa con la sintassi seguente:

```
CREATE TABLE nome_relazione (nome_attributo tipo [, ...])
```

In questo modo, all'interno delle parentesi vengono semplicemente elencati i nomi degli attributi seguiti dal tipo di dati che in essi possono essere contenuti. L'esempio seguente rappresenta l'istruzione necessaria a creare una relazione composta da cinque attributi, contenenti rispettivamente informazioni su: codice, cognome, nome, indirizzo e numero di telefono.

```
CREATE TABLE Indirizzi (
        Codice          integer,
        Cognome         char(40),
        Nome            char(40),
        Indirizzo       varchar(60),
        Telefono        varchar(40)
    )
```

74.4.3.2 Valori predefiniti

Quando si inseriscono delle tuple all'interno della relazione, in linea di principio è possibile che i valori corrispondenti ad attributi particolari non siano inseriti esplicitamente. Se si verifica questa situazione (purché ciò sia consentito dai vincoli), viene assegnato a questi elementi mancanti un valore predefinito. Questo può essere stabilito all'interno delle specifiche di creazione della relazione; in mancanza di tale definizione, viene assegnato '**NULL**', corrispondente al valore indefinito.

La sintassi necessaria a creare una relazione contenente le indicazioni sui valori predefiniti da utilizzare è la seguente:

```
CREATE TABLE nome_relazione (
                    nome_attributo tipo
                        [DEFAULT espressione]
                    [,…]
        )
```

L'esempio seguente crea la stessa relazione già vista nell'esempio precedente, specificando come valore predefinito per l'indirizzo, la stringa di caratteri: «sconosciuto».

```
CREATE TABLE Indirizzi (
        Codice          integer,
        Cognome         char(40),
        Nome            char(40),
        Indirizzo       varchar(60)   DEFAULT 'sconosciuto',
        Telefono        varchar(40)
    )
```

74.4.3.3 Vincoli interni alla relazione

«

Può darsi che in certe situazioni, determinati valori all'interno di una tupla non siano ammissibili, a seconda del contesto a cui si riferisce la relazione. I vincoli interni alla relazione sono quelli che possono essere risolti senza conoscere informazioni esterne alla relazione stessa.

Il vincolo più semplice da esprimere è quello di non ammissibilità dei valori indefiniti. La sintassi seguente ne mostra il modo.

```
CREATE TABLE nome_relazione (
                    nome_attributo  tipo
                        [NOT NULL]
                    [,…]
        )
```

L'esempio seguente crea la stessa relazione già vista negli esempi precedenti, specificando che il codice, il cognome, il nome e il telefono non possono essere indeterminati.

```
CREATE TABLE Indirizzi (
        Codice          integer         NOT NULL,
        Cognome         char(40)        NOT NULL,
        Nome            char(40)        NOT NULL,
        Indirizzo       varchar(60)     DEFAULT 'sconosciuto',
        Telefono        varchar(40)     NOT NULL
    )
```

Un altro vincolo importante è quello che permette di definire che un gruppo di attributi deve rappresentare dati unici in ogni tupla, cioè che non siano ammissibili tuple che per quel gruppo di attributi abbiano dati uguali. Segue lo schema sintattico relativo:

```
CREATE TABLE nome_relazione (
                nome_attributo tipo
                [,…],
                UNIQUE ( nome_attributo [,…] )
                [,…]
        )
```

L'indicazione dell'unicità può riguardare più gruppi di attributi in modo indipendente. Per ottenere questo si possono indicare più opzioni '**UNIQUE**'.

È il caso di osservare che il vincolo '**UNIQUE**' non è sufficiente per impedire che i dati possano essere indeterminati. Infatti, il valore indeterminato, '**NULL**', è diverso da ogni altro '**NULL**'.

L'esempio seguente crea la stessa relazione già vista negli esempi precedenti, specificando che i dati dell'attributo del codice devono

essere unici per ogni tupla.

```
CREATE TABLE Indirizzi (
        Codice          integer         NOT NULL,
        Cognome         char(40)        NOT NULL,
        Nome            char(40)        NOT NULL,
        Indirizzo       varchar(60)     DEFAULT 'sconosciuto',
        Telefono        varchar(40)     NOT NULL,
        UNIQUE (Codice)
    )
```

Quando un attributo, o un gruppo di attributi, costituisce un riferimento importante per identificare le varie tuple che compongono la relazione, si può utilizzare il vincolo '**`PRIMARY KEY`**', che può essere utilizzato una sola volta. Questo vincolo stabilisce anche che i dati contenuti, oltre a non poter essere doppi, non possono essere indefiniti.

```
CREATE TABLE nome_relazione (
                    nome_attributo tipo
                    [,...],
                    PRIMARY KEY ( nome_attributo [,...] )
        )
```

L'esempio seguente crea la stessa relazione già vista negli esempi precedenti specificando che l'attributo del codice deve essere considerato come chiave primaria.

```
CREATE TABLE Indirizzi (
        Codice          integer,
        Cognome         char(40)        NOT NULL,
        Nome            char(40)        NOT NULL,
        Indirizzo       varchar(60)     DEFAULT 'sconosciuto',
        Telefono        varchar(40)     NOT NULL,
        PRIMARY KEY (Codice)
    )
```

74.4.3.4 Vincoli esterni alla relazione

I vincoli esterni alla relazione riguardano principalmente la connessione con altre relazioni e la necessità che i riferimenti a queste siano validi. La definizione formale di questa connessione è molto complessa e qui non viene descritta. Si tratta, in ogni caso, dell'opzione '**`FOREIGN KEY`**' seguita da '**`REFERENCES`**'.

Vale la pena però di considerare i meccanismi che sono coinvolti. Infatti, nel momento in cui si inserisce un valore, il sistema può impedire l'operazione perché non valida in base all'assenza di quel valore in un'altra relazione esterna specificata. Il problema nasce però nel momento in cui nella relazione esterna viene eliminata o modificata una tupla che è oggetto di un riferimento da parte della prima. Si pongono le alternative seguenti.

Vincolo	Descrizione
`CASCADE`	Se nella relazione esterna il dato a cui si fa riferimento è stato cambiato, viene cambiato anche il riferimento nella relazione di partenza; se nella relazione esterna la tupla corrispondente viene rimossa, viene rimossa anche la tupla della relazione di partenza.
`SET NULL`	Se viene a mancare l'oggetto a cui si fa riferimento, viene modificato il dato attribuendo il valore indefinito.

Vincolo	Descrizione
`SET DEFAULT`	Se viene a mancare l'oggetto a cui si fa riferimento, viene modificato il dato attribuendo il valore predefinito.
`NO ACTION`	Se viene a mancare l'oggetto a cui si fa riferimento, non viene modificato il dato contenuto nella relazione di partenza.

Le azioni da compiere si possono distinguere in base all'evento che ha causato la rottura del riferimento: cancellazione della tupla della relazione esterna o modifica del suo contenuto.

74.4.3.5 Modifica della struttura della relazione

La modifica della struttura di una relazione riguarda principalmente la sua organizzazione in attributi. Le cose più semplici che si possono desiderare di fare sono l'aggiunta di nuovi attributi e l'eliminazione di attributi già esistenti. Vedendo il problema in questa ottica, la sintassi si riduce ai due casi seguenti:

```
ALTER TABLE nome_relazione ADD [COLUMN]
            nome_attributo tipo [altre_caratteristiche]
```

```
ALTER TABLE nome_relazione DROP [COLUMN] nome_attributo
```

Nel primo caso si aggiunge un attributo, del quale si deve specificare il nome, il tipo ed eventualmente i vincoli; nel secondo si tratta solo di indicare l'attributo da eliminare. A livello di singolo attributo può essere eliminato o assegnato un valore predefinito.

```
ALTER TABLE nome_relazione ALTER [COLUMN]
            nome_attributo DROP DEFAULT
```

```
ALTER TABLE nome_relazione ALTER [COLUMN]
            nome_attributo SET DEFAULT valore_predefinito
```

74.4.3.6 Eliminazione di una relazione

L'eliminazione di una relazione, con tutto il suo contenuto, è un'operazione semplice che dovrebbe essere autorizzata solo all'utente che l'ha creata.

```
DROP TABLE nome_relazione
```

74.5 DML

DML, ovvero *Data manipulation language*, è il linguaggio usato per inserire, modificare e accedere ai dati. In questa sezione viene trattato il linguaggio SQL per ciò che riguarda specificatamente l'inserimento, la lettura e la modifica del contenuto delle relazioni.

74.5.1 Inserimento, eliminazione e modifica dei dati

L'inserimento, l'eliminazione e la modifica dei dati di una relazione è un'operazione che interviene sempre a livello delle tuple. Infatti, come già definito, la tupla è l'elemento che costituisce l'unità di dati più piccola che può essere inserita o cancellata da una relazione.

74.5.1.1 Inserimento di tuple

«

L'inserimento di una nuova tupla all'interno di una relazione viene eseguito attraverso l'istruzione '**INSERT**'. Dal momento che nel modello di SQL gli attributi sono ordinati, è sufficiente indicare ordinatamente l'elenco dei valori della tupla da inserire, come mostra la sintassi seguente:

```
INSERT INTO nome_relazione VALUES (espressione_1 [,...espressione_n])
```

Per esempio, l'inserimento di una tupla nella relazione '**Indirizzi**' già mostrata in precedenza, potrebbe avvenire nel modo seguente:

```
INSERT INTO Indirizzi
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    )
```

Se i valori inseriti sono meno del numero degli attributi della relazione, i valori mancanti, in coda, ottengono quanto stabilito come valore predefinito, o '**NULL**' in sua mancanza (sempre che ciò sia concesso dai vincoli della relazione).

L'inserimento dei dati può avvenire in modo più chiaro e sicuro elencando prima i nomi degli attributi, in modo da evitare di dipendere dalla sequenza degli attributi memorizzata nella relazione. La sintassi seguente mostra il modo di ottenere questo.

```
INSERT INTO nome_relazione (attributo_1 [,...attributo_n]) ]
    VALUES (espressione_1 [,...espressione_n] )
```

L'esempio già visto potrebbe essere tradotto nel modo seguente, più prolisso, ma anche più chiaro:

```
INSERT INTO Indirizzi (
        Codice,
        Cognome,
        Nome,
        Indirizzo,
        Telefono
    )
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    )
```

Questo modo esplicito di fare riferimento agli attributi garantisce anche che eventuali modifiche di lieve entità nella struttura della relazione non debbano necessariamente riflettersi nei programmi. L'esempio seguente mostra l'inserimento di alcuni degli attributi della tupla, lasciando che gli altri ottengano l'assegnamento di un valore predefinito.

```
INSERT INTO Indirizzi (
        Codice,
        Cognome,
        Nome,
        Telefono
```

```
    )
    VALUES (
        01,
        'Pinco',
        'Pallino',
        '0222,222222'
    )
```

74.5.1.2 Aggiornamento delle tuple

La modifica delle tuple può avvenire attraverso una scansione della relazione, dalla prima all'ultima tupla, eventualmente controllando la modifica in base all'avverarsi di determinate condizioni. La sintassi per ottenere questo risultato, leggermente semplificata, è la seguente:

```
UPDATE relazione
    SET attributo_1=espressione_1 [ ,...attributo_n=espressione_n ]
    [WHERE condizione ]
```

L'istruzione '**UPDATE**' esegue tutte le sostituzioni indicate dalle coppie ***attributo=espressione***, per tutte le tuple in cui la condizione posta dopo la parola chiave '**WHERE**' si avvera. Se tale condizione manca, l'effetto delle modifiche si riflette su tutte le tuple della relazione.

L'esempio seguente aggiunge un attributo alla relazione degli indirizzi, per contenere il nome del comune di residenza; successivamente viene inserito il nome del comune «Sferopoli» in base al prefisso telefonico.

```
ALTER TABLE Indirizzi ADD COLUMN Comune char(30)
```

```
UPDATE Indirizzi
    SET Comune='Sferopoli'
    WHERE Telefono >= '022' AND Telefono < '023'
```

Eventualmente, al posto dell'espressione si può indicare la parola chiave '**DEFAULT**' che fa in modo di assegnare il valore predefinito per quel attributo.

74.5.1.3 Eliminazione di tuple

La cancellazione di tuple da una relazione è un'operazione molto semplice. Richiede solo l'indicazione del nome della relazione e la condizione in base alla quale le tuple devono essere cancellate.

```
DELETE FROM relazione [WHERE condizione]
```

Se la condizione non viene indicata, si cancellano tutte le tuple!

74.5.2 Interrogazioni di relazioni

L'interrogazione di una relazione è l'operazione con cui si ottengono i dati contenuti al suo interno, in base a dei criteri di filtro determinati. L'interrogazione consente anche di combinare assieme dati provenienti da relazioni differenti, in base a dei «collegamenti» che possono intercorrere tra queste.

74.5.2.1 Interrogazioni elementari

La forma più semplice di esprimere la sintassi necessaria a interrogare **una** sola relazione è quella espressa dallo schema seguente:

```
SELECT espress_col_1 [ , ...espress_col_n ]
    FROM relazione
    [WHERE condizione ]
```

In questo modo è possibile definire gli attributi che si intendono utilizzare per il risultato, mentre le tuple si specificano, eventualmente, con la condizione posta dopo la parola chiave '**WHERE**'. L'esempio seguente mostra la proiezione degli attributi del cognome e nome della relazione di indirizzi già vista negli esempi delle altre sezioni, senza porre limiti alle tuple.

```
SELECT Cognome, Nome FROM Indirizzi
```

Quando si vuole ottenere una selezione composta dagli stessi attributi della relazione originale, nel suo stesso ordine, si può utilizzare un carattere jolly particolare, l'asterisco ('*'). Questo rappresenta l'elenco di tutti gli attributi della relazione indicata.

```
SELECT * FROM Indirizzi
```

È bene osservare che gli attributi si esprimono attraverso un'espressione, questo significa che gli attributi a cui si fa riferimento sono quelle del risultato finale, cioè della relazione che viene restituita come selezione o proiezione della relazione originale. L'esempio seguente emette un solo attributo contenente un ipotetico prezzo scontato del 10 %, in pratica viene moltiplicato il valore di un attributo contenente il prezzo per 0,90, in modo da ottenerne il 90 % (100 % meno lo sconto).

```
SELECT Prezzo * 0.90 FROM Listino
```

In questo senso si può comprendere l'utilità di assegnare esplicita-

mente un nome agli attributi del risultato finale, come indicato dalla sintassi seguente:

```
SELECT espress_col_1 AS nome_col_1 [,...espress_col_n AS nome_col_n]
    FROM relazione
    [WHERE condizione]
```

In questo modo, l'esempio precedente può essere trasformato come segue, dando un nome all'attributo generato e chiarendone così il contenuto.

```
SELECT Prezzo * 0.90 AS Prezzo_Scontato FROM Listino
```

Finora è stata volutamente ignorata la condizione che controlla le tuple da selezionare. Anche se potrebbe essere evidente, è bene chiarire che la condizione posta dopo la parola chiave '**WHERE**' può fare riferimento solo ai dati originali della relazione da cui si attingono. Quindi, non è valida una condizione che utilizza un riferimento a un nome che appare dopo la parola chiave '**AS**' abbinata alle espressioni degli attributi.

Per qualche motivo che viene chiarito in seguito, può essere conveniente associare un alias alla relazione da cui estrarre i dati. Anche in questo caso si utilizza la parola chiave '**AS**', come indicato dalla sintassi seguente:

```
SELECT specificazione_dell'attributo_1 [,...specificazione_dell'attributo_n]
    FROM relazione AS alias
    [WHERE condizione]
```

Quando si vuole fare riferimento al nome di un attributo, se per qual-

che motivo questo nome dovesse risultare ambiguo, si può aggiungere anteriormente il nome della relazione a cui appartiene, separandolo attraverso l'operatore punto ('.'). L'esempio seguente è la proiezione dei cognomi e dei nomi della solita relazione degli indirizzi. In questo caso, le espressioni degli attributi rappresentano solo gli attributi corrispondenti della relazione originaria, con l'aggiunta dell'indicazione esplicita del nome della relazione stessa.

```
SELECT Indirizzi.Cognome, Indirizzi.Nome FROM Indirizzi
```

A questo punto, se al nome della relazione viene abbinato un alias, si può esprimere la stessa cosa indicando il nome dell'alias al posto di quello della relazione, come nell'esempio seguente:

```
SELECT Ind.Cognome, Ind.Nome FROM Indirizzi AS Ind
```

74.5.2.2 Interrogazioni ordinate

«

Per ottenere un elenco ordinato in base a qualche criterio, si utilizza l'istruzione '**SELECT**' con l'indicazione di un'espressione in base alla quale effettuare l'ordinamento. Questa espressione è preceduta dalle parole chiave '**ORDER BY**':

```
SELECT espress_col_1 [,...espress_col_n]
    FROM relazione
    [WHERE condizione]
    ORDER BY espressione [ASC|DESC] [,...]
```

L'espressione può essere il nome di un attributo, oppure un'espressione che genera un risultato da uno o più attributi; l'aggiunta eventuale della parola chiave '**ASC**', o '**DESC**', permette di specificare un

ordinamento crescente, o discendente. Come si vede, le espressioni di ordinamento possono essere più di una, separate con una virgola.

```
SELECT Cognome, Nome FROM Indirizzi ORDER BY Cognome
```

L'esempio mostra un'applicazione molto semplice del problema, in cui si ottiene un elenco dei soli attributi '**Cognome**' e '**Nome**', della relazione '**Indirizzi**', ordinato per '**Cognome**'.

```
SELECT Cognome, Nome FROM Indirizzi ORDER BY Cognome, Nome
```

Questo esempio, aggiunge l'indicazione del nome nella chiave di ordinamento, in modo che in presenza di cognomi uguali, la scelta venga fatta in base al nome.

```
SELECT Cognome, Nome
    FROM Indirizzi
    ORDER BY TRIM( Cognome ), TRIM( Nome )
```

Questo ultimo esempio mostra l'utilizzo di due espressioni come chiave di ordinamento. Per la precisione, la funzione ***TRIM()***, usata in questo modo, serve a eliminare gli spazi iniziali e finali superflui. In questo modo, se i nomi e i cognomi sono stati inseriti con degli spazi iniziali, questi non vanno a influire sull'ordinamento.

74.5.2.3 Interrogazioni simultanee di più relazioni

Se dopo la parola chiave '**FROM**' si indicano più relazioni (ciò vale anche se si indica più volte la stessa relazione), si intende fare riferimento a una relazione generata dal prodotto di queste. Se per esempio si vogliono abbinare due relazioni, una di tre tuple con due attributi e un'altra di due tuple con due attributi, quello che si ottiene è una relazione con quattro attributi composta da sei tuple. Infatti,

ogni tupla della prima relazione risulta abbinata con ogni tupla della seconda.

```
SELECT specificazione_dell'attributo_1 [ , ...specificazione_dell'attributo_n ]
    FROM specificazione_della_relazione_1 [ , ...specificazione_della_relazione_n ]
    [WHERE condizione ]
```

Viene proposto un esempio banalizzato, con il quale poi si vuole eseguire un'elaborazione (figura 74.54).

Figura 74.54. Relazioni '**Articoli**' e '**Movimenti**' di una gestione del magazzino ipotetica.

Articoli	
Codice	**Descrizione**
vite30	Vite 3 mm
dado30	Dado 3 mm
rond50	Rondella 5 mm

Movimenti			
Codice	**Data**	**Carico**	**Scarico**
dado30	01/01/2012	1200	
vite30	01/01/2012		800
vite30	03/01/2012	2000	
rond50	03/01/2012		500

Da questa situazione si vuole ottenere la congiunzione della relazione '**Movimenti**' con tutte le informazioni corrispondenti della relazione '**Articoli**', basando il riferimento sull'attributo '**Codice**'. In pratica si vuole ottenere la relazione della figura 74.55.

Tabella 74.55. Risultato del join che si intende ottenere tra la relazione '**Movimenti**' e la relazione '**Articoli**'.

Codice	Data	Carico	Scarico	Descrizione
dado30	01/01/2012	1200		Dado 3 mm
vite30	01/01/2012	2000		Vite 3 mm
vite30	03/01/2012		800	Vite 3 mm
rond50	03/01/2012		500	Rondella 5 mm

Considerato che da un'istruzione '**SELECT**' contenente il riferimento a più relazioni si genera il prodotto tra queste, si pone poi il problema di eseguire una proiezione degli attributi desiderati e, soprattutto, di selezionare le tuple. In questo caso, la selezione deve essere basata sulla corrispondenza tra l'attributo '**Codice**' della prima relazione, con lo stesso attributo della seconda. Dovendo fare riferimento a due attributi di relazioni differenti, aventi però lo stesso nome, diviene indispensabile indicare i nomi degli attributi prefissandoli con i nomi delle relazioni rispettive.

```
SELECT
    Movimenti.Codice,
    Movimenti.Data,
    Movimenti.Carico,
    Movimenti.Scarico,
    Articoli.Descrizione
    FROM Movimenti, Articoli
    WHERE Movimenti.Codice = Articoli.Codice;
```

L'interrogazione simultanea di più relazioni si presta anche per elaborazioni della stessa relazione più volte. In tal caso, diventa obbligatorio l'uso degli alias. Si osservi il caso seguente:

```
SELECT Ind1.Cognome, Ind1.Nome
    FROM Indirizzi AS Ind1, Indirizzi AS Ind2
    WHERE
        Ind1.Cognome = Ind2.Cognome
    AND Ind1.Nome <> Ind2.Nome
```

Il senso di questa interrogazione, che utilizza la stessa relazione degli indirizzi per due volte con due alias differenti, è quello di ottenere l'elenco delle persone che hanno lo stesso cognome, avendo però un nome differente.

Esiste anche un'altra situazione in cui si ottiene l'interrogazione simultanea di più relazioni: l'***unione***. Si tratta semplicemente di attaccare il risultato di un'interrogazione su una relazione con quello di un'altra relazione, quando gli attributi finali appartengono allo stesso tipo di dati.

```
SELECT specificazione_attributo_1 [ , ...specificazione_attributo_n ]
        FROM specificazione_relazione_1 [ , ...specificazione_relazione_n ]
        [WHERE condizione ]
    UNION
        SELECT specificatore_attributo_1 [ , ...specificazione_attributo_n ]
            FROM specificazione_relazione_1 [ , ...specificazione_relazione_n ]
            [WHERE condizione ]
```

Lo schema sintattico dovrebbe essere abbastanza esplicito: si uniscono due istruzioni '**SELECT**' in un risultato unico, attraverso la parola chiave '**UNION**'.

74.5.2.4 Condizioni

La condizione che esprime la selezione delle tuple può essere composta come si vuole, purché il risultato sia di tipo logico e i dati a cui si fa riferimento provengano dalle relazioni di partenza. Quindi si possono usare anche altri operatori di confronto, funzioni e operatori booleani.

È bene ricordare che il valore indefinito, rappresentato da '**NULL**', è diverso da qualunque altro valore, compreso un altro valore indefinito. Per verificare che un valore sia o non sia indefinito, si deve usare l'operatore '**IS NULL**' oppure '**IS NOT NULL**'.

74.5.2.5 Aggregazioni

L'aggregazione è una forma di interrogazione attraverso cui si ottengono risultati riepilogativi del contenuto di una relazione, in forma di relazione contenente una sola tupla. Per questo si utilizzano delle funzioni speciali al posto dell'espressione che esprime gli attributi del risultato. Queste funzioni restituiscono un solo valore e come tali concorrono a creare un'unica tupla. Le funzioni di aggregazione sono: ***COUNT()***, ***SUM()***, ***MAX()***, ***MIN()***, ***AVG()***. Per intendere il problema, si osservi l'esempio seguente:

```
SELECT COUNT(*) FROM Movimenti WHERE …
```

In questo caso, quello che si ottiene è solo il numero di tuple della relazione '**Movimenti**' che soddisfano la condizione posta dopo la parola chiave '**WHERE**' (qui non è stata indicata). L'asterisco posto co-

me parametro della funzione ***COUNT()*** rappresenta effettivamente l'elenco di tutti i nomi degli attributi della relazione '**Movimenti**'.

Quando si utilizzano funzioni di questo tipo, occorre considerare che l'elaborazione si riferisce alla relazione virtuale generata dopo la selezione posta da '**WHERE**'.

La funzione ***COUNT()*** può essere descritta attraverso la sintassi seguente:

```
COUNT( * )
```

```
COUNT( [DISTINCT|ALL] lista_attributi)
```

Utilizzando la forma già vista, quella dell'asterisco, si ottiene solo il numero delle tuple della relazione. L'opzione '**DISTINCT**', seguita da una lista di nomi di attributi, fa in modo che vengano contate le tuple contenenti valori differenti per quel gruppo di attributi. L'opzione '**ALL**' è implicita quando non si usa '**DISTINCT**' e indica semplicemente di contare tutte le tuple.

Il conteggio delle tuple esclude in ogni caso quelle in cui il contenuto di tutti gli attributi selezionati è indefinito ('**NULL**').

Le altre funzioni aggreganti non prevedono l'asterisco, perché fanno riferimento a un'espressione che genera un risultato per ogni tupla ottenuta dalla selezione.

```
SUM( [DISTINCT|ALL] espressione)
```

```
MAX( [DISTINCT|ALL] espressione)
```

```
MIN( [DISTINCT|ALL] espressione)
```

```
AVG( [DISTINCT|ALL] espressione)
```

In linea di massima, per tutti questi tipi di funzioni aggreganti, l'espressione deve generare un risultato numerico, sul quale calcolare la sommatoria, ***SUM()***, il valore massimo, ***MAX()***, il valore minimo, ***MIN()***, la media, ***AVG()***.

L'esempio seguente calcola lo stipendio medio degli impiegati, ottenendo i dati da un'ipotetica relazione '**`Emolumenti`**', limitandosi ad analizzare le tuple riferite a un certo settore.

```
SELECT AVG( Stipendio ) FROM Emolumenti
    WHERE Settore = 'Amministrazione'
```

L'esempio seguente è una variante in cui si estraggono rispettivamente lo stipendio massimo, medio e minimo.

```
SELECT MAX( Stipendio ), AVG( Stipendio ), MIN( Stipendio )
    FROM Emolumenti WHERE Settore = 'Amministrazione'
```

L'esempio seguente è invece volutamente **errato**, perché si mescolano funzioni aggreganti assieme a espressioni di attributi normali.

```
-- Esempio errato
SELECT MAX( Stipendio ), Settore FROM Emolumenti
    WHERE Settore = 'Amministrazione'
```

74.5.2.6 Raggruppamenti

«

Le aggregazioni possono essere effettuate in riferimento a gruppi di tuple, distinguibili in base al contenuto di uno o più attributi. In questo tipo di interrogazione si può generare solo una relazione composta da tanti attributi quanti sono quelli presi in considerazione dall'opzione di raggruppamento, assieme ad altre contenenti solo espressioni di aggregazione.

Alla sintassi normale già vista nelle sezioni precedenti, si aggiunge la clausola '**GROUP BY**'.

```
SELECT specificazione_dell'attributo_1 [ , ...specificazione_dell'attributo_n ]
    FROM specificazione_della_relazione_1 [ , ...specificazione_della_relazione_n ]
    [WHERE condizione ]
    GROUP BY attributo_1 [ , ... ]
```

Per comprendere l'effetto di questa sintassi, si deve scomporre idealmente l'operazione di selezione da quella di raggruppamento:

1. la relazione ottenuta dall'istruzione '**SELECT**...**FROM**' viene filtrata dalla condizione '**WHERE**';
2. la relazione risultante viene riordinata in modo da raggruppare le tuple in cui i contenuti degli attributi elencati dopo l'opzione '**GROUP BY**' sono uguali;
3. su questi gruppi di tuple vengono valutate le funzioni di aggregazione.

Figura 74.62. Carichi e scarichi in magazzino.

Movimenti				
Codice	**Data**	**Carico**	**Scarico**	**...**
vite40	01/01/2012	1200		...
vite30	01/01/2012		800	...
vite40	01/01/2012	1500		...
vite30	02/01/2012		1000	...
vite30	03/01/2012	2000		...
rond50	03/01/2012		500	...
vite40	04/01/2012	2200		...

Si osservi la relazione riportata in figura 74.62, mostra la solita sequenza di carichi e scarichi di magazzino. Si potrebbe porre il problema di conoscere il totale dei carichi e degli scarichi per ogni articolo di magazzino. La richiesta può essere espressa con l'istruzione seguente:

```
SELECT Codice, SUM( Carico ), SUM( Scarico ) FROM Movimenti
    GROUP BY Codice
```

Quello che si ottiene appare nella figura 74.64.

Figura 74.64. Carichi e scarichi totali.

Codice	**SUM(Carico)**	**SUM(Scarico)**
vite40	4900	
vite30	2000	1800
rond50		500

Volendo si possono fare i raggruppamenti in modo da avere i totali distinti anche in base al giorno, come nell'istruzione seguente:

```
SELECT Codice, Data, SUM( Carico ), SUM( Scarico )
    FROM Movimenti GROUP BY Codice, Data
```

Come già affermato, la condizione posta dopo la parola chiave '**WHERE**' serve a filtrare inizialmente le tuple da considerare nel raggruppamento. Se quello che si vuole è filtrare ulteriormente il risultato di un raggruppamento, occorre usare la clausola '**HAVING**'.

```
SELECT specificazione_dell'attributo_1 [, ...specificazione_dell'attributo_n]
    FROM specificazione_della_relazione_1 [, ...specificazione_della_relazione_n]
    [WHERE condizione]
    GROUP BY attributo_1 [, ...]
    HAVING condizione
```

L'esempio seguente serve a ottenere il raggruppamento dei carichi e scarichi degli articoli, limitando però il risultato a quelli per i quali sia stata fatta una quantità di scarichi consistente (superiore a 1000 unità).

```
SELECT Codice, SUM( Carico ), SUM( Scarico ) FROM Movimenti
    GROUP BY Codice
    HAVING SUM( Scarico ) > 1000
```

Dall'esempio già visto in figura 74.64 risulterebbe escluso l'articolo '**rond50**'.

74.5.3 Trasferimento di dati in un'altra relazione

«

Alcune forme particolari di interrogazioni SQL possono essere utilizzate per inserire dati in relazioni esistenti o per crearne di nuove.

74.5.3.1 Creazione di una nuova relazione a partire da altre

L'istruzione '**SELECT**' può servire per creare una nuova relazione a partire dai dati ottenuti dalla sua interrogazione.

```
SELECT specificazione_dell'attributo_1 [ , ...specificazione_dell'attributo_n ]
    INTO TABLE relazione_da_generare
    FROM specificazione_della_relazione_1 [ , ...specificazione_della_relazione_n ]
    [WHERE condizione ]
```

L'esempio seguente crea la relazione '**Mia_prova**' come risultato della fusione delle relazioni '**Indirizzi**' e '**Presenze**'.

```
SELECT
    Presenze.Giorno,
    Presenze.Ingresso,
    Presenze.Uscita,
    Indirizzi.Cognome,
    Indirizzi.Nome
    INTO TABLE Mia_prova
    FROM Presenze, Indirizzi
    WHERE Presenze.Codice = Indirizzi.Codice;
```

74.5.3.2 Inserimento in una relazione esistente

L'inserimento di dati in una relazione esistente prelevando da dati contenuti in altre, può essere fatta attraverso l'istruzione '**INSERT**' sostituendo la clausola '**VALUES**' con un'interrogazione ('**SELECT**').

```
INSERT INTO nome_relazione [(attributo_1...attributo_n)]
    SELECT espressione_1, ... espressione_n
        FROM relazioni_di_origine
        [WHERE condizione]
```

L'esempio seguente aggiunge alla relazione dello storico delle presenze le registrazioni vecchie che poi vengono cancellate.

```
INSERT INTO PresenzeStorico (
        PresenzeStorico.Codice,
        PresenzeStorico.Giorno,
        PresenzeStorico.Ingresso,
        PresenzeStorico.Uscita
    )
    SELECT
        Presenze.Codice,
        Presenze.Giorno,
        Presenze.Ingresso,
        Presenze.Uscita
        FROM Presenze
        WHERE Presenze.Giorno <= '01/01/1999';

DELETE FROM Presenze WHERE Giorno <= '01/01/1999';
```

74.5.4 Viste

«

Le viste sono delle relazioni virtuali ottenute a partire da relazioni vere e proprie o da altre viste, purché non si formino ricorsioni. Il concetto non dovrebbe risultare strano. In effetti, il risultato delle interrogazioni è sempre in forma di relazione. La vista crea una sorta di interrogazione permanente che acquista la personalità di una relazione normale.

```
CREATE VIEW nome_vista [(attributo_1 [,...attributo_n)]]
    AS richiesta
```

Dopo la parola chiave '**AS**' deve essere indicato ciò che compone un'istruzione '**SELECT**'. L'esempio seguente, genera la vista dei movimenti di magazzino del solo articolo '**vite30**'.

```
CREATE VIEW Movimenti_Vite30
    AS SELECT Codice, Data, Carico, Scarico
        FROM Movimenti
        WHERE Codice = 'vite30'
```

L'eliminazione di una vista si ottiene con l'istruzione '**DROP VIEW**', come illustrato dallo schema sintattico seguente:

```
DROP VIEW nome_vista
```

Volendo eliminare la vista '**Movimenti_Vite30**', si può intervenire semplicemente come nell'esempio seguente:

```
DROP VIEW Movimenti_Vite30
```

74.5.5 Cursori

Quando il risultato di un'interrogazione SQL deve essere gestito all'interno di un programma, si pone un problema nel momento in cui ciò che si ottiene è composto da più di una sola tupla. Per poter scorrere un elenco ottenuto attraverso un'istruzione '**SELECT**', tupla per tupla, si deve usare un ***cursore***.

La dichiarazione e l'utilizzo di un cursore avviene all'interno di una transazione. Quando la transazione si chiude attraverso un'istruzione '**COMMIT**' o '**ROLLBACK**', si chiude anche il cursore.

74.5.5.1 Dichiarazione e apertura

«

L'SQL prevede due fasi prima dell'utilizzo di un cursore: la dichiarazione e la sua apertura:

```
DECLARE cursore [INSENSITIVE] [SCROLL] CURSOR FOR
    SELECT …
```

```
OPEN cursore
```

Nella dichiarazione, la parola chiave '**INSENSITIVE**' serve a stabilire che il risultato dell'interrogazione che si scandisce attraverso il cursore, non deve essere sensibile alle variazioni dei dati originali; la parola chiave '**SCROLL**' indica che è possibile estrarre più tuple simultaneamente attraverso il cursore.

```
DECLARE Mio_cursore CURSOR FOR
    SELECT
        Presenze.Giorno,
        Presenze.Ingresso,
        Presenze.Uscita,
        Indirizzi.Cognome,
        Indirizzi.Nome
        FROM Presenze, Indirizzi
        WHERE Presenze.Codice = Indirizzi.Codice;
```

L'esempio mostra la dichiarazione del cursore '**Mio_cursore**', abbinato alla selezione degli attributi composti dal collegamento di due relazioni, '**Presenze**' e '**Indirizzi**', dove le tuple devono avere lo stesso numero di codice. Per attivare questo cursore, lo si deve aprire come nell'esempio seguente:

```
OPEN Mio_cursore
```

74.5.5.2 Scansione

La scansione di un'interrogazione inserita in un cursore, avviene attraverso l'istruzione '**FETCH**'. Il suo scopo è quello di estrarre una tupla alla volta, in base a una posizione, relativa o assoluta.

```
FETCH [ [ NEXT | PRIOR | FIRST | LAST
              | { ABSOLUTE | RELATIVE } n ]
    FROM cursore ]
    INTO :variabile [,…]
```

Le parole chiave '**NEXT**', '**PRIOR**', '**FIRST**', '**LAST**', permettono rispettivamente di ottenere la tupla successiva, quella precedente, la prima e l'ultima. Le parole chiave '**ABSOLUTE**' e '**RELATIVE**' sono seguite da un numero, corrispondente alla scelta della tupla ***n***-esima, rispetto all'inizio del gruppo per il quale è stato definito il cursore ('**ABSOLUTE**'), oppure della tupla ***n***-esima rispetto all'ultima tupla estratta da un'istruzione '**FETCH**' precedente.

Le variabili indicate dopo la parola chiave '**INTO**', che in particolare sono precedute da due punti ('**:**'), ricevono ordinatamente il contenuto dei vari attributi della tupla estratta. Naturalmente, le variabili in questione devono appartenere a un linguaggio di programmazione che incorpora l'SQL, dal momento che l'SQL stesso non fornisce questa possibilità.

```
FETCH NEXT FROM Mio_cursore
```

L'esempio mostra l'uso tipico di questa istruzione, dove si legge la tupla successiva (se non ne sono state lette fino a questo punto, si

tratta della prima), dal cursore dichiarato e aperto precedentemente. L'esempio seguente è identico dal punto di vista funzionale.

```
FETCH RELATIVE 1 FROM Mio_cursore
```

I due esempi successivi sono equivalenti e servono a ottenere la tupla precedente.

```
FETCH PRIOR FROM Mio_cursore
```

```
FETCH RELATIVE -1 FROM Mio_cursore
```

74.5.5.3 Chiusura

«

Il cursore, al termine dell'utilizzo, deve essere chiuso:

```
CLOSE cursore
```

Seguendo gli esempi visti in precedenza, per chiudere il cursore '**`Mio_cursore`**' basta l'istruzione seguente:

```
CLOSE Mio_cursore
```

74.6 DCL

«

DCL, ovvero *Data control language*, è il linguaggio usato per il «controllo» delle basi di dati. In questa sezione viene trattato il linguaggio SQL per ciò che riguarda la gestione delle basi di dati, degli utenti, dei privilegi assegnati loro e il controllo delle transazioni.

74.6.1 Gestione delle utenze

«

La gestione degli accessi in una base di dati è molto importante e potenzialmente indipendente dall'eventuale gestione degli utenti del sistema operativo sottostante. Per quanto riguarda i sistemi Unix, il

DBMS può riutilizzare la definizione degli utenti del sistema operativo, farvi riferimento, oppure astrarsi completamente (spesso vale questa ultima ipotesi).

Un DBMS SQL richiede la presenza di almeno un amministratore complessivo, che come tale abbia sempre tutti i privilegi necessari a intervenire come vuole nel DBMS. Il nome simbolico predefinito per questo utente dal linguaggio SQL standard è '**_SYSTEM**'.

Il sistema di definizione e controllo delle utenze è esterno al linguaggio SQL standard; tuttavia, i DBMS principali utilizzano istruzioni abbastanza uniformi per questo scopo.

Per la creazione di un utente si dispone normalmente dell'istruzione '**CREATE USER**', con opzioni che dipendono dalle caratteristiche particolari del DBMS, nella gestione delle utenze. I due modelli sintattici successivi si riferiscono, rispettivamente, a Oracle e a PostgreSQL, ma omettono varie opzioni specifiche e presumono che l'utente debba essere identificato attraverso una parola d'ordine:

```
CREATE USER nome_utente IDENTIFIED BY 'parola_d'ordine'
```

```
CREATE USER nome_utente [WITH PASSWORD 'parola_d'ordine']
```

Nel caso di MySQL, invece di introdurre un'istruzione che non esiste nello standard, si estende quella con cui si concedono i privilegi (descritta in un'altra sezione):

```
GRANT privilegi
    ON risorsa [, ...]
    TO utente
    IDENTIFIED BY 'parola_d'ordine'
    [WITH GRANT OPTION]
```

In tal modo, attribuendo dei privilegi a un utente, se questo non esiste ancora, viene creato contestualmente. Si osservi comunque, che le versioni più recenti di MySQL dispongono di un'istruzione '**CREATE USER**' simile a quella di altri DBMS.

Per l'eliminazione di un utente si dispone normalmente dell'istruzione '**DROP USER**', con opzioni che di solito consentono l'eliminazione contestuale di tutto ciò che appartiene a tale utente:

```
DROP USER nome_utente
```

Per modificare la parola d'ordine di un utente, si dispone normalmente dell'istruzione '**ALTER USER**', con la quale si potrebbero cambiare anche altre opzioni legate ai privilegi generali di cui può disporre tale utente. I due modelli sintattici successivi si riferiscono, rispettivamente, a Oracle e a PostgreSQL, omettendo opzioni che non sono indispensabili:

```
ALTER USER nome_utente IDENTIFIED BY 'parola_d'ordine'
```

```
ALTER USER nome_utente [WITH PASSWORD 'parola_d'ordine']
```

Nel caso di MySQL si usa una forma differente:

```
SET PASSWORD FOR nome_utente = PASSWORD('parola_d'ordine')
```

74.6.2 Gestione delle basi di dati

La creazione e l'eliminazione delle basi di dati è una funzione non considerata dallo standard SQL, anche se è normale che un DBMS consenta la gestione di più basi di dati simultaneamente. Pertanto, i vari DBMS offrono delle istruzioni SQL abbastanza uniformi:

```
CREATE DATABASE nome_base_di_dati
```

Di solito, salvo indicazione diversa derivante da opzioni particolari aggiunte all'istruzione, l'utente che crea la base di dati ne diviene il proprietario, con ogni facoltà sulla stessa, anche quella di eliminarla.

È il caso di osservare che uno dei problemi tecnici da considerare nella creazione di una base di dati sta nel definire la codifica da usare per la memorizzazione delle informazioni testuali. Di solito, questo genere di cose viene definito tramite delle opzioni specifiche, che si aggiungono al modello sintetico e generalizzato mostrato qui.

L'eliminazione di una base di dati richiede generalmente un'istruzione altrettanto semplice:

```
DROP DATABASE nome_base_di_dati
```

La differenza più importante tra i vari DBMS consiste nel modo di comportarsi di fronte a questo comando, quando la base di dati non è vuota. Se esiste un contenuto, la cancellazione potrebbe essere ri-

fiutata, oppure potrebbe essere ammessa se si aggiungono opzioni specifiche che servono a confermarne l'eliminazione. Tuttavia, non si può contare su un controllo di questo genere e la cancellazione di una base di dati richiede sempre la dovuta prudenza.

74.6.3 Gestione dei privilegi standard

«

L'utente che crea una relazione, o un'altra risorsa, è il suo creatore. Su tale risorsa è l'unico utente che possa modificarne la struttura e che possa eliminarla. In pratica è l'unico che possa usare le istruzioni '**`DROP`**' e '**`ALTER`**'. Chi crea una relazione, o un'altra risorsa, può concedere o revocare i privilegi degli altri utenti su di essa.

I privilegi che si possono concedere o revocare su una risorsa sono di vario tipo, espressi attraverso una parola chiave particolare. È bene considerare i casi seguenti:

Privilegio	Descrizione
`SELECT`	rappresenta l'operazione di lettura del valore di un oggetto della risorsa, per esempio dei valori di una tupla da una relazione (in pratica si riferisce all'uso dell'istruzione '**`SELECT`**');
`INSERT`	rappresenta l'azione di inserire un nuovo oggetto nella risorsa, come l'inserimento di una tupla in una relazione;
`UPDATE`	rappresenta l'operazione di aggiornamento del valore di un oggetto della risorsa, per esempio la modifica del contenuto di una tupla di una relazione;
`DELETE`	rappresenta l'eliminazione di un oggetto dalla risorsa, come la cancellazione di una tupla da una relazione;

Privilegio	Descrizione
`ALL PRIVILEGES`	rappresenta simultaneamente tutti i privilegi possibili riferiti a un oggetto.

I privilegi su una relazione, o su un'altra risorsa, vengono concessi attraverso l'istruzione '**GRANT**':

```
GRANT privilegi
    ON risorsa [,…]
    TO utenti
    [WITH GRANT OPTION]
```

Nella maggior parte dei casi, le risorse da controllare coincidono con una relazione. L'esempio seguente permette all'utente '**Pippo**' di leggere il contenuto della relazione '**Movimenti**':

```
GRANT SELECT ON Movimenti TO Pippo
```

L'esempio seguente, concede tutti i privilegi sulla relazione '**Movimenti**' agli utenti '**Pippo**' e '**Arturo**':

```
GRANT ALL PRIVILEGES ON Movimenti TO Pippo, Arturo
```

L'opzione '**WITH GRANT OPTION**' permette agli utenti presi in considerazione di concedere a loro volta tali privilegi ad altri utenti. L'esempio seguente concede all'utente '**Pippo**' di accedere in lettura al contenuto della relazione '**Movimenti**' e gli permette di concedere lo stesso privilegio ad altri:

```
GRANT SELECT ON Movimenti TO Pippo WITH GRANT OPTION
```

I privilegi su una relazione, o un'altra risorsa, vengono revocati attraverso l'istruzione '**REVOKE**':

```
REVOKE privilegi
    ON risorsa [, ...]
    FROM utenti
```

L'esempio seguente toglie all'utente '**Pippo**' il permesso di accedere in lettura al contenuto della relazione '**Movimenti**':

```
REVOKE SELECT ON Movimenti FROM Pippo
```

L'esempio seguente toglie tutti i privilegi sulla relazione '**Movimenti**' agli utenti '**Pippo**' e '**Arturo**':

```
REVOKE ALL PRIVILEGES ON Movimenti FROM Pippo, Arturo
```

74.6.4 Controllo delle transazioni

«

Una transazione SQL, è una sequenza di istruzioni che rappresenta un corpo unico dal punto di vista della memorizzazione effettiva dei dati. In altre parole, secondo l'SQL, la registrazione delle modifiche apportate alla base di dati avviene in modo asincrono, raggruppando assieme l'effetto di gruppi di istruzioni determinati.

Una transazione inizia nel momento in cui l'interprete SQL incontra, generalmente, l'istruzione '**START TRANSACTION**', terminando con l'istruzione '**COMMIT**', oppure '**ROLLBACK**': nel primo caso si conferma la transazione che viene memorizzata regolarmente, mentre nel secondo si richiede di annullare le modifiche apportate dalla transazione.

```
START TRANSACTION
```

```
COMMIT [WORK]
```

```
ROLLBACK [WORK]
```

Stando così le cose, si intende la necessità di utilizzare regolarmente l'istruzione '**COMMIT**' per memorizzare i dati quando non esiste più la necessità di annullare le modifiche.

```
START TRANSACTION
…
COMMIT

INSERT INTO Indirizzi
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    )

COMMIT
```

L'esempio mostra un uso intensivo dell'istruzione '**COMMIT**', dove dopo l'inserimento di una tupla nella relazione '**Indirizzi**', viene confermata immediatamente la transazione.

```
START TRANSACTION
…
COMMIT

INSERT INTO Indirizzi
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    )

ROLLBACK
```

Questo esempio mostra un ripensamento (per qualche motivo). Dopo l'inserimento di una tupla nella relazione '**Indirizzi**', viene annullata la transazione, riportando la relazione allo stato precedente.

74.7 Riferimenti

«

- Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Riccardo Torlone, *Basi di dati, concetti, linguaggi e architetture*, McGraw-Hill
- James Hoffmann, *Introduction to Structured Query Language*, *http://www.highcroft.com/highcroft/sql_intro.pdf*
- *JCC's SQL Std. Page*, *http://www.jcc.com/sql.htm*
- *SQL Reference Page*, *http://www.contrib.andrew.cmu.edu/~shadow/sql.html*

- *SQL92 BNF*, http://www.contrib.andrew.cmu.edu/~shadow/sql/sql2bnf.aug92.txt
- *ISO/IEC 9075:1992, Database Language SQL*, http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt
- *BNF Grammar for ISO/IEC 9075:1999 - Database Language SQL (SQL-99)*, http://savage.net.au/SQL/sql-99.bnf , http://savage.net.au/SQL/sql-99.bnf.html
- *PostgreSQL*, http://www.postgresql.org/
- *MySQL*, http://www.mysql.com/
- *MaxDB*, http://www.mysql.com/sap/
- *Firebird*, http://firebird.sourceforge.net/

75.1 Struttura e preparazione

«

PostgreSQL[1] è un DBMS (*Data base management system*) relazionale, esteso agli oggetti. In questo capitolo si vuole introdurre al suo utilizzo e accennare alla sua struttura, senza affrontare le particolarità del linguaggio di interrogazione. Il nome lascia intendere che si tratti di un DBMS in grado di comprendere le istruzioni SQL.

75.1.1 Struttura dei dati nel file system

PostgreSQL, a parte i programmi binari, gli script e la documentazione, colloca i file di gestione delle basi di dati a partire da una certa directory, che nella documentazione originale viene definita '**PGDATA**'. Questo è il nome di una variabile di ambiente che può essere utilizzata per informare i vari programmi di PostgreSQL della sua collocazione; tuttavia, di solito questo meccanismo della variabile di ambiente non viene utilizzato, specificando tale directory in fase di compilazione dei sorgenti oppure avviando i programmi con opzioni appropriate.

Tutti i programmi che compongono il sistema di PostgreSQL, che hanno la necessità di sapere dove si trovano i dati, oltre al meccanismo della variabile di ambiente '**PGDATA**' permettono di indicare tale directory attraverso un'opzione della riga di comando. I programmi più importanti riconoscono l'opzione '**-D**'. Come si può intuire, l'utilizzo di questa opzione, o di un'altra equivalente per gli altri programmi, fa in modo che l'indicazione della variabile '**PGDATA**' non abbia effetto.

Questa directory è contenuta solitamente nella directory iniziale dell'utente di sistema per l'amministrazione di PostgreSQL, che dovrebbe essere '**postgres**'. La directory iniziale dell'utente '**postgres**' (ovvero '`~postgres/`') è normalmente '`/var/lib/postgres/`'; la directory usata normalmente per collocarvi le basi di dati è normalmente '`~postgres/data/`', ovvero '`/var/lib/postgres/data/`'. Di norma, tutto ciò che si trova a partire da '`~postgres/`' appartiene all'utente '**postgres**', anche se

i permessi per il gruppo e gli altri utenti variano a seconda della circostanza.

Inizialmente, la directory che costituisce l'inizio delle basi di dati (‘`~postgres/data/`’) dovrebbe contenere dei file di configurazione, una basi di dati amministrativa (trasparente) e una base di dati da usare come modello per la produzione di altre (‘**`template1`**’) o semplicemente per accedere al DBMS quando non se ne può indicare un'altra. Naturalmente, se per qualche ragione si utilizza l'utenza ‘**`postgres`**’ in modo normale, nella sua directory personale (‘`~postgres/`’) potrebbero apparire dei file che riguardano la personalizzazione di questo utente (‘`.profile`’, ‘`.bash_history`’, o altre cose simili, in funzione dei programmi che si utilizzano).

75.1.2 Amministratore

L'amministratore dei servizi offerti dal DBMS PostgreSQL potrebbe essere una persona diversa dall'amministratore del sistema operativo (l'utente ‘**`root`**’) e corrisponde di solito all'utente ‘**`postgres`**’. In condizioni normali, tale utente del DBMS viene riconosciuto implicitamente da PostgreSQL, purché acceda localmente utilizzando un'utenza del sistema operativo con lo stesso nome.

Quando la propria distribuzione GNU è già predisposta per PostgreSQL, l'utente ‘**`postgres`**’ dovrebbe essere stato previsto (non importa il numero UID che gli sia stato abbinato), ma quasi sicuramente la parola d'ordine per l'accesso al sistema operativo dovrebbe essere «impossibile», come nell'esempio seguente:

```
postgres:!:101:101:PostgreSQL Server:/var/lib/postgres:/bin/bash
```

Come si vede, in questo esempio il campo della parola d'ordine è occupato da un punto esclamativo che di fatto impedisce l'accesso

all'utente '**postgres**'.

A questo punto si pongono due alternative, a seconda che si voglia affidare la gestione del DBMS allo stesso utente '**root**' oppure che si voglia incaricare per questo un altro utente. Nel primo caso non occorrono cambiamenti: l'utente '**root**' può diventare '**postgres**' quando vuole con il comando '**su**'.

```
# su postgres [Invio]
```

Nel secondo caso, l'attribuzione di una parola d'ordine all'utente '**postgres**' permetterebbe a una persona diversa di amministrare il DBMS.

```
# passwd postgres [Invio]
```

Di solito, nella sua configurazione iniziale, l'utente '**postgres**' ha la facoltà di accedere localmente al DBMS, senza bisogno di altre forme di autenticazione, a parte il fatto di essere riconosciuto dal sistema operativo proprio con quello stesso nome. Ciò dipende principalmente dalla configurazione contenuta nel file '`pg_hba.conf`', che viene descritto in seguito, all'interno di questo capitolo. Negli esempi che si mostrano qui, si presume proprio che l'utente '**postgres**' del sistema operativo, in quanto tale, sia riconosciuto così anche dal DBMS; se così non fosse, a causa della configurazione, è probabile vedere apparire la richiesta di introdurre una parola d'ordine, riferita però al DBMS.

75.1.3 Creazione del sistema di basi di dati

La prima volta che si installa PostgreSQL, è molto probabile che venga predisposta automaticamente la directory '`~postgres/`'. Se così non fosse, o se per qualche motivo si dovesse intervenire manualmente, si può utilizzare '**initdb**', che però potrebbe risiedere al di fuori dei percorsi normali contenuti nella variabile '**$PATH**'; precisamente potrebbe trattarsi della directory '`/usr/lib/postgresql/bin/`'.

[*percorso*]`initdb` [*opzioni*] [`--pgdata=`*directory* | `-D` *directory*]

Lo schema sintattico mostra in modo molto semplice l'uso di '**initdb**'. Se si definisce correttamente la variabile di ambiente '**PGDATA**', si può fare anche a meno delle opzioni, diversamente diventa necessario dare questa informazione attraverso l'opzione '**-D**'.

Volendo fare tutto da zero, occorre predisporre la directory iniziale in modo che appartenga dell'utente fittizio '**postgres**':

```
# mkdir ~postgres[Invio]

# chown postgres: ~postgres[Invio]
```

Prima di avviare '**initdb**', è bene utilizzare l'identità dell'utente amministratore di PostgreSQL:

```
# su postgres[Invio]
```

Successivamente, si deve avviare '**initdb**' specificando la directory a partire dalla quale si devono articolare i file che costituisco-

no le basi di dati. Come già descritto, la directory in questione è normalmente '`~postgres/data/`':

```
postgres$ /usr/lib/postgresql/bin/initdb ↩
↪         --locale=it_IT.UTF-8 ↩
↪         --encoding=UNICODE ↩
↪         --pgdata=/var/lib/postgres/data [Invio]

  The files belonging to this database system will be owned
  by user "postgres".
  This user must also own the server process.

  The database cluster will be initialized with locale
  it_IT.UTF-8.

  creating directory /var/lib/postgres/data... ok
  creating directory /var/lib/postgres/data/base... ok
  creating directory /var/lib/postgres/data/global... ok
  creating directory /var/lib/postgres/data/pg_xlog... ok
  creating directory /var/lib/postgres/data/pg_clog... ok
  selecting default max_connections... 100
  selecting default shared_buffers... 1000
  creating configuration files... ok
  creating template1 database in
  /var/lib/postgres/data/base/1... ok
  initializing pg_shadow... ok
  enabling unlimited row size for system tables... ok
  initializing pg_depend... ok
  creating system views... ok
  loading pg_description... ok
  creating conversions... ok
  setting privileges on built-in objects... ok
  creating information schema... ok
  vacuuming database template1... ok
  copying template1 to template0... ok
```

```
Success. The database server should be started
automatically.
If not, you can start the database server using:

    /etc/init.d/postgresql start
```

Nell'esempio sono state usate anche due opzioni il cui significato dovrebbe risultare intuitivo.

Tabella 75.3. Alcune opzioni per l'uso di '**initdb**'.

Opzione	Descrizione
`--pgdata=`***directory_pgdata*** `-D` ***directory_pgdata***	Stabilisce la directory iniziale del sistema di basi di dati di PostgreSQL che si vuole creare. Di solito deve corrispondere a '`~postgres/data/`'.
`--locale=`***sigla_locale***	Stabilisce la configurazione locale. Se non viene utilizzata questa opzione si usa il contenuto delle variabili di ambiente '**LANG**' ed eventualmente '**LC_***'.
`--encoding=`***codifica*** `-E` ***codifica***	Stabilisce la codifica della base di dati usata come modello ('**template1**'), diventando di conseguenza la codifica predefinita per le nuove basi di dati. Tra le varie sigle che si possono usare vale la pena di ricordare '**UNICODE**', '**SQL_ASCII**'.

Teoricamente, '**initdb**' fa tutto quello che è necessario fare; in pratica potrebbe non essere così. La prima cosa da considerare sono i file di configurazione, che, seguendo l'esempio mostrato, vengono collocati nella directory '`~postgres/data/`'. Molto probabilmente la propria distribuzione GNU è organizzata per avere i file di configurazione in una directory '`/etc/postgresql/`', o si-

mile. Se le cose stanno così, bisogna provvedere a sostituire i file di configurazione nella directory '`~postgres/data/`' con dei collegamenti simbolici appropriati.

Le distribuzioni GNU possono avere la necessità di passare alcune informazioni, tramite variabili di ambiente, all'utente fittizio '**`postgres`**', cosa che si ottiene con un file '`~postgres/.profile`' appropriato. Se si vuole ricreare la directory '`~postgres/`' da zero, ma si nota la presenza di file di configurazione della shell, è necessario accertarsi del loro contenuto e provvedere di conseguenza nella ricostruzione della directory.

Un'ultima questione importante da sistemare è la directory '`~postgres/dumpall/`', che serve a contenere versioni vecchie degli eseguibili di PostgreSQL, con lo scopo di recuperare i dati dalle versioni vecchie delle basi di dati. Normalmente è sufficiente recuperare la directory già usata in precedenza.

75.1.4 Avvio del servizio

Il DBMS di PostgreSQL si basa su un sistema cliente-servente, in cui, il programma che vuole interagire con una base di dati determinata deve farlo attraverso delle richieste inviate a un servente. In questo modo, il servizio può essere esteso anche attraverso la rete.

L'organizzazione di PostgreSQL prevede la presenza di un demone sempre in ascolto (può trattarsi di un socket di dominio Unix o anche di una porta TCP, che di solito corrisponde al numero 5432). Quando questo riceve una richiesta valida per iniziare una connessione, attiva una copia del servente vero e proprio (*back-end*), a cui affida la connessione con il cliente. Il demone in ascolto per le ri-

chieste di nuove connessioni è '**postmaster**', mentre il servente è '**postgres**'.

Purtroppo, la scelta del nome «postmaster» è un po' infelice, dal momento che potrebbe far pensare all'amministratore del servizio di posta elettronica. Come al solito occorre un po' di attenzione al contesto in cui ci si trova.

Generalmente, il demone '**postmaster**' viene avviato attraverso la procedura di inizializzazione del sistema, in modo indipendente dal supervisore dei servizi di rete. In pratica, di solito si utilizza uno script collocato all'interno di '`/etc/init.d/`', o in un'altra collocazione simile, per l'avvio e l'interruzione del servizio.

Durante il funzionamento del sistema, quando alcuni clienti sono connessi, si può osservare una dipendenza del tipo rappresentato dallo schema seguente:

```
--postmaster-+-postgres
             |-postgres
             `-postgres
```

Il demone '**postmaster**' si occupa di restare in ascolto in attesa di una richiesta di connessione con un servente '**postgres**' (il programma terminale, o *back-end* in questo contesto). Quando riceve questo tipo di richiesta mette in connessione il cliente (programma frontale, o *front-end*) con una nuova copia del servente '**postgres**'.

`postmaster` [***opzioni***]

Per poter compiere il suo lavoro, il demone deve essere a conoscenza di alcune notizie essenziali, tra cui in particolare: la collocazione del programma '**postgres**' (se questo non è in uno dei percorsi della variabile '**PATH**') e la directory da cui si dirama il sistema di file che costituisce l'insieme delle varie basi di dati. Queste notizie possono essere predefinite, nella configurazione usata al momento della compilazione dei sorgenti, oppure possono essere indicate attraverso la riga di comando.

Il demone '**postmaster**' e i processi terminali da lui controllati, gestiscono dei file che compongono le varie basi di dati del sistema. Trattandosi di un sistema di gestione dei dati molto complesso, è bene evitare di inviare il segnale '**SIGKILL**' (9), perché con questo si provoca la conclusione immediata del processo destinatario e di tutti i suoi discendenti, senza permettere una conclusione corretta. Al contrario, gli altri segnali sono accettabili, come per esempio un '**SIGTERM**' che viene dato in modo predefinito quando si utilizza il comando '**kill**'.

Tabella 75.5. Alcune opzioni per l'avvio di '**postmaster**'.

Opzione	Descrizione
`-D` ***directory_dei_dati***	Permette di specificare la directory di inizio della struttura dei dati del DBMS.

Opzione	Descrizione
`-S`	Specifica che il programma deve funzionare in modo «silenzioso», senza emettere alcuna segnalazione, diventando un processo discendente direttamente da quello iniziale (Init), disassociandosi dalla shell e quindi dal terminale da cui è stato avviato. Questa opzione viene utilizzata particolarmente per avviare il programma all'interno della procedura di inizializzazione del sistema, quando non sono necessari dei controlli di funzionamento.
`-b` ***percorso_del_programma_terminale***	Se il programma terminale, ovvero '**`postgres`**', non si trova in uno dei percorsi contenuti nella variabile di ambiente '**`PATH`**', è necessario specificare la sua collocazione (il percorso assoluto) attraverso questa opzione.

Opzione	Descrizione
`-d` [***livello_di_diagnosi***]	Questa opzione permette di attivare la segnalazione di messaggi diagnostici (*debug*), da parte di '**`postmaster`**' e da parte dei programmi terminali, a più livelli di dettaglio: 1, segnala solo il traffico di connessione; 2, o superiore, attiva la segnalazione diagnostica anche nei programmi terminali, oltre ad aggiungere dettagli sul funzionamento di '**`postmaster`**'. Di norma, i messaggi diagnostici vengono emessi attraverso lo standard output da parte di '**`postmaster`**', anche quando si tratta di messaggi provenienti dai programmi terminali. Perché abbia significato l'uso di questa opzione, occorre avviare '**`postmaster`**' **senza l'opzione** '**`-S`**'.
`-i`	Abilita le connessioni TCP/IP. Senza l'indicazione di questa opzione, sono ammissibili solo le connessioni locali attraverso socket di dominio Unix (*Unix domain socket*).

Opzione	Descrizione
`-p` ***porta***	Se viene avviato in modo da accettare le connessioni attraverso la rete (l'opzione '**`-i`**'), specifica una porta di ascolto diversa da quella predefinita (5432).

Segue la descrizione di alcuni esempi.

- `# su postgres -c 'postmaster -S ↩`
  `↪ -D/var/lib/postgres/data'` [*Invio*]

  L'utente '**`root`**', avvia '**`postmaster`**' dopo essersi trasformato temporaneamente nell'utente '**`postgres`**' (attraverso '**`su`**'), facendo in modo che il programma si disassoci dalla shell e dal terminale, diventando un discendente da Init. Attraverso l'opzione '**`-D`**' si specifica la directory di inizio dei file della base di dati.

- `# su postgres -c 'postmaster -i -S ↩`
  `↪ -D/var/lib/postgres/data'` [*Invio*]

  Come nell'esempio precedente, specificando che si vuole consentire, in modo preliminare, l'accesso attraverso la rete.

  Per consentire in pratica l'accesso attraverso la rete, occorre anche intervenire all'interno del file di configurazione '`~postgres/pg_hda.conf`'.

- `# su postgres -c 'nohup postmaster ↩`
  `↪ -D/var/lib/postgres/data ↩`
  `↪ > /var/log/pglog 2>&1 &'` [*Invio*]

L'utente '**root**', avvia '**postmaster**' in modo simile al precedente, dove in particolare viene diretto lo standard output all'interno di un file, per motivi diagnostici. Si osservi l'utilizzo di '**nohup**' per evitare l'interruzione del funzionamento di '**postmaster**' all'uscita del programma '**su**'.

- `# su postgres -c 'nohup postmaster ↩`
  `↪ -D/var/lib/postgres -d 1 ↩`
  `↪ > /var/log/pglog 2>&1 &'` [*Invio*]

Come nell'esempio precedente, con l'attivazione del primo livello diagnostico nei messaggi emessi.

Riquadro 75.6. Controllo diagnostico.

Inizialmente, l'utilizzo di PostgreSQL si può dimostrare poco intuitivo, soprattutto per ciò che riguarda le segnalazioni di errore, spesso troppo poco esplicite. In caso di difficoltà, per permettere di avere una visione un po' più chiara di ciò che accade, sarebbe bene fare in modo che '**postmaster**' produca dei messaggi diagnostici, possibilmente diretti a un file o a una console virtuale inutilizzata.
Per avere una visione immediata di ciò che accade, l'esempio seguente avvia '**postmaster**' in modo manuale e, oltre a conservare le informazioni diagnostiche in un file, le visualizza continuamente attraverso una console virtuale inutilizzata, che in questo caso è l'ottava:

`# su postgres` [*Invio*]

`$ nohup postmaster -D/var/lib/postgres/data -d 1 ↩`
`↪> /var/log/pglog 2>&1 &` [*Invio*]

`$ exit` [*Invio*]

`# nohup tail -f /var/lib/postgres > /dev/tty8 &` [*Invio*]

75.1.5 Configurazione del DBMS

«

Come già accennato, è possibile influenzare il comportamento del servente PostgreSQL attraverso opzioni della riga di comando e variabili di ambiente. Oltre a questi metodi, è possibile intervenire nel file '`~postgres/data/postgresql.conf`', attraverso direttive che assomigliano all'assegnamento di variabili. Il loro significato dovrebbe risultare intuitivo. Viene mostrato un estratto di esempio di questo file:

```
# PostgreSQL configuration file
…
#
# TCP/IP access is allowed by default, but the default
# access given in pg_hba.conf will permit it only from
# localhost, not other machines.
#
tcpip_socket = true
…
#
#       Message display
#
log_connections = true
log_pid = true
log_timestamp = true
…
#
#       Syslog
#
syslog = 2      # range 0-2
…
#
#       Misc
#
```

```
dynamic_library_path = ↩
↪'/usr/share/postgresql:↩
↪/usr/lib/postgresql:/usr/lib/postgresql/lib'
…
#
# How (by default) to present dates to the frontend; the
# user can override this setting for his own session.
# The choices are:
#   Style      Date              Timestamptz
#   ---------------------------------------------------
#   ISO        1999-07-17        1999-07-17 07:09:18+01
#   SQL        17/07/1999        17/07/1999 07:09:19 BST
#   POSTGRES   17-07-1999        Sat 17 Jul 07:09:19 1999 BST
#   GERMAN     17.07.1999        17.07.1999 07:09:19 BST
#
# It is also possible to specify month-day or day-month
# ordering in date input and output.  Americans tend to use
# month-day; Europeans use day-month.  Specify European or
# US.  This is used for interpreting date input, even if the
# output format is ISO.  Separate the two parameters
# by a comma with no spaces
#
datestyle = 'ISO,European'
…
LC_MESSAGES = 'C'
LC_MONETARY = 'C'
LC_NUMERIC = 'C'
LC_TIME = 'C'
```

Si può osservare la direttiva '**`tcpip_socket = true`**', che abilita l'accesso al servente attraverso la rete, ma che richiede di specificare meglio le possibilità di accesso attraverso il file '`~postgres/data/pg_hba.conf`'.

Tabella 75.8. Elenco dei formati di data gestibili con PostgreSQL.

Stile	Descrizione	Esempio
`ISO`	ISO 8601	2012-12-31
`SQL`	Tipo tradizionale	12/31/2012
`POSTGRESQL`	Tipo specifico di PostgreSQL	12-31-2012
`GERMAN`		31.12.2012

Nel caso particolare della distribuzione GNU/Linux Debian, può essere controllato tutto a partire dai file che si trovano nella directory '`/etc/postgresql/`'. In particolare, si trova in questa directory il file '`pg_hba.conf`' e il file '`postgresql.conf`', già descritti in altre sezioni; inoltre, si trova un file aggiuntivo che viene interpretato dallo script della procedura di inizializzazione del sistema che si occupa di avviare e di arrestare il servizio. Si tratta dei file '`/etc/postgresql/postmaster.conf`', attraverso il quale si possono controllare delle piccole cose a cui non si può accedere con il file '`postgresql.conf`', che altrimenti richiederebbero di intervenire attraverso le opzioni della riga di comando del demone relativo.

75.1.6 Accesso e autenticazione

«

L'accesso alle basi di dati viene permesso attraverso un sistema di autenticazione. I sistemi di autenticazione consentiti possono essere diversi e dipendono dalla configurazione di PostgreSQL fatta all'atto della compilazione dei sorgenti.

Il file di configurazione ‘`pg_hba.conf`’ (*Host-based authentication*), che si trova nella directory ‘`~postgres/data/`’, serve per controllare il sistema di autenticazione una volta installato PostgreSQL.

L’autenticazione degli utenti può avvenire in modo incondizionato (‘**`trust`**’), dove ci si fida del nome fornito come utente del DBMS, senza richiedere altro.

L’autenticazione può essere semplicemente disabilitata, nel senso di impedire qualunque accesso incondizionatamente. Questo può servire per impedire l’accesso da parte di un certo gruppo di nodi.

L’accesso può essere controllato attraverso l’abbinamento di una parola d’ordine agli utenti di PostgreSQL.

Inoltre, l’autenticazione può avvenire attraverso un sistema Kerberos, oppure attraverso il protocollo IDENT (sezione 43.3). In questo ultimo caso, ci si fida di quanto riportato dal sistema remoto il quale conferma o meno che la connessione appartenga a quell’utente che si sta connettendo.

Il file ‘`~postgres/data/pg_hba.conf`’ (ma spesso questo è un collegamento simbolico che punta a ‘`/etc/postgresql/pg_hba.conf`’ o a un’altra posizione simile) permette di definire quali nodi possono accedere al servizio DBMS di PostgreSQL, eventualmente stabilendo anche un abbinamento specifico tra basi di dati, utenti e nodi di rete.

Le righe vuote e il testo preceduto dal simbolo ‘**`#`**’ vengono ignorati. I record (cioè le righe contenenti le direttive del file in questione) sono suddivisi in campi separati da spazi o caratteri di tabulazione. Il formato può essere semplificato nei due modelli sintattici seguenti,

tenendo conto che esistono comunque altri casi:

```
local base_di_dati utente_dbms autenticazione_utente [mappa]
```

```
host base_di_dati utente_dbms indirizzo_ip maschera_degli_indirizzi ↩
↪ autenticazione_utente [mappa]
```

Nel primo caso si intendono controllare gli accessi provenienti da programmi clienti avviati nello stesso sistema locale, utilizzando un socket di dominio Unix per il collegamento; nel secondo si fa riferimento ad accessi attraverso la rete (connessioni TCP).

- Il secondo campo del record serve a indicare il nome di una base di dati per la quale autorizzare l'accesso; in alternativa si può usare la parola chiave '**all**', in modo da specificare tutte le basi di dati in una sola volta.
- Il terzo campo del record serve a indicare il nome dell'utente del DBMS da autorizzare; in alternativa si può usare la parola chiave '**all**', in modo da rendere indifferente chi sia l'utente.
- I campi ***indirizzo_ip*** e ***maschera_degli_indirizzi*** rappresentano un gruppo di indirizzi di nodi che hanno diritto di accedere a quella base di dati determinata.
- Il campo ***autenticazione_utente*** rappresenta il tipo di autenticazione attraverso una parola chiave. Le più comuni sono elencate nella tabella 75.9.
- L'ultimo campo dipende dal penultimo. Nel caso di autenticazione '**ident**', si utilizza quasi sempre la parola chiave '**sameuser**'

per indicare a PostgreSQL che i nomi usati dagli utenti nei sistemi remoti da cui possono accedere, coincidono con quelli predisposti per la gestione del DBMS. Nel caso di autenticazione '**`password`**', l'ultimo campo potrebbe rappresentare il nome del file di testo contenente le parole d'ordine.

Tabella 75.9. Parole chiave che possono essere usate nel campo ***autenticazione_utente***.

Autenticazione	Descrizione
`trust`	L'autenticazione non ha luogo e si accetta il nome fornito dall'utente senza alcuna verifica.
`reject`	La connessione viene rifiutata in ogni caso.
`password`	Viene richiesta una parola d'ordine riferita all'utente del DBMS.

`md5`
 	
`crypt`
	Viene richiesta una parola d'ordine riferita all'utente del DBMS, che però viene trasmessa in modo cifrato. Le due parole chiave si riferiscono a sistemi differenti; si osservi che, di solito, solo uno dei due sistemi può essere utilizzato, perché dipende dal modo in cui sono memorizzate le parole d'ordine. Pertanto, se uno dei due non funziona, si può tentare con l'altro (dopo aver verificato che comunque l'accesso con le parole d'ordine in chiaro funziona regolarmente).

Autenticazione	Descrizione
`ident sameuser` `ident` ***mappa***	L'autenticazione avviene attraverso il sistema operativo locale, oppure con il protocollo IDENT per gli accessi remoti (sezione 43.3). Si usa questa modalità di riconoscimento, prevalentemente per gli accessi locali, ma in tal caso si mette quasi sicuramente anche l'opzione '**`sameuser`**', per fare riferimento allo stesso utente del sistema operativo. Se non si utilizza la parola chiave '**`sameuser`**', al suo posto va messo il nome di una «mappa», da definire in un altro file.
`pam` [***servizio***]	L'autenticazione avviene attraverso il sistema PAM (*Pluggable authentication modules*) del sistema operativo. Se non viene indicato il servizio PAM, si intende '**`postgresql`**'.

Segue la descrizione di alcuni esempi.

- ```
  local   all     all                                     trust
  ```
  Concede a tutti gli utenti di accedere localmente (tramite un socket di dominio Unix), a qualunque base di dati, senza bisogno di alcun riconoscimento (si accetta il nome e basta).

- ```
  host    all     all     127.0.0.1   255.255.255.255  trust
  ```
  Concede a tutti gli utenti di accedere localmente, ma tramite un socket di dominio Internet (l'indirizzo 127.0.0.1 e normalmente quello di ogni nodo, dal punto di vista locale), senza bisogno di alcun riconoscimento.

- ```
  local   all     all                                     ident sameuser
  ```

  Concede a tutti gli utenti di accedere localmente (tramite un socket di dominio Unix), a qualunque base di dati, sulla base del riconoscimento fatto dal sistema operativo (si intende che ci si affida ai privilegi che ha ottenuto il programma usato per accedere).

- ```
  host  all  all  127.0.0.1  255.255.255.255  ident sameuser
  ```

  Concede a tutti gli utenti di accedere localmente, ma tramite un socket di dominio Internet, sulla base del riconoscimento ottenuto tramite l'uso del protocollo di rete IDENT. Questo metodo può essere usato in alternativa a quello dell'esempio precedente, se per qualche ragione il riconoscimento locale (senza rete), non dovesse funzionare.

- ```
  host   gazie   pippo  192.168.0.0   255.255.0.0   password
  ```

  Concede all'utente '**`pippo`**' di accedere alla base di dati '**`gazie`**', da un nodo qualunque tra quelli che hanno indirizzi del tipo 192.168.∗.∗, attraverso l'indicazione di una parola d'ordine, che viene trasmessa in chiaro.

L'esempio seguente rappresenta una configurazione che potrebbe essere considerata «ragionevole», per poter utilizzare l'utente '**`postgres`**', localmente, senza bisogno di fornire una parola d'ordine (come richiesto dagli esempi mostrati in questo capitolo), consentendo agli altri utenti di accedere da una rete locale qualunque (lo si determina in base al fatto che si fa riferimento a indirizzi IPv4 privati), ma in tal caso si richiede un riconoscimento basato su una parola d'ordine:
  ```

```
#
# TYPE DATABASE USER   IP-ADDRESS   IP-MASK      METHOD
#
local  all      postgres                        ident sameuser
#
local  all      all                             password
host   all      all    127.0.0.1    255.0.0.0    password
host   all      all    192.168.0.0  255.255.0.0  password
host   all      all    172.16.0.0   255.240.0.0  password
host   all      all    10.0.0.0     255.0.0.0    password
#
host   all      all    0.0.0.0      0.0.0.0      reject
```

75.2 Gestione del DBMS

«

La gestione di un DBMS richiede di occuparsi di utenze e di basi di dati. PostgreSQL mette a disposizione degli script per facilitare la loro creazione ed eliminazione, ma in generale è meglio avvalersi di istruzioni SQL, anche se non sono standard.

Per poter impartire comandi in forma di istruzioni SQL, occorre collegarsi al DBMS attraverso un programma appropriato (di solito '**psql**'); per poter maneggiare gli utenti e le basi di dati è necessario disporre dei privilegi necessari. Generalmente, le prime volte si compiono queste operazioni in qualità di amministratore, pertanto con l'utenza '**postgres**'.

Per accedere al DBMS, occorre indicare una basi di dati, anche se le funzioni in questione non interagiscono direttamente con questa. Di solito, dato che inizialmente non è disponibile altro, ci si collega alla basi di dati '**template1**'.

Teoricamente, PostgreSQL non distingue tra lettere maiuscole e minuscole quando si tratta di nominare le basi di dati, le relazioni (le tabelle o gli oggetti a seconda della definizione che si preferisce utilizzare) e gli elementi che compongono delle relazioni. Tuttavia, in certi casi si verificano degli errori inspiegabili dovuti alla scelta dei nomi che in generale conviene indicare sempre solo con lettere minuscole.

75.2.1 Accesso a una base di dati

L'accesso a una base di dati avviene attraverso un cliente, ovvero un programma frontale, o *front-end*, secondo la documentazione di PostgreSQL. Questo programma si avvale generalmente della libreria LibPQ. PostgreSQL fornisce un programma cliente standard, '**`psql`**', che si comporta come una sorta di shell tra l'utente e la base di dati stessa.

Il programma cliente tipico, dovrebbe riconoscere le variabili di ambiente '**`PGHOST`**' e '**`PGPORT`**'. La prima serve a stabilire l'indirizzo o il nome a dominio del servente, indicando implicitamente che la connessione avviene attraverso una connessione TCP e non con un socket di dominio Unix; la seconda specifica il numero della porta, ammesso che si voglia utilizzare un numero diverso da 5432. L'uso di queste variabili non è indispensabile, ma serve solo per non dover specificare queste informazioni attraverso opzioni della riga di comando.

Il programma '**`psql`**' permette un utilizzo interattivo attraverso una serie di comandi impartiti dall'utente su una riga di comando; op-

pure può essere avviato in modo da eseguire il contenuto di un file o di un solo comando fornito tra gli argomenti. Per quanto riguarda l'utilizzo interattivo, il modo più semplice per avviarlo è quello che si vede nell'esempio seguente, dove si indica semplicemente il nome della base di dati sulla quale intervenire.

```
$ psql mio_db[Invio]

  Welcome to the POSTGRESQL interactive sql monitor:
    Please read the file COPYRIGHT for copyright terms of
    POSTGRESQL

     type \? for help on slash commands
     type \q to quit
     type \g or terminate with semicolon to execute query
   You are currently connected to the database: mio_db

  mio_db=>_
```

Da questo momento si possono inserire le istruzioni SQL per la base di dati selezionata, in questo caso '**mio_db**', oppure si possono inserire dei comandi specifici di '**psql**'. Questi ultimi si notano perché sono composti da una barra obliqua inversa ('****'), seguita da un carattere.

Il comando interno di '**psql**' più importante è '**\h**' che permette di visualizzare una guida rapida alle istruzioni SQL che possono essere utilizzate.

```
=> \h[Invio]
```

```
type \h <cmd> where <cmd> is one of the following:
    abort                   abort transaction       alter table
    begin                   begin transaction       begin work
    cluster                 close                   commit
…
type \h * for a complete description of all commands
```

Nello stesso modo, il comando '**\?**' fornisce un riepilogo dei comandi interni di '**psql**'.

=> **\?** [*Invio*]

```
  \?            -- help
  \a            -- toggle field-alignment (currenty on)
  \C [<captn>] -- set html3 caption (currently '')
…
```

Tutto ciò che '**psql**' non riesce a interpretare come un suo comando interno viene trattato come un'istruzione SQL. Dal momento che queste istruzioni possono richiedere più righe, è necessario informare '**psql**' della conclusione di queste, per permettergli di analizzarle e inviarle al servente. Queste istruzioni possono essere terminate con un punto e virgola ('**;**'), oppure con il comando '**\g**'.

Si può osservare, utilizzando '**psql**', che l'invito mostrato cambia leggermente a seconda del contesto: inizialmente appare nella forma '**=>**', mentre quando è in corso l'inserimento di un'istruzione SQL non ancora terminata si trasforma in '**->**'. Il comando '**\g**' viene usato prevalentemente in questa situazione.

-> **\g** [*Invio*]

Le istruzioni SQL possono anche essere raccolte in un file di testo normale. In tal caso si può utilizzare il comando '**\i**' per fare

in modo che '**psql**' interpreti il suo contenuto, come nell'esempio seguente, dove il file in questione è '`mio_file.sql`'.

=> **\i mio_file.sql** [*Invio*]

Nel momento in cui si utilizza questa possibilità (quella di scrivere le istruzioni SQL in un file facendo in modo che poi questo venga letto e interpretato), diventa utile il poter annotare dei commenti. Questi sono iniziati da una sequenza di due trattini ('**--**'), come prescrive lo standard, e tutto quello che vi appare dopo viene ignorato.

La conclusione del funzionamento di '**psql**' si ottiene con il comando '**\q**'.

=> **\q** [*Invio*]

Per l'avvio di '**psql**' si può usare la sintassi seguente. L'opzione '**-f**' consente di indicare un file contenente istruzioni SQL da eseguire subito; in alternativa, un file di questo tipo può essere fornito attraverso lo standard input.

```
psql [opzioni] [base_di_dati]
```

```
psql -f file_di_istruzioni [altre_opzioni] [base_di_dati]
```

```
cat file_di_istruzioni | psql [opzioni] [base_di_dati]
```

Il programma '**psql**' può funzionare solo in abbinamento a una base di dati determinata. In questo senso, se non viene indicato il nome di una base di dati nella riga di comando, '**psql**' tenta di uti-

lizzarne una con lo stesso nome dell'utente. Per la precisione, si fa riferimento alla variabile di ambiente '**USER**'.

Questo dettaglio dovrebbe permettere di comprendere il significato della segnalazione di errore che si ottiene se si tenta di avviare '**psql**' senza indicare una base di dati, quando non ne esiste una con lo stesso nome dell'utente.

Tabella 75.19. Alcune opzioni per l'avvio di '**psql**'.

Opzione	Descrizione
`-c` ***istruzione_sql*** `--command` ***istruzione_sql***	Permette di fornire un'istruzione SQL già nella riga di comando, ottenendone il risultato attraverso lo standard output e facendo terminare subito dopo l'esecuzione di '**psql**'. Questa opzione viene usata particolarmente in abbinamento a '**-q**'.
`-d` ***base_di_dati*** `--dbname` ***base_di_dati***	Permette di indicare il nome della base di dati da utilizzare. Può essere utile quando per qualche motivo potrebbe essere ambigua l'indicazione del suo nome come ultimo argomento.
`-f` ***file_di_istruzioni*** `--file` ***file_di_istruzioni***	Permette di fornire a '**psql**' un file da interpretare contenente le istruzioni SQL (oltre agli eventuali comandi specifici di '**psql**'), senza avviare così una sessione di lavoro interattiva.

Opzione	Descrizione
`-h` ***nodo*** `--host` ***nodo***	Permette di specificare il nodo a cui connettersi per l'interrogazione del servente PostgreSQL.
`-H` `--html`	Fa in modo che l'emissione in forma tabellare avvenga utilizzando il formato HTML. In pratica, ciò è utile per costruire un risultato da leggere attraverso un navigatore ipertestuale.
`-o` ***file_output*** `--output` ***file_output***	Fa in modo che tutto l'output venga inviato nel file specificato dall'argomento.
`-p` ***porta*** `--port` ***porta***	Nel caso in cui '**`postmaster`**' sia in ascolto su una porta TCP diversa dal numero 5432 (corrispondente al valore predefinito), si può specificare con questa opzione il numero corretto da utilizzare.
`-q` `--quiet`	Fa sì che '**`psql`**' funzioni in modo «silenzioso», limitandosi all'emissione pura e semplice di quanto generato dalle istruzioni impartite. Questa opzione è utile quando si utilizza '**`psql`**' all'interno di script che devono occuparsi di rielaborare il risultato ottenuto.
`-t` `--tuple-only`	Disattiva l'emissione dei nomi degli attributi. Questa opzione viene utilizzata particolarmente in abbinamento con '**`-c`**' o '**`-q`**'.

Opzione	Descrizione
`-T` ***opzioni_tabelle_html*** `--table-attr` ***opzioni_tabelle_html***	Questa opzione viene utilizzata in abbinamento con '**`-H`**', per definire le opzioni HTML delle tabelle che si generano. In pratica, si tratta di ciò che può essere inserito all'interno del marcatore di apertura della tabella: '**`<TABLE ...>`**'.
`-U` ***utente*** `--username` ***utente***	Consente di specificare il nome dell'utente del DBMS.
`-W` `--password`	Forza '**`psql`**' a richiedere una parola d'ordine, in ogni caso.

Tabella 75.20. Alcuni comandi che '**`psql`**' riconosce durante il funzionamento interattivo.

Comando	Descrizione
`\h` [***comando***]	L'opzione '**`\h`**' usata da sola, elenca le istruzioni SQL che possono essere utilizzate. Se viene indicato il nome di una di queste, viene mostrata in breve la sintassi relativa.
`\?`	Elenca i comandi interni di '**`psql`**', cioè quelli che iniziano con una barra obliqua inversa ('**`\`**').

Comando	Descrizione
`\l`	Elenca tutte le basi di dati presenti nel servente. Ciò che si ottiene è una tabella contenente rispettivamente: i nomi delle basi di dati, i numeri di identificazione dei rispettivi amministratori (gli utenti che li hanno creati) e il nome della directory in cui sono collocati fisicamente.
`\connect` ***base_di_dati*** ↩ ↪ [***nome_utente***]	Chiude la connessione con la base di dati in uso precedentemente e tenta di accedere a quella indicata. Se il sistema di autenticazione lo consente, si può specificare anche il nome dell'utente con cui si intende operare sulla nuova base di dati. Generalmente, ciò dovrebbe essere impedito. Se si utilizza un'autenticazione basata sul file '`pg_hba.conf`', l'autenticazione di tipo '**`trust`**' consente questo cambiamento di identificazione, altrimenti, il tipo '**`ident`**' lo impedisce.
`\d` [***relazione***]	L'opzione '**`\d`**' usata da sola, elenca le relazioni contenute nella base di dati, altrimenti, se viene indicato il nome di una di queste relazioni, si ottiene l'elenco degli attributi. Se si utilizza il comando '**`\d *`**', si ottiene l'elenco di tutte le relazioni con le informazioni su tutti gli attributi rispettivi.
`\i` ***file***	Con questa opzione si fa in modo che '**`psql`**' esegua di seguito tutte le istruzioni contenute nel file indicato come argomento.

Comando	Descrizione
`\q`	Termina il funzionamento di '**psql**'.

Segue la descrizione di alcuni esempi.

- `$` **psql mio_db** [*Invio*]

  Cerca di connettersi con la base di dati '**mio_db**' nel nodo locale, riferendosi alla stessa utenza riconosciuta dal sistema operativo, utilizzando il meccanismo del socket di dominio Unix.

- `$` **psql -d mio_db** [*Invio*]

  Esattamente come nell'esempio precedente, con l'uso dell'opzione '**-d**' che serve a evitare ambiguità sul fatto che '**mio_db**' sia il nome della base di dati.

- `$` **psql -U tizio -d mio_db** [*Invio*]

  Come nell'esempio precedente, ma specificando che si intende accedere in qualità di utente '**tizio**'.

- `$` **psql -U tizio -W -d mio_db** [*Invio*]

  Come nell'esempio precedente, ma forzando in ogni caso la richiesta di inserimento di una parola d'ordine.

- `$` **psql -U tizio -W -h dinkel.brot.dg -d mio_db** [*Invio*]

  Come nell'esempio precedente, ma questa volta l'accesso viene fatto a una base di dati con lo stesso nome presso il nodo *dinkel.brot.dg*.

- `$` **psql -U tizio -W -f istruzioni.sql -d mio_db** [*Invio*]

Cerca di connettersi con la base di dati '**mio_db**' nel nodo locale, utilizzando il meccanismo del socket di dominio Unix, quindi esegue le istruzioni contenute nel file '`istruzioni.sql`'.

- `$` **`psql -U tizio -W -d mio_db < istruzioni.sql`**[*Invio*]

  Come nell'esempio precedente, ricevendo il contenuto del file '`istruzioni.sql`' dallo standard input.

75.2.1.1 Variabile di ambiente «PAGER»

Il programma '**psql**' è sensibile alla presenza o meno della variabile di ambiente '**PAGER**'. Se questa esiste e non è vuota, '**psql**' utilizza il programma indicato al suo interno per controllare l'emissione dell'output generato. Per esempio, se contiene '**less**', come si vede nell'esempio seguente che fa riferimento a una shell POSIX o compatibile con quella di Bourne, si fa in modo che l'output troppo lungo venga controllato da Less:

```
PAGER=less
export PAGER
```

Per eliminare l'impostazione di questa variabile, in modo da ritornare allo stato predefinito, basta annullare il contenuto della variabile nel modo seguente:

```
PAGER=
export PAGER
```

75.2.2 Organizzazione degli utenti

La creazione di un utente per il DBMS si ottiene con l'istruzione '**CREATE USER**', che nel modello seguente appare in modo semplificato:

```
CREATE USER nome_utente
    [WITH [PASSWORD 'parola_d'ordine' ]
        [CREATEDB|NOCREATEDB|CREATEUSER|NOCREATEUSER] ]
```

Si comprende intuitivamente il significato delle parole chiave delle opzioni finali, con le quali è possibile concedere o negare i privilegi di creare o eliminare delle basi di dati e di creare o eliminare degli utenti. Si osservi che la parola d'ordine va indicata esattamente tra apici singoli. Se si omettono le opzioni finali, i privilegi relativi vengono negati, come se fossero state specificate implicitamente le parole chiave '**NOCREATEDB**' e '**NOCREATEUSER**'.

L'uso della parola chiave '**CREATEUSER**' nella creazione o nella modifica di un'utenza, concede a questa la facoltà di creare o eliminare delle utenze, senza limitazioni, oltre che di creare ed eliminare delle basi di dati. In altri termini, dà all'utente il ruolo di amministrazione del DBMS, con tutti i poteri necessari.

L'esempio seguente mostra i passaggi per la creazione, presso il DBMS locale, dell'utente '**tizio**' (con una parola d'ordine di esempio) a cui viene concesso di creare delle basi di dati, ma non di gestire delle utenze:

```
# su postgres [Invio]

postgres$ psql template1 [Invio]

template1=# CREATE USER tizio WITH PASSWORD 'segreta' ↩
↪           CREATEDB NOCREATEUSER; [Invio]
```

```
  CREATE USER

template1=# \q[Invio]

postgres$ exit[Invio]
```

Così come è stato creata, le caratteristiche di un'utenza possono essere modificate con l'istruzione '**ALTER USER**', per esempio per modificare la parola d'ordine:

```
ALTER USER nome_utente
    [WITH [PASSWORD 'parola_d'ordine' ]
          [CREATEDB|NOCREATEDB]
          [CREATEUSER|NOCREATEUSER] ]
```

Logicamente, se si tratta di modificare la parola d'ordine, può essere lo stesso utente che esegue questa istruzione; altrimenti, per cambiare i privilegi, è necessario che intervenga un utente che ha maggiori facoltà. Nell'esempio seguente, l'utente '**tizio**' creato in quello precedente, modifica la sua parola d'ordine; si osservi che la scelta della base di dati '**template1**' è puramente casuale:

```
$ psql --username tizio template1[Invio]

Password: digitazione_all'oscuro [Invio]

template1=> ALTER USER tizio ↩
↪                       WITH PASSWORD 'segretissima'; [Invio]

  ALTER USER

template1=> \q[Invio]
```

L'eliminazione di un'utenza avviene con un'istruzione molto semplice, senza opzioni particolari:

```
DROP USER nome_utente
```

L'esempio seguente elimina l'utenza '**tizio**' e l'operazione viene svolta dall'utente '**postgres**':

```
# su postgres[Invio]

postgres$ psql template1[Invio]

template1=# DROP USER tizio;[Invio]

  DROP USER

template1=# \q[Invio]

postgres$ exit[Invio]
```

75.2.3 Creazione ed eliminazione delle basi di dati

La creazione di una base di dati è consentita agli amministratori e agli utenti che hanno ottenuto questo privilegio. La base di dati può essere creata per sé, oppure per farla gestire da un altro utente.

```
CREATE DATABASE nome_base_di_dati
    [[WITH] [OWNER [=] utente_proprietario]
            [ENCODING [=] 'codifica']]
```

Il modello sintattico mostrato omette alcune opzioni, di utilizzo meno frequente. In particolare, sarebbe possibile specificare il modello di riferimento per la creazione della base di dati, ma in modo prede-

finito viene utilizzata la base di dati '**template1**' per crearne una di nuova.

Nell'esempio seguente, l'utente '**tizio**' crea la base di dati '**mia_db**', specificando espressamente la codifica; inizialmente accede facendo riferimento alla base di dati '**template1**':

```
$ psql --username tizio template1[Invio]

Password: digitazione_all'oscuro [Invio]

template1=> CREATE DATABASE mia_db ENCODING 'UNICODE';[Invio]

  CREATE DATABASE

template1=> \q[Invio]
```

L'eliminazione di una base di dati si ottiene con l'uso di un'istruzione molto semplice:

```
DROP DATABASE nome_base_di_dati
```

Questa istruzione può essere usata da un amministratore, oppure dall'utente che ne è proprietario. Nell'esempio seguente, l'utente '**tizio**' elimina la sua base di dati '**mia_db**'; per farlo, accede facendo riferimento a un'altra (la solita '**template1**'):

```
$ psql --username tizio template1[Invio]

Password: digitazione_all'oscuro [Invio]

template1=> DROP DATABASE mia_db;[Invio]

  DROP DATABASE

template1=> \q[Invio]
```

75.2.4 La base di dati amministrativa

PostgreSQL memorizza le informazioni sugli utenti e sulle basi di dati all'interno di una sorta di base di dati amministrativa, senza nome. Alle relazioni di questa base di dati trasparente, si accede da qualunque posizione; in pratica, le relazioni sono accessibili quando si apre la base di dati '**template1**', o qualunque altra, ma ovviamente, solo l'amministratore del DBMS ha la facoltà di modificarle direttamente.

Come conseguenza del fatto che le relazioni della base di dati amministrativa sono accessibili da qualunque posizione, si comprende che i nomi di queste relazioni non si possono utilizzare per la costruzione di nuove.

La documentazione originale di PostgreSQL individua queste relazioni, definendole «cataloghi». In questo documento, si preferisce indicarle come relazioni o tabelle della base di dati amministrativa.

Le relazioni più importanti della base di dati amministrativa sono '**pg_user**', '**pg_shadow**' e '**pg_database**'. Vale la pena di osservare il loro contenuto.

```
postgres:~$ psql -d template1 [Invio]
```

La relazione '**pg_user**' è in realtà una vista del catalogo '**pg_shadow**', che contiene le informazioni sugli utenti di PostgreSQL. La figura 75.28 mostra un esempio di come potrebbe essere composta. La consultazione della relazione si ottiene con l'istruzione SQL seguente:

```
template1=> SELECT * FROM pg_user; [Invio]
```

Figura 75.28. Esempio del contenuto di '**pg_user**'.

```
 usename  |usesysid|usecreatedb|usesuper|usecatupd| passwd |valuntil|useconfig
----------+--------+-----------+--------+---------+--------+--------+---------
postgres  |       1|t          |t       |t        |********|        |
pgnanouser|     100|t          |f       |f        |********|        |
tizio     |    1001|t          |t       |t        |********|        |
```

Si può osservare che l'utente '**postgres**' ha tutti gli attributi booleani attivi ('**usecreatedb**', '**usesuper**', '**usecatupd**') e questo per permettergli di compiere tutte le operazioni all'interno delle basi di dati. In particolare, l'attributo '**usecreatedb**' permette all'utente di creare una base di dati e '**usesuper**' permette di aggiungere utenti. In effetti, osservando l'esempio della figura, l'utente '**tizio**' ha praticamente gli stessi privilegi dell'amministratore '**postgres**'.

La relazione '**pg_shadow**' è il contenitore delle informazioni sugli utenti, a cui si accede normalmente tramite la vista '**pg_user**'. Il suo scopo è quello di conservare in un file più sicuro, in quanto non accessibile agli utenti comuni, i dati delle parole d'ordine degli utenti che intendono usare le forme di autenticazione basate su queste. L'esempio della figura 75.29 mostra gli stessi utenti a cui non viene abbinata alcuna parola d'ordine (probabilmente perché accedono localmente e vengono identificati dal sistema operativo). La consultazione della relazione si ottiene con l'istruzione SQL seguente:

```
template1=> SELECT * FROM pg_shadow; [Invio]
```

Figura 75.29. Esempio del contenuto di '**pg_shadow**'.

```
 usename   | usesysid | usecreatedb | usesuper | usecatupd | passwd | valuntil | useconfig
-----------+----------+-------------+----------+-----------+--------+----------+-----------
 postgres  |        1 | t           | t        | t         |        |          |
 pgnanouser|      100 | t           | f        | f         |        |          |
 tizio     |     1001 | t           | t        | t         |        |          |
```

La relazione ‘**pg_database**’ contiene le informazioni sulle basi di dati esistenti. La figura 75.30 mostra un esempio di come potrebbe essere composta. La consultazione della relazione si ottiene con l’istruzione SQL:

```
template1=> SELECT * FROM pg_database; [Invio]
```

Figura 75.30. Esempio del contenuto di ‘**pg_database**’, diviso in due parti per motivi tipografici.

```
  datname  | datdba | encoding | datistemplate | datallowconn | datlastsysoid |
-----------+--------+----------+---------------+--------------+---------------+--
 nanodb    |    100 |        6 | f             | t            |         17140 |
 template1 |      1 |        6 | t             | t            |         17140 |
 template0 |      1 |        6 | t             | f            |         17140 |

      | datvacuumxid | datfrozenxid | datpath | datconfig |          datacl
     -+--------------+--------------+---------+-----------+--------------------------
      |         8159 |   3221233632 |         |           |
      |         8271 |   3221233744 |         |           | {postgres=C*T*/postgres}
      |          464 |          464 |         |           | {postgres=C*T*/postgres}
```

Il primo attributo rappresenta il nome della base di dati, il secondo riporta il numero di identificazione dell’utente che rappresenta il suo DBA, cioè colui che l’ha creata o che comunque deve amministrarla. Per esempio, si può osservare che la base di dati ‘**nanodb**’ è stata creata dall’utente identificato dal numero 100, che da quanto riportato in ‘**pg_user**’ è ‘**pgnanouser**’.

75.2.5 Manutenzione delle basi di dati

Un problema comune dei DBMS è quello della riorganizzazione periodica dei dati, in modo da semplificare e accelerare le elaborazioni successive. Nei sistemi più semplici si parla a volte di «ricostruzione indici», o di qualcosa del genere. Nel caso di PostgreSQL, si utilizza un comando specifico che è estraneo all’SQL standard: ‘**VACUUM**’.

```
VACUUM [altre_opzioni] [VERBOSE] [nome_relazione]
```

```
VACUUM [altre_opzioni] [VERBOSE] ANALYZE ↩
↪      [nome_relazione [(attributo_1 [, ... attributo_n])]]
```

L'operazione di pulizia si riferisce alla base di dati aperta in quel momento. L'opzione '**VERBOSE**' permette di ottenere i dettagli sull'esecuzione dell'operazione; '**ANALYZE**' serve invece per indicare specificatamente una relazione, o addirittura solo alcuni attributi (le colonne delle tabelle) una relazione e avere informazioni su questi. Eventualmente, sono disponibili altre opzioni per ottenere una riorganizzazione dei dati più importante.

Anche se non si tratta di un comando SQL standard, per PostgreSQL è importante che venga eseguita periodicamente una ripulitura con il comando '**VACUUM**', eventualmente attraverso uno script simile a quello seguente, da avviare per mezzo del sistema Cron:

```
#!/bin/sh
su postgres -c "psql $1 -c 'VACUUM'"
```

In pratica, richiamando questo script con i privilegi dell'utente '**root**', indicando come argomento il nome della base di dati (viene inserito al posto di '**$1**' dalla shell), si ottiene di avviare il comando '**VACUUM**' attraverso '**psql**'.

Per riuscire a fare il lavoro in serie per tutte le basi di dati, si potrebbe scrivere uno script più complesso, come quello seguente. In questo caso, lo script deve essere avviato con i privilegi dell'utente '**postgres**'.

```
#!/bin/sh
#
BASI_DATI=`psql template1 -t -c "SELECT datname from pg_database"`
#
echo "Procedimento di ripulitura e sistemazione delle basi di dati"
echo "di PostgreSQL."
echo "Se l'operazione dovesse essere interrotta accidentalmente,"
echo "potrebbe essere necessaria l'eliminazione del file pg_vlock"
echo "contenuto nella directory della base di dati relativa."
#
for BASE_DATI in $BASI_DATI
do
    printf "$BASE_DATI: "
    psql $BASE_DATI -c "VACUUM"
done
```

In breve, si utilizza la prima volta '**psql**' in modo da aprire la base di dati '**template1**' (quella usata come modello, che si ha la certezza di trovare sempre), accedendo alla relazione '**pg_database**', che fa parte della base di dati amministrativa, per leggere l'attributo contenente i nomi delle basi di dati. In particolare, l'opzione '**-t**' serve a evitare di inserire il nome dell'attributo stesso. L'elenco che si ottiene viene inserito nella variabile di ambiente '**BASI_DATI**', che in seguito viene scandita da un ciclo '**for**', all'interno del quale si utilizza '**psql**' per ripulire ogni singola base di dati.

75.2.6 Copie di sicurezza

Prima di poter pensare a copiare o a spostare una base di dati occorre avere chiaro in mente che si tratta di file «binari» (nel senso che non si tratta di file di testo), contenenti informazioni collegate l'una all'altra in qualche modo più o meno oscuro. Queste informazioni possono a volte essere espresse anche in forma numerica; in tal caso dipende dall'architettura in cui sono state create. Ciò implica due

cose fondamentali: la copia deve essere fatta in modo che non si perdano dei pezzi per la strada; lo spostamento dei dati in forma binaria, in un'altra architettura, non è ammissibile.

La copia di sicurezza binaria, di tutto ciò che serve a PostgreSQL per la gestione delle sue basi di dati, si ottiene semplicemente archiviando quanto contenuto a partire da '`~postgres/`', così come si può comprendere intuitivamente. Ciò che conta è che il ripristino dei dati avvenga nello stesso contesto (architettura, sistema operativo, librerie, versione di PostgreSQL e configurazione).

Per una copia di sicurezza più «sicura», è necessario archiviare i dati in modo indipendente da tutto. Si ottiene questo generando un file di testo, contenente istruzioni SQL con le quali ricostruire poi una sola base di dati o anche tutte assieme. Per questo vengono in aiuto due programmi di PostgreSQL: '**`pg_dump`**' e '**`pg_dumpall`**'.

Non sempre il procedimento di trasferimento dei dati in forma di comandi SQL può essere portato a termine con successo. Può succedere che delle relazioni troppo complesse o con dati troppo grandi, non siano tradotte correttamente nella fase di archiviazione. Questo problema deve essere preso in considerazione già nel momento della progettazione di una base di dati, avendo cura di verificare, sperimentandolo, che il procedimento di scarico e recupero dei dati possa funzionare.

Lo scarico di una sola base di dati si ottiene attraverso il programma '**`pg_dump`**', che, eventualmente, potrebbe risiedere al di fuori dei percorsi normali contenuti nella variabile '**`$PATH`**' e potrebbe trovarsi nella directory '`/usr/lib/postgresql/bin/`':

```
pg_dump [opzioni] base_di_dati
```

Se non si indicano delle opzioni e ci si limita a specificare la base di dati su cui intervenire, si ottiene il risultato attraverso lo standard output, composto in pratica dai comandi necessari a '**psql**' per ricostruire le relazioni che compongono la base di dati (la base di dati stessa deve essere ricreata manualmente). Tanto per chiarire subito il senso della cosa, se si utilizza '**pg_dump**' nel modo seguente, si ottiene il file di testo '`mio_db.dump`':

```
$ pg_dump mio_db > mio_db.dump[Invio]
```

Questo file va verificato, ricercando la presenza eventuale di segnalazioni di errore che vengono generate in presenza di dati che non possono essere riprodotti fedelmente; eventualmente, il file può anche essere modificato se si conosce la sintassi dei comandi che vengono inseriti in questo script. Per fare in modo che le relazioni della base di dati vengano ricreate e caricate, si può utilizzare '**psql**' nel modo seguente:

```
$ psql -e mio_db < mio_db.dump[Invio]
```

Tabella 75.33. Alcune opzioni che possono essere usate con '**pg_dump**'.

Autenticazione	Descrizione

`-d`
 	
`--inserts`
	In condizioni normali, '**pg_dump**' salva i dati delle relazioni (le tabelle secondo l'SQL) in una forma compatibile con il comando '**COPY**', che però non è compatibile con lo standard SQL. Con l'opzione '**-d**', utilizza il comando '**INSERT**' tradizionale.

Autenticazione	Descrizione
`-D` `--column-inserts`	Come con l'opzione '**`-d`**', con l'aggiunta dell'indicazione degli attributi (le colonne secondo l'SQL) in cui vanno inseriti i dati. In pratica, questa opzione permette di generare uno script più preciso e dettagliato.
`-f` ***file*** `--file=`***file***	Permette di definire un file diverso dallo standard output, che si vuole generare con il risultato dell'elaborazione di '**`pg_dump`**'.
`-h` ***nodo*** `--host=`***nodo***	Permette di specificare il nodo a cui connettersi per l'interrogazione del servente PostgreSQL. In pratica, se l'accesso è consentito, è possibile scaricare una base di dati gestita presso un nodo remoto.
`-p` ***porta*** `--port=`***porta***	Nel caso in cui '**`postmaster`**' sia in ascolto su una porta TCP diversa dal numero 5432 (corrispondente al valore predefinito), si può specificare con questa opzione il numero corretto da utilizzare.
`-s` `--schema-only`	Scarica soltanto la struttura delle relazioni, senza occuparsi del loro contenuto. In pratica, serve per poter riprodurre le relazioni vuote.
`-t` ***nome_relazione*** `--table=`***nome_relazione***	Utilizzando questa opzione, indicando il nome di una relazione, si ottiene lo scarico solo di quella.
`-U` ***nome***	Specifica con quale nominativo utente identificarsi per eseguire l'operazione.

Autenticazione	Descrizione
`-W`	Forza la richiesta di inserire una parola d'ordine, che comunque dovrebbe essere chiesta automaticamente se il DBMS la richiede.

Per copiare o trasferire tutte le basi di dati del sistema di PostgreSQL, si può utilizzare '**pg_dumpall**', che, eventualmente, potrebbe risiedere al di fuori dei percorsi normali contenuti nella variabile '**$PATH**' e potrebbe trovarsi nella directory '`/usr/lib/postgresql/bin/`':

[*percorso*]`pg_dumpall` [*opzioni*]

Il programma '**pg_dumpall**' provvede a scaricare tutte le basi di dati, assieme alle informazioni necessarie per ricreare il catalogo '**pg_shadow**' (la vista '**pg_user**' si ottiene di conseguenza). Come si può intuire, si deve utilizzare '**pg_dumpall**' con i privilegi dell'amministratore del DBMS (di solito l'utente '**postgres**').

`postgres$` **pg_dumpall > basi_dati.dump**[*Invio*]

L'esempio mostra il modo più semplice di utilizzare '**pg_dumpall**' per scaricare tutte le basi di dati in un file unico. In questo caso, si ottiene il file di testo '`basi_dati.dump`'. Questo file va verificato alla ricerca di segnalazioni di errore che potrebbero essere generate in presenza di dati che non possono essere riprodotti fedelmente; eventualmente, può essere modificato se si conosce la sintassi dei comandi che vengono inseriti in questo script.

Il recupero dell'insieme completo delle basi di dati avviene normal-

mente in un ambiente PostgreSQL, in cui il sistema delle basi di dati sia stato predisposto, ma non sia stata creata alcuna base di dati (a parte quelle standard, come '**template1**'). Come si può intuire, il comando necessario per ricaricare le basi di dati, assieme alle informazioni sugli utenti (la relazione '**pg_shadow**'), è quello seguente:

```
postgres$ psql -e template1 < basi_dati.dump [Invio]
```

La situazione tipica in cui è necessario utilizzare '**pg_dumpall**' per scaricare tutto il sistema delle basi di dati, è quella del momento in cui ci si accinge ad aggiornare la versione di PostgreSQL. In breve, in quella occasione, si devono eseguire i passaggi seguenti:

1. con la versione vecchia di PostgreSQL, si deve utilizzare '**pg_dumpall**' in modo da scaricare tutto il sistema delle basi di dati in un solo file di testo;
2. si aggiorna PostgreSQL;
3. si elimina il contenuto della directory '`~postgres/data/`', ovvero quella che altrimenti viene definita '**PGDATA**' (prima conviene forse fare una copia di sicurezza del suo contenuto, tale e quale, in forma binaria);
4. si ricrea il sistema delle basi di dati, vuoto, attraverso '**initdb**';
5. si ricaricano le basi di dati precedenti, assieme alle informazioni sugli utenti, attraverso '**psql**', utilizzando il file generato in precedenza attraverso '**pg_dumpall**'.

Quello che manca, eventualmente, è la configurazione di PostgreSQL, in particolare per ciò che riguarda i sistemi di accesso e au-

tenticazione (il file '`~postgres/data/pg_hda.conf`'), che deve essere ripristinata manualmente.

75.2.7 Importazione ed esportazione dei dati

Al posto di utilizzare gli script già pronti per la copia e il recupero dei dati, è possibile avvalersi di comandi SQL. PostgreSQL fornisce un'istruzione speciale per permettere l'importazione e l'esportazione dei dati da e verso un file indipendente dalla piattaforma. Si tratta dell'istruzione '**`COPY`**', la cui sintassi semplificata è quella seguente:

```
COPY relazione TO { 'file' | STDIN }
    [ [WITH]
        [BINARY]
        [DELIMITER [AS] 'delimitatore' ] ]
```

```
COPY relazione FROM { 'file' | STDIN }
    [ [WITH]
        [BINARY]
        [DELIMITER [AS] 'delimitatore' ] ]
```

Nella prima delle due forme, si esportano i dati verso un file o verso lo standard input; nella seconda si importano da un file o dallo standard output.

Se si usa l'opzione '**`BINARY`**' si ottiene un file «binario» indipendente dalla piattaforma; diversamente si ottiene un file di testo tradizionale. Nel caso del file di testo, ogni riga corrisponde a una tupla della relazione; gli attributi sono separati da un carattere di delimitazione, che in mancanza della definizione tramite l'opzione '**`DELIMITER`**

AS’ è un carattere di tabulazione. In ogni caso, anche se si specifica tale opzione, può trattarsi solo di un carattere. In pratica, sempre nell’ipotesi di creazione di un file di testo, ogni riga è organizzata secondo lo schema seguente:

attributo_1 *x* ***attributo_2*** *x* ··· *x* ***attributo_n***

Nello schema, ***x*** rappresenta il carattere di delimitazione, che, come si può vedere, non viene inserito all’inizio e alla fine.

Quando l’istruzione ‘**COPY**’ viene usata per importare dati dallo standard input, in formato testo, è necessario che dopo l’ultima riga che contiene attributi da inserire nella relazione, sia presente una sequenza di escape speciale: una barra obliqua inversa seguita da un punto (‘**\.**’). Il file ottenuto quando si esporta verso lo standard output contiene questo simbolo di conclusione.

Il file di testo in questione può contenere anche altre sequenze di escape, che si trovano descritte nella tabella 75.34.

Tabella 75.34. Sequenze di escape nei file di testo generati e utilizzati da ‘**COPY**’.

Escape	Descrizione
\\	Una barra obliqua inversa.
\.	Simbolo di conclusione del file.
\N	‘**NULL**’.
delimitatore	Protegge il simbolo che viene già utilizzato come delimitatore.

Escape	Descrizione
`\`<*LF*>	Tratta <*LF*> in modo letterale.
`\b`	<*BS*>.
`\f`	<*FF*>.
`\n`	<*LF*>.
`\r`	<*CR*>.
`\t`	<*HT*> (tabulazione orizzontale).
`\v`	<*VT*> (tabulazione verticale).
`\`***ooo***	Codice per un byte espresso in ottale.

È importante fare mente locale al fatto che l'istruzione viene eseguita dal servente. Ciò significa che i file, quando non si tratta di standard input o di standard output, sono creati o cercati secondo il file system che questo servente si trova ad avere sotto di sé.

Segue la descrizione di alcuni esempi.

- ```
  COPY Indirizzi TO STDOUT;
  ```

  L'esempio mostra l'istruzione necessaria a emettere attraverso lo standard output del programma cliente ('**psql**') la trasformazione in testo del contenuto della relazione '**Indirizzi**'.
  ```

- `COPY Indirizzi TO STDOUT BINARY;`

  Come nell'esempio precedente, generando però un formato binario, indipendente dalla piattaforma.

- `COPY Indirizzi TO '/tmp/prova' WITH DELIMITER AS '|';`

  In questo caso, si genera il file di testo '`/tmp/prova`' nel file system dell'elaboratore servente, inoltre gli attributi sono separati attraverso una barra verticale ('`|`').

- `COPY Indirizzi FROM STDIN;`

  In questo caso, si aggiungono tuple alla relazione '**`Indirizzi`**', utilizzando quanto proviene dallo standard input, che si attende essere un file di testo (alla fine deve apparire la sequenza di escape '`\.`').

- `COPY Indirizzi FROM STDIN BINARY;`

  Come nell'esempio precedente, attendendo i dati in formato binario.

- `COPY Indirizzi FROM '/tmp/prova' WITH DELIMITER AS '|';`

  Si aggiungono tuple alla relazione '**`Indirizzi`**', utilizzando quanto proviene dal file '`/tmp/prova`', che si trova nel file system dell'elaboratore servente. Il file deve essere in formato testo e gli attributi si intendono separati da una barra verticale ('`|`').

75.3 Il linguaggio

«

PostgreSQL è un ORDBMS, ovvero un *Object-relational DBMS*, cioè un DBMS relazionale a oggetti. La sua documentazione utilizza terminologie differenti, a seconda delle preferenze dei rispettivi autori. In generale si possono distinguere tre modalità, riferite

a tre punti di vista: la programmazione a oggetti, la teoria generale sui DBMS e il linguaggio SQL. Le equivalenze dei termini sono riassunte dallo schema seguente:

classi	istanze	attributi	tipi di dati contenibili negli attributi
relazioni	tuple	attributi	domini
tabelle	righe	colonne	tipi di dati contenibili nelle colonne

In questo capitolo si intende usare la terminologia tradizionale dei DBMS, dove i dati sono organizzati in relazioni, tuple e attributi, affiancando eventualmente i termini del linguaggio SQL tradizionale (tabelle, righe e colonne). Inoltre, la sintassi delle istruzioni (interrogazioni) SQL che vengono mostrate è limitata alle funzionalità più semplici, sempre compatibilmente con le possibilità di PostgreSQL. Per una visione più estesa delle funzionalità SQL di PostgreSQL conviene consultare la sua documentazione.

75.3.1 Prima di iniziare

Per fare pratica con il linguaggio SQL, il modo migliore è quello di utilizzare il programma '**psql**' con il quale si possono eseguire interrogazioni interattive con il servente. Quello che conta è tenere a mente che per poterlo utilizzare occorre avere già creato una base di dati (vuota), in cui vanno poi inserite delle nuove relazioni, con le quali si possono eseguire altre operazioni.

Attraverso le istruzioni SQL si fa riferimento sempre a un'unica base di dati: quella a cui ci si collega quando si avvia '**psql**'.

Utilizzando '**psql**', le istruzioni devono essere terminate con il punto e virgola ('**;**'), oppure dal comando interno '**\g**' (*go*).

75.3.2 Tipi di dati e rappresentazione

«

I tipi di dati gestibili sono un punto delicato della compatibilità tra un DBMS e lo standard SQL. Vale la pena di riepilogare i tipi più comuni, compatibili con lo standard SQL, che possono essere trovati nella tabella 75.42.

Tabella 75.42. Elenco dei tipi di dati standard utilizzabili con PostgreSQL.

Tipo	Standard	Descrizione
`CHAR` `CHARACTER`	SQL92	Un carattere singolo.
`CHAR(`***n***`)` `CHARACTER(`***n***`)`	SQL92	Una stringa di lunghezza fissa, di ***n*** caratteri, completata da spazi.

`VARCHAR(`***n***`)`
`CHARACTER VARYING(`***n***`)`
`CHAR VARYING(`***n***`)`
	SQL92	Una stringa di lunghezza variabile con un massimo di ***n*** caratteri.

`INTEGER`	SQL92	Intero (al massimo nove cifre numeriche).
`SMALLINT`	SQL92	Intero più piccolo di '**`INTEGER`**' (al massimo quattro cifre numeriche).
`FLOAT`	SQL92	Numero a virgola mobile.
`FLOAT(`***n***`)`	SQL92	Numero a virgola mobile lungo ***n*** bit.

Tipo	Standard	Descrizione
`REAL`	SQL92	Numero a virgola mobile (teoricamente più preciso di '**`FLOAT`**').
`DOUBLE PRECISION`	SQL92	Numero a virgola mobile (più o meno equivalente a '**`REAL`**').

`NUMERIC`
`NUMERIC` (***precisione*** [, ***scala***])
`DECIMAL`
`DECIMAL` (***precisione*** [, ***scala***])
`DEC`
`DEC(`***precisione*** [, ***scala***]`)`
	SQL92	Numero composto da un massimo di tante cifre numeriche quante indicate dalla precisione, cioè il primo argomento tra parentesi. Se viene specificata anche la scala, si intende riservare quella parte di cifre per quanto appare dopo la virgola.

`DATE`	SQL92	Data, di solito nella forma '***mm/gg/aaaa***'.
`TIME`	SQL92	Orario, nella forma '***hh:mm:ss***', oppure solo '***hh:mm***'.
`TIMESTAMP`	SQL92	Informazione completa data-orario.
`INTERVAL`	SQL92	Intervallo di tempo.
`BIT(`***n***`)`	SQL92	Stringa binaria di dimensione fissa.
`BIT VARYING(`***n***`)`	SQL92	Stringa binaria di dimensione variabile.

Tipo	Standard	Descrizione
`BOOLEAN`	SQL99	Valore logico booleano.

Oltre ai tipi di dati gestibili, è necessario conoscere il modo di rappresentarli in forma costante. In particolare, è bene osservare che PostgreSQL ammette solo l'uso degli apici singoli come delimitatori; pertanto, per rappresentare un apice in una stringa delimitata in questo modo, lo si può raddoppiare, oppure si può usare la sequenza di escape '`\'`'. La tabella 75.43 mostra alcuni esempi.

Tabella 75.43. Esempi di rappresentazione dei valori costanti. Si osservi che in alcuni casi, conviene dichiarare il tipo di valore, seguito da una costante stringa che lo rappresenta, come in questi esempi a proposito di valori data-orario.

Tipo di valore in generale	Esempi di rappresentazione in forma di costante letterale

`CHAR`
`CHARACTER`
`CHAR(`***n***`)`
`CHARACTER(`***n***`)`
`VARCHAR(`***n***`)`
`CHARACTER VARYING(`***n***`)`
`CHAR VARYING(`***n***`)`
	`'a'`
	`'ciao'`
	`'Ciao'`
	`'123/der:876'`

Tipo di valore in generale	Esempi di rappresentazione in forma di costante letterale
INTEGER SMALLINT	1 123 -987
FLOAT FLOAT(*n*) REAL DOUBLE PRECISION NUMERIC NUMERIC(*precisione* [, *scala*]) DECIMAL DECIMAL(*precisione* [, *scala*]) DEC DEC(*precisione* [, *scala*])	123.45 -45.3 123.45e+10 123.45e-10

Tipo di valore in generale	Esempi di rappresentazione in forma di costante letterale
DATE	DATE '31.12.2012' DATE '12/31/2012' DATE '2012-12-31'
TIME	TIME '15:55:27' TIME '15:59'
TIMESTAMP	TIMESTAMP '2012-12-31 15:55:27' TIMESTAMP '2012-12-31 15:55:27+1'
INTERVAL	INTERVAL '15:55:27' INTERVAL '15 HOUR 59 MINUTE' INTERVAL '- 15 HOUR'
BIT BIT VARYING(*n*)	B'1' B'101' X'2F'

Tipo di valore in generale	Esempi di rappresentazione in forma di costante letterale
BOOLEAN	1 'y' 'yes' 't' 'true' 0 'n' 'no' 'f' 'false'

In particolare, le costanti stringa possono contenere delle sequenze di escape, rappresentate da una barra obliqua inversa seguita da un simbolo. La tabella 75.44 mostra le sequenze di escape tipiche e inserisce anche il caso del raddoppio degli apici singoli.

Tabella 75.44. Sequenze di escape utilizzabili all'interno delle stringhe di caratteri costanti.

Escape	Significato
\n	*<LF>*

Escape	Significato
`\r`	*<CR>*
`\b`	*<BS>*
`\'`	'
`''`	'
`\"`	"
`\\`	\
`\%`	*%*
`\_`	_

75.3.3 Funzioni

PostgreSQL, come altri DBMS SQL, offre una serie di funzioni che fanno parte dello standard SQL, assieme ad altre non standard che però sono ampiamente diffuse e di grande utilità. Le tabelle 75.45 e 75.46 ne riportano alcune.

Tabella 75.45. Funzioni SQL riconosciute da PostgreSQL.

Funzione	Descrizione
`POSITION(`***stringa_1*** `IN` ***stringa_2***`)`	Posizione di ***stringa_1*** in ***stringa_2***.
`SUBSTRING(`***stringa*** `[FROM` ***n***`] [FOR` ***m***`])`	Sottostringa da ***n*** per ***m*** caratteri.

Funzione	Descrizione
`TRIM([LEADING|TRAILING|BOTH] ↩ ↪ ['x'] FROM [stringa])`	Ripulisce all'inizio e alla fine del testo.

Tabella 75.46. Alcune funzioni riconosciute dal linguaggio di PostgreSQL.

Funzione	Descrizione
`UPPER(`***stringa***`)`	Converte la stringa in caratteri maiuscoli.
`LOWER(`***stringa***`)`	Converte la stringa in caratteri minuscoli.
`INITCAP(`***stringa***`)`	Converte la stringa in modo che le parole inizino con la maiuscola.
`SUBSTR(`***stringa***`,`***n***`,`***m***`)`	Estrae la stringa che inizia dalla posizione ***n***, lunga ***m*** caratteri.
`LTRIM(`***stringa***`, 'x')`	Ripulisce la stringa a sinistra (*Left trim*).
`RTRIM(`***stringa***`, 'x')`	Ripulisce la stringa a destra (*Right trim*).

Segue la descrizione di alcuni esempi.

- ```
  SELECT POSITION( 'o' IN 'Topo' )
  ```
  Restituisce il valore due.

- ```
  SELECT POSITION( 'ino' IN Cognome ) FROM Indirizzi
  ```
  Restituisce un elenco delle posizioni in cui si trova la stringa '**ino**' all'interno dell'attributo '**Cognome**', per tutte le tuple della relazione '**Indirizzi**'.

- ```
  SELECT SUBSTRING( 'Daniele' FROM 3 FOR 2 )
  ```
  ```

Restituisce la stringa '**ni**'.

- ```
  SELECT TRIM( LEADING '*' FROM '*****Ciao****' )
  ```

  Restituisce la stringa '**Ciao****'.

- ```
  SELECT TRIM( TRAILING '*' FROM '*****Ciao****' )
  ```

  Restituisce la stringa '*****Ciao**'.

- ```
  SELECT TRIM( BOTH '*' FROM '*****Ciao****' )
  ```

  Restituisce la stringa '**Ciao**'.

- ```
  SELECT TRIM( BOTH ' ' FROM '    Ciao    ' )
  ```

  Restituisce la stringa '**Ciao**'.

- ```
  SELECT TRIM( '    Ciao    ' )
  ```

  Esattamente come nell'esempio precedente, dal momento che lo spazio normale è il carattere predefinito e che la parola chiave '**BOTH**' è anche predefinita.

- ```
  SELECT LTRIM( '*****Ciao****', '*' )
  ```

  Restituisce la stringa '**Ciao****'.

- ```
  SELECT RTRIM( '*****Ciao****', '*' )
  ```

  Restituisce la stringa '*****Ciao**'.

75.3.4 Esempi comuni

«

Nelle sezioni seguenti vengono mostrati alcuni esempi comuni di utilizzo del linguaggio SQL, limitato alle possibilità di PostgreSQL. La sintassi non viene descritta, salvo quando la differenza tra quella standard e quella di PostgreSQL è importante.

Negli esempi si fa riferimento frequentemente a una relazione di indirizzi, il cui contenuto è visibile nella figura 75.57.
  ```

Figura 75.57. La relazione '**`Indirizzi (Codice, Cognome, Nome, Indirizzo, Telefono)`**' usata in molti esempi del capitolo.

Indirizzi				
Codice	**Cognome**	**Nome**	**Indirizzo**	**Telefono**
1	Pallino	Pinco	Via Biglie 1	0222,222222
2	Tizi	Tizio	Via Tazi 5	0555,555555
3	Cai	Caio	Via Caini 1	0888,888888
4	Semproni	Sempronio	Via Sempi 7	0999,999999

75.3.4.1 Creazione di una relazione

La relazione di esempio mostrata nella figura 75.57, potrebbe essere creata nel modo seguente:

```
CREATE TABLE Indirizzi (
        Codice          integer,
        Cognome         char(40),
        Nome            char(40),
        Indirizzo       varchar(60),
        Telefono        varchar(40)
    );
```

Quando si inseriscono i valori per una tupla, può capitare che venga omesso l'inserimento di alcuni attributi. In questi casi, il campo corrispondente riceve il valore '**`NULL`**', cioè un valore indefinito, oppure il valore predefinito attraverso quanto specificato con l'espressione che segue la parola chiave '**`DEFAULT`**'.

In alcuni casi non è possibile definire un valore predefinito e nem-

meno è accettabile che un dato resti indefinito. In tali situazioni si può aggiungere l'opzione '**NOT NULL**', dopo la definizione del tipo.

75.3.4.2 Modifica della relazione

La modifica di una relazione implica l'intervento sulle caratteristiche degli attributi, oppure la loro aggiunta ed eliminazione. Seguono due esempi, con cui si aggiunge un attributo e poi lo si elimina:

```
ALTER TABLE Indirizzi ADD COLUMN Comune char(30);
```

```
ALTER TABLE Indirizzi DROP COLUMN Comune;
```

L'esempio seguente modifica il tipo di un attributo già esistente:

```
ALTER TABLE Indirizzi ALTER COLUMN Codice TYPE REAL;
```

Naturalmente, la conversione del tipo di un attributo può avere significato solo se i valori contenuti nelle tuple esistenti, in corrispondenza di quell'attributo, sono convertibili.

75.3.4.3 Inserimento dati in una relazione

L'esempio seguente mostra l'inserimento dell'indirizzo dell'impiegato «Pinco Pallino».

```
INSERT INTO Indirizzi
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    );
```

In questo caso, si presuppone che i valori inseriti seguano la sequenza degli attributi, così come è stata creata la relazione in origine.

Se si vuole indicare un comando più leggibile, occorre aggiungere l'indicazione della sequenza degli attributi da compilare, come nell'esempio seguente:

```
INSERT INTO Indirizzi (
        Codice,
        Cognome,
        Nome,
        Indirizzo,
        Telefono
    )
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    );
```

In questo stesso modo, si può evitare di compilare il contenuto di un attributo particolare, indicando espressamente solo gli attributi che si vogliono fornire; in tal caso gli altri attributi ricevono il valore predefinito o '**NULL**' in mancanza d'altro. Nell'esempio seguente viene indicato solo il codice e il nominativo:

```
INSERT INTO Indirizzi (
        Codice,
        Cognome,
        Nome,
    )
    VALUES (
        01,
        'Pallino'
        'Pinco',
    );
```

75.3.4.4 Eliminazione di una relazione

Una relazione può essere eliminata completamente attraverso l'istruzione '**DROP**'. L'esempio seguente elimina la relazione degli indirizzi degli esempi già mostrati:

```
DROP TABLE Indirizzi;
```

75.3.4.5 Interrogazioni semplici

L'esempio seguente emette tutto il contenuto della relazione degli indirizzi già vista negli esempi precedenti:

```
SELECT * FROM Indirizzi;
```

Seguendo l'esempio fatto in precedenza si dovrebbe ottenere l'elenco riportato sotto, equivalente a tutto il contenuto della relazione.

```
codice  cognome   nome       indirizzo     telefono

     1  Pallino   Pinco      Via Biglie 1  0222,222222
     2  Tizi      Tizio      Via Tazi 5    0555,555555
     3  Cai       Caio       Via Caini 1   0888,888888
     4  Semproni  Sempronio  Via Sempi 7   0999,999999
```

Per ottenere un elenco ordinato in base al cognome e al nome (in caso di ambiguità), lo stesso comando si completa nel modo seguente:

```
SELECT * FROM Indirizzi ORDER BY Cognome, Nome;
```

```
codice  cognome   nome       indirizzo     telefono

     3  Cai       Caio       Via Caini 1   0888,888888
     1  Pallino   Pinco      Via Biglie 1  0222,222222
     4  Semproni  Sempronio  Via Sempi 7   0999,999999
     2  Tizi      Tizio      Via Tazi 5    0555,555555
```

La selezione degli attributi permette di ottenere un risultato che contenga solo quelli desiderati, permettendo anche di cambiarne l'intestazione. L'esempio seguente permette di mostrare solo i nominativi e il telefono, cambiando un po' le intestazioni:

```
SELECT Cognome as cognomi, Nome as nomi,
       Telefono as numeri_telefonici
    FROM Indirizzi;
```

Quello che si ottiene è simile all'elenco seguente:

```
cognomi   nomi       numeri_telefonici

Pallino   Pinco      0222,222222
Tizi      Tizio      0555,555555
Cai       Caio       0888,888888
Semproni  Sempronio  0999,999999
```

La selezione delle tuple può essere fatta attraverso la condizione che segue la parola chiave '**WHERE**'. Nell'esempio seguente vengono selezionate le tuple in cui l'iniziale dei cognomi è compresa tra '**N**' e '**T**'.

```
SELECT * FROM Indirizzi
         WHERE Cognome >= 'N' AND Cognome <= 'T';
```

Dall'elenco che si ottiene, si osserva che '**Caio**' è stato escluso:

```
codice  cognome   nome       indirizzo     telefono

     1  Pallino   Pinco      Via Biglie 1  0222,222222
     2  Tizi      Tizio      Via Tazi 5    0555,555555
     4  Semproni  Sempronio  Via Sempi 7   0999,999999
```

Per evitare ambiguità possono essere indicati i nomi degli attributi prefissati dal nome della relazione a cui appartengono, separando le due parti con l'operatore punto ('.'). Nell'esempio seguente si selezionano solo il cognome, il nome e il numero telefonico, specificando il nome della relazione a cui appartengono gli attributi:

```
SELECT Indirizzi.Cognome, Indirizzi.Nome,
       Indirizzi.Telefono
    FROM Indirizzi;
```

Ecco il risultato:

```
cognome   nome       telefono

Pallino   Pinco      0222,222222
Tizi      Tizio      0555,555555
Cai       Caio       0888,888888
Semproni  Sempronio  0999,999999
```

75.3.4.6 Interrogazioni simultanee di più relazioni

«

Se dopo la parola chiave '**FROM**' si indicano più relazioni (ciò vale anche se si indica più volte la stessa relazione), si intende fare riferimento a una relazione generata dal «prodotto» di queste. Si immagini di abbinare alla relazione '**Indirizzi**' la relazione '**Presenze**' contenente i dati visibili nella figura 75.76.

Figura 75.76. La relazione '**Presenze (Codice, Giorno, Ingresso, Uscita)**'.

Presenze			
Codice	**Giorno**	**Ingresso**	**Uscita**
1	01/01/2012	07:30	13:30
2	01/01/2012	07:35	13:37
3	01/01/2012	07:45	14:00
4	01/01/2012	08:30	16:30
1	01/02/2012	07:35	13:38
2	01/02/2012	08:35	14:37
4	01/02/2012	07:30	13:30

Come si può intendere, il primo attributo, '**Codice**', serve a identificare la persona per la quale è stata fatta l'annotazione dell'ingresso e dell'uscita. Tale codice viene interpretato in base al contenuto della relazione '**Indirizzi**'. Si immagini di volere ottenere un elenco contenente tutti gli ingressi e le uscite, indicando chiaramente il cognome e il nome della persona a cui si riferiscono.

```
SELECT
    Presenze.Giorno,
    Presenze.Ingresso,
    Presenze.Uscita,
    Indirizzi.Cognome,
    Indirizzi.Nome
    FROM Presenze, Indirizzi
    WHERE Presenze.Codice = Indirizzi.Codice;
```

Ecco quello che si dovrebbe ottenere:

```
giorno      ingresso  uscita    cognome   nome

01-01-2012  07:30:00  13:30:00  Pallino   Pinco
01-01-2012  07:35:00  13:37:00  Tizi      Tizio
01-01-2012  07:45:00  14:00:00  Cai       Caio
01-01-2012  08:30:00  16:30:00  Semproni  Sempronio
01-02-2012  07:35:00  13:38:00  Pallino   Pinco
01-02-2012  08:35:00  14:37:00  Tizio     Tizi
01-02-2012  07:40:00  13:30:00  Semproni  Sempronio
```

75.3.4.7 Alias

«

Una stessa relazione può essere presa in considerazione come se si trattasse di due o più relazioni differenti. Per distinguere tra questi punti di vista diversi, si devono usare degli alias, che sono in pratica dei nomi alternativi. Gli alias si possono usare anche solo per questioni di leggibilità. L'esempio seguente è la semplice ripetizione di quello mostrato nella sezione precedente, con l'aggiunta però della definizione degli alias '**Pre**' e '**Nom**'.

```
SELECT
    Pre.Giorno,
    Pre.Ingresso,
    Pre.Uscita,
    Nom.Cognome,
    Nom.Nome
    FROM Presenze AS Pre, Indirizzi AS Nom
    WHERE Pre.Codice = Nom.Codice;
```

75.3.4.8 Viste

«

Attraverso una vista, è possibile definire una relazione virtuale:

```
CREATE VIEW Presenze_dettagliate AS
SELECT
    Presenze.Giorno,
    Presenze.Ingresso,
    Presenze.Uscita,
    Indirizzi.Cognome,
    Indirizzi.Nome
    FROM Presenze, Indirizzi
    WHERE Presenze.Codice = Indirizzi.Codice;
```

L'esempio mostra la creazione della vista '**Presenze_dettagliate**', ottenuta dalle relazioni '**Presenze**' e '**Indirizzi**'. In pratica, questa vista permette di interrogare direttamente la relazione virtuale '**Presenze_dettagliate**', invece di utilizzare ogni volta un comando '**SELECT**' molto complesso, per ottenere lo stesso risultato.

75.3.4.9 Aggiornamento delle tuple

«

La modifica di tuple già esistenti avviene attraverso l'istruzione '**UPDATE**', la cui efficacia viene controllata dalla condizione posta dopo la parola chiave '**WHERE**'. Se tale condizione manca, l'effetto delle modifiche si riflette su tutte le tuple della relazione.

L'esempio seguente, aggiunge un attributo alla relazione degli indirizzi, per contenere il nome del comune di residenza degli impiegati; successivamente viene inserito il nome del comune '**Sferopoli**' in base al prefisso telefonico.

```
ALTER TABLE Indirizzi ADD COLUMN Comune char(30);
```

```
UPDATE Indirizzi
    SET Comune='Sferopoli'
    WHERE Telefono >= '022' AND Telefono < '023';
```

In pratica, viene aggiornata solo la tupla dell'impiegato '**Pinco Pallino**'.

75.3.4.10 Cancellazione delle tuple

«

L'esempio seguente elimina dalla relazione delle presenze le tuple riferite alle registrazioni del giorno 01/01/2012 e le eventuali antecedenti.

```
DELETE FROM Presenze WHERE Giorno <= '01/01/2012';
```

75.3.4.11 Creazione di una nuova relazione a partire da altre

«

L'esempio seguente crea la relazione '**mia_prova**' dalla fusione della relazioni degli indirizzi e delle presenze, come già mostrato in un esempio precedente:

```
SELECT
    Presenze.Giorno,
    Presenze.Ingresso,
    Presenze.Uscita,
    Indirizzi.Cognome,
    Indirizzi.Nome
    INTO TABLE mia_prova
    FROM Presenze, Indirizzi
    WHERE Presenze.Codice = Indirizzi.Codice;
```

75.3.4.12 Inserimento in una relazione esistente

L'esempio seguente aggiunge alla relazione dello storico delle presenze le registrazioni vecchie che poi vengono cancellate.

```
INSERT INTO PresenzeStorico (
        PresenzeStorico.Codice,
        PresenzeStorico.Giorno,
        PresenzeStorico.Ingresso,
        PresenzeStorico.Uscita
    )
    SELECT
        Presenze.Codice,
        Presenze.Giorno,
        Presenze.Ingresso,
        Presenze.Uscita
        FROM Presenze
        WHERE Presenze.Giorno <= '2012/01/01';

DELETE FROM Presenze WHERE Giorno <= '2012/01/01';
```

75.3.4.13 Controllare gli accessi a una relazione

Quando si creano delle relazioni in una base di dati, tutti gli altri utenti che sono stati registrati nel sistema del DBMS, potrebbero accedervi e fare le modifiche che vogliono. Per controllare questi accessi, l'utente proprietario delle relazioni (di solito è colui che le ha create), può usare le istruzioni '**GRANT**' e '**REVOKE**'. La prima permette a un gruppo di utenti di eseguire operazioni determinate, la seconda toglie dei privilegi.

```
GRANT {ALL | SELECT | INSERT | UPDATE | DELETE
              | RULE} [,…]
    ON relazione [,…]
    TO {PUBLIC | GROUP gruppo | utente}
```

```
REVOKE {ALL | SELECT | INSERT | UPDATE | DELETE
              | RULE} [,…]
    ON relazione [,…]
    FROM {PUBLIC | GROUP gruppo | utente}
```

La sintassi delle due istruzioni è simile, basta fare attenzione a cambiare la parola chiave '**TO**' con '**FROM**'. I gruppi e gli utenti sono nomi che fanno riferimento a quanto registrato all'interno del DBMS.

L'esempio seguente toglie a tutti gli utenti ('**PUBLIC**') tutti i privilegi sulle relazioni delle presenze e degli indirizzi; successivamente vengono ripristinati tutti i privilegi solo per l'utente '**tizio**':

```
REVOKE ALL
    ON Presenze, Indirizzi
    FROM PUBLIC;

GRANT ALL
    ON Presenze, Indirizzi
    TO tizio;
```

75.3.5 Controllo delle transazioni

La gestione delle transazioni richiede che queste siano introdotte dall'istruzione '**START TRANSACTION**':

```
START TRANSACTION
```

L'esempio seguente mostra il caso in cui si voglia isolare l'inserimento di una tupla nella relazione '**Indirizzi**' all'interno di una transazione, che alla fine viene confermata regolarmente con l'istruzione '**COMMIT**':

```
START TRANSACTION;

INSERT INTO Indirizzi
    VALUES (
        05,
        'De Pippo',
        'Pippo',
        'Via Pappo, 5',
        '0333,3333333'
    );

COMMIT;
```

Nell'esempio seguente, si rinuncia all'inserimento della tupla con l'istruzione '**ROLLBACK**' finale:

```
START TRANSACTION;

INSERT INTO Indirizzi
    VALUES (
        05,
        'De Pippo',
        'Pippo',
        'Via Pappo, 5',
        '0333,3333333'
    );

ROLLBACK;
```

75.3.6 Cursori

«

La gestione dei cursori da parte di PostgreSQL è abbastanza compatibile con lo standard, a parte il fatto che avviene fuori dal contesto previsto, che viene consentito un accesso in sola lettura e che non è possibile assegnare i dati a delle variabili.

La gestione dei cursori riguarda generalmente gli accessi a un DBMS tramite codice SQL incorporato in un programma (che usa un altro linguaggio), mentre PostgreSQL estende il loro utilizzo anche se il «programma» in questione è costituito esclusivamente da codice SQL.

La dichiarazione di un cursore si ottiene nel modo solito, con la differenza che questa deve avvenire esplicitamente in una transazione. In particolare, con PostgreSQL, il cursore viene aperto automaticamente nel momento della dichiarazione, per cui l'istruzione '**`OPEN`**' non è disponibile.

```
START TRANSACTION;

DECLARE Mio_cursore INSENSITIVE CURSOR FOR
    SELECT * FROM Indirizzi ORDER BY Cognome, Nome;

-- L'apertura del cursore non esiste in PostgreSQL
-- OPEN Mio_cursore;
…
```

L'esempio mostra la dichiarazione dell'inizio di una transazione, assieme alla dichiarazione del cursore '**Mio_cursore**', per selezionare tutta la relazione '**Indirizzi**' in modo ordinato per '**Cognome**'. Si osservi che per PostgreSQL la selezione che si ingloba nella gestione di un cursore non può aggiornarsi automaticamente se i dati originali cambiano, per cui è come se fosse sempre definita la parola chiave '**INSENSITIVE**'.

```
…
FETCH NEXT FROM Mio_cursore;
…
COMMIT;
```

L'esempio mostra l'uso tipico dell'istruzione '**FETCH**', in cui si preleva la tupla successiva rispetto alla posizione corrente del cursore e più avanti si conclude la transazione con un '**COMMIT**'. L'esempio seguente è identico, con la differenza che si indica espressamente il passo.

```
…
FETCH RELATIVE 1 FROM Mio_cursore;
…
COMMIT;
```

Un cursore dovrebbe essere chiuso attraverso una richiesta esplicita,

con l'istruzione '**CLOSE**', ma la chiusura della transazione chiude implicitamente il cursore, se questo dovesse essere rimasto aperto. L'esempio seguente riepiloga quanto visto sopra, completato dell'istruzione '**CLOSE**'.

```
START TRANSACTION;

DECLARE Mio_cursore INSENSITIVE CURSOR FOR
    SELECT * FROM Indirizzi ORDER BY Cognome, Nome;

-- L'apertura del cursore non esiste in PostgreSQL
-- OPEN Mio_cursore;

FETCH NEXT FROM Mio_cursore;

CLOSE Mio_cursore;

COMMIT;
```

75.3.7 Impostazione dell'ora locale

«

Il linguaggio SQL dispone dell'istruzione '**SET TIME ZONE**' per definire l'ora locale e di conseguenza lo scostamento dal tempo universale. PostgreSQL dispone della stessa istruzione che funziona in modo molto simile allo standard; per la precisione, la definizione dell'ora locale avviene attraverso le definizioni riconosciute dal sistema operativo (nel caso di GNU/Linux si tratta delle definizioni che si articolano a partire dalla directory '`/usr/share/zoneinfo/`').

```
SET TIME ZONE { 'definizione_ora_locale' | LOCAL }
```

Per esempio, per definire che si vuole fare riferimento all'ora locale italiana, si potrebbe usare il comando seguente:

```
SET TIME ZONE 'Europe/Rome';
```

Questa impostazione riguarda la visione del programma cliente, mentre il programma servente può essere stato preconfigurato attraverso le variabili di ambiente '**LC_***' oppure la variabile '**LANG**', che in questo caso hanno effetto sullo stile di rappresentazione delle informazioni data-orario. Anche il programma cliente può essere preconfigurato attraverso la variabile di ambiente '**PGTZ**', assegnandole gli stessi valori che si possono utilizzare per l'istruzione '**SET TIME ZONE**'.

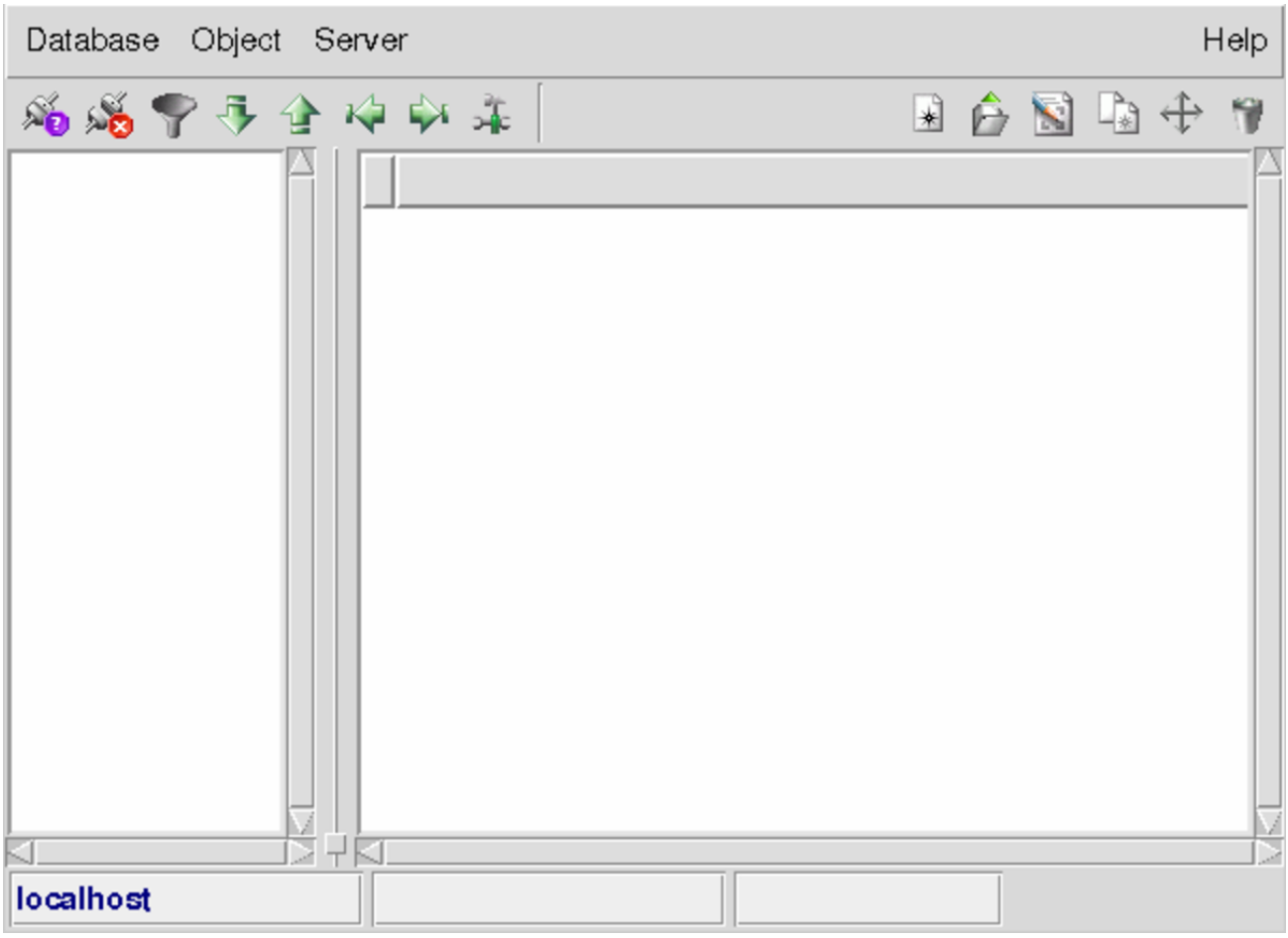
75.4 Accesso attraverso PgAccess

PgAccess [2] (ovvero PostgreSQL Access) è un componente di una libreria Tcl/Tk: LibPgTcl. A volte viene distribuito come un pacchetto autonomo, che comunque dipende dalla libreria indicata, oppure viene incluso nello stesso pacchetto della libreria. PgAccess è un programma frontale (che utilizza l'interfaccia grafica) per accedere alle funzionalità di PostgreSQL.

Prima di poter utilizzare qualunque programma frontale per PostgreSQL, occorre ricordare di configurare correttamente PostgreSQL stesso, in modo che questo consenta gli accessi previsti.

PgAccess è costituito in pratica dall'eseguibile '**pgaccess**', che si utilizza senza argomenti e si presenta inizialmente come si vede nella figura 75.94.

Figura 75.94. Finestra iniziale di PgAccess, quando viene avviato per la prima volta dall'utente.



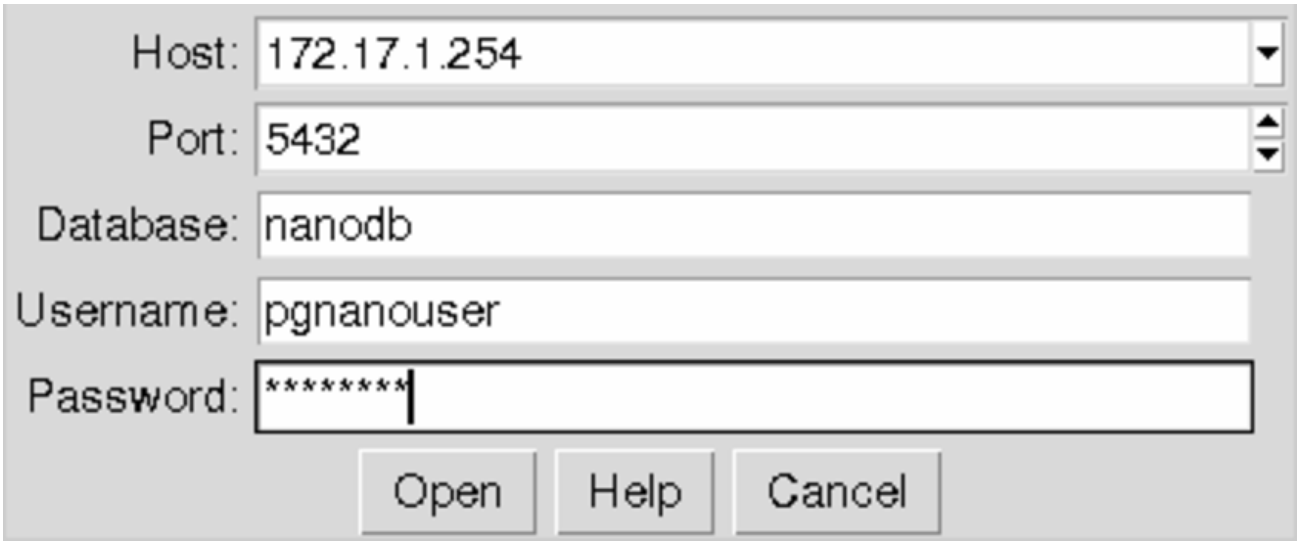
Mentre lo si usa, PgAccess memorizza alcune informazioni nella directory '`~/.pgaccess/`' e questo fatto facilita successivamente le operazioni di accesso alla base di dati da parte dell'utente.

Purtroppo, l'uso di programmi come questo, che mediano la comunicazione con un DBMS attraverso delle finestre grafiche, può risultare più complicato della scrittura manuale del codice SQL necessario. In questo capitolo, le figure appartengono a versioni diverse del programma, perché alcune funzionalità essenziali della versione aggiornata, si sono rivelate inaffidabili.

75.4.1 Accesso alla base di dati

PostgreSQL è un DBMS in grado di gestire diverse basi di dati simultaneamente; pertanto, con PgAccess è necessario stabilire per prima cosa quale sia la base di dati. Dal menù *Database*, si seleziona la funzione *Open*, ottenendo la mascherina che si vede nella figura 75.95. Da lì si possono indicare tutte le informazioni necessarie alla connessione con la base di dati desiderata; in particolare, per quanto riguarda le informazioni sull'autenticazione, queste sono richieste solo in base al modo in cui sono stati regolati i permessi di accesso da parte di PostgreSQL.

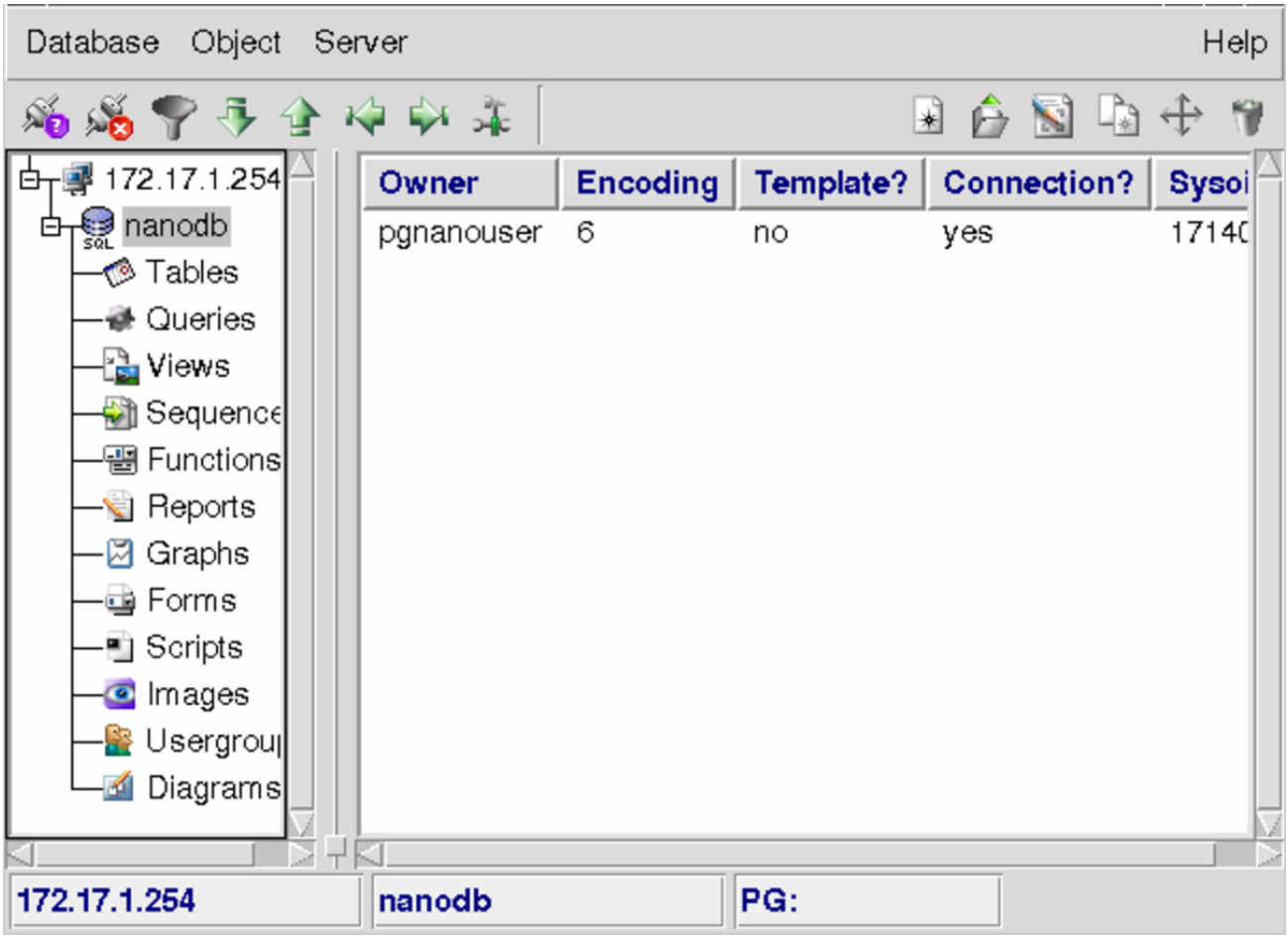
Figura 75.95. Connessione alla base di dati '**nanodb**', presso il nodo 172.17.1.254, come utente '**pgnanouser**'.



Attraverso PgAccess non è possibile creare una base di dati. Per questo occorre usare le funzioni di PostgreSQL, descritto nella sezione 75.2.

La base di dati aperta, assieme all'indicazione del nodo presso il quale si trova il DBMS con cui si interagisce, appare in basso, nella finestra principale di PgAccess.

Figura 75.96. Quando è attiva una connessione con una base di dati, lo si vede dalle informazioni che appaiono in basso nella finestra principale di PgAccess.



È importante ricordare che PgAccess tiene nota dell'ultima base di dati aperta attraverso i file di configurazione contenuti in '`~/.pgaccess/`'; in questo modo la connessione viene ritentata automaticamente all'avvio del programma la volta successiva che lo si utilizza.

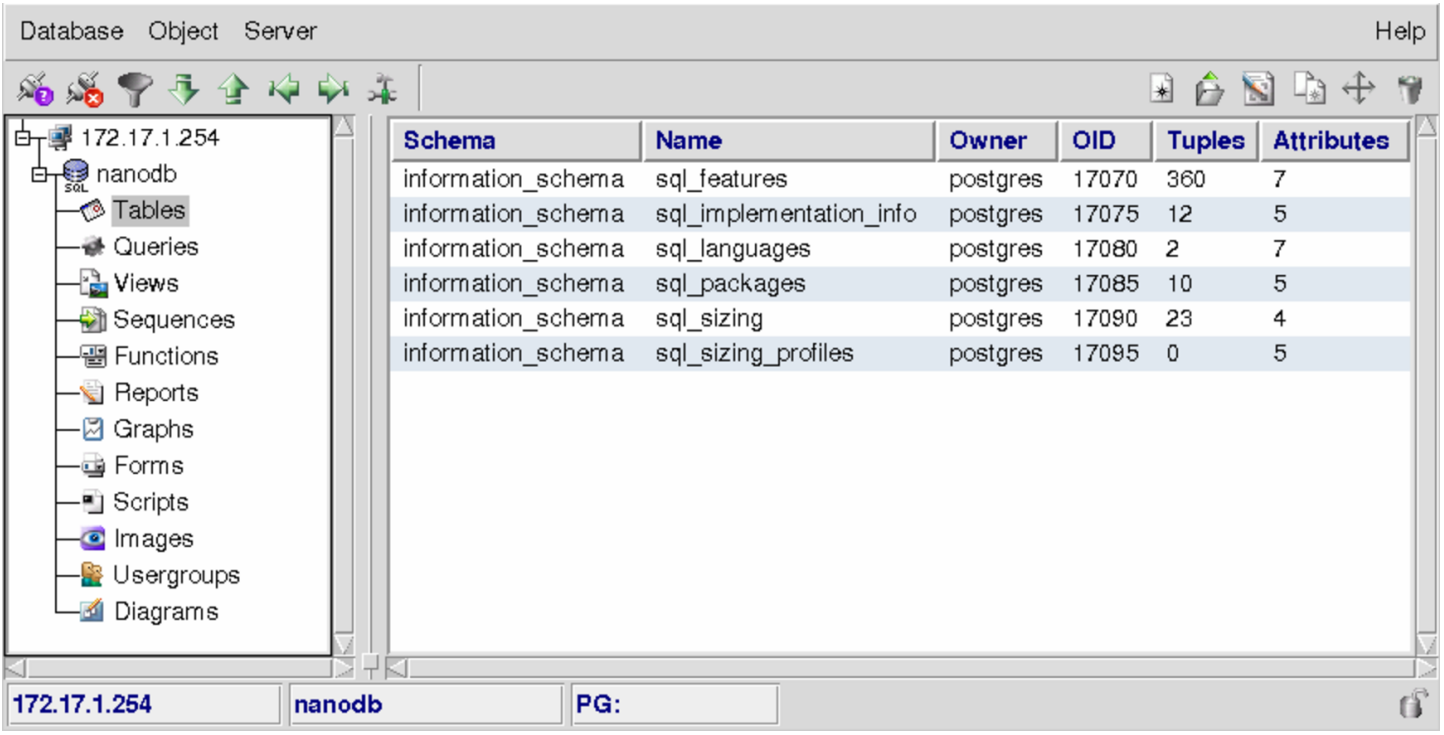
75.4.2 Gli «oggetti» secondo PgAccess

Dal punto di vista di PgAccess, una base di dati contiene degli «oggetti» (secondo la stessa filosofia di PostgreSQL). Questi possono

essere delle relazioni, il risultato di interrogazioni SQL, delle viste, delle stampe o altro ancora.

Per intervenire su ognuno di questi oggetti basta selezionare la voce relativa che si trova sulla parte sinistra (nella figura 75.97 si vede selezionata la gestione delle «tabelle», ovvero delle relazioni).

Figura 75.97. L'aspetto di PgAccess quando viene evidenziata a sinistra la voce *Tables*.



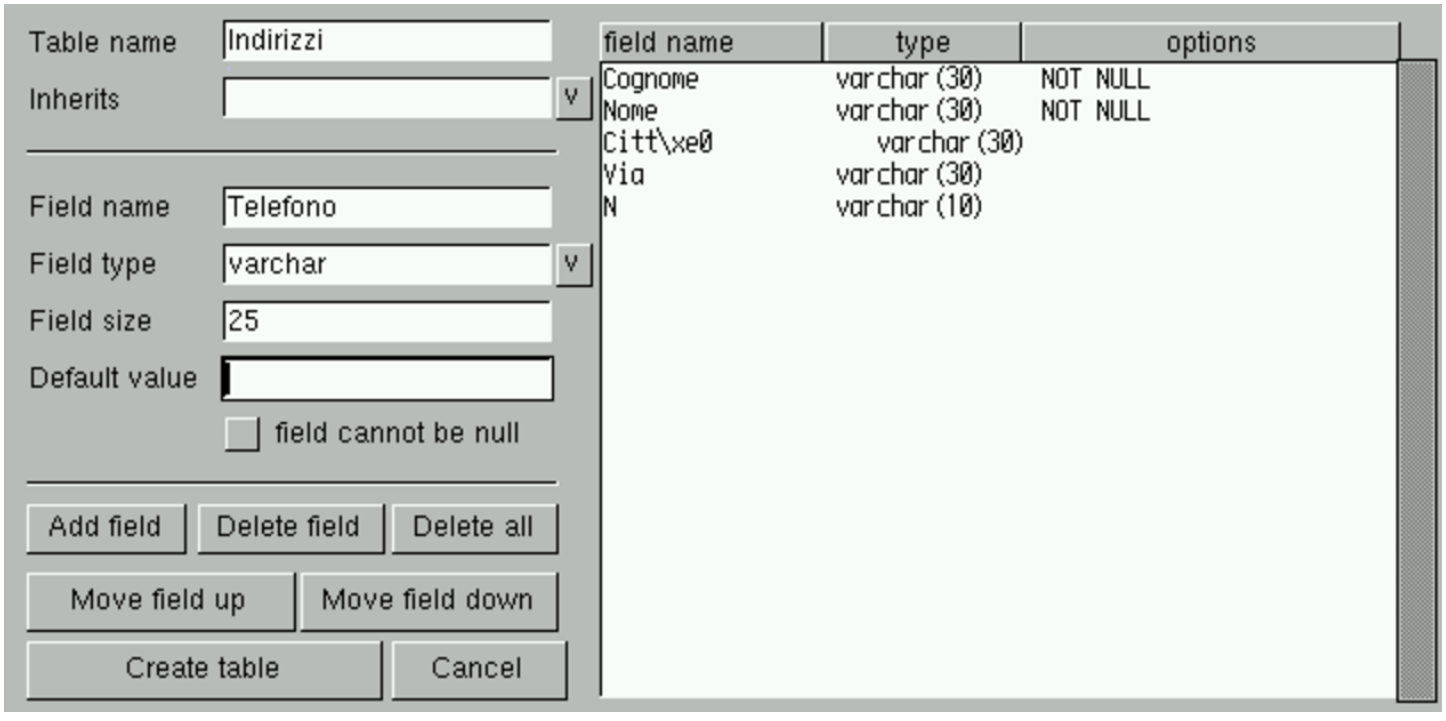
Premendo il tasto destro del mouse quando il puntatore si trova nel riquadro centrale, si ottiene un menù a scomparsa, con il quale è possibile modificare gli oggetti a cui fa riferimento la voce selezionata a sinistra; in alternativa, le stesse voci sono disponibili dal menù *Object*. In particolare, la voce *New* serve a creare un oggetto nuovo, *Open* serve ad accedervi e *Design* serve a modificarne la struttura (ammesso che ciò sia consentito in base al tipo di oggetto). Eventualmente è possibile anche modificare il nome dell'oggetto e visualizzarne la struttura.

PgAccess gestisce una serie aggiuntiva di oggetti rispetto a quanto fa PostgreSQL. Per realizzarli, PgAccess gestisce delle relazioni proprie, che non vengono mostrate all'utente, distinguibili per il fatto di avere un nome che inizia per '**pga_**'. In generale, queste relazioni hanno tutti i permessi di accesso per tutti gli utenti di PostgreSQL.

75.4.3 Relazioni

La figura 75.98 mostra l'esempio della creazione di una relazione molto semplice, per contenere una serie di indirizzi. Alla creazione della relazione, dopo avere selezionato la voce relativa a questo tipo di oggetto, si accede selezionando la voce *New* del menù *Object*.

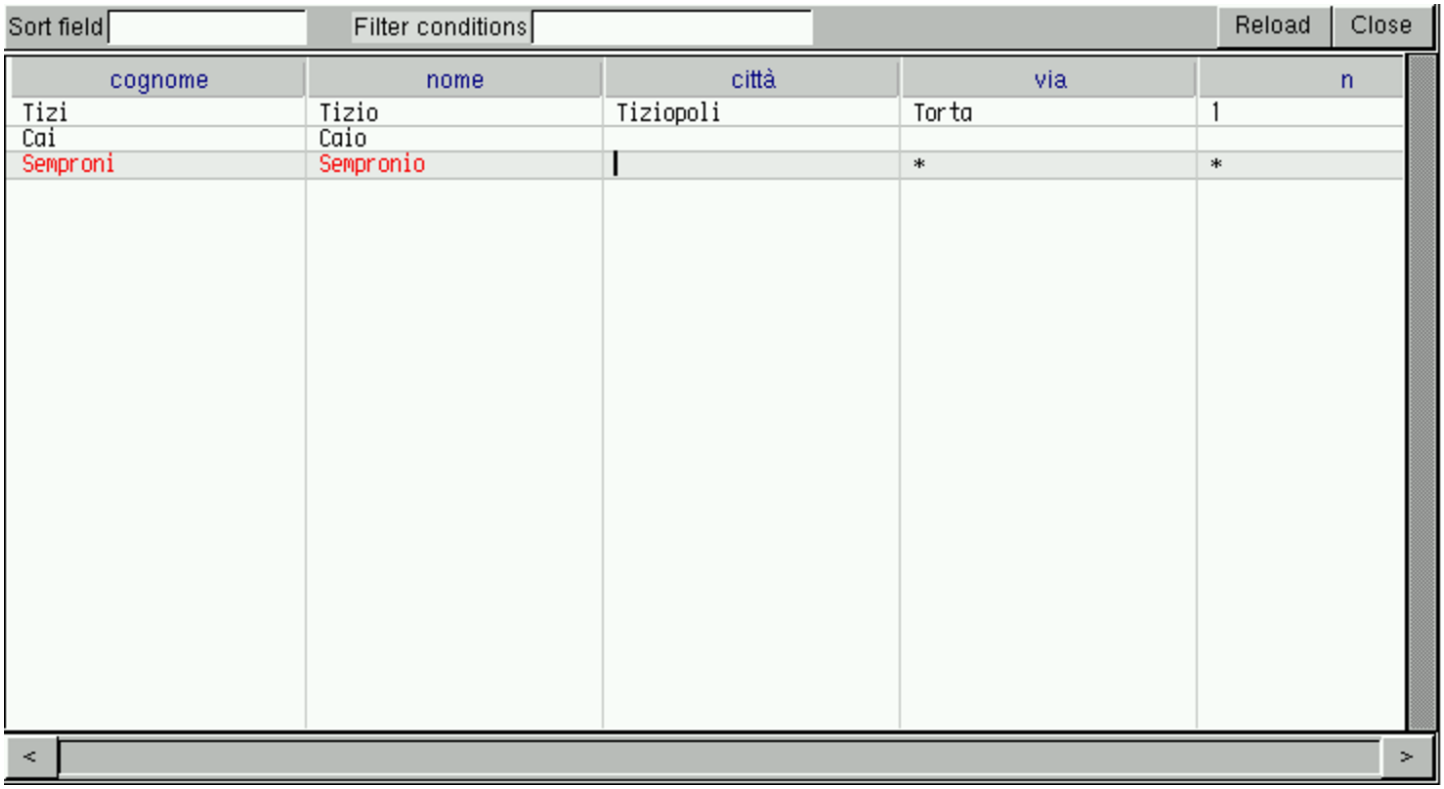
Figura 75.98. Finestra per la creazione di una relazione.



Una volta creata la relazione (si ottiene questo confermando con il pulsante grafico CREATE TABLE), il suo nome appare nella parte

centrale della finestra principale del programma; per accedere al suo contenuto basta selezionare la voce *Open* dal menù Object, ottenendo così una tabella di scorrimento con la quale si possono aggiungere e modificare righe preesistenti. La figura 75.99 mostra l'inserimento di alcuni nomi. Si osservi in particolare il fatto che, eventualmente, si può richiedere espressamente l'aggiunta di una riga nuova premendo il terzo tasto del mouse.

Figura 75.99. Finestra per lo scorrimento del contenuto di una relazione.



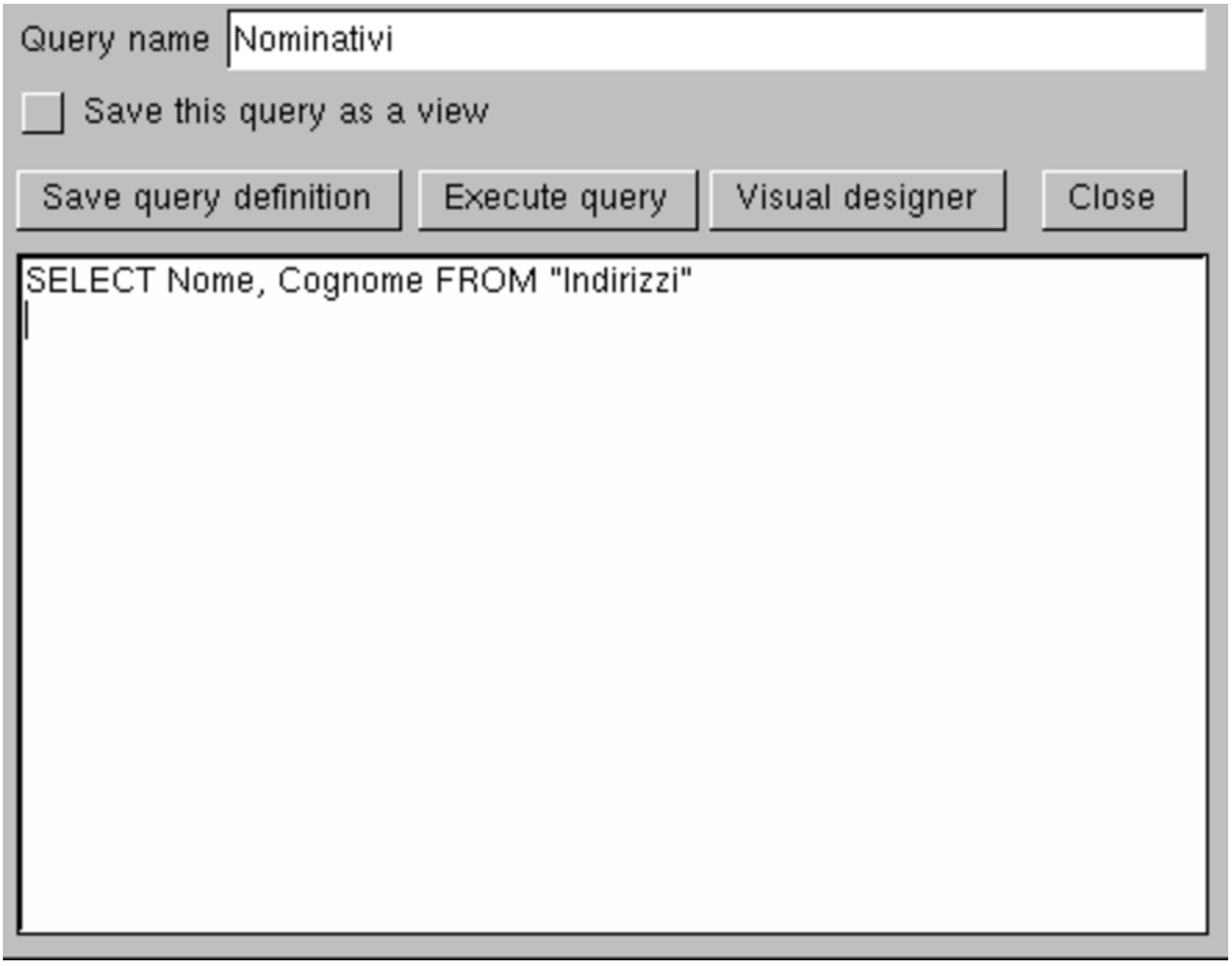
Vale la pena di osservare che la maschera di scorrimento e inserimento dati nella relazione, permette di leggere le tuple in ordine, in base a un certo attributo, filtrando eventualmente le tuple in base a una condizione. Si stabilisce questo mettendo il nome di un attributo nella casella '**`Sort field`**' e mettendo l'espressione della condizione di filtro nella casella '**`Filter conditions`**': se poi si seleziona il pulsante grafico RELOAD (che comunque appare come un'icona),

si riottiene il contenuto ordinato e filtrato in base alle preferenze indicate.

75.4.4 Interrogazioni e viste

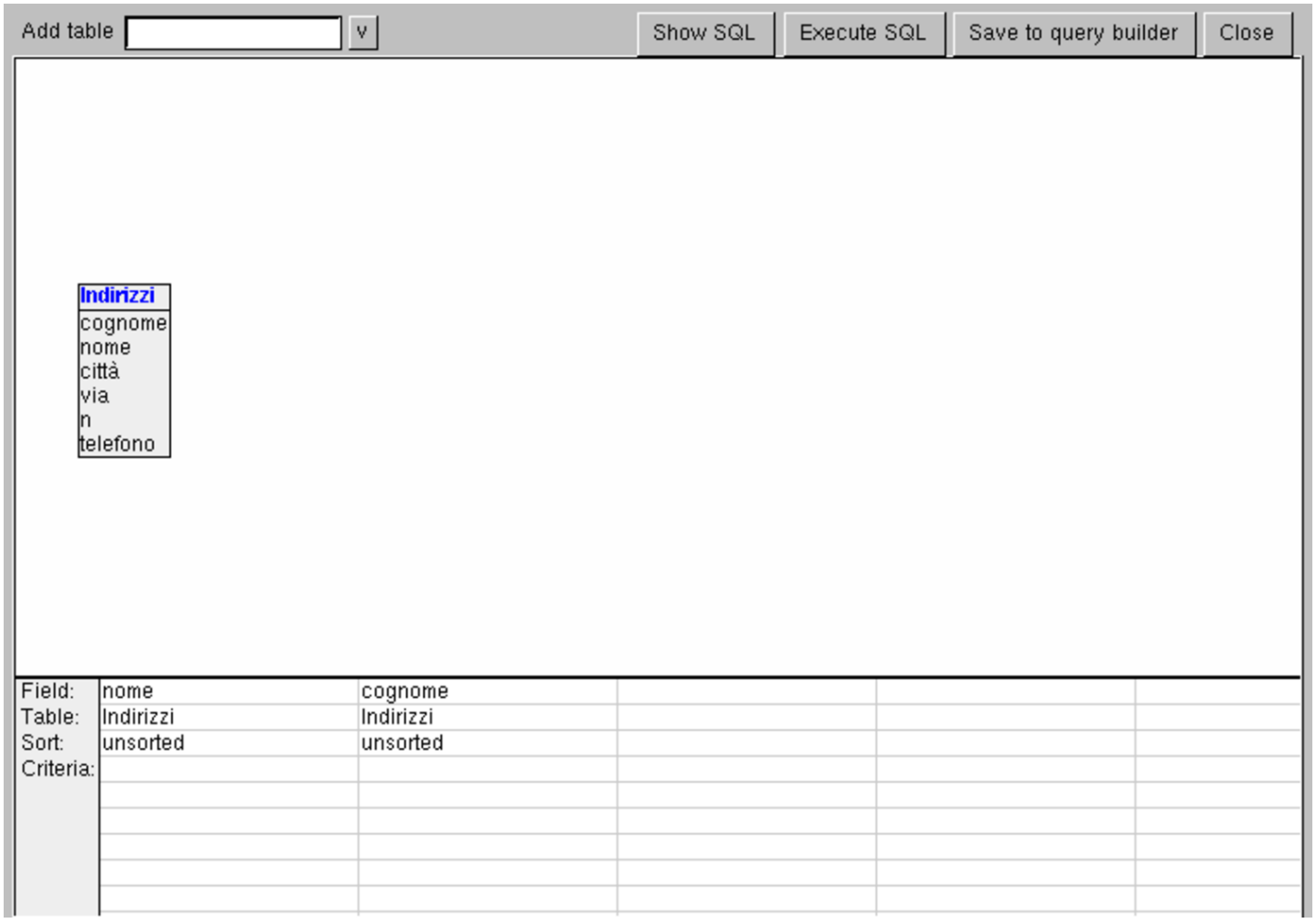
È possibile realizzare facilmente dei modelli di interrogazione e delle viste, attraverso la selezione delle voci *Queries* e *Views* (nella parte sinistra della finestra, sotto alla voce *Tables*). Nel primo caso si tratta di interrogazioni SQL che vengono memorizzate da PgAccess e richiamate a piacere, mentre nel secondo si tratta di viste vere e proprie. A livello operativo, con PgAccess le due cose sono praticamente identiche, per cui si passa generalmente per la creazione di un'interrogazione SQL che poi, eventualmente, si salva come vista. La figura 75.100 mostra la definizione dell'interrogazione '**`Nominativi`**', abbinata al comando '**`SELECT Cognome, Nome FROM "Indirizzi"`**', scritto manualmente dall'utilizzatore.

Figura 75.100. Finestra per la creazione di un'interrogazione.



Nella figura si può osservare che è disponibile una casella di selezione attraverso la quale si può richiedere di salvare come vista. In particolare, con il pulsante grafico SAVE QUERY DEFINITION si salva il modello dell'interrogazione, con il nome fissato in alto; ma volendo, con il pulsante grafico VISUAL DESIGNER, si accede a una maschera per la definizione grafica dell'interrogazione, come si vede nella figura 75.101.

Figura 75.101. Finestra per la creazione visuale di un'interrogazione.



In alto appare una casella in cui si deve indicare il nome di una relazione da cui si vogliono prelevare i campi; una volta fatto, appare un riepilogo di questi campi, in un riquadro. Questi nomi possono essere trascinati con il puntatore del mouse, in basso, dove vengono elencati i campi da includere nell'interrogazione; se si sbaglia, gli elementi che si vogliono togliere possono essere cancellati premendo il tasto [*Canc*] ([*Del*] nelle tastiere inglesi). Nella figura mostrata, sono già stati trascinati e depositati i campi del nome e del cognome.

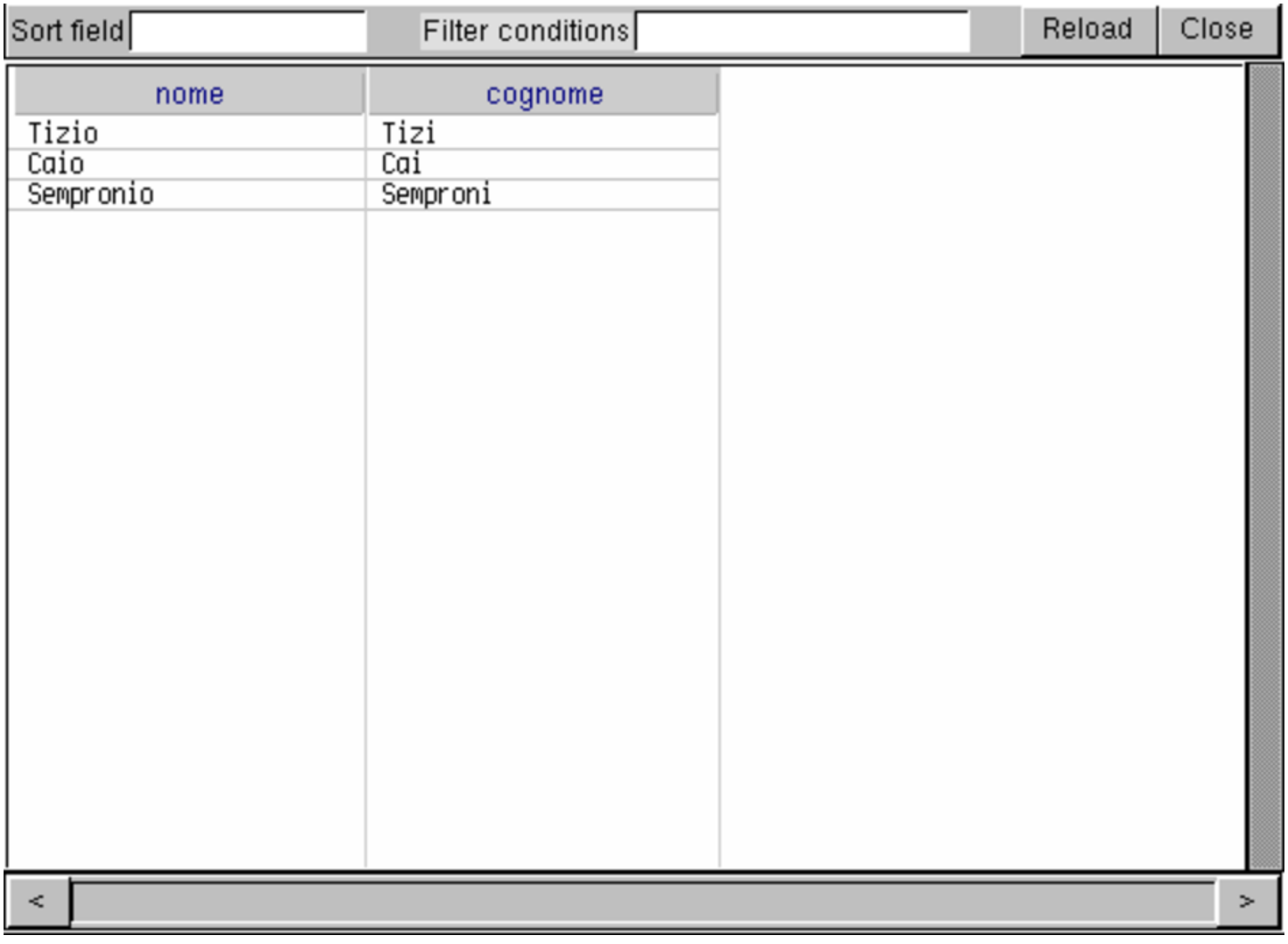
Al termine, se si è soddisfatti del risultato, si può confermare con

il pulsante grafico SAVE TO QUERY BUILDER, ritrovando poi nella finestra precedente l'interrogazione corrispondente alle scelte fatte, che può essere ritoccata a mano se lo si desidera. Nel caso dell'esempio mostrato, l'interrogazione SQL che si ottiene è:

```
select t0.nome, t0.cognome from "Indirizzi" t0
```

L'apertura di un'interrogazione o di una vista, genera lo scorrimento del risultato dell'interrogazione, oppure della vista, come si vede nella figura 75.103 che fa sempre riferimento agli esempi precedenti.

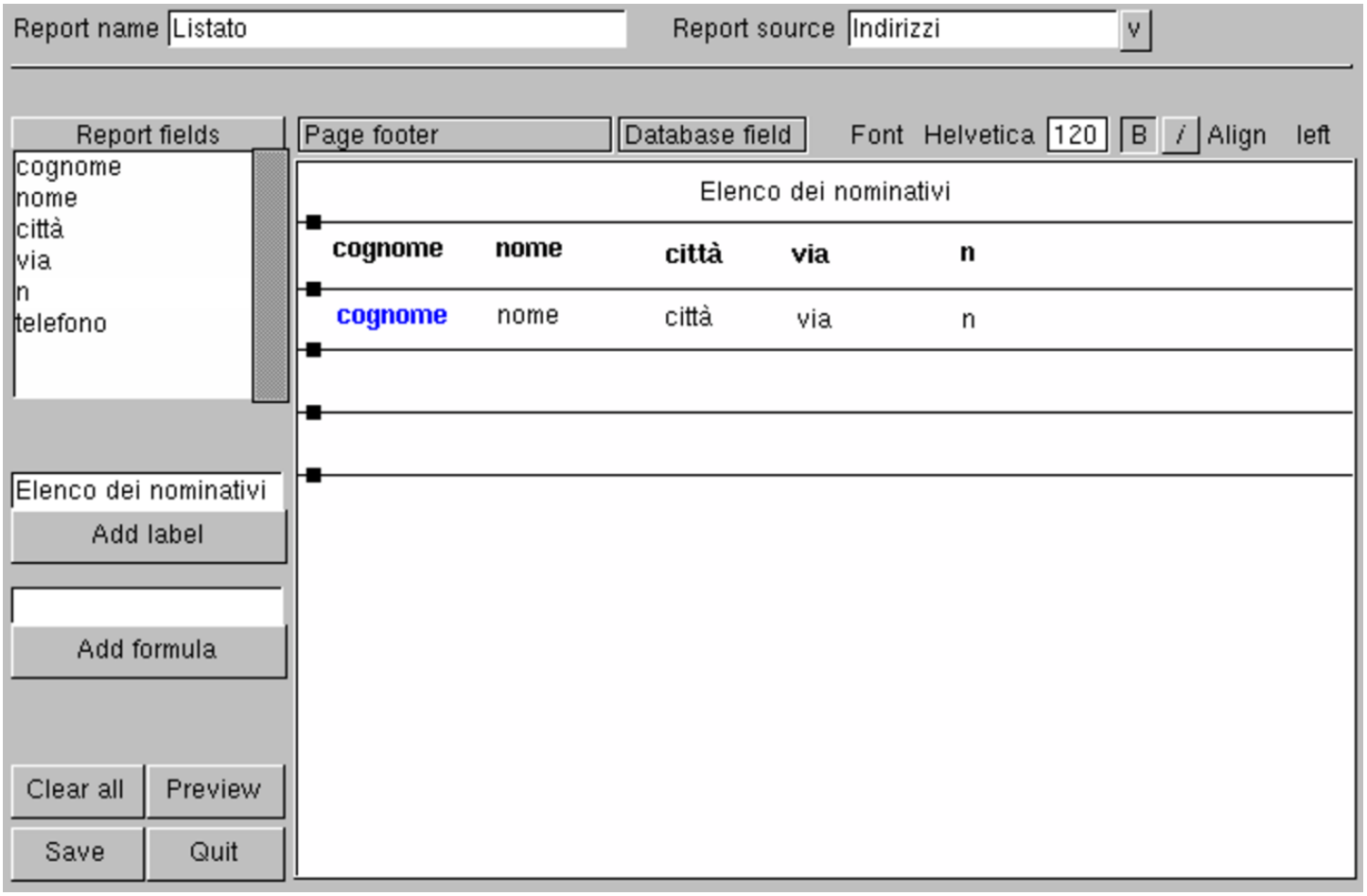
Figura 75.103. Scorrimento di una vista.



75.4.5 Stampe

Con PgAccess è possibile definire anche delle stampe, nel senso di rapporti stampati contenenti il risultato di un'interrogazione SQL. La figura 75.104 mostra la finestra che si utilizza per questo scopo, dove è già iniziata la compilazione dello schema di stampa.

Figura 75.104. Creazione di un tabulato.



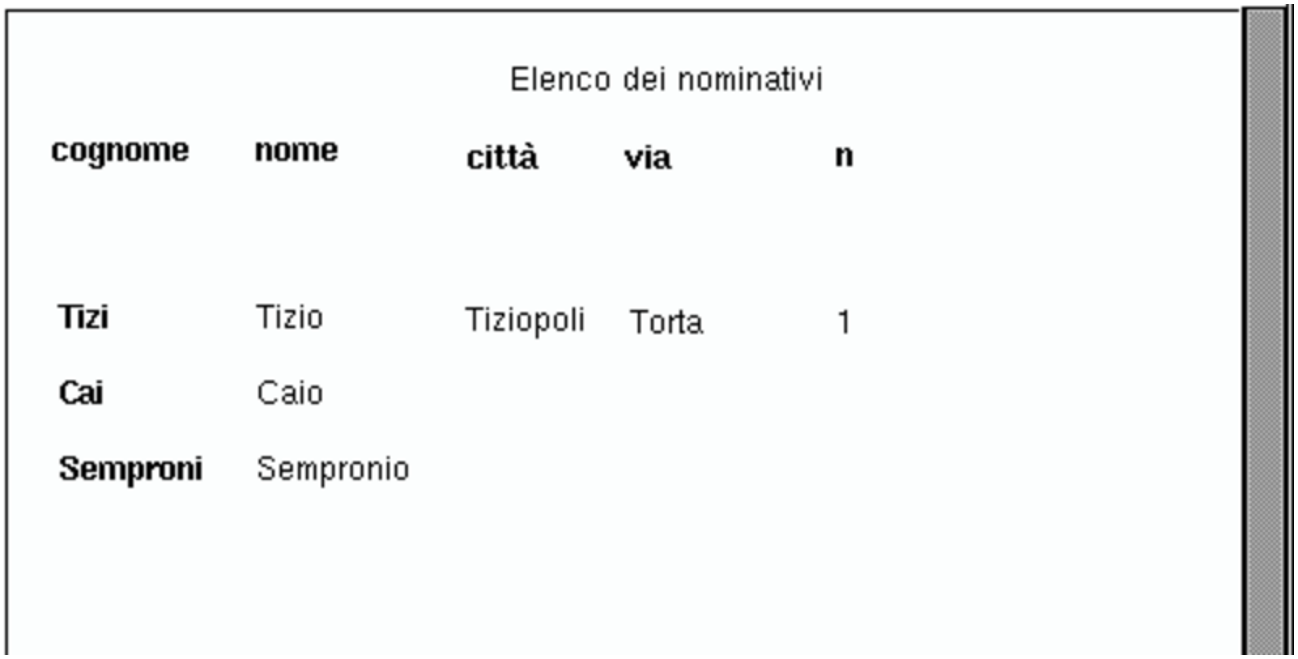
Una volta selezionata la relazione da cui prelevare i campi, dopo aver indicato il nome del tabulato che si vuole generare, basta fare un clic con il tasto sinistro del mouse mentre si punta sul nome del campo che si vuole inserire sullo schema di destra (che rappresenta il modello della stampa). Una volta che sono apparsi i nomi nello spazio a destra, questi possono essere trascinati dove si vuole, even-

tualmente possono anche essere cancellati usando il tasto [*Canc*]. Nell'esempio della figura, si vede anche che è stato inserito un titolo.

Spostando il puntatore del mouse sullo spazio che rappresenta lo schema di stampa, si vede cambiare la sua descrizione in alto. Nella figura mostrata viene indicato '**`Page footer`**', perché in quel momento il puntatore del mouse era nella penultima riga di quello schema.

Per verificare il risultato, è disponibile anche un'anteprima, che si ottiene selezionando il pulsante grafico PREVIEW. Seguendo gli esempi precedenti, la figura 75.105 mostra questa anteprima. Da lì si può passare alla stampa, che però potrebbe limitarsi a generare un file PostScript.

Figura 75.105. Anteprima di stampa.



75.5 Accesso attraverso WWW-SQL

WWW-SQL [3] è un programma CGI in grado di creare pagine HTML a partire dalle informazioni ottenute da una base di dati PostgreSQL o MySQL. In questo capitolo si vuole vedere in particolare l'interazione rispetto alle basi di dati di PostgreSQL. In ogni caso, per poter leggere questo capitolo, occorre sapere cosa sia un programma CGI e come interagisce con un servente HTTP, come spiegato nel capitolo 40.

È molto probabile che la propria distribuzione GNU abbia organizzato due pacchetti distinti, in base all'uso che se ne intende fare, per l'abbinamento con PostgreSQL, oppure con MySQL. In questo modo, il nome del programma CGI a cui si deve fare riferimento può cambiare leggermente, anche da una distribuzione all'altra. Qui si fa riferimento al nome '**`www-pgsql`**' per quello che riguarda l'uso con PostgreSQL.

75.5.1 Principio di funzionamento

Ammesso che il pacchetto organizzato dalla propria distribuzione sia stato realizzato nel modo corretto, l'eseguibile '**`www-pgsql`**' dovrebbe trovarsi nella directory più adatta per i programmi CGI, ovvero quella a cui si accede normalmente con l'URI *http://localhost/cgi-bin/*. In tal caso, per accedere a questo programma, basta avviare il proprio navigatore preferito e puntare sull'indirizzo *http://localhost/cgi-bin/www-pgsql*. Ma non basta, dal momento che il programma in questione ha bisogno di interpretare un file HTML speciale dal quale restituisce poi un risultato. Per capire come funziona la cosa, prima ancora di avere affrontato lo studio del linguaggio specifico di WWW-SQL, si può provare con un file HTML normale:

si supponga di avere a disposizione il file ‘`http://localhost/index.html`’; per fare in modo che WWW-SQL lo analizzi, basta indicare l’URI *http://localhost/cgi-bin/www-pgsql/index.html*. Il risultato è identico all’originale, ma per arrivare a questo si passa attraverso l’elaborazione del programma CGI, dimostrando così il suo funzionamento.

Volendo, se il proprio programma servente HTTP è Apache, è possibile rendere la cosa più elegante attraverso una configurazione opportuna del file ‘`srm.conf`’. Per esempio si potrebbe fare in modo che i file che terminano con l’estensione ‘`.pgsql`’ vengano elaborati automaticamente attraverso il programma CGI in questione:

```
Action www-pgsql /cgi-bin/www-pgsql
AddHandler www-pgsql pgsql
```

Tuttavia, occorre considerare che alcune installazioni di Apache sono state predisposte in modo da impedire l’utilizzazione dell’istruzione ‘**`Action`**’. Se dopo le modifiche di questo file, il servizio di Apache non si riavvia, ciò potrebbe essere un sintomo di questo problema.

75.5.2 Preparazione delle basi di dati e accesso

«

Perché il programma CGI possa accedere alle basi di dati di PostgreSQL, occorre ricordare di predisporre gli utenti e i permessi necessari all’interno della gestione delle basi di dati stesse. Potrebbe essere conveniente prevedere la possibilità di accesso per l’utente di sistema usato dal processo elaborativo del servente HTTP, quando esegue i programmi CGI, in modo da semplificare l’istruzione necessaria alla connessione. Supponendo che si tratti dell’utente ‘**`www-cgi`**’,

volendo procedere in questo modo, occorre aggiungere tale utente, con lo stesso nome, nel sistema di PostgreSQL:

```
postgres$ createuser www-cgi [Invio]
```

Quindi occorre intervenire nelle basi di dati regolando i permessi attraverso i comandi '**GRANT**' e '**REVOKE**', tenendo conto che a questo proposito si può consultare quanto già spiegato nella sezione 75.1. Per fare un esempio, volendo concedere l'accesso in lettura alla relazione '**Indirizzi**', della base di dati '**anagrafe**', all'utente '**www-cgi**', si potrebbe agire come si vede di seguito:

```
postgres$ psql anagrafe [Invio]

anagrafe=> GRANT SELECT ON Indirizzi TO www-cgi; [Invio]
```

75.5.3 Linguaggio di WWW-SQL

WWW-SQL interpreta un file HTML alla ricerca di istruzioni secondo il formato schematizzato di seguito:

```
<! SQL comando [argomento...] >
```

Come si vede, queste istruzioni assomigliano a dei commenti per l'HTML, ma anche se non lo sono realmente, di solito i navigatori ignorano dei marcatori di questo tipo. Tuttavia, questa si può considerare solo come una misura di sicurezza, dal momento che questi file non dovrebbero essere raggiunti direttamente, ma solo attraverso l'intermediazione di WWW-SQL.

Le istruzioni di WWW-SQL rappresentano un linguaggio di programmazione, semplice, ma efficace per lo scopo che ci si prefigge. Si osservi che il «comando» è una parola chiave che rappresen-

ta il tipo di azione che si intende svolgere; inoltre, gli argomenti possono essere presenti o meno, in funzione del comando. Gli argomenti di un comando possono essere racchiusi tra apici doppi (‘"…"’): all’interno di queste stringhe si possono indicare delle variabili da espandere e si possono usare anche delle sequenze di escape per rappresentare simboli speciali che altrimenti avrebbero un altro significato.

Le parole chiave che costituiscono le istruzioni di WWW-SQL possono essere scritte indipendentemente utilizzando lettere maiuscole o minuscole. Inoltre, lo spazio dopo il delimitatore iniziale ‘**<!**’ e lo spazio prima del delimitatore finale ‘**>**’ sono facoltativi.

Per iniziare subito con un esempio che faccia capire la logica di funzionamento di WWW-SQL, si osservi il «programma» seguente, rappresentato dal file ‘`variabili.pgsql`’:

```
<HTML>
<HEAD>
    <TITLE>Esempio sul funzionamento delle
           variabili con WWW-SQL</TITLE>
</HEAD>
<BODY>
<H1>Esempio sul funzionamento delle variabili
    con WWW-SQL</H1>

<P><! SQL PRINT "var = $var" ></P>

<FORM ACTION="variabili.pgsql" METHOD="GET">
    <P><INPUT NAME="var">
    <INPUT TYPE="submit">
</FORM>
```

```
</BODY>
```

L'unica istruzione per WWW-SQL è '**<!SQL PRINT**...**>**', con la quale si vuole ottenere la visualizzazione di una stringa tra apici doppi. Si osservi che '**$var**' è il riferimento alla variabile '**var**', che viene espanso, come parte della valutazione della stringa.

Come si può intuire leggendo l'esempio, i campi definiti attraverso i modelli (gli elementi '**FORM**'), si traducono in variabili per WWW-SQL.

Per verificare il funzionamento di questo programma, supponendo di avere collocato il file '`variabili.pgsql`' nella directory iniziale dei documenti HTML offerti dal servente HTTP, basta puntare il navigatore sull'indirizzo *http://localhost/cgi-bin/www-pgsql/variabili.pgsql* (sempre ammettendo che l'indirizzo *http://localhost/cgi-bin/www-pgsql* corrisponda all'avvio del programma CGI che costituisce in pratica WWW-SQL).

Quello che si ottiene dovrebbe essere un modulo HTML molto semplice, dove si può inserire un testo. Inviando il modulo compilato, dovrebbe essere restituito lo stesso modulo, con la stringa iniziale aggiornata, dove viene mostrato che è stato recepito il dato inserito (nella figura 75.108 si vede che è stata inviata la stringa «Saluti».

Figura 75.108. Risultato dell'interpretazione del file '`variabili.pgsql`' attraverso WWW-SQL.

I sorgenti di WWW-SQL possono essere compilati in modo differente. In particolare, si può distinguere tra due tipi di scansione: il tipo vecchio non permette l'uso di istruzioni che prevedono un'iterazione. In pratica, in quel caso, non funzionano i cicli iterativi e gli altri comandi correlati.

75.5.3.1 Espressioni

Si distinguono due tipi di espressioni che si possono valutare all'interno delle istruzioni di WWW-SQL: quelle che si applicano ai valori numerici e quelle che si applicano alle stringhe. Le tabelle 75.109 e 75.110 elencano gli operatori che possono essere utilizzati a questo proposito. Si osservi in particolare l'operatore '`:`', che permette di fare un confronto tra una stringa e un'espressione regolare.

Tabella 75.109. Elenco degli operatori utilizzabili con operandi numerici.

Operatore e operandi	Descrizione
+op	Non ha alcun effetto.
−op	Inverte il segno dell'operando.
op1 + *op2*	Somma i due operandi.
op1 − *op2*	Sottrae dal primo il secondo operando.
op1 * *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo.
op1 % *op2*	Modulo: il resto della divisione tra il primo e il secondo operando.
op1 ^ *op2*	Eleva il primo operando alla potenza del secondo.
op1 == *op2*	*Vero* se gli operandi sono uguali.
op1 = *op2*	*Vero* se gli operandi sono uguali (sinonimo di '**==**').
op1 != *op2*	*Vero* se gli operandi sono differenti.
op1 > *op2*	*Vero* se il primo operando è maggiore del secondo.
op1 < *op2*	*Vero* se il primo operando è minore del secondo.
op1 >= *op2*	*Vero* se il primo operando è maggiore o uguale al secondo.

Operatore e operandi	Descrizione
op1 `<=` ***op2***	*Vero* se il primo operando è minore o uguale al secondo.
`!` ***op***	Negazione logica.
op1 `&&` ***op2***	AND logico.
op1 `&` ***op2***	AND logico (sinonimo di '**`&&`**').
op1 `||` ***op2***	OR logico.
op1 `|` ***op2***	OR logico (sinonimo di '**`||`**').

Tabella 75.110. Elenco degli operatori utilizzabili con operandi di tipo stringa.

Operatore e operandi	Descrizione
op1 `==` ***op2***	*Vero* se gli operandi sono uguali.
op1 `!=` ***op2***	*Vero* se gli operandi sono differenti.
op1 `>` ***op2***	*Vero* se il primo operando è lessicograficamente successivo al secondo.
op1 `<` ***op2***	*Vero* se il primo operando è lessicograficamente precedente al secondo.
op1 `>=` ***op2***	*Vero* se il primo operando non è lessicograficamente precedente al secondo.
op1 `<=` ***op2***	*Vero* se il primo operando non è lessicograficamente successivo al secondo.
str `:` ***regexp***	*Vero* se l'espressione regolare corrisponde alla stringa.

All'interno delle stringhe è prevista l'espansione di variabili e sono anche riconosciute alcune sequenze di escape (tabella 75.111). Le variabili in questione vanno intese come parte del linguaggio di WWW-SQL; alcune di queste sono la ripetizione di variabili di ambiente corrispondenti, altre sono variabili interne del programma (come elencato nella tabella 75.112), altre ancora possono essere definite all'interno del «programma» stesso, o meglio ancora, attraverso dei moduli, come è stato mostrato nell'esempio iniziale. Le variabili vengono riconosciute in quanto scritte secondo lo schema seguente:

prefisso nome_della_variabile

Il prefisso è un simbolo a scelta tra: '**$**', '**@**', '**?**', '**#**'. In pratica, '**$var**', '**@var**', '**?var**', e '**#var**', sono riferimenti identici alla stessa variabile '**var**'. Per questo motivo, se si vogliono usare i simboli corrispondenti a questi prefissi in modo letterale, occorre usare una sequenza di escape.

Tabella 75.111. Sequenze di escape utilizzabili all'interno delle stringhe.

Escape	Significato
\\	\
\"	"
\n	<*LF*>
\t	<*HT*> (tabulazione)

Escape	Significato
`\$`	$
`\@`	@
`\#`	#
`\?`	?
`\~`	~

Tabella 75.112. Variabili interne di WWW-SQL.

Variabile	Descrizione
`AFFECTED_ROWS`	Numero di righe coinvolte dall'ultima interrogazione.
`NUM_FIELDS`	Numero di campi restituiti dall'ultima interrogazione.
`NUM_ROWS`	Numero di righe restituiti dall'ultima interrogazione.
`WWW_SQL_VERSION`	Versione di WWW-SQL.
`GATEWAY_INTERFACE`	Versione dell'interfaccia CGI.
`HOSTTYPE`	Tipo di macchina del servente HTTP.
`HTTPHOST`	Nome del nodo servente.
`HTTP_REFERER`	Pagina da cui proviene il cliente.
`HTTP_USER_AGENT`	Nome del programma di navigazione (cliente).

Variabile	Descrizione
`OSTYPE`	Nome del sistema operativo del servente.
`PATH_INFO`	Percorso relativo dello script attuale.
`PATH_TRANSLATED`	Percorso assoluto del file corrispondente allo script attuale.
`REMOTE_ADDR`	Indirizzo del nodo remoto.
`REMOTE_HOST`	Nome del nodo remoto.
`SERVER_ADMIN`	Indirizzo di posta elettronica dell'amministratore.
`SERVER_NAME`	Nome del servente.
`SERVER_PORT`	Numero della porta utilizzata per la connessione con il servente.
`SERVER_PROTOCOL`	Nome e versione del protocollo (HTTP).
`SERVER_SOFTWARE`	Nome del software usato come servente HTTP.
`SCRIPT_FILENAME`	Percorso del programma CGI (l'eseguibile di WWW-SQL).
`SCRIPT_NAME`	Percorso relativo del programma CGI (l'eseguibile di WWW-SQL).
`REQUEST_URI`	Indirizzo richiesto.

Per prendere confidenza con le variabili interne di WWW-SQL, si può realizzare lo script seguente ('`interne.pgsql`'), che con l'istruzione '**`<!SQL DUMPVARS>`**' le elenca tutte. La figura 75.114 mostra il risultato che si potrebbe ottenere.

```
<HTML>
<HEAD>
    <title>Visualizzazione delle variabili interne</title>
</HEAD>
<BODY>
<H1>Visualizzazione delle variabili interne</H1>

<! SQL DUMPVARS >

</BODY>
```

Figura 75.114. Esempio del contenuto delle variabili interne attraverso l'istruzione '**<!SQL DUMPVARS>**'.

```
          Visualizzazione delle variabili interne

WWW_SQL_VERSION = 0.5.5
SERVER_SOFTWARE = Apache/1.3.3 (Unix) Debian/GNU
SERVER_PROTOCOL = HTTP/1.0
SERVER_PORT = 80
SERVER_NAME = dinkel.brot.dg
SERVER_ADMIN = webmaster@dinkel.brot.dg
SCRIPT_FILENAME = /usr/lib/cgi-bin/www-pgsql
SCRIPT_NAME = /cgi-bin/www-pgsql
REQUEST_URI = /cgi-bin/www-pgsql/interne.pgsql
REMOTE_ADDR = 127.0.0.1
QUERY_STRING =
PATH_TRANSLATED = /var/www/interne.pgsql
PATH_INFO = /interne.pgsql
HTTP_USER_AGENT = Lynx/2.8.1rel.2 libwww-FM/2.14
HTTP_HOST = localhost
GATEWAY_INTERFACE = CGI/1.1
DOCUMENT_ROOT = /var/www
```

75.5.3.2 Strutture di controllo

Attraverso le istruzioni di WWW-SQL, si possono realizzare le strutture di controllo che sono comuni nei linguaggi di programmazione. È prevista la struttura condizionale e il ciclo iterativo.

```
<! SQL IF espressione >
    ...
[<! SQL ELSIF espressione >]
    ...
[<! SQL ELSE >]
    ...
<! SQL ENDIF >
```

La struttura condizionale che si vede nello schema, permette di delimitare uno spazio da filtrare in base all'esito delle espressioni condizionali coinvolte. Si osservi l'esempio seguente:

```
<! SQL IF $NUM_ROWS == 10 >
    <P>Il numero delle righe è uguale a 10.</P>
<! SQL ELSE >
    <P>Il numero delle righe non corrisponde a
    quanto previsto.</P>
<! SQL ENDIF >
```

In questo modo si condiziona la visualizzazione di una frase in base al fatto che la variabile '**NUM_ROWS**' contenga o meno il valore 10.

È importante osservare che l'espressione usata come condizione di controllo potrebbe restituire un risultato numerico e non logico. In tal caso, lo zero corrisponde a *Falso*, mentre qualunque altro valore corrisponde a *Vero*.

```
<! SQL WHILE espressione >
    ...
<! SQL DONE >
```

La struttura iterativa che si vede nello schema, permette di delimitare uno spazio da interpretare ripetitivamente, finché l'espressione condizionale introduttiva continua a restituire il valore *Vero* (o un valore numerico diverso da zero).

```
<! SQL SET contatore 10 >
<! SQL WHILE $contatore > 0 >
    <P>Il contatore ha raggiunto il livello
       <! SQL PRINT "$contatore" >.</P>
    <! SQL SETEXPR contatore $contatore - 1 >
<! SQL DONE >
```

L'esempio mostra l'inizializzazione di una variabile, denominata '**contatore**', al valore iniziale 10; quindi inizia un ciclo iterativo che si arresta quando tale variabile raggiunge lo zero. A ogni ciclo, viene visualizzato il contenuto della variabile, che subito dopo viene ridotto di un'unità.

Se l'istruzione '**<!SQL WHILE**...**>**' non viene riconosciuta, significa che non è disponibile la scansione iterativa.

Nell'ambito di un'iterazione, possono essere usate delle istruzioni per interrompere il ciclo in corso o per interrompere tutta l'iterazione:

```
<! SQL CONTINUE >
```

```
<! SQL BREAK >
```

La prima della due istruzioni interrompe il ciclo attuale, facendo riprendere immediatamente l'iterazione, mentre il secondo interrompe l'iterazione del tutto.

Esiste anche un altro tipo di iterazione, il cui scopo è la scansione delle righe ottenute dall'interrogazione di una base di dati:

```
<! SQL PRINT_LOOP riferimento_all'interrogazione >
    ...
<! SQL DONE >
```

Anche all'interno di questa struttura si possono usare le istruzioni '**<!SQL CONTINUE>**' e '**<!SQL BREAK>**'.

75.5.4 Istruzioni

Le istruzioni «normali» di WWW-SQL, ovvero quelle che non servono a descrivere delle strutture di controllo, sono descritte in questa sezione e in quelle seguenti. In particolare si può notare che WWW-SQL offre delle istruzioni per la lettura semplificata dell'esito di un'interrogazione SQL e altre per la lettura dettagliata, fino ad arrivare a distinguere tupla per tupla e attributo per attributo.

È importante chiarire che, anche se un'«interrogazione» serve principalmente per leggere dati da una relazione di una base di dati, nello stesso modo, attraverso WWW-SQL si potrebbero fare delle modifiche ai dati.

Segue un elenco di istruzioni di tipo vario, mentre nelle sezioni seguenti vengono raccolte altre istruzioni più specifiche.

- Emissione di una stringa con espansione di variabili:

```
<! SQL PRINT stringa >
```

  L'istruzione '**<!SQL PRINT** ...**>**' permette di emettere una stringa. Dal momento che un file HTML non ha bisogno di accorgimenti particolari per mostrare una stringa costante, è evidente che il senso di questa istruzione sta nella possibilità di indicare delle variabili da espandere, come nell'esempio seguente:

```
<P>Il contatore ha raggiunto il livello
   <! SQL PRINT "$contatore" >.</P>
```

- Risultato di un'espressione:

```
<! SQL EVAL espressione >
```

  L'istruzione '**<!SQL EVAL** ...**>**' è simile a '**<!SQL PRINT** ...**>**', con la differenza che l'argomento non è più una stringa, ma un'espressione differente, il cui risultato viene emesso alla fine.

- Impostazione di una variabile:

```
<! SQL SET nome_variabile valore_da_assegnare >
```

L'istruzione '**<!SQL SET ...>**' permette di definire e inizializzare una variabile. L'esempio seguente definisce la variabile '**contatore**', inizializzandola a zero:

```
<! SQL SET contatore 0 >
```

- Impostazione di una variabile attraverso un'espressione:

```
<! SQL SETEXPR nome_variabile espressione >
```

L'istruzione '**<!SQL SETEXPR ...>**' permette di definire e inizializzare una variabile; in particolare, il valore che si assegna può essere il risultato della valutazione di un'espressione. L'esempio seguente definisce la variabile '**contatore**', inizializzandola con il risultato dell'espressione '**$contatore - 1**'. In pratica viene decrementato il contenuto della variabile '**contatore**':

```
<! SQL SETEXPR contatore $contatore - 1 >
```

- Definizione di un valore predefinito per il contenuto di una variabile:

```
<! SQL SETDEFAULT nome_variabile valore_da_assegnare >
```

L'istruzione '**<!SQL SETDEFAULT ...>**' permette di stabilire un valore predefinito per una variabile; a differenza di '**<!SQL SET ...>**' la variabile non viene modificata se esiste già e ha un valore. L'esempio seguente definisce la variabile '**contatore**', solo se necessario, inizializzandola con il valore 10:

```
<! SQL SETDEFAULT contatore 10 >
```

- Elenco variabili:

```
<! SQL DUMPVARS >
```

L'istruzione '**<!SQL DUMPVARS>**' emette l'elenco delle variabili esistenti, assieme al valore che contengono. Può essere usato per scopo diagnostico, quando si cerca di capire cosa succede realmente.

75.5.4.1 Apertura e chiusura di una connessione, e accesso a una base di dati

«

L'interrogazione di una base di dati deve essere preceduta dalla connessione a un servente DBMS e dalla selezione di una base di dati; inoltre, al termine delle interrogazioni, si passa normalmente alla chiusura di una connessione, in pratica secondo lo schema seguente:

```
<! SQL CONNECT … >
<! SQL DATABASE nome_della_base_di_dati >
…
…
<! SQL CLOSE >
```

In breve: '**<!SQL CONNECT** ...**>**' serve a iniziare una connessione con un servente per l'accesso a una base di dati; '**<!SQL DATABASE** ...**>**' serve a indicare la base di dati specifica presso il servente; '**<!SQL CLOSE>**' chiude la connessione.

- Accesso a un servente DBMS:

```
<! SQL CONNECT [nodo [utente [parola_d'ordine] ] ] >
```

L'istruzione '**<!SQL CONNECT ...>**' permette di iniziare una connessione con un DBMS. Dipende dal DBMS stesso se è possibile accedere senza alcun sistema di autenticazione. In generale, se non si indica il nodo a cui accedere, si intende *localhost*; inoltre, se non si indica l'utente, si fa riferimento al numero UID con il quale funziona il programma servente del servizio HTTP (che a sua volta avvia il programma CGI). L'esempio che segue richiede di connettersi al servente DBMS PostgreSQL che opera nello stesso elaboratore locale, utilizzando l'identità dell'utente '**pgnanouser**' e senza specificare alcuna parola d'ordine:

```
<! SQL CONNECT localhost pgnanouser >
```

- Selezione di una base di dati specifica:

```
<! SQL DATABASE nome_base_di_dati >
```

L'istruzione '**<!SQL DATABASE ...>**' permette di aprire una base di dati specifica; per la precisione, utilizzando PostgreSQL, l'accesso al servente avviene solo dopo che è stata specificata la base di dati.

- Chiusura di una connessione:

```
<! SQL CLOSE >
```

La chiusura di una connessione (e quindi anche di una base di dati aperta), si ottiene con l'istruzione '**<!SQL CLOSE>**'.

Prima di passare alla descrizione delle istruzioni che permettono l'interrogazione del contenuto di una base di dati, viene mostrato un esempio che si limita a elencare la relazione '**Indirizzi**' della base di dati '**anagrafe**':

```
<HTML>
<HEAD>
    <TITLE>Esempio di interrogazione</TITLE>
</HEAD>
<BODY>
<H1>Esempio di interrogazione</H1>
<! SQL CONNECT localhost nobody >
<! SQL DATABASE anagrafe >
<! SQL QUERY "SELECT * FROM Indirizzi" RICHIESTA_1 >
<! SQL QTABLE RICHIESTA_1 >
<! SQL FREE RICHIESTA_1 >
<! SQL CLOSE >
</BODY>
```

75.5.4.2 Istruzioni di interrogazione normali

«

L'interrogazione di una base di dati avviene attraverso la definizione di un riferimento, che si apre e si chiude come se fosse un flusso di file nei linguaggi di programmazione comuni. Per aprire questo riferimento si inizia con l'invio di un'interrogazione SQL; successivamente è possibile leggere l'esito dell'interrogazione attraverso il riferimento che è stato aperto; infine si passa alla chiusura del riferimento:

```
<! SQL QUERY stringa_di_interrogazione_sql riferimento >
…
…
<! SQL FREE riferimento >
```

- Apertura di un'interrogazione:

```
<! SQL QUERY stringa_di_interrogazione_sql riferimento >
```

L'istruzione '**<!SQL QUERY ...>**' definisce una stringa di interrogazione da inviare al servente DBMS. A questa interrogazione viene abbinato un riferimento costituito da un nome, che in seguito deve essere usato per leggere l'esito dell'interrogazione. Nell'esempio che appare nella sezione precedente, si vedeva l'istruzione seguente con la quale si selezionano tutte le tuple della relazione '**Indirizzi**', abbinando questo risultato al nome '**RICHIESTA_1**':

```
<! SQL QUERY "SELECT * FROM Indirizzi" RICHIESTA_1 >
```

- Tabella rapida:

```
<! SQL QTABLE riferimento [borders] >
```

L'istruzione '**<!SQL QTABLE ...>**' consente di rappresentare rapidamente il risultato di un'interrogazione attraverso una tabella HTML. In particolare, utilizzando la parola chiave '**borders**', la tabella che si genera ha i bordi delle caselle visibili. L'esempio seguente mostra in che modo visualizzare rapidamente il risultato dell'interrogazione abbinata al nome '**RICHIESTA_1**':

```
<! SQL QTABLE RICHIESTA_1 >
```

- Elenco rapido:

```
<! SQL QLONGFORM riferimento >
```

L'istruzione '**<!SQL QLONGFORM ...>**' si utilizza in modo simile a '**<!SQL QTABLE ...>**', per rappresentare il risultato di un'in-

terrogazione attraverso un elenco dettagliato, senza una tabella HTML.

- Chiusura del riferimento all'interrogazione:

```
<! SQL FREE riferimento >
```

Come è stato mostrato all'inizio, l'istruzione '**<!SQL FREE ...>**' serve a chiudere il riferimento a un'interrogazione.

- Realizzazione di un elenco di voci da selezionare:

```
<! SQL QSELECT riferimento variabile_modulo_html >
```

Con l'istruzione '**<!SQL QSELECT ...>**' si ottiene un elenco di voci di un modulo di selezione. In generale, la cosa corrisponde a:

```
<SELECT NAME="variabile_modulo_html">
<! SQL PRINT_ROWS riferimento ↩
↪"<OPTION name=\"@riferimento.0\">riferimento.1">
</SELECT>
```

L'istruzione '**<!SQL PRINT_ROWS ...>**' è descritta nella prossima sezione.

75.5.4.3 Istruzioni per la selezione dettagliata di tuple e attributi

«

È possibile selezionare in maniera più precisa le tuple e gli attributi da ciò che si ottiene da un'interrogazione SQL. Attraverso l'istruzione '**<!SQL FETCH *riferimento*>**' si preleva la tupla attuale dall'interrogazione a cui si fa riferimento. Questo prelievo permette

di fare riferimento agli attributi della tupla attraverso una notazione particolare:

```
@riferimento . n
```

In pratica, è come se fosse l'espansione di una variabile, con la differenza che si indica il nome di un riferimento a un'interrogazione aperta, aggiungendo un'estensione numerica, separata da un punto, dove lo zero corrisponde al primo attributo e $n-1$ corrisponde all'attributo n-esimo.

- Spostamento della tupla attuale all'interno del risultato di un'interrogazione:

  ```
  <! SQL SEEK riferimento n_riga >
  ```

  L'istruzione '**<!SQL SEEK ...>**' permette di spostare la tupla attuale all'interno di un'interrogazione. Per indicare il numero della tupla da raggiungere, occorre tenere presente che lo zero corrisponde alla prima. L'esempio seguente fa in modo che la tupla attuale diventi la seconda del riferimento '**RICHIESTA_1**':

  ```
  <! SQL SEEK RICHIESTA_1 1 >
  ```

- Prelievo della tupla attuale di un certo riferimento:

  ```
  <! SQL FETCH riferimento >
  ```

  L'istruzione '**<!SQL FETCH ...>**' permette di rendere disponibile il contenuto della tupla attuale di un certo riferimento. L'esempio seguente preleva il contenuto della tupla attuale del riferimento '**RICHIESTA_1**'; quindi mostra il primo e il secondo attributo di

questa tupla, che si presume corrispondano al cognome e al nome di una persona:

```
<! SQL FETCH RICHIESTA_1 >
<P>Cognome: <! SQL PRINT "@RICHIESTA_1.0" ></P>
<P>Nome: <! SQL PRINT "@RICHIESTA_1.1" ></P>
```

- Emissione di una stringa per ogni tupla:

```
<! SQL PRINT_ROWS riferimento stringa >
```

L'istruzione '**<!SQL PRINT_ROWS ...>**' è una sorta di istruzione '**<!SQL PRINT ...>**' ripetuta per tutte le tuple di un'interrogazione, a partire da quella corrente. L'esempio seguente mostra la visualizzazione dei primi due attributi di tutte le tuple di un'interrogazione, a cui si fa riferimento con il nome '**Q**':

```
<! SQL SEEK Q 0 >
<! SQL PRINT_ROWS Q "<P>Cognome: @Q.0</P>\n<P>Nome: @Q.1</P>\n" >
```

L'esempio seguente mostra la realizzazione di un modulo per la selezione di un articolo, attraverso l'invio del codice corrispondente. A questo proposito, si suppone che il primo attributo del risultato dell'interrogazione a cui si fa riferimento con il nome '**ELENCO**', corrisponda al codice dell'articolo, mentre il secondo corrisponda a una sua descrizione:

```
<FORM ACTION="ordine.pgsql">
<P><SELECT NAME="codice">
<! SQL PRINT_ROWS ELENCO "<OPTION name=\"@ELENCO.0\">@ELENCO.1" >
</SELECT>
<INPUT TYPE="submit">
</FORM>
```

Dal momento che si fa riferimento alle prime due colonne, la stessa cosa avrebbe potuto essere realizzata con l'istruzione '**<!SQL QSELECT ...>**', nel modo seguente:

```
<FORM ACTION="ordine.pgsql">
<! SQL QSELECT ELENCO codice >
<INPUT TYPE="submit">
</FORM>
```

75.6 Riferimenti

«

- *PostgreSQL*, *http://www.postgresql.org/*
- The PostgreSQL Global Development Group, *PostgreSQL Documentation*, *http://www.postgresql.org/docs/manuals/*
- *PgAccess*, *http://sourceforge.net/projects/pgaccess/*
- James Henstridge, *WWW-SQL*, *http://www.jamesh.id.au/software/www-sql/www-sql.html*

[1] **PostgreSQL** software libero con licenza speciale

[2] **PgAccess** software libero con licenza speciale

[3] **WWW-SQL** GNU GPL

76.1 Struttura e preparazione

MySQL [1] è un DBMS (*Data base management system*) relazionale. Il nome contiene la sigla «SQL», a indicare che si tratta di un DBMS in grado di comprendere le istruzioni SQL.

L'installazione del servente MySQL può essere molto laboriosa, se non si dispone di un pacchetto già pronto per la propria distribuzione

GNU. La descrizione di come installare MySQL viene omessa, perché questa appare nella documentazione di MySQL in modo molto dettagliato. Qui si fa riferimento a una situazione relativamente «comune», a seguito dell'installazione automatica di un pacchetto pronto.

In generale, il servente MySQL è costituito dal demone '**mysqld**', avviato attraverso uno script della procedura di inizializzazione del sistema, che potrebbe corrispondere a '`/etc/init.d/mysql`'. Assieme a questo demone ci sono altri programmi di servizio che servono principalmente per l'amministrazione e la manutenzione delle basi di dati, il più importante dei quali è '**mysqladmin**'.

L'installazione del servente MySQL richiede, nell'ambito del sistema operativo, la preparazione di un utente e un gruppo speciali, entrambi con il nome '**mysql**'. Questa utenza dovrebbe essere già disponibile oppure potrebbe essere creata automaticamente dal sistema di installazione dei pacchetti della propria distribuzione. La directory personale associata a questo utente speciale dovrebbe rappresentare il contenitore dei file che compongono le basi di dati gestite da MySQL: '`~mysql/`'.

Il servente MySQL è sottoposto al controllo di un file di configurazione, che in condizioni normali potrebbe corrispondere a '`/etc/mysql/my.cnf`', oppure solo '`/etc/my.cnf`'. Tuttavia, questo file è suddiviso in sezioni e contiene anche la configurazione relativa ai programmi clienti, mentre una configurazione specifica del solo servente può essere collocata nel file '`~mysql/my.cnf`'.

L'amministratore del DBMS con MySQL si chiama convenzional-

mente ‘**root**’ e si prevede che in un sistema Unix coincida con l’utente ‘**root**’, ma ciò non è strettamente necessario. L’accesso da parte di questo utente, come degli altri, è sottoposto alla presentazione di una parola d’ordine che viene stabilita con l’ausilio di ‘**mysqladmin**’, o attraverso istruzioni SQL appropriate; pertanto, anche se ci può essere una corrispondenza tra utenze di MySQL e utenze del sistema operativo, le parole d’ordine usate non sono collegate tra loro.

MySQL ha una gestione particolare delle proprie utenze, distinguendole in base alla provenienza degli accessi. A seconda del contesto può capitare di utilizzare notazioni del tipo ‘**tizio@dinkel.brot.dg**’, dove si intende fare riferimento all’utente denominato ‘**tizio**’ che accede dal nodo *dinkel.brot.dg*. Deve essere subito chiaro che non si tratta di un indirizzo di posta elettronica e nemmeno di un’utenza Unix. Pertanto, anche l’utente ‘**root**’ (l’amministratore del DBMS), deve essere qualificato meglio e solitamente si fa riferimento a ‘**root@localhost**’ (l’utente ‘**root**’ che accede al servente attraverso lo stesso elaboratore che offre il servizio).

76.1.1 Configurazione del servente

Come accennato, la configurazione generale di MySQL è contenuta nel file ‘`/etc/mysql/my.cnf`’, oppure ‘`/etc/my.cnf`’, a seconda dell’organizzazione della propria distribuzione GNU, con la possibilità di usare il file ‘`~mysql/my.cnf`’ per ciò che riguarda strettamente il funzionamento del servente. Molto probabilmente, il file di configurazione generale è già predisposto per consentire un accesso locale al servente MySQL; di solito può essere utile verificare

o ritoccare solo alcune direttive, specialmente per ciò che riguarda l'abilitazione dell'accesso attraverso la rete.

Il file di configurazione contiene direttive che assomigliano in pratica all'assegnamento di variabili, nella forma seguente:

```
nome = valore
```

Queste direttive sono suddivise in sezioni, dichiarate tra parentesi quadre:

```
[sezione]
direttiva
...
```

È attraverso la presenza di queste sezioni che è possibile distinguere le direttive relative al servente da quelle di altre componenti.

Come spesso accade nei file di configurazione comuni, il simbolo '`#`' introduce un commento, fino alla fine della riga; inoltre, le righe bianche o vuote sono ignorate.

Le direttive che riguardano il funzionamento del servente sono contenute nelle sezioni '`[mysqld_safe]`' e '`[mysqld]`'. Per il momento conviene soffermarsi solo su alcune della seconda di queste sezioni:

```
[mysqld]
user            = mysql
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
port            = 3306
log             = /var/log/mysql.log
basedir         = /usr
datadir         = /var/lib/mysql
tmpdir          = /tmp
language        = /usr/share/mysql/english
```

Si comprende intuitivamente il significato di queste direttive: il demone '**`mysqld`**' viene avviato con i privilegi dell'utente '**`mysql`**'; viene usato il file '`/var/run/mysqld/mysqld.pid`' per annotare il numero PID del demone in funzione; viene utilizzato il file '`/var/run/mysqld/mysqld.sock`' come socket di dominio Unix per le comunicazioni locali, oppure la porta 3306 per le comunicazioni attraverso la rete; nel file '`/var/log/mysql.log`' vengono annotate le informazioni relative al suo funzionamento; la directory '`/usr/`' è il punto di partenza dell'installazione dei programmi e di altre componenti; la directory '`/var/lib/mysql/`' è il punto di inizio dei file che compongono le basi di dati (equivale a '`~mysql/`'); la directory usata per i file temporanei è '`/tmp/`'; infine, i messaggi mostrati dal demone sono quelli contenuti nella directory '`/usr/share/mysql/english/`', ovvero quelli espressi in lingua inglese.

Alcune direttive particolari non hanno la forma dell'assegnamento e contengono una sola stringa, nella forma seguente:

```
skip-nome
```

La presenza di queste direttive indica la disabilitazione di ciò che

viene descritto sinteticamente dalla parte finale del loro nome; per esempio, la direttiva '**skip-networking**' serve a disabilitare l'accesso attraverso la rete. Di solito, la configurazione predefinita di MySQL prevede proprio l'uso di questa direttiva per impedire l'accesso attraverso la rete, rendendo necessaria la modifica di questo file per consentirlo in modo esplicito.

Un altro tipo di direttiva, che comunque rientra nel genere normale di assegnamento, serve a dichiarare delle variabili, intese come opzioni di funzionamento, a cui si assegna anche un valore. Si tratta di direttive che hanno l'aspetto seguente:

```
set-variable = nome=valore
```

L'utilizzo di queste variabili può dipendere dal modo in cui si compilano i sorgenti di MySQL; pertanto può essere necessario stabilire su cosa si può intervenire, avviando il demone '**mysqld**' con l'opzione '**--help**', anche senza privilegi particolari:

```
$ /usr/bin/mysqld --help [Invio]
```

Oltre alla sintassi relativa alla riga di comando, si dovrebbe osservare la presenza dell'elenco seguente, che qui viene abbreviato:

```
Variables (--variable-name=value)
and boolean options {FALSE|TRUE}  Value (after reading options)
--------------------------------- ----------------------------
auto-rehash                       TRUE
character-sets-dir                (No default value)
default-character-set             (No default value)
compress                          FALSE
database                          (No default value)
…
net_buffer_length                 16384
select_limit                      1000
max_join_size                     1000000
```

76.1.1.1 Controllo dell'accesso dalla rete

Un problema comune che si può avvertire quando si cerca di mettere in funzione un servente MySQL, sta nel concedere gli accessi attraverso la rete.

Generalmente, la configurazione contenuta nel file '`/etc/mysql/my.cnf`' (o solo '`/etc/my.cnf`') prevede una direttiva che limita gli accessi al solo elaboratore locale. Ci sono due possibilità:

```
[mysqld]
…
skip-networking
…
bind-address = 127.0.0.1
…
```

Se si usa la direttiva '**`skip-networking`**', si intende concedere esclusivamente un accesso locale, tramite un socket di dominio Unix; se si usa la direttiva '**`bind-address = 127.0.0.1`**' (senza però usare anche '**`skip-networking`**'), si concede l'accesso

attraverso la rete, ma in pratica si concede esclusivamente un accesso locale, tramite l'indirizzo 127.0.0.1.

In pratica, per consentire un accesso remoto al DBMS, occorre eliminare (o commentare) entrambe queste direttive.

76.1.2 Avvio e arresto del servente

«

L'avvio e l'arresto del servente dovrebbe essere gestito da uno script della procedura di inizializzazione del sistema, che in generale potrebbe corrispondere a '`/etc/init.d/mysql`'. Se le cose stanno così, è sufficiente avviare il servizio con il comando seguente:

```
# /etc/init.d/mysql start [Invio]
```

Nello stesso modo, per arrestare il servizio basta il comando seguente:

```
# /etc/init.d/mysql stop [Invio]
```

Può essere interessante approfondire cosa succede realmente all'interno di questo script per comprendere come è organizzato MySQL nell'ambito del proprio sistema operativo.

In generale ci sono da considerare due aspetti: il demone '**mysqld**' non viene avviato direttamente, ma attraverso un altro programma, '**mysqld_safe**'; in secondo luogo, l'arresto del funzionamento del servente avviene attraverso '**mysqladmin**', che richiede l'indicazione di una parola d'ordine.

L'avvio del servente attraverso '**mysqld_safe**' non richiede la comprensione di altre questioni, a parte la necessità di accertarsi che non ci sia già un servente MySQL in funzione. Comunque, la presenza di questo programma ('**mysqld_safe**') fa sì che esista anche

una sezione apposita nella configurazione, denominata nello stesso modo:

```
[mysqld_safe]
err-log         = /var/log/mysql/mysql.err
socket          = /var/run/mysqld/mysqld.sock
```

Il problema dell'arresto del servizio è invece più complesso, in quanto si deve usare un altro programma, '**mysqladmin**', che può portare a termine l'operazione soltanto se si utilizzano i privilegi dell'amministratore del servizio, per ottenere i quali occorre fornire la parola d'ordine relativa:

```
# mysqladmin --password="ciaociao" shutdown [Invio]
```

L'amministratore di MySQL ha il nome '**root**' (in questo caso sarebbe precisamente '**root@localhost**'), che, come accennato nella premessa, coincide volutamente con il nome dell'amministrazione di un sistema Unix, pur non essendo la stessa cosa. Dal momento che ogni utente di MySQL può predisporre nella propria directory personale un file di configurazione personalizzato, corrispondente a '`~/.my.cnf`', in questo file si può anche inserire la propria parola d'ordine, con una direttiva della sezione '**[client]**', che in questo caso è riferita all'utente '**root**':

```
[client]
user      = root
password  = ciaociao
```

Così facendo, come viene chiarito in seguito, quando un utente del sistema operativo accede a un servente MySQL locale, se l'utenza di MySQL coincide con il nominativo usato nell'ambito del sistema operativo, può fare a meno di fornire la propria parola d'ordine perché già definita nella configurazione personale. Secondo questo

principio, l'utente '**root**' del sistema operativo potrebbe fare altrettanto per consentire a script che vengono avviati automaticamente di svolgere il loro compito.

Esistono comunque dei raggiri differenti, per evitare di costringere l'utente '**root**' del sistema operativo a inserire la parola d'ordine nel file di configurazione '`~/.my.cnf`'. In particolare, la distribuzione GNU/Linux Debian definisce un utente speciale, denominato '**debian-sys-maint**' ('**debian-sys-maint@localhost**'), con i privilegi necessari, a cui associa il file di configurazione '`/etc/mysql/debian.cnf`', avviando poi '**mysqladmin**' con l'opzione '**--defaults-extra-file**':

```
# mysqladmin --defaults-extra-file=/etc/mysql/debian.cnf ... [Invio]
```

Naturalmente, il file '`/etc/mysql/debian.cnf`' non deve essere leggibile agli utenti comuni del sistema operativo, dal momento che contiene la parola d'ordine per le funzioni amministrative gestite in modo automatico dal sistema stesso:

```
# Automatically generated for Debian scripts. DO NOT TOUCH!
[client]
host     = localhost
user     = debian-sys-maint
password = sddgetGRFhyjftuP
```

76.1.2.1 Struttura iniziale dei dati

«

Perché il servente MySQL possa essere avviato e funzionare, è necessario che sia stata predisposta una struttura iniziale di dati, che serve successivamente ad accogliere le basi di dati. Si tratta di file di una base di dati speciale, che si colloca assieme alle altre nella directory '`~mysql/`'.

Questa struttura iniziale si crea con il comando '**mysql_install_db**', che corrisponde a uno script. Si usa semplicemente così:

```
# mysql_install_db [Invio]
```

Può succedere che la propria distribuzione GNU, pur predisponendo un pacchetto per il servente MySQL, non provveda alla creazione di questa struttura iniziale dei dati; in tal caso non si riesce ad avviare il servente e occorre provvedere da soli a usare il comando '**mysql_install_db**'.

76.1.3 Utenze

MySQL distingue le proprie utenze in base a un nominativo e al nome dell'elaboratore da cui questi accedono. Il nominativo può anche essere assente e in tal caso si parla di utenze anonime. L'accesso può essere sottoposto al controllo di una parola d'ordine; inoltre, può anche essere consentito l'accesso a una base di dati senza essere utenti conosciuti.

Dal momento che l'utente è sempre riferito a un certo elaboratore, diventa necessario provvedere a un sistema di risoluzione dei nomi, anche se si tratta solo del file '`/etc/hosts`'; è comunque assolutamente indispensabile che sia stato definito il nome '**localhost**', riferito all'indirizzo IPv4 127.0.0.1, per consentire almeno l'accesso all'utente '**root@localhost**' che è l'amministratore quando accede dallo stesso elaboratore che ospita il servente MySQL.

Un primo controllo di accesso si ottiene con la configurazione del servente MySQL, dove è possibile escludere l'accesso attraverso la rete con la direttiva '**skip-networking**', cosa che è utile fare quando il sistema di utenze e dei privilegi relativi non è ancora stato definito.

Per poter iniziare a organizzare le basi di dati e le utenze di MySQL, occorre agire con i privilegi dell'utente '**root**'. Quando si installa MySQL per la prima volta, è molto probabile che questo utente risulti essere sprovvisto di parola d'ordine, consentendo in pratica a qualunque utente di dichiararsi '**root**', senza bisogno di altre forme di riconoscimento (ecco perché inizialmente è bene che la configurazione impedisca almeno l'accesso dall'esterno, attraverso la rete). Per associare una parola d'ordine all'utente '**root**' presso l'elaboratore che ospita il servente MySQL (quindi '**root@localhost**') quando questo ne è ancora sprovvisto, basta usare il comando seguente:

```
# mysqladmin -u root password 'ciaociao' [Invio]
```

Come si intende, la parola d'ordine che appare nell'esempio è proprio «ciaociao».

A partire dal momento in cui la parola d'ordine è stata definita, ogni accesso al servente compiuto da questo utente richiede la sua indicazione, a meno che questa parola d'ordine risulti inserita nel file di configurazione personale '`~/.my.cnf`':

```
[client]
user     = root
password = ciaociao
```

In questo caso, facendo riferimento alla sezione '**[client]**', la con-

figurazione riguarda qualunque situazione in cui si accede al servente; in pratica riguarda qualunque programma. Eventualmente, si possono definire contesti più precisi; per esempio come nel caso seguente, in cui si consente di non specificare la parola d'ordine solo quando si utilizza il programma '**mysqladmin**':

```
[mysqladmin]
user     = root
password = ciaociao
```

Il sistema di gestione delle utenze di MySQL è abbastanza complesso, dove per esempio, è possibile anche concedere dei privilegi di accesso a utenti qualunque provenienti da un certo nodo, o da un certo gruppo di nodi.

Inizialmente dovrebbe essere previsto l'utente '**root@localhost**', come già descritto più volte, ma potrebbe esistere anche un utente '**root@*hostname***', ovvero l'utente '**root**' che accede dal nodo corrispondente al nome che si ottiene con il comando '**hostname**'. Entrambe queste utenze hanno tutti i privilegi possibili ed entrambe potrebbero essere inizialmente senza parola d'ordine.
Assieme alle utenze '**root@**...' potrebbero essere stati definiti dei privilegi di accesso molto limitati per qualunque nominativo-utente che accede dal nodo locale, ovvero '**@localhost**' (ma si rappresenta necessariamente come '**''@localhost**'). Di solito ciò consente di fare degli esperimenti con la base di dati denominata '**test**', prima ancora di avere studiato i criteri di controllo degli accessi.

Per quanto riguarda l'indicazione dei nominativi-utente, esistono solo due possibilità: indicare un nome preciso, oppure non indicarlo affatto. Nel secondo caso si intende fare riferimento a qualunque utente nell'ambito del nodo o del gruppo di nodi a cui la definizione dei privilegi è associata.

Il nodo di accesso può essere indicato per nome, per numero, oppure si può fare riferimento a un gruppo di nodi con l'uso di caratteri jolly oppure associando a un indirizzo una maschera di rete. I caratteri jolly in questione sono il simbolo di percentuale ('`%`') e il trattino basso ('_'). Segue una tabellina che descrive alcuni esempi di associazione tra utenti e nodi di accesso.

utente*@*nodo	Descrizione
tizio@dinkel.brot.dg	L'utente '**`tizio`**' che accede dal nodo *dinkel.brot.dg*.
@dinkel.brot.dg	Qualunque utente che accede dal nodo *dinkel.brot.dg*.
tizio@%	L'utente '**`tizio`**' che accede da qualunque nodo.
@%	Qualunque utente che accede da qualunque nodo.
tizio@%.brot.dg	L'utente '**`tizio`**' che accede da un nodo che appartiene al dominio *brot.dg*.
tizio@brot.brot.%	L'utente '**`tizio`**' che accede da un nodo che appartiene a domini che iniziano per *dinkel.brot.*$*$.
tizio@111.112.113.114	L'utente '**`tizio`**' che accede dal nodo corrispondente all'indirizzo IPv4 111.112.113.114.

utente@*nodo*	Descrizione
tizio@111.112.113.%	L'utente '**tizio**' che accede da nodi con indirizzi IPv4 111.112.113.∗.
tizio@111.112.113.0/255.255.255.0	L'utente '**tizio**' che accede da nodi con indirizzi IPv4 111.112.113.∗ (come nell'esempio precedente).

A seconda del contesto, le coordinate di un utente si indicano come nella tabella, inserendo il carattere '**@**' per separare le due parti, oppure in campi distinti. In molti casi, nell'ambito delle istruzioni di MySQL è necessario proteggere il nome dell'utente e l'indicazione del nodo o dei nodi di accesso attraverso apici singoli. Per esempio, per poter indicare qualunque utente, come nel secondo esempio della tabella, è necessario delimitare la stringa nulla: '**''@dinkel.brot.dg**'. Così, quando si usa il carattere jolly '**%**' è indispensabile delimitare l'indicazione riferita al nodo: '**tizio@'%.brot.dg'**'. In generale non si sbaglia se si delimitano sempre queste due componenti, anche se non dovesse servire: '**''tizio'@'dinkel.brot.dg'**'.

76.1.4 Accesso comune al servente

Prima di poter definire delle politiche di accesso, è necessario prendere un po' di confidenza con il programma '**mysql**', che consente di interagire con il servente MySQL. Si è già accennato al fatto che inizialmente dovrebbe risultare permesso l'accesso da parte di

qualunque utente, inoltre dovrebbe essere disponibile la base di dati '**test**', a cui qualunque utente può accedere.

```
mysql [opzioni] [base_di_dati]
```

```
cat file_sql | mysql [opzioni] [base_di_dati]
```

I modelli sintattici mostrano la possibilità di indicare delle opzioni ed eventualmente il nome di una base di dati da aprire inizialmente. Nel secondo modello, si vede in particolare come si può alimentare il programma '**mysql**' con uno script SQL. La tabella successiva descrive alcune delle opzioni che possono essere utilizzate.

Opzione	Descrizione
`-h` ***nodo*** `--host=`***nodo***	Specifica il nome del nodo a cui ci si vuole connettere.
`-p` [***parola_d'ordine***] `--password` [`=`***parola_d'ordine***]	Specifica la parola d'ordine da fornire per ottenere l'accesso; se non viene indicata come argomento dell'opzione, viene richiesta in modo interattivo. In generale è meglio evitare di fornire la parola d'ordine già nella riga di comando, per non renderla evidente nell'elenco dei processi elaborativi.

Opzione	Descrizione
`-P` ***n_porta*** `--port=`***n_porta***	Specifica la porta da utilizzare per accedere al servizio; in modo predefinito è la numero 3306.
`-u` ***utente*** `--user=`***utente***	Specifica il nominativo-utente con il quale si vuole accedere.

Il programma '**`mysql`**' è uno di quelli che prende in considerazione la configurazione di MySQL, soprattutto per quanto riguarda il file '`~/.my.cnf`', dove gli utenti possono inserire il proprio nominativo-utente da utilizzare con MySQL e la parola d'ordine, per non dover ogni volta utilizzare le opzioni '**`-u`**' e '**`-p`**':

```
[client]
user     = tizio
password = supersegreta
```

Per esempio, se un certo utente del sistema operativo ha la necessità di identificarsi con il nominativo '**`tizio`**' e la sua parola d'ordine:

`$` **`mysql -u tizio -p`** … [*Invio*]

`Enter password:` ***digitazione_all'oscuro*** [*Invio*]

Oppure, disponendo del file '`~/.my.cnf`' mostrato sopra, può fare a meno di queste opzioni e dell'indicazione della parola d'ordine.

Viene mostrato un esempio riferito all'utente '**`tizio`**' che accede a un servente MySQL locale:

`$` **`mysql -u tizio`** [*Invio*]

Molto probabilmente non è necessario inserire alcuna parola d'ordine, dal momento che inizialmente dovrebbe essere stata inserita automaticamente l'utenza anonima '**@localhost**' senza parola d'ordine:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 14 to server version: 4.0.13-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

Da questo momento appare un invito speciale, rappresentato dalla stringa '**mysql>**'. La prima cosa che può essere conveniente fare è verificare la disponibilità di basi di dati a cui si può accedere:

```
mysql> SHOW DATABASES; [Invio]
```

Molto probabilmente si ottiene precisamente quanto segue, ovvero il riferimento all'unica base di dati, '**test**', a cui è consentito accedere con questo utente:

```
+----------+
| Database |
+----------+
| test     |
+----------+
1 row in set (0.03 sec)
```

Si può selezionare la base di dati con il comando seguente:

```
mysql> USE test; [Invio]

Database changed
```

Quindi si può tentare di consultare l'elenco delle relazioni disponibili, con il comando seguente, ma inizialmente la basi di dati '**test**' non ne contiene alcuna:

```
mysql> SHOW TABLES; [Invio]

  Empty set (0.00 sec)
```

A titolo di esempio si vuole creare la relazione '**Indirizzi**' con il codice SQL seguente:

```
CREATE TABLE Indirizzi (
        Codice          INTEGER,
        Cognome         CHAR(40),
        Nome            CHAR(40),
        Indirizzo       VARCHAR(60),
        Telefono        VARCHAR(40)
    );
```

Con il programma '**mysql**' si può fare così:

```
mysql> CREATE TABLE Indirizzi ( [Invio]

    -> Codice INTEGER, [Invio]

    -> Cognome CHAR(40), [Invio]

    -> Nome CHAR(40), [Invio]

    -> Indirizzo VARCHAR(60), [Invio]

    -> Telefono VARCHAR(40) [Invio]

    -> ); [Invio]

  Query OK, 0 rows affected (0.06 sec)
```

Come si può osservare, finché l'istruzione SQL non risulta completa, appare un invito differente: '**->**'.

Per completare l'esempio si può inserire qualche dato e poi si può visualizzare il contenuto della relazione:

```
mysql> INSERT INTO Indirizzi VALUES (01, 'Pallino', 'Pinco', [Invio]
    -> 'Via Biglie 1', '0222,222222'); [Invio]

  Query OK, 1 row affected (0.06 sec)

mysql> SELECT * FROM Indirizzi; [Invio]

  +--------+---------+-------+-------------+-------------+
  | Codice | Cognome | Nome  | Indirizzo   | Telefono    |
  +--------+---------+-------+-------------+-------------+
  |      1 | Pallino | Pinco | Via Biglie 1 | 0222,222222 |
  +--------+---------+-------+-------------+-------------+
  1 row in set (0.00 sec)
```

Per concludere il funzionamento di '**mysql**' basta il comando interno '**quit**', che si può esprimere anche come '**\q**':

```
mysql> \q[Invio]

  Bye
```

76.1.5 Base di dati amministrativa

«

MySQL gestisce le proprie informazioni amministrative all'interno della base di dati '**mysql**', a cui si dovrebbe poter accedere soltanto in qualità di utente '**root**' (possibilmente '**root@localhost**'), o comunque solo attraverso utenze speciali.

```
# mysql -u root -p[Invio]

Enter password: digitazione all'oscuro [Invio]
```

In questo modo si vuole accedere al servente MySQL locale, in qualità di utente '**root**' ('**root@localhost**'), fornendo anche la

parola d'ordine.

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 15 to server version: 4.0.13-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

Si controlla la disponibilità di basi di dati esistenti:

```
mysql> SHOW DATABASES; [Invio]

+----------+
| Database |
+----------+
| mysql    |
| test     |
+----------+
2 rows in set (0.01 sec)
```

Si accede alla base di dati '**mysql**':

```
mysql> USE mysql; [Invio]

Database changed
```

Si elencano le relazioni disponibili:

```
mysql> SHOW TABLES; [Invio]
```

```
+-----------------+
| Tables_in_mysql |
+-----------------+
| columns_priv    |
| db              |
| func            |
| host            |
| tables_priv     |
| user            |
+-----------------+
6 rows in set (0.00 sec)
```

Il significato dettagliato di queste relazioni può essere studiato nella documentazione originale che accompagna MySQL. Ciò che è importante comprendere è che non si può consentire l'accesso a queste relazioni a utenti che non hanno un ruolo amministrativo. Tuttavia, durante la fase di studio di MySQL da parte di chi deve poi amministrarne il servizio, è utile imparare a consultare queste relazioni. Per esempio, è utile essere al corrente delle utenze che sono effettivamente previste:

```
mysql> SELECT User, Host, Password FROM user; [Invio]

+------------------+-----------+------------------+
| user             | host      | password         |
+------------------+-----------+------------------+
| root             | localhost | 576gtd435e967361 |
| root             | dinkel    |                  |
|                  | localhost |                  |
|                  | dinkel    |                  |
| debian-sys-maint | localhost | 69b3c178kbvcd325 |
+------------------+-----------+------------------+
5 rows in set (0.00 sec)
```

Quello che si vede qui è quanto si potrebbe ottenere dopo aver installato MySQL in una distribuzione GNU/Linux Debian, attraverso pacchetti appositi, in un elaboratore dove il comando '**hostname**' restituisce il nome '**dinkel**'. Si può notare la presenza dell'utente speciale '**debian-sys-maint**', a cui si è già accennato, ma si deve portare attenzione alle utenze «normali». Si può osservare che ci sono due utenti '**root**'; precisamente si tratta di '**root@localhost**' e di '**root@dinkel**'. La seconda di queste due utenze rappresenta l'utente '**root**' che accede localmente, ma attraverso la rete, da un'interfaccia che è associata al nome dell'elaboratore (in questo caso *dinkel.brot.dg*). Può anche darsi che un'utenza annotata in questo modo non risulti funzionante, ma rimane il problema, dato il fatto che si tratta di un'utenza importante e senza parola d'ordine per accedere (per quanto riguarda l'utenza '**root@localhost**' è già stato mostrato come definire la parola d'ordine ed è per questo che appare qualcosa nella colonna '**password**'). Eventualmente, si possono fare considerazioni simili per l'utenza anonima '**@dinkel**'.

Probabilmente è superfluo precisare che nella colonna '**password**', se esiste una parola d'ordine associata all'utenza, appare una stringa che rappresenta la parola d'ordine cifrata.

```
mysql> \q[Invio]

  Bye
```

Durante la fase della prima installazione di MySQL uno script si occupa di creare le directory che devono ospitare i file che compongono le basi di dati, inserendo la base di dati amministrativa ('**mysql**'). In pratica, è questo script che definisce anche le utenze iniziali, senza parola d'ordine. Questo script è '**mysql_install_db**'.

76.2 Gestione del DBMS

Una volta superata la fase della configurazione del servizio di MySQL; una volta compreso come utilizzare gli strumenti essenziali per interagire con questo, si può passare alla gestione del DBMS, che implica l'amministrazione delle utenze e delle basi di dati.

76.2.1 Controllo delle utenze e degli accessi

Il modo normale per creare un'utenza con MySQL è quello di concedere dei privilegi di accesso, attraverso l'istruzione '**`GRANT`**'; se poi quell'utenza esiste già, i privilegi in questione vengono solo aggiunti a quelli già esistenti. Per eliminare un'utenza, invece, non è sufficiente privarla di tutti i privilegi con l'istruzione '**`REVOKE`**', perché occorre anche eliminare la tupla corrispondente all'utenza nella relazione '**`user`**' della base di dati '**`mysql`**'.

Il comportamento di MySQL è abbastanza diverso rispetto allo standard ANSI, per quanto riguarda l'identificazione delle utenze e l'attribuzione o l'eliminazione dei privilegi relativi. In particolare, è importante il fatto che MySQL distingua le utenze in base al nodo di provenienza; inoltre è possibile concedere privilegi complessivi: per una base di dati completa, o anche per tutte le basi di dati esistenti. A questo si aggiunga che è possibile fare riferimento a una relazione di una base di dati indicando esplicitamente la base di dati relativa, con la forma: '***base_di_dati*** **.** ***relazione***'.

Gli schemi seguenti mostrano la sintassi semplificata per l'uso delle istruzioni '**`GRANT`**' e '**`REVOKE`**' con MySQL:

```
GRANT privilegio [, privilegio] ...
  ON {relazione | * | base_di_dati.* | *.*}
  TO utenza [IDENTIFIED BY [PASSWORD] 'parola_d'ordine']
    [, utenza [IDENTIFIED BY [PASSWORD] 'parola_d'ordine']] ...
  [WITH GRANT OPTION]
```

```
REVOKE privilegio [, privilegio] ...
    ON {relazione | * | base_di_dati.* | *.*}
    FROM utenza [, utenza] ...
```

I privilegi che possono essere concessi o revocati sono espressi attraverso una parola chiave. Alcuni di questi privilegi sono descritti nell'elenco seguente:

Privilegio	Descrizione
`ALL [PRIVILEGES]`	concede tutti i privilegi disponibili;
`ALTER`	consente la modifica delle relazioni;
`CREATE`	consente la creazione delle relazioni;
`DELETE`	consente la cancellazione dei dati contenuti nelle relazioni;
`DROP`	consente l'eliminazione delle relazioni;
`INDEX`	consente di creare ed eliminare degli indici;
`INSERT`	consente l'inserimento di dati nelle relazioni;

Privilegio	Descrizione
`SELECT`	consente la lettura dei dati nelle relazioni;
`UPDATE`	consente la modifica dei dati all'interno delle relazioni;
`USAGE`	serve solo a creare un utente senza privilegi;
`GRANT OPTION`	consente di dare ad altri i propri privilegi.

Si osservi che MySQL prevede anche privilegi «particolari», che dipendono dalle proprie specificità rispetto allo standard ANSI. In generale, si può considerare che l'utente '**`root`**' deve possedere tutti i privilegi, mentre possono esistere delle utenze amministrative fittizie (come nel caso di '**`debian-sys-maint`**') con privilegi particolari legati alla possibilità di arrestare il funzionamento del servente MySQL.

Lo standard ANSI prevede la concessione di privilegi su relazioni singole, mentre MySQL permette di fare riferimento a basi di dati complete. Pertanto, nel modello sintattico mostrato sono apparse delle notazioni speciali:

Modello	Descrizione
relazione	rappresenta una relazione singola, che può contenere anche l'indicazione della basi di dati che la contiene, nella forma '***base_di_dati . relazione***';
`*`	l'asterisco rappresenta tutte le relazioni della basi di dati attiva, ma se non è stata selezionata una base di dati in precedenza, si fa riferimento a tutte le basi di dati;

Modello	Descrizione
base_di_dati . *	l'indicazione di una base di dati con un asterisco al posto del nome della relazione, serve a fare riferimento a tutta la base di dati nel complesso;
* . *	l'indicazione di due asterischi separati da un punto serve a fare riferimento esplicito a tutte le relazioni di tutte le basi di dati.

Quando si utilizza l'istruzione '**GRANT**' è consentito l'uso dei caratteri jolly '_' e '%', con il significato comune nell'ambito del linguaggio SQL. Ciò consente di fare riferimento a gruppi di relazioni e a gruppi di basi di dati, secondo la corrispondenza del modello. Tuttavia, nel caso questi simboli debbano essere usati in modo letterale, è necessario proteggerli con la barra obliqua inversa: '_' e '\%' (in generale, è improbabile che si dia un nome a una base di dati o a una relazione che contenga il simbolo di percentuale, ma è più probabile che si pensi invece di usare il trattino basso).

Come già accennato altre volte, l'utenza tiene conto anche della provenienza dell'accesso, secondo la forma seguente:

nominativo_utente [@*nodo_di_provenienza*]

Pertanto, l'indicazione del nominativo è obbligatoria, mentre si può omettere la specificazione del nodo di provenienza, se si vuole fare riferimento a qualunque origine.

Sono già stati descritti i modi in cui si può rappresentare un utente secondo il modello '*utente*@*nodo*'; in particolare è già stato descrit-

to l'utilizzo dei caratteri jolly (anche se sono stati mostrati soltanto esempi con il simbolo '`%`').

Si ricorda comunque che si possono usare i caratteri jolly soltanto nella parte che descrive i nodi di provenienza

Infine, è già stata indicata la necessità di usare gli apici singoli per delimitare separatamente il nominativo e la specificazione del nodo di provenienza quando questi nomi contengono caratteri «particolari», compresi i caratteri jolly '`%`' e '_'.

Si possono presentare delle ambiguità nell'individuazione dei privilegi delle utenze. Per esempio, può esistere l'utente anonimo '**`''@dinkel.brot.gd`**', l'utente '**`tizio@'%'`**' e l'utente '**`tizio@dinkel.brot.gd`**': quando un utente accede valgono per lui i privilegi più specifici che gli si possono individuare, altrimenti, in presenza di utenze anonime, i privilegi di queste prevarrebbero.

Come si vede dal modello sintattico dell'istruzione '**`GRANT`**', è possibile specificare una parola d'ordine. Quando si concedono dei privilegi a un utente che non è ancora stato definito e non si stabilisce la parola d'ordine, l'utenza in questione rimane priva di parola d'ordine; pertanto, per accedere non deve essere fornita. Se invece l'utenza esiste già, i privilegi vengono aggiunti e la presenza di una parola d'ordine eventuale serve solo per cambiare quella preesistente. Tuttavia, è possibile cambiare una parola d'ordine anche con l'i-

struzione '**SET PASSWORD**', se si tratta della propria o se si hanno i privilegi dell'amministratore:

```
SET PASSWORD FOR utenza = PASSWORD('parola_d'ordine')
```

In alternativa, è anche possibile usare il comando '**mysqladmin**' (dal sistema operativo), ma solo per cambiare la propria parola d'ordine:

```
$ mysqladmin password 'supersegreta' [Invio]
```

Se si utilizza l'opzione '**GRANT OPTION**', o '**WITH GRANT OPTION**', si permette all'utenza a cui si fa riferimento di concedere ad altri gli stessi privilegi di cui si dispone.

Come accennato, l'eliminazione di un'utenza richiede prima l'eliminazione di tutti i privilegi e quindi la cancellazione dalla relazione '**user**' della base di dati '**mysql**'.

Nel seguito vengono descritti alcuni esempi attraverso una sequenza lineare di operazioni, a cominciare dall'avvio del programma '**mysql**' per interagire con il servente.

```
$ mysql -u root -p [Invio]

Enter password: parola_d'ordine [Invio]

  Welcome to the MySQL monitor.  Commands end with ; or \g.
  Your MySQL connection id is 6 to server version: 4.0.13-log

  Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

Si crea una base di dati nuova:

```
mysql> CREATE DATABASE Magazzino; [Invio]
```

```
Query OK, 1 row affected (0.26 sec)
```

Si crea un utente amministratore per la base di dati appena creata, che può accedere da dove vuole:

```
mysql> GRANT ALL ON Magazzino.* TO amministratore@'%' ↩
↪      IDENTIFIED BY 'segreta' WITH GRANT OPTION; [Invio]

Query OK, 0 rows affected (0.35 sec)
```

Si osservi che se esiste un'utenza anonima riferita al nodo locale (utenza che verrebbe indicata come '**''@localhost**'), se l'utente appena creato volesse accedere localmente, non verrebbe identificato come amministratore, ma solo come utente anonimo. Per risolvere il problema si potrebbe aggiungere l'utente '**amministratore@localhost**', ma in questo caso si preferisce eliminare l'utenza anonima che interferisce e non si vuole mantenere:

```
mysql> REVOKE ALL ON *.* FROM ''@localhost; [Invio]

Query OK, 0 rows affected (0.15 sec)

mysql> USE mysql; [Invio]

Database changed

mysql> DELETE FROM user WHERE user = '' ↩
↪      AND host = 'localhost'; [Invio]

Query OK, 0 rows affected (0.15 sec)
```

Si crea un'altra utenza in grado di consultare e di modificare i dati delle relazioni che deve contenere la base di dati '**Magazzino**'; anche in questo caso si consente l'accesso da qualunque nodo:

```
mysql> GRANT DELETE, INSERT, SELECT, UPDATE ON Magazzino.* ↩
↪      TO tizio@'%' IDENTIFIED BY 'ottimo'; [Invio]

  Query OK, 0 rows affected (0.05 sec)
```

Supponendo di avere installato MySQL nell'elaboratore *dinkel.brot.dg*, dovrebbero essere presenti anche le utenze '**''@dinkel**' e '**root@dinkel**', che si preferisce eliminare:

```
mysql> REVOKE ALL ON *.* FROM ''@dinkel; [Invio]

  Query OK, 0 rows affected (0.15 sec)

mysql> REVOKE ALL ON *.* FROM root@dinkel; [Invio]

  Query OK, 0 rows affected (0.15 sec)

mysql> USE mysql; [Invio]

  Database changed

mysql> DELETE FROM user WHERE user = '' ↩
↪       AND host = 'dinkel'; [Invio]

  Query OK, 0 rows affected (0.15 sec)

mysql> DELETE FROM user WHERE user = 'root' ↩
↪       AND host = 'dinkel'; [Invio]

  Query OK, 0 rows affected (0.15 sec)

mysql> \q [Invio]

  Bye
```

La documentazione di MySQL descrive anche come compiere tutte queste operazioni amministrative intervenendo esclusivamente nella base di dati '**mysql**'. Nel caso si preferisca intervenire in

quel modo, è necessario concludere le operazioni di modifica o aggiornamento delle relazioni amministrative con l'istruzione '**`FLUSH PRIVILEGES`**'.

76.2.2 Amministrazioni varie attraverso il sistema operativo

«

In questo capitolo si è fatto riferimento più volte al programma '**`mysqladmin`**' a proposito della possibilità di assegnare e modificare la parola d'ordine di un utente. Questo programma ha anche altre funzionalità; in particolare consente di creare ed eliminare una base di dati e di arrestare il funzionamento del servente MySQL, senza bisogno di utilizzare istruzioni SQL. Vengono sintetizzate le sintassi da utilizzare per le varie occasioni:

Comando	Descrizione

`mysqladmin [-u root] [-p] ↩`
`↪[-h` ***nodo*** `] ↩`
`↪create database` ***base_di_dati***
	crea la base di dati indicata;

`mysqladmin [-u root] [-p] ↩`
`↪[-h` ***nodo*** `] ↩`
`↪drop database` ***base_di_dati***
	elimina la base di dati indicata con tutto il suo contenuto;

`mysqladmin [-u` ***utente*** `] [-p] ↩`
`↪[-h` ***nodo*** `] ↩`
`↪password` ***parola_d'ordine***
	assegna o cambia la parola d'ordine per accedere;

`mysqladmin [-u root] [-p] ↩` `↪[-h` ***nodo*** `] shutdown`	arresta il funzionamento del servente.

76.2.3 Ripristino della parola d'ordine dell'amministratore

Nel caso fosse necessario modificare la parola d'ordine dell'amministratore del DBMS ('**root**'), senza poter conoscere quella precedente, esiste un procedimento che è bene annotare. Per prima cosa si deve arrestare il servente, nel caso questo fosse in funzione; dovrebbe essere possibile farlo così:

```
# /etc/init.d/mysql stop[Invio]
```

Successivamente si deve avviare '**mysqld_safe**', con l'opzione '**--skip-grant-tables**', in modo da ignorare completamente il controllo dei privilegi concessi (o negati) agli utenti del DBMS:

```
# mysqld_safe --skip-grant-tables &[Invio]
```

Se il servizio si avvia regolarmente, è possibile accedere alle basi di dati senza vincoli; pertanto è possibile cambiare parola d'ordine intervenendo direttamente nella base di dati amministrativa '**mysql**':

```
# mysql -u root[Invio]

  Welcome to the MySQL monitor.  Commands end with ; or \g.
  Your MySQL connection id is 2 to server version: ↩
↪4.0.24_Debian-10-log

  Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE mysql;[Invio]

  Reading table information for completion of table and column
  names. You can turn off this feature to get a quicker startup
  with -A

  Database changed
```

```
mysql> UPDATE user SET password=password("supersegreta") ↩
↪      WHERE user="root"; [Invio]

  Query OK, 2 rows affected (0.00 sec)
  Rows matched: 2  Changed: 2  Warnings: 0

mysql> \q[Invio]

  Bye
```

Un modo alternativo, ma drastico, di ripristinare l'utenza dell'amministratore senza parola d'ordine, consiste nell'eliminazione «manuale» della base di dati '**mysql**', da ricostruire con l'aiuto di '**mysql_install_db**'. Naturalmente, in questo modo si perdono le informazioni su tutti gli altri utenti del DBMS.

76.2.4 Archiviazione e recupero delle basi di dati

«

MySQL gestisce le sue basi di dati come sottodirectory di '`~mysql/`', che di solito corrisponde a '`/var/lib/mysql/`'. Per esempio, la base di dati '**prova**' corrisponde alla struttura che si articola a partire da '`~mysql/prova/`'.

Per archiviare una base di dati è sufficiente fare una copia della struttura che la riguarda; per ripristinare una base di dati è sufficiente rimpiazzare la struttura esistente con la sua copia fatta in precedenza; per duplicare una base di dati è sufficiente fare la copia della struttura di origine, utilizzando un nome differente. Per esempio, disponendo della base di dati '**prova**', è possibile creare un'altra identica, ma con nome differente, così:

```
# cp -dpRv ~mysql/prova ~mysql/prova2[Invio]
```

In base all'esempio si ottiene una seconda base di dati con il nome '**prova2**', con lo stesso contenuto di '**prova**'.

Tuttavia, con il procedere dello sviluppo di MySQL bisogna considerare la possibilità che il formato dei file usati per descrivere le relazioni delle basi di dati cambi nel tempo. Pertanto, per archiviare una base di dati in modo abbastanza duraturo, è necessario creare un file di testo contenente comandi SQL. Si ottiene questo con il programma '**mysqldump**':

```
# mysqldump -u root -p prova > prova.sql [Invio]
```

L'esempio mostra in che modo archiviare la base di dati '**prova**' nel file '`prova.sql`'. Per fare questo, logicamente, ci si identifica in qualità di amministratore (con il nominativo '**root**') e nell'esempio si usa l'opzione '**-p**', per richiedere l'inserimento della parola d'ordine, supponendo che ciò sia necessario.

Per ripristinare un file ottenuto in questo modo, occorre prima creare o ricreare la base di dati; nell'esempio seguente si cancella prima la base di dati '**prova**', quindi la si ricrea vuota:

```
# mysql -u root -p [Invio]

Enter password: digitazione_all'oscuro [Invio]

  Welcome to the MySQL monitor.  Commands end with ; or \g.
  Your MySQL connection id is 6 to server version: 4.0.13-log

  Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> DROP DATABASE prova; [Invio]

  Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CREATE DATABASE prova; [Invio]

  Query OK, 1 row affected (0.00 sec)

mysql> \q[Invio]

  Bye
```

Per recuperare la base di dati archiviata in forma di comandi SQL, dopo un controllo visivo del suo contenuto, si può procedere così:

```
# mysql -u root -p prova < prova.sql [Invio]

Enter password: digitazione_all'oscuro [Invio]
```

In questo modo, si fa leggere a '**mysql**' il contenuto del file '`prova.sql`', che dovrebbe contenere le istruzioni per la rigenerazione delle relazioni della base di dati originaria.

Si osservi che nella riga di comando di '**mysql**' appare l'indicazione della base di dati di destinazione; pertanto, si potrebbe benissimo ricreare una base di dati con un nome differente da quello che aveva nel momento dell'archiviazione in forma di comandi SQL. Di conseguenza, questa tecnica di archiviazione e recupero è anche quella più appropriata per duplicare una base di dati.

76.3 Riferimenti

«

- *MySQL reference manual*, http://dev.mysql.com/doc/mysql/en/index.html

[1] **MySQL** GNU GPL

77.1 Utilizzo generale

«

SQLite[1] è una libreria in grado di fornire le funzionalità di un DBMS relazionale, basato sul linguaggio SQL, utilizzando come basi di dati dei file singoli. In pratica, SQLite gestisce una basi di dati intera in un file singolo, senza alcuna amministrazione esterna; pertanto, le utenze e i privilegi sono definiti dal sistema operativo, in quanto gli accessi sono regolati dai permessi del file che contiene la basi di dati.

In qualità di libreria, SQL consente ai programmi che la utilizzano di avere accesso a una basi di dati anche senza bisogno di collegarsi a un servizio DBMS tradizionale, facilitando così la realizzazione di piccoli progetti.

Di norma, la libreria SQL è accompagnata da un programma per l'esecuzione interattiva di istruzioni SQLite, che consente di accedere alle sue basi di dati in modo generalizzato.

77.1.1 Utilizzo di «sqlite3»

«

Per creare o accedere a una basi di dati di SQL, si può usare il programma '**`sqlite3`**', o '**`sqlite`**' nelle versioni più vecchie, che accompagna normalmente la libreria vera e propria:

```
sqlite3 [opzioni] [file_base_di_dati]
```

Generalmente, alla fine della riga di comando si indica il nome del file base di dati a cui si vuole accedere; se poi il file non dovesse esistere, il fatto di nominarlo implicherebbe comunque la sua creazione (come base di dati vuota). Se il nome di questo file non viene fornito attraverso la riga di comando, occorre usare un'istruzione apposita,

durante il funzionamento di '**sqlite**'. L'esempio seguente mostra l'avvio e la conclusione del programma, allo scopo di accedere alla base di dati contenuta nel file '`mia_prova.db`':

```
$ sqlite3 mia_prova.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite3> .quit[Invio]
```

Come si può intendere, si tratta di un programma che si comporta sostanzialmente come altri programmi frontali simili, usati per accedere in modo diretto ad altri DBMS basati su SQL.

Tabella 77.2. Alcune opzioni per l'avvio di '**sqlite3**'.

Opzione	Descrizione
`-init` ***file***	Consente di indicare un file contenente comandi per '**sqlite3**'; può trattarsi di comandi interni al programma, oppure di istruzioni SQL.
`-column`	Richiede di mostrare i risultati in modo incolonnato.
`-html`	Richiede di generare i risultati delle interrogazioni in forma di tabella HTML.

Oltre all'uso dell'opzione '**-init**', il programma '**sqlite3**' può ricevere un file contenente comandi e istruzioni SQL semplicemente dallo standard input.

Tabella 77.3. Alcuni comandi interni di '**sqlite3**'.

Comando	Descrizione
`.help`	Mostra una guida interna all'uso del programma.
`.exit` `.quit`	Termina il funzionamento del programma.
`.dump` [*relazione*] ...	Consente di «scaricare» il contenuto della base di dati, oppure delle sole relazioni indicate, generando le istruzioni necessarie a ricostruire le informazioni relative.
`.read` *file*	Legge ed esegue i comandi contenuti nel file indicato.
`.mode column`	Fa in modo di visualizzare il risultato delle interrogazioni in una forma incolonnata.
`.mode list`	Fa in modo di visualizzare il risultato delle interrogazioni nel modo usuale di '**sqlite3**'.
`.header on` `.header off`	attiva o disattiva l'intestazione delle colonne.

77.1.2 Copie di sicurezza

«

È evidente che la copia di una base di dati realizzata con SQLite è un'operazione molto semplice, essendo tutto contenuto in un file. Tuttavia, per garantire la trasferibilità in un'architettura differente, è necessario procedere alla generazione di un file di testo contenente le istruzioni SQL con cui poi ricostruire la base di dati. Si ottiene lo scarico in questa forma usando il comando '**.dump**':

```
echo ".dump" | sqlite3 file_base_di_dati > file_sql
```

Per esempio, per generare il file ‘`backup.sql`’ dalla base di dati contenuta nel file ‘`mio.db`’, si potrebbe usare il comando seguente:

```
$ echo ".dump" | sqlite3 mio.db > backup.sql [Invio]
```

Quello che si ottiene nel file ‘`backup.sql`’ potrebbe avere l’aspetto seguente:

```
BEGIN TRANSACTION;
CREATE TABLE Indirizzi (codice integer, cognome char(40), ↩
↪nome char(40), indirizzo varchar(60), telefono varchar(40));
INSERT INTO "Indirizzi" VALUES(1, 'Pallino', 'Pinco', ↩
↪'Via Biglie, 1', '0222,222222');
…
CREATE TABLE presenze (codice integer, giorno date, ↩
↪ingresso time, uscita time);
INSERT INTO "presenze" VALUES(1, '2005-09-17', '11:20:00', '13:30:00');
…
CREATE VIEW ingressi as select indirizzi.cognome, indirizzi.nome, ↩
↪presenze.giorno, presenze.ingresso ↩
↪from indirizzi, presenze ↩
↪where indirizzi.codice = presenze.codice;
COMMIT;
```

Per ricostruire una base di dati da un file del genere, basta fornire il file a ‘**`sqlite3`**’ dallo standard input:

```
sqlite3 file_base_di_dati < file_sql
```

Per esempio, per creare una base di dati nuova nel file ‘`nuova.db`’, a partire dallo stesso file appena creato, si potrebbe usare il comando seguente:

```
$ sqlite3 nuova.db < backup.sql [Invio]
```

77.1.3 Accesso simultaneo a più basi di dati

«

SQLite consente di accedere a più di una basi di dati alla volta. Per fare questo è disponibile un'istruzione SQL che non è standard:

```
ATTACH [DATABASE] file_base_di_dati AS nome_interno
```

In pratica, si indica il file della basi di dati a cui collegarsi, tenendo conto che può essere necessario indicarlo tra virgolette, ma le si deve associare un nome. Nell'esempio seguente, durante il funzionamento di '**sqlite3**' viene collegata la basi di dati contenuta nel file '`seconda.db`', associandola al nome '**seconda**':

```
sqlite> ATTACH 'seconda.db' AS seconda; [Invio]
```

La base di dati già aperta all'atto dell'avvio di '**sqlite3**' acquista automaticamente il nome interno '**main**'. La distinzione delle basi di dati con questi nomi interni, consente di accedere a relazioni che altrimenti avrebbero lo stesso nome: per indicare la relazione '**indirizzi**' della base di dati '**seconda**', si indica '**seconda.indirizzi**'; per indicare la relazione con lo stesso nome, contenuta nella base di dati aperta all'avvio del programma, si indica '**main.indirizzi**'. Comunque, si osservi che non è necessario indicare dei nomi completi della base di dati se non ci sono ambiguità tra le relazioni.

Eventualmente, per chiudere il collegamento con una base di dati c'è un'altra istruzione:

```
DETACH [DATABASE] nome_interno
```

77.2 Esempi comuni

Nelle sezioni seguenti vengono mostrati alcuni esempi comuni di utilizzo del linguaggio SQL, limitato alle possibilità di SQLite. La sintassi non viene descritta. Negli esempi si fa riferimento frequentemente a una relazione di indirizzi, il cui contenuto è visibile nella figura successiva.

Figura 77.5. La relazione '**`Indirizzi (Codice, Cognome, Nome, Indirizzo, Telefono)`**' usata negli esempi del capitolo.

Indirizzi				
Codice	**Cognome**	**Nome**	**Indirizzo**	**Telefono**
1	Pallino	Pinco	Via Biglie 1	0222,222222
2	Tizi	Tizio	Via Tazi 5	0555,555555
3	Cai	Caio	Via Caini 1	0888,888888
4	Semproni	Sempronio	Via Sempi 7	0999,999999

77.2.1 Creazione di una relazione

La relazione di esempio, denominata '**`Indirizzi`**', potrebbe essere creata nel modo seguente:

```
sqlite> CREATE TABLE Indirizzi ( [Invio]
   ...>         Codice          integer, [Invio]
   ...>         Cognome         char(40), [Invio]
   ...>         Nome            char(40), [Invio]
```

```
...>          Indirizzo       varchar(60), [Invio]

...>          Telefono        varchar(40) [Invio]

...>      ); [Invio]
```

77.2.2 Modifica della relazione

SQLite consente soltanto l'aggiunta di attributi alle relazioni, mentre la loro eliminazione o il cambiamento del dominio (il tipo), non è ammissibile. Segue un esempio con cui si aggiunge un attributo:

```
sqlite> ALTER TABLE Indirizzi ADD COLUMN Comune char(30);
[Invio]
```

77.2.3 Inserimento dati in una relazione

L'esempio seguente mostra l'inserimento dell'indirizzo dell'impiegato «Pinco Pallino»:

```
sqlite> INSERT INTO Indirizzi VALUES ([Invio]

...>          01, [Invio]

...>          'Pallino', [Invio]

...>          'Pinco', [Invio]

...>          'Via Biglie 1', [Invio]

...>          '0222,222222'); [Invio]
```

In questo caso, si presuppone che i valori inseriti seguano la sequenza degli attributi, così come è stata creata la relazione in origine; tuttavia, si osservi che se la relazione ha degli attributi in più, si ottiene una segnalazione di errore e l'inserimento viene rifiutato.

Per indicare un comando più leggibile, evitando anche problemi quando dovessero esserci attributi ulteriori, che però non si vogliono prendere in considerazione, occorre aggiungere l'indicazione della sequenza degli attributi da compilare, come nell'esempio seguente:

```
sqlite> INSERT INTO Indirizzi (Codice, Cognome, Nome, [Invio]
   ...>                        Indirizzo, Telefono) [Invio]
   ...>        VALUES (01, 'Pallino', 'Pinco', [Invio]
   ...>                'Via Biglie 1', '0222,222222'); [Invio]
```

In questo modo, gli attributi che non vengono indicati, ricevono il valore predefinito (se esiste), oppure '**NULL**' in mancanza d'altro.

77.2.4 Eliminazione di una relazione

Una relazione può essere eliminata completamente attraverso l'istruzione '**DROP**'. L'esempio seguente elimina la relazione degli indirizzi degli esempi già mostrati:

```
sqlite> DROP TABLE Indirizzi; [Invio]
```

77.2.5 Interrogazioni semplici

L'esempio seguente emette tutto il contenuto della relazione degli indirizzi già vista negli esempi precedenti:

```
sqlite> SELECT * FROM Indirizzi; [Invio]
```

Il risultato può apparire in formati differenti; di solito si ottiene così:

```
1|Pallino|Pinco|Via Biglie 1|0222,222222
2|Tizi|Tizio|Via Tazi 5|0555,555555
3|Cai|Caio|Via Caini 1|0888,888888
4|Semproni|Sempronio|Via Sempi 7|0999,999999
```

Per ottenere un elenco incolonnato, occorre usare il comando '**.mode column**', ma in tal caso l'ampiezza delle colonne è fissa e le informazioni potrebbero apparire troncate:

```
sqlite> .mode column[Invio]

sqlite> SELECT * FROM Indirizzi;[Invio]

1          Pallino     Pinco       Via Biglie 1   0222,222222
2          Tizi        Tizio       Via Tazi 5     0555,555555
3          Cai         Caio        Via Caini 1    0888,888888
4          Semproni    Sempronio   Via Sempi 7    0999,999999
```

Per visualizzare anche l'intestazione delle colonne che appaiono, occorre utilizzare il comando '**.header on**':

```
sqlite> .header on[Invio]

sqlite> SELECT * FROM Indirizzi;[Invio]

Codice    Cognome     Nome        Indirizzo      Telefono
--------  ----------  ----------  -------------  -----------
1         Pallino     Pinco       Via Biglie 1   0222,222222
2         Tizi        Tizio       Via Tazi 5     0555,555555
3         Cai         Caio        Via Caini 1    0888,888888
4         Semproni    Sempronio   Via Sempi 7    0999,999999
```

Per ottenere un elenco ordinato in base al cognome e al nome (in caso di ambiguità), lo stesso comando si completa nel modo seguente:

```
sqlite> SELECT * FROM Indirizzi ORDER BY Cognome, Nome;[Invio]

  Codice    Cognome     Nome         Indirizzo      Telefono
  --------  ----------  ----------  -------------  -----------
  3         Cai         Caio        Via Caini 1    0888,888888
  1         Pallino     Pinco       Via Biglie 1   0222,222222
  4         Semproni    Sempronio   Via Sempi 7    0999,999999
  2         Tizi        Tizio       Via Tazi 5     0555,555555
```

La selezione degli attributi permette di ottenere un risultato che contenga solo quelli desiderati, permettendo anche di cambiarne l'intestazione. L'esempio seguente permette di mostrare solo i nominativi e il telefono, cambiando un po' le intestazioni:

```
sqlite> SELECT Cognome as cognomi, Nome as nomi,[Invio]

   ...>         Telefono as numeri_telefonici[Invio]

   ...>         FROM Indirizzi;[Invio]
```

Quello che si ottiene è simile all'elenco seguente:

```
  cognomi     nomi        numeri_telefonici
  ----------  ----------  -----------------
  Pallino     Pinco       0222,222222
  Tizi        Tizio       0555,555555
  Cai         Caio        0888,888888
  Semproni    Sempronio   0999,999999
```

La selezione delle tuple può essere fatta attraverso la condizione che segue la parola chiave '**WHERE**'. Nell'esempio seguente vengono selezionate le tuple in cui l'iniziale dei cognomi è compresa tra '**N**' e '**T**':

```
sqlite> SELECT * FROM Indirizzi[Invio]
```

```
   ...>        WHERE Cognome >= 'N' AND Cognome <= 'T'; [Invio]
```

Dall'elenco che si ottiene, si osserva che '**Caio**' è stato escluso:

```
Codice     Cognome      Nome         Indirizzo       Telefono
--------   ----------   ----------   -------------   -----------
1          Pallino      Pinco        Via Biglie 1    0222,222222
2          Tizi         Tizio        Via Tazi 5      0555,555555
4          Semproni     Sempronio    Via Sempi 7     0999,999999
```

Per evitare ambiguità possono essere indicati i nomi degli attributi prefissati dal nome della relazione a cui appartengono, separando le due parti con l'operatore punto ('.'). Nell'esempio seguente si selezionano solo il cognome, il nome e il numero telefonico, specificando il nome della relazione a cui appartengono gli attributi:

```
sqlite> SELECT Indirizzi.Cognome, Indirizzi.Nome,
Indirizzi.Telefono [Invio]

   ...>        FROM Indirizzi; [Invio]
```

Ecco il risultato:

```
Cognome      Nomi         Telefono
----------   ----------   -----------
Pallino      Pinco        0222,222222
Tizi         Tizio        0555,555555
Cai          Caio         0888,888888
Semproni     Sempronio    0999,999999
```

77.2.6 Interrogazioni simultanee di più relazioni

«

Se dopo la parola chiave '**FROM**' si indicano più relazioni (ciò vale anche se si indica più volte la stessa relazione), si intende fare riferimento a una relazione generata dal «prodotto» di queste. Si immagi-

ni di abbinare alla relazione '**Indirizzi**' la relazione '**Presenze**' contenente i dati visibili nella figura 77.13.

Figura 77.13. La relazione '**Presenze (Codice, Giorno, Ingresso, Uscita)**'.

Presenze			
Codice	**Giorno**	**Ingresso**	**Uscita**
1	01/01/2012	07:30	13:30
2	01/01/2012	07:35	13:37
3	01/01/2012	07:45	14:00
4	01/01/2012	08:30	16:30
1	01/02/2012	07:35	13:38
2	01/02/2012	08:35	14:37
4	01/02/2012	07:30	13:30

Ecco le istruzioni per crearla e per inserire la prima tupla dell'esempio:

```
sqlite> CREATE TABLE Presenze (Codice INTEGER, Giorno DATE,
[Invio]

   ...>                              Ingresso TIME, USCITA Time);
[Invio]

sqlite> INSERT INTO Presenze[Invio]

   ...>     VALUES (1, '2012-01-01', '07:30', '13:30');[Invio]
```

Come si può intendere, il primo attributo, '**Codice**', serve a identificare la persona per la quale è stata fatta l'annotazione dell'ingresso

e dell'uscita. Tale codice viene interpretato in base al contenuto della relazione '**Indirizzi**'. Si immagini di volere ottenere un elenco contenente tutti gli ingressi e le uscite, indicando chiaramente il cognome e il nome della persona a cui si riferiscono.

```
sqlite> SELECT Presenze.Giorno, Presenze.Ingresso,
Presenze.Uscita, [Invio]
   ...>        Indirizzi.Cognome, Indirizzi.Nome[Invio]
   ...>        FROM Presenze, Indirizzi[Invio]
   ...>        WHERE Presenze.Codice = Indirizzi.Codice; [Invio]
```

Ecco quello che si dovrebbe ottenere:

```
giorno      ingresso    uscita      cognome     nome
----------  ----------  ----------  ----------  ----------
01-01-2012    07:30:00    13:30:00    Pallino     Pinco
01-01-2012    07:35:00    13:37:00    Tizi        Tizio
01-01-2012    07:45:00    14:00:00    Cai         Caio
01-01-2012    08:30:00    16:30:00    Semproni    Sempronio
01-02-2012    07:35:00    13:38:00    Pallino     Pinco
01-02-2012    08:35:00    14:37:00    Tizio       Tizi
01-02-2012    07:40:00    13:30:00    Semproni    Sempronio
```

77.2.7 Alias

«

Una stessa relazione può essere presa in considerazione come se si trattasse di due o più relazioni differenti. Per distinguere tra questi punti di vista diversi, si devono usare degli alias, che sono in pratica dei nomi alternativi. Gli alias si possono usare anche solo per questioni di leggibilità. L'esempio seguente è la semplice ripetizione di quello mostrato nella sezione precedente, con l'aggiunta però della definizione degli alias '**Pre**' e '**Nom**'.

```
sqlite> SELECT Pre.Giorno, Pre.Ingresso, Pre.Uscita, [Invio]
   ...>      Nom.Cognome, Nom.Nome [Invio]
   ...>      FROM Presenze AS Pre, Indirizzi AS Nom [Invio]
   ...>      WHERE Pre.Codice = Nom.Codice; [Invio]
```

77.2.8 Viste

Attraverso una vista, è possibile definire una relazione virtuale:

```
sqlite> CREATE VIEW Presenze_dettagliate AS [Invio]
   ...>      SELECT Presenze.Giorno, Presenze.Ingresso, [Invio]
   ...>             Presenze.Uscita, [Invio]
   ...>             Indirizzi.Cognome, Indirizzi.Nome [Invio]
   ...>          FROM Presenze, Indirizzi [Invio]
   ...>          WHERE Presenze.Codice = Indirizzi.Codice;
[Invio]
```

L'esempio mostra la creazione della vista '**Presenze_dettagliate**', ottenuta dalle relazioni '**Presenze**' e '**Indirizzi**'. In pratica, questa vista permette di interrogare direttamente la relazione virtuale '**Presenze_dettagliate**', invece di utilizzare ogni volta un comando '**SELECT**' molto complesso, per ottenere lo stesso risultato.

77.2.9 Aggiornamento delle tuple

La modifica di tuple già esistenti avviene attraverso l'istruzione '**UPDATE**', la cui efficacia viene controllata dalla condizione posta

dopo la parola chiave '**WHERE**'. Se tale condizione manca, l'effetto delle modifiche si riflette su tutte le tuple della relazione.

L'esempio seguente, aggiunge un attributo alla relazione degli indirizzi, per contenere il nome del comune di residenza degli impiegati; successivamente viene inserito il nome del comune '**Sferopoli**' in base al prefisso telefonico.

```
sqlite> ALTER TABLE Indirizzi ADD COLUMN Comune char(30);
[Invio]

sqlite> UPDATE Indirizzi[Invio]

   ...>     SET Comune='Sferopoli'[Invio]

   ...>     WHERE Telefono >= '022' AND Telefono < '023';
[Invio]
```

In pratica, viene aggiornata solo la tupla dell'impiegato '**Pinco Pallino**'.

77.2.10 Cancellazione delle tuple

«

L'esempio seguente elimina dalla relazione delle presenze le tuple riferite alle registrazioni del giorno 01/01/2012 e le eventuali antecedenti.

```
sqlite> DELETE FROM Presenze WHERE Giorno <= '2012-01-01';
[Invio]
```

77.2.11 Inserimento in una relazione esistente

«

L'esempio seguente aggiunge alla relazione dello storico delle presenze le registrazioni vecchie che poi vengono cancellate:

```
sqlite> INSERT INTO PresenzeStorico ([Invio]
```

```
   ...>          PresenzeStorico.Codice, [Invio]
   ...>          PresenzeStorico.Giorno, [Invio]
   ...>          PresenzeStorico.Ingresso, [Invio]
   ...>          PresenzeStorico.Uscita) [Invio]
   ...>      SELECT Presenze.Codice, Presenze.Giorno, [Invio]
   ...>             Presenze.Ingresso, Presenze.Uscita [Invio]
   ...>          FROM Presenze [Invio]
   ...>          WHERE Presenze.Giorno <= '2012-01-01'; [Invio]
sqlite> DELETE FROM Presenze [Invio]
   ...>         WHERE Giorno <= '2012-01-01'; [Invio]
```

77.3 Riferimenti

«

- *SQLite*, *http://www.sqlite.org*

[1] **SQLite** dominio pubblico

ODBC

ODBC, ovvero *Open database connectivity* è un metodo standardizzato per l'accesso ai DBMS. Si attua inserendo un servizio intermedio, tra i DBMS e le applicazioni che devono accedere ai dati: le applicazioni comunicano con il servizio ODBC, mentre il servizio ODBC comunica con i DBMS sottostanti, preoccupandosi di adattarsi alle loro particolarità. Ciò consente di scrivere applicazioni che, attraverso ODBC, sono in grado di utilizzare qualsiasi DBMS con cui il servizio ODBC sia in grado di interagire.

Generalmente, il sistema ODBC non fa tutto da solo, in quanto di solito si avvale di librerie aggiuntive (*driver*) per la gestione dei DBMS reali.

78.1 DSN

Un DSN, ovvero *Data source name*, è una base di dati virtuale, del sistema ODBC, individuata da un nome.

Generalmente, un DSN fa riferimento a una base di dati di un certo DBMS reale, con cui il sistema ODBC è in grado di interagire; tuttavia, teoricamente, potrebbe trattarsi di qualunque cosa in grado di comportarsi come una base di dati vera e propria.

78.2 unixODBC

Nei sistemi Unix è disponibile unixODBC,[1] che consente di interagire con un discreto numero di DBMS comuni, attraverso delle librerie aggiuntive per la gestione dei vari DBMS, alcune delle quali vengono elencate qui brevemente:

- PostgreSQL ODBC,[2] per la comunicazione con DBMS PostgreSQL;
- MyODBC,[3] per la comunicazione con DBMS MySQL.

Una volta installato unixODBC e le librerie per la comunicazione con i DBMS di proprio interesse, occorre provvedere a configurare unixODBC in modo da poterle utilizzare. Per fare questo si interviene nel file '`/etc/odbcinst.ini`'; l'esempio seguente riguarda le librerie per comunicare con MySQL e PostgreSQL rispettivamente:

```
[MySQL]
Description     = MySQL driver
Driver          = /usr/lib/odbc/libmyodbc.so
Setup           = /usr/lib/odbc/libodbcmyS.so
CPTimeout       =
CPReuse         =
FileUsage       = 1

[PostgreSQL]
Description     = PostgreSQL ODBC driver
Driver          = /usr/lib/odbc/psqlodbc.so
Setup           = /usr/lib/odbc/libodbcpsqlS.so
Debug           = 0
CommLog         = 1
FileUsage       = 1
```

Come si può vedere e intuire, alcune direttive non sono comuni per tutti i tipi di DBMS; in pratica, l'inserimento di una sezione per l'accesso a un DBMS richiede un modello da copiare e adattare, per quel caso specifico.

Una volta configurate le librerie per l'accesso ai DBMS, è possibile

definire dei DSN, con i quali fare riferimento a delle basi di dati reali presso tali DBMS. unixODBC offre tre modi per memorizzare le informazioni sui DSN, in base al campo di azione previsto per questi: il sistema nel suo insieme, una rete locale, il singolo utente.

Attraverso il file '`/etc/odbc.ini`' è possibile dichiarare dei DSN validi nell'ambito del sistema locale, per tutti gli utenti; attraverso dei file che corrispondono al modello '`/etc/ODBCDataSources/`***nome***`.dsn`', è possibile dichiarare un DSN che, teoricamente, potrebbe essere condiviso da più elaboratori in una rete locale, con la stessa condivisione della directory '`/etc/ODBCDataSources/`'; attraverso i file '`~/.odbc.ini`', ogni utente può dichiarare i propri DSN personali.

Questi file di configurazione dei DSN, possono contenere direttive raggruppate in sezioni, corrispondenti al nome del DSN. Per esempio, l'estratto seguente dichiara l'accesso alla base di dati, presso l'elaboratore locale, denominata '**`nanodb`**', accedendo con l'utenza '**`pgnanouser`**' (secondo il DBMS); il nome del DSN è '**`mio`**':

```
[mio]
Description             = PostgreSQL
Driver                  = PostgreSQL
Trace                   = No
TraceFile               =
Database                = nanodb
Servername              = localhost
Username                = pgnanouser
Password                =
Port                    = 5432
Protocol                = 6.4
ReadOnly                = No
RowVersioning           = No
```

```
ShowSystemTables        = No
ShowOidColumn           = No
FakeOidIndex            = No
ConnSettings            =
```

L'estratto seguente, invece, dichiara l'accesso alla base di dati, presso l'elaboratore locale, denominata '**nanodb**', accedendo con l'utenza '**mynanouser**' (secondo il DBMS); il nome del DSN è '**mio2**':

```
[mio2]
Description     = MySQL
Driver          = MySQL
Server          = localhost
Database        = nanodb
Username        = mynanouser
Port            =
Socket          =
Option          =
Stmt            =
```

78.3 ODBCConfig

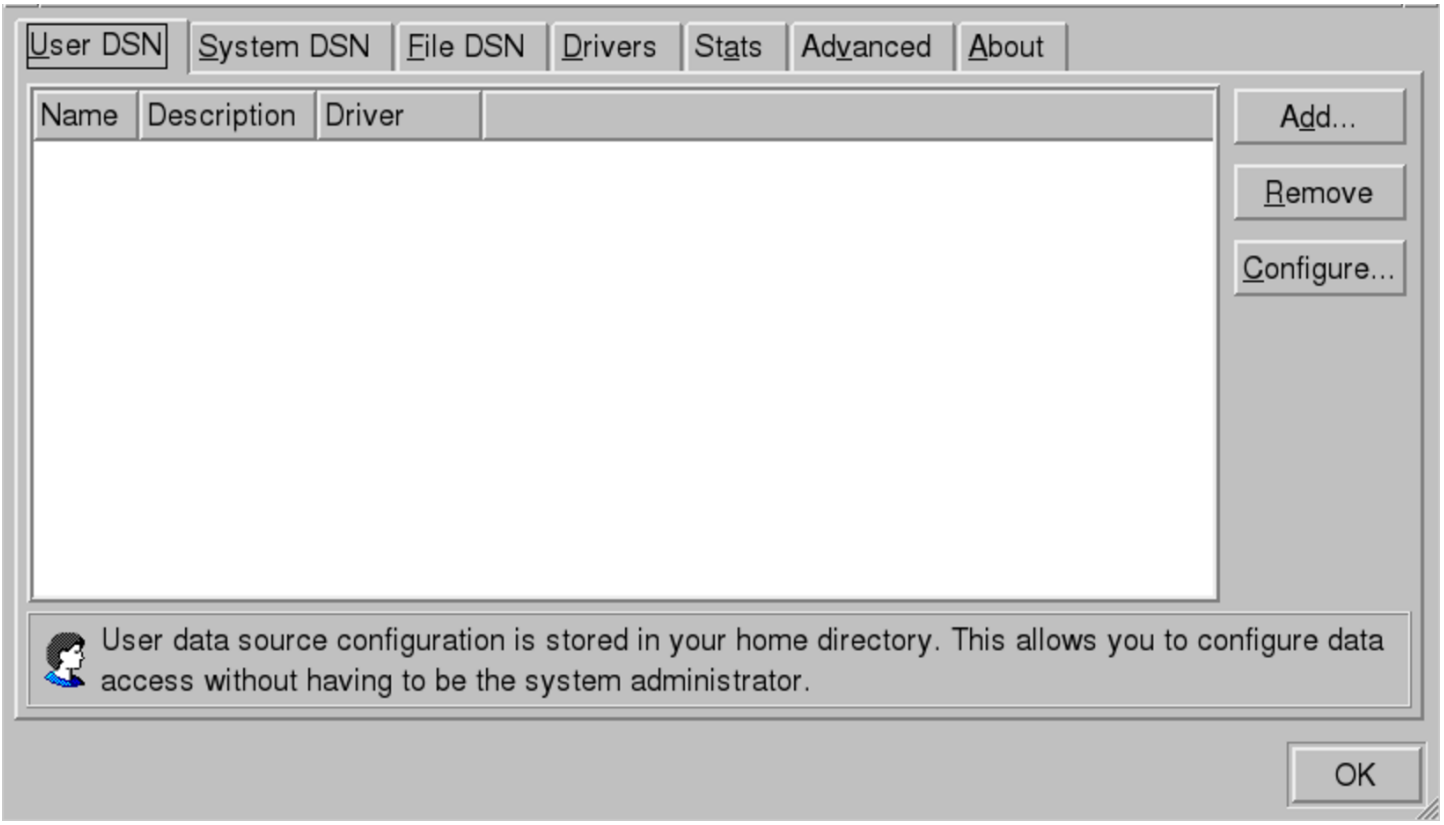
«

Il programma ODBCConfig, che fa parte di unixODBC, consente di configurare i DSN in modo guidato, proponendo dei valori predefiniti appropriati al tipo di DBMS a cui questi vanno associati. Eventualmente, sarebbe possibile anche intervenire nella configurazione di '`/etc/odbcinst.ini`', ma questo non è conveniente, perché in tal caso manca la guida necessaria. Il programma si avvia generalmente senza argomenti:

```
ODBCConfig
```

Se il programma viene avviato con i privilegi necessari, può intervenire nella configurazione di ‘`/etc/odbc.ini`’ o dei file contenuti nella directory ‘`/etc/ODBCDataSources/`’, altrimenti può operare esclusivamente nel file personale ‘`~/.odbc.ini`’.

Figura 78.4. Aspetto di ODBCConfig all’avvio.

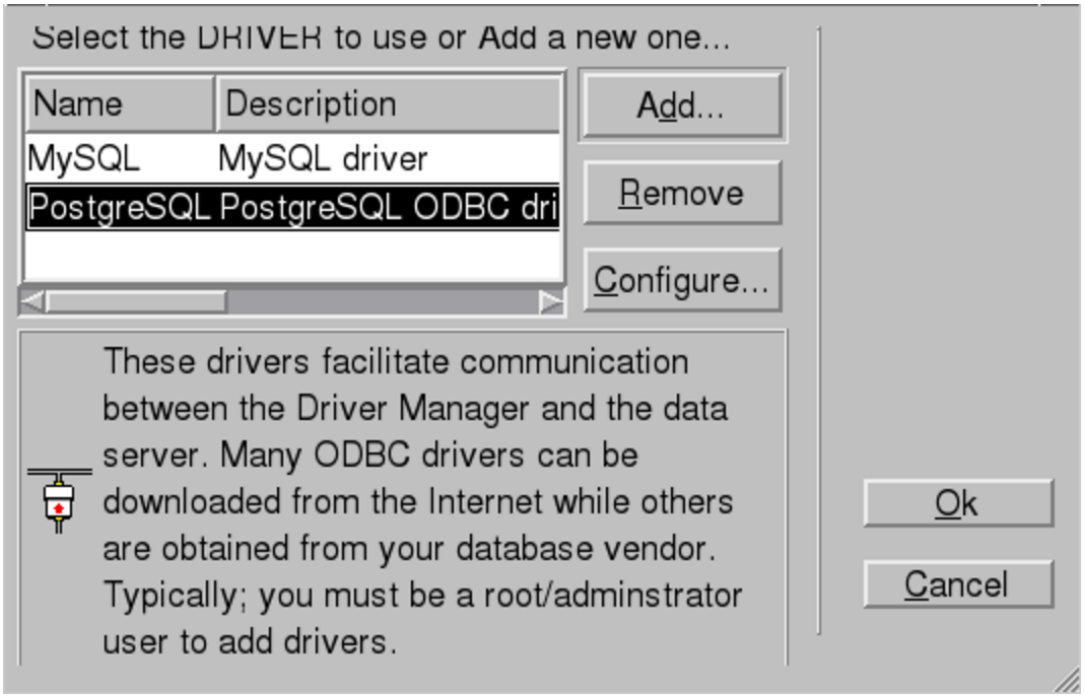


Osservando il programma in funzione, si vedono alcuni lembi posti sul lato superiore. I primi tre (*User DSN*, *System DSN*, *File DSN*) permettono di selezionare una scheda riferita, rispettivamente, alla configurazione dei DSN personali (‘`~/.odbc.ini`’), di quelli di sistema (‘`/etc/odbc.ini`’) e di quelli condivisibili (‘`/etc/ODBCDataSources/`’). La configurazione con una qualsiasi di queste tre schede è uniforme alle altre; quello che cambia sono i privilegi

necessari a modificare i file di configurazione rispettivi.

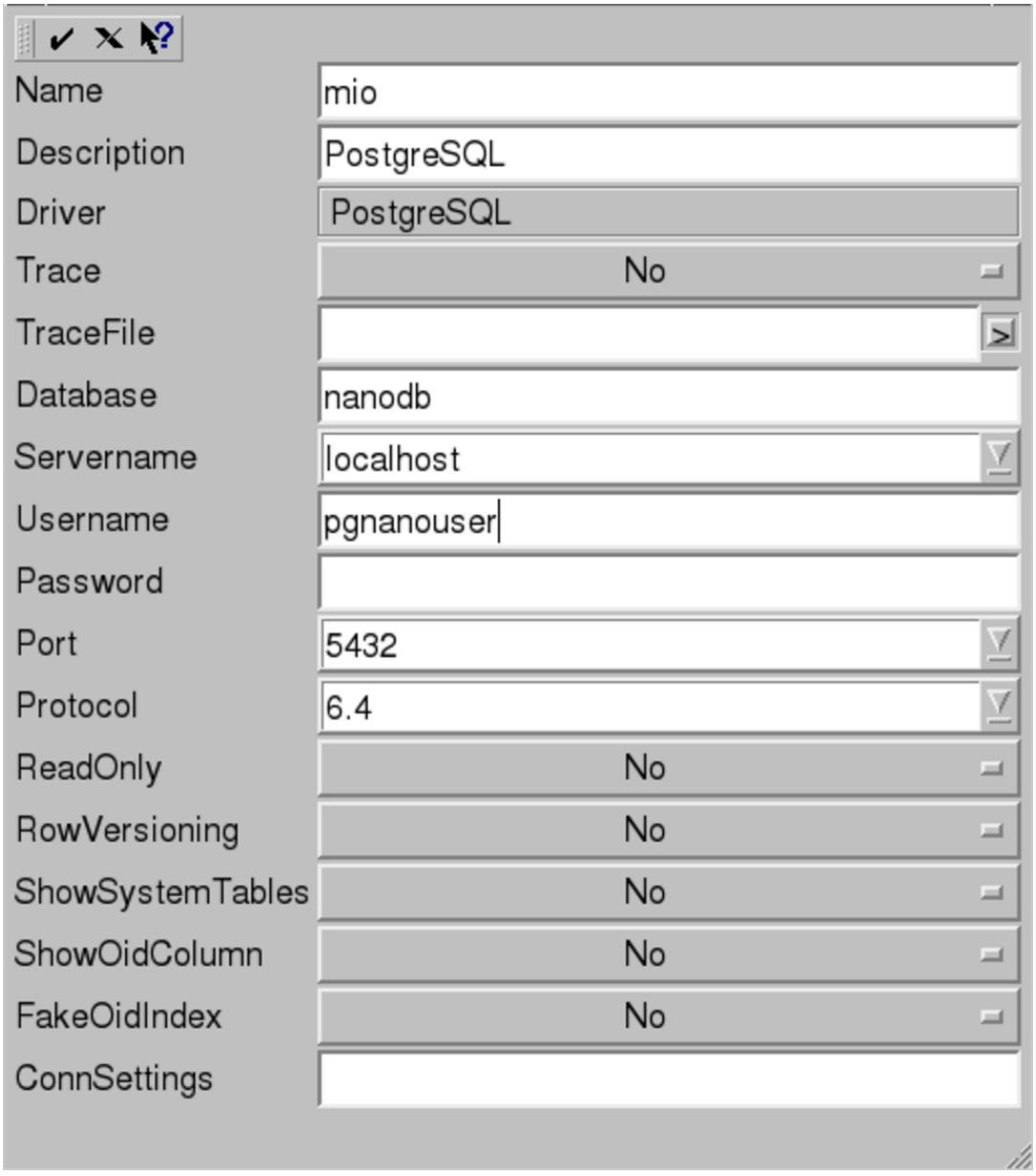
Avendo selezionato una delle tre schede che consentono di intervenire sui DSN, sul lato destro della finestra appaiono dei pulsanti grafici, con i quali è possibile creare, eliminare o modificare dei DSN. Volendo creare un DSN (pulsante ADD), appare la richiesta di specificare il tipo di DBMS (*driver*).

Figura 78.5. La maschera con cui si specifica la libreria adatta a comunicare con il DBMS di proprio interesse. Una volta evidenziato, come si vede in questo caso a proposito di PostgreSQL, si deve selezionare il pulsante grafico OK.



Successivamente viene proposta una maschera che riproduce, sostanzialmente, le direttive specifiche per quel tipo di libreria, da inserire nel file di configurazione. Fortunatamente, la maschera contiene già dei valori predefiniti appropriati per la maggior parte delle direttive.

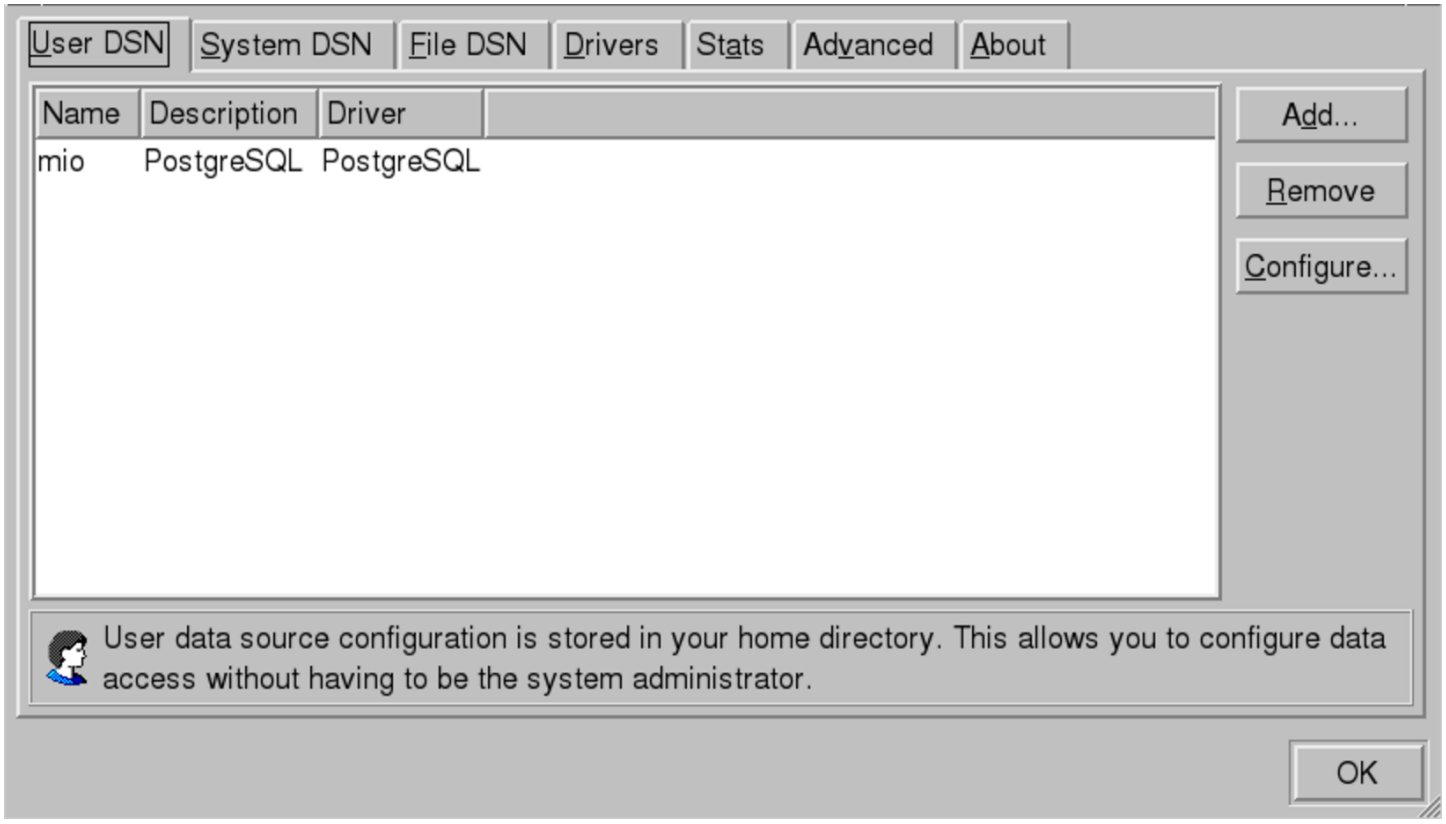
Figura 78.6. La maschera con cui si definiscono le direttive per il DSN, nel caso della libreria per il collegamento a und DBMS PostgreSQL. In questo caso viene dichiarato il DSN denominato '**mio**', abbinato alla base di dati '**nanodb**', presso l'elaboratore locale, a cui si accede con l'utenza '**pgnanouser**' (la parola d'ordine, ammesso che sia necessaria per accedere, rimane da specificare al momento del collegamento.



Nella maschera di inserimento delle direttive, appare un menù di icone, di solito sul lato superiore sinistro. Con l'icona che mostra una

«X», si annullano le modifiche, mentre con quella che assomiglia a una «V», si confermano gli inserimenti.

Figura 78.7. Aspetto di ODBCConfig dopo l'inserimento del DSN '**mio**' nella configurazione personale dell'utente.



Tra le altre schede del programma, è interessante osservare ciò che è contenuto in quella denominata *Advanced*. Lì dovrebbe essere visibile il percorso di un file usato per annotare l'esito delle interrogazioni avvenute con le basi di dati reali, per poter risalire alla causa di un problema, quando l'utilizzo di ODBC sembra fallire senza un motivo apparente. In ogni caso, questo file dovrebbe corrispondere a '`/tmp/sql.log`'.

78.4 Accesso a ODBC tramite «isql» o «iusql»

unixODBC include due programmi equivalenti, per accedere a un DSN attraverso istruzioni SQL impartite interattivamente. Attraverso questi programmi si può verificare in pratica il funzionamento di un certo DSN:

```
isql nome_dsn [nome_utente [parola_d'ordine]] [opzioni]
```

```
iusql nome_dsn [nome_utente [parola_d'ordine]] [opzioni]
```

La differenza tra i due programmi dovrebbe consistere nella migliore disposizione del secondo verso la codifica universale.

Dalla sintassi mostrata sull'uso dei due programmi, si può osservare che è obbligatorio l'inserimento del nome del DSN con cui ci si vuole connettere, mentre gli altri dati sono facoltativi, perché potrebbero essere memorizzati nella configurazione del DSN stesso, oppure, nel caso della parola d'ordine, potrebbe non essere richiesta dal DBMS per accedere. A titolo di esempio, si suppone di collegarsi al DSN '**mio**', per il quale è già stato specificato il nominativo utente da usare e la parola d'ordine non è richiesta:

```
$ isql mio[Invio]
```

```
+--------------------------------------+
| Connected!                           |
|                                      |
| sql-statement                        |
| help [tablename]                     |
| quit                                 |
|                                      |
+--------------------------------------+
```

Come si vede, viene suggerito ciò che si può fare: inserire istruzioni SQL, usare il comando '**help**' o '**quit**'. A questo punto non c'è molto da spiegare; si comprende che possono essere impartite delle istruzioni SQL (secondo i canoni di ODBC) e che al termine si può concludere con il comando '**quit**'.

`SQL>` **quit** [*Invio*]

Il problema nell'uso di un programma come questo, semmai, sta nel fatto che, di fronte a un errore, la spiegazione che si ottiene è estremamente scarna e occorre leggere il file in cui i messaggi del DBMS reale vengono scaricati (di solito è '`/tmp/sql.log`').

78.5 Riferimenti

«

- *unixODBC*, *http://www.unixodbc.org*

[1] **unixODBC** GNU GPL e GNU LGPL

[2] **PostgreSQL ODBC** GNU GPL

[3] **MyODBC** GNU GPL

Prima di poter iniziare a eseguire gli esercizi di questo capitolo, dedicato alle basi di dati e al linguaggio SQL, è necessario verificare di disporre degli strumenti adatti ed essere sicuri di saperli utilizzare.

Per facilitare l'esecuzione di queste esercitazioni, sia gli esercizi, sia le verifiche sono realizzabili con SQLite, attraverso il program-

ma '**sqlite3**'. Pertanto gli esercizi prevedono l'uso di basi di dati personali, ognuna contenuta tutta in un file.

Le verifiche associate a queste esercitazioni portano a produrre dei fogli stampati, che gli studenti devono avere la cura di controllarle in base a quanto indicato nella traccia delle verifiche stesse, prima della consegna all'insegnante.

Per poter svolgere gli esercizi e le verifiche, ogni studente deve essere in grado di scrivere e modificare file di testo, con un programma adatto (per esempio va bene il programma Gedit). In questi file di testo vanno inserite le istruzioni SQL necessarie allo svolgimento del lavoro; per evitare confusione, i file che contengono codice SQL vengono nominati con l'estensione '`.sql`'.

Per eseguire le istruzioni SQL contenute in un file, si usa il programma '**sqlite3**' nel modo seguente:

`$` **sqlite3** ***file_db*** **<** ***file_sql*** [*Invio*]

In questo caso, le istruzioni contenute nel file ***file_sql***, vengono applicate alla base di dati contenuta nel file ***file_db***.

Per essere certi di sapere usare gli strumenti occorre fare una prova. Si realizzi il file di testo denominato '`prova-istruzioni.sql`', contenente quanto segue, sostituendo le metavariabili ***cognome***, ***nome***, ***classe*** e ***data*** con qualcosa di appropriato:

```
-- Esercizio di prova di: cognome  nome  classe
-- Data: data
-- File: prova-istruzioni.sql

CREATE TABLE Prova (Codice  INTEGER,
                    Cognome VARCHAR(60),
                    Nome    VARCHAR(60));
```

Una volta salvato il file con il nome stabilito, lo si esegue nella base di dati contenuta nel file '`prova.db`'. Dal momento che il file '`prova.db`' non esiste, essendo la prima volta che viene utilizzato questo nome, l'esecuzione delle istruzioni comporta automaticamente la creazione della base di dati relativa:

```
$ sqlite3 prova.db < prova-istruzioni.sql[Invio]
```

Se non si vedono segnalazioni di alcun genere, le istruzioni contenute nel file '`prova-istruzioni.sql`' sono state eseguite tutte con successo.

Le istruzioni contenute nel file '`prova-istruzioni.sql`' servono a produrre una ***relazione***, denominata '**`Prova`**', contenente alcuni ***attributi*** ('**`Codice`**', '**`Cognome`**' e '**`Nome`**').

Se il file contenente le istruzioni SQL contiene degli errori, o viene eseguito quando ciò non deve essere fatto, è probabile vedere apparire dei messaggi, che vanno letti attentamente. Per esempio, se venisse eseguito nuovamente il file '`prova-istruzioni.sql`' nella stessa base di dati, si otterrebbe una segnalazione che avvisa del fatto che la relazione '**`Prova`**' esiste già (e non può essere creata nuovamente):

```
$ sqlite3 prova.db < prova-istruzioni.sql[Invio]
```

```
CREATE TABLE Prova (Codice  INTEGER,
                    Cognome VARCHAR(60),
                    Nome    VARCHAR(60));
SQL error: table Prova already exists
```

Il programma '**sqlite3**' può essere usato anche interattivamente. Per farlo, si avvia senza indicare il file contenente le istruzioni SQL. Si proceda in questo modo:

```
$ sqlite3 prova.db[Invio]
```

A questo punto appare l'invito del programma '**sqlite3**', che indica la sua attesa di comandi o di istruzioni SQL, digitati direttamente:

```
sqlite>
```

Con il comando '**.schema**' (si osservi il fatto che il comando inizia con un punto) è possibile visualizzare l'elenco delle relazioni esistenti, in forma di istruzione SQL. Si proceda inserendo questo comando:

```
sqlite> .schema[Invio]

CREATE TABLE Prova (Codice  INTEGER,
                    Cognome VARCHAR(60),
                    Nome    VARCHAR(60));
```

Si proceda inserendo l'istruzione necessaria a eliminare la relazione '**Prova**' creata poco prima:

```
sqlite> DROP TABLE Prova;[Invio]
```

Si conclude con il funzionamento di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit [Invio]
```

Prima di passare alle sezioni successive, vanno eliminati i file ‘`prova-istruzioni.sql`’ e ‘`prova.db`’ che non servono più.

In questi esercizi vengono creati i file seguenti, elencati in ordine alfabetico, con il riferimento alla sezione in cui sono utilizzati:

79.1 Creazione ed eliminazione delle relazioni

«

Una ***relazione*** è un insieme di ***tuple***, suddivise in ***attributi***, tutte nello stesso modo. Una relazione si rappresenta normalmente in forma

tabellare, dove le tuple sono costituite dalle righe e gli attributi dalle colonne.

Figura 79.4. Relazione «Articoli (Articolo, Descrizione, UM, Listino, ScortaMin)».

Articolo	**Descrizione**	**UM**	**Listino**	**ScortaMin**
1	Dischetti da 9 cm 1440 Kibyte	pz	0,20	500
2	Dischetti da 9 cm 1440 Kibyte colorati	pz	0,25	500
101	CD-R 16x	pz	0,50	500
102	CD-R 52x	pz	1,00	500
201	CD-RW 4x	pz	1,00	200
202	CD-RW 8x	pz	1,50	200
301	DVD-R 8x	pz	1,00	200
302	DVD-R 16x	pz	2,00	200
401	DVD+R 8x	pz	1,00	200
402	DVD+R 16x	pz	2,00	200
501	DVD-RW 8x	pz	2,00	200
601	DVD+RW 8x	pz	2,00	200

I valori che si possono inserire nelle celle della tabella dipendono dal ***dominio*** dell'attributo relativo. Per esempio, l'attributo '**`Listino`**' (corrispondente alla quarta colonna) della relazione '**`Articoli`**', può contenere solo valori numerici positivi, con un massimo di due decimali.

79.1.1 Creazione di una relazione

Con l'ausilio di un programma per la scrittura e la modifica di file di testo puro, si crei il file '`prova-creazione-articoli.`

`sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-creazione-articoli.sql

CREATE TABLE Articoli (Articolo    INTEGER        NOT NULL,
                       Descrizione CHAR(60)       NOT NULL,
                       UM          CHAR(7)        DEFAULT 'pz',
                       Listino     NUMERIC(14,2)  DEFAULT 0,
                       ScortaMin   NUMERIC(12,3)  DEFAULT 0,
                       PRIMARY KEY (Articolo));

INSERT INTO Articoli VALUES
    (1, 'Dischetti da 9 cm 1440 Kibyte', 'pz', 0.2, 500);
INSERT INTO Articoli VALUES
    (2, 'Dischetti da 9 cm 1440 Kibyte colorati', 'pz', 0.25, 500);
```

L'istruzione '**`CREATE TABLE`**' permette la creazione della relazione '**`Articoli`**', stabilendo dei vincoli, per cui gli attributi '**`Articolo`**' e '**`Descrizione`**' non possono contenere un valore nullo; inoltre viene stabilito il valore predefinito per gli altri attributi. Si stabilisce anche che l'attributo '**`Articolo`**' deve essere una chiave primaria, comportando la necessità che non appaiano tuple con lo stesso codice articolo.

Le istruzioni '**`INSERT`**', inseriscono le prime due tuple della relazione. A questo proposito, si osservi che i dati numerici, come il prezzo di listino e il livello della scorta minima, si indicano così come sono, con l'accortezza di usare **il punto per la separazione dei decimali**, mentre **le stringhe** (le informazioni testuali) **vanno delimitate da apici singoli**.

Si controlli di avere scritto il file ‘`prova-creazione-articoli.sql`’ in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con li nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-creazione-articoli.sql[Invio]
```

Se il programma mostra dei messaggi, si tratta di errori, che devono essere verificati attentamente, prima di proseguire.

Si ricorda che il file ‘`mag.db`’, contenente la base di dati, viene creato automaticamente se non dovesse già essere presente.

79.1.2 Eliminazione di una relazione

Per eliminare una relazione si usa l’istruzione ‘**`DROP TABLE`**’, come nell’esempio seguente:

```
DROP TABLE Articoli;
```

Si vuole eliminare la relazione ‘**`Articoli`**’ appena creata nella base di dati contenuta nel file ‘`mag.db`’, ma trattandosi di un’operazione molto semplice, è meglio usare il programma ‘**`sqlite3`**’ in modo interattivo. Si avvii il programma ‘**`sqlite3`**’ e si eseguano i comandi successivi, come descritto qui di seguito, utilizzando anche il comando ‘**`.schema`**’ per avere l’elenco delle relazioni esistenti, prima di cancellare effettivamente quella stabilita:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
```

```
sqlite> .schema [Invio]

  CREATE TABLE Articoli (Articolo     INTEGER         NOT NULL,
                         Descrizione  CHAR(60)        NOT NULL,
                         UM           CHAR(7)         DEFAULT 'pz',
                         Listino      NUMERIC(14,2)   DEFAULT 0,
                         ScortaMin    NUMERIC(12,3)   DEFAULT 0,
                         PRIMARY KEY (Articolo));

sqlite> DROP TABLE Articoli; [Invio]

sqlite> .quit [Invio]
```

Si ricorda che se, a seguito dell'inserimento dell'istruzione '**DROP TABLE**', il programma mostra dei messaggi, si tratta di errori che devono essere verificati attentamente, prima di proseguire.

79.1.3 Verifica sulla creazione e popolazione della relazione «Articoli»

«

Per poter svolgere questa verifica, gli studenti devono essere in grado di realizzare un file di testo contenente codice SQL, con le istruzioni necessarie alla creazione di una relazione e con quelle che permettono l'inserimento delle tuple. Inoltre, devono essere in grado di utilizzare il programma '**sqlite3**' in modo interattivo, per visualizzare l'elenco delle relazioni esistenti nella base di dati e per eliminare una relazione.

Si riprenda il file '`prova-creazione-articoli.sql`' e lo si salvi con il nome '`creazione-articoli.sql`'. Il file '`creazione-articoli.sql`' va poi modificato aggiungendo le istruzioni necessarie a completare l'inserimento degli articoli che sono visibili nella figura 79.4.

Una volta completato e salvato il file ‘`creazione-articoli.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < creazione-articoli.sql[Invio]
```

Se si ottengono degli errori, si deve eliminare la relazione ‘**`Articoli`**’ dalla base di dati contenuta nel file ‘`mag.db`’, utilizzando il programma ‘**`sqlite3`**’ in modo interattivo, quindi, dopo le correzioni, si deve riprovare.

Una volta eseguita l’operazione con successo, si stampi il file ‘`creazione-articoli.sql`’ e lo si consegni per la correzione all’insegnante.

Nella valutazione viene controllata la correttezza del contenuto del file e la coerenza estetica nella scrittura delle istruzioni SQL.

79.1.4 Verifica sulla creazione e popolazione della relazione «Causali»

Prima di svolgere questa verifica, è necessario avere svolto quella precedente, della quale valgono anche gli stessi requisiti.

Si crei il file ‘`creazione-causali.sql`’, inserendo le istruzioni necessarie a creare la relazione ‘**`Causali`**’, con il contenuto che si vede nella figura 79.9, tenendo conto che:

1. l’attributo ‘**`Causale`**’ è di tipo ‘**`INTEGER`**’, non ammette il valore nullo e costituisce la chiave primaria;
2. l’attributo ‘**`Descrizione`**’ è di tipo ‘**`CHAR`**’ a 60 caratteri e non ammette il valore nullo;
3. l’attributo ‘**`Variazione`**’ è di tipo ‘**`NUMERIC`**’, a una sola cifra, senza decimali, con un valore predefinito pari a zero.

Figura 79.9. Relazione **`Causali (Causale, Descrizione, Variazione)`**.

Causale	Descrizione	Variazione
1	Carico per acquisto	+1
2	Scarico per vendita	-1
3	Reso da cliente	+1
4	Reso a fornitore	-1
5	Rettifica aumento acquisto	+1
6	Rettifica aumento vendite	-1
7	Rettifica diminuzione vendite	+1
8	Rettifica diminuzione acquisti	-1
9	Carico da produzione	+1
10	Scarico a produzione	-1
11	Carico da altro magazzino	+1
12	Scarico ad altro magazzino	-1
13	Saldo iniziale	+1

Figura 79.10. Scheletro del file ‘`creazione-causali.sql`’, da completare.

```
-- Creazione della relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: creazione-causali.sql

CREATE TABLE Causali …

INSERT INTO Causali …
…
```

Una volta completato e salvato il file ‘`creazione-causali.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < creazione-causali.sql [Invio]
```

Se si ottengono degli errori, si deve eliminare la relazione '**Causali**' dalla base di dati contenuta nel file '`prova.db`', utilizzando il programma '**sqlite3**' in modo interattivo, quindi, dopo le correzioni, si deve riprovare.

Una volta eseguita l'operazione con successo, si stampi il file '`creazione-causali.sql`' e lo si consegni per la correzione all'insegnante.

Nella valutazione viene controllata la correttezza del contenuto del file e la coerenza estetica nella scrittura delle istruzioni SQL.

79.1.5 Verifica sulla creazione e popolazione della relazione «Fornitori»

Prima di svolgere questa verifica, è necessario avere svolto quelle precedenti, delle quali valgono anche gli stessi requisiti.

Si crei il file '`creazione-fornitori.sql`', inserendo le istruzioni necessarie a creare la relazione '**Fornitori**', con il contenuto che si vede nella figura 79.11, tenendo conto che:

1. l'attributo '**Fornitore**' è di tipo '**INTEGER**', non ammette il valore nullo e costituisce la chiave primaria;
2. l'attributo '**RagioneSociale**' è di tipo '**VARCHAR**' a 120 caratteri e non ammette il valore nullo;
3. l'attributo '**Paese**' è di tipo '**CHAR**' a 30 caratteri e il suo valore predefinito è '**ITALIA**';
4. l'attributo '**Indirizzo**' è di tipo '**VARCHAR**' a 120 caratteri e non ammette il valore nullo;

5. l'attributo '**CAP**' è di tipo '**CHAR**' a 10 caratteri;
6. l'attributo '**Citta**' è di tipo '**VARCHAR**' a 120 caratteri e non ammette il valore nullo;
7. l'attributo '**Prov**' è di tipo '**CHAR**' a 2 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla;
8. l'attributo '**Telefono**' è di tipo '**CHAR**' a 20 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla;
9. l'attributo '**Fax**' è di tipo '**CHAR**' a 20 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla;
10. l'attributo '**CFPI**' (codice fiscale o partita IVA) è di tipo '**CHAR**' a 30 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla.

Figura 79.11. Relazione **Fornitori (Fornitore, RagioneSociale, Paese, Indirizzo, CAP, Citta, Prov, Telefono, Fax, CFPI)**.

Fornitore	RagioneSociale	Paese	Indirizzo	CAP	Citta	Prov	Telefono	Fax	CFPI
1	Tizio Tizi	ITALIA	via Tazio, 11	31100	Treviso	TV	0422,111111	0422,222222	12345678901
2	Caio Cai	ITALIA	via Caino, 22	31033	Castelfranco Veneto	TV	0423,222222	0423,333333	23456789012
3	Sempronio Semproni	ITALIA	via Salina, 33	31057	Silea	TV	0422,333333	0422,444444	34567890123

Figura 79.12. Scheletro del file ‘`creazione-fornitori.sql`’, da completare.

```
-- Creazione della relazione "Fornitori"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: creazione-fornitori.sql

CREATE TABLE Fornitori …

INSERT INTO Fornitori …
…
```

Una volta completato e salvato il file ‘`creazione-fornitori.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < creazione-fornitori.sql [Invio]
```

Se si ottengono degli errori, si deve eliminare la relazione ‘**`Fornitori`**’ dalla base di dati contenuta nel file ‘`mag.db`’, utilizzando il programma ‘**`sqlite3`**’ in modo interattivo, quindi, dopo le correzioni, si deve riprovare.

Una volta eseguita l’operazione con successo, si stampi il file ‘`creazione-fornitori.sql`’ e lo si consegni per la correzione all’insegnante.

Nella valutazione viene controllata la correttezza del contenuto del file e la coerenza estetica nella scrittura delle istruzioni SQL.

79.1.6 Verifica sulla creazione e popolazione della relazione «Clienti»

Prima di svolgere questa verifica, è necessario avere svolto quelle precedenti, delle quali valgono anche gli stessi requisiti.

Si crei il file ‘`creazione-clienti.sql`’, inserendo le istruzioni necessarie a creare la relazione ‘**`Clienti`**’, con il contenuto che si vede nella figura 79.13, tenendo conto che:

1. l’attributo ‘**`Cliente`**’ è di tipo ‘**`INTEGER`**’, non ammette il valore nullo e costituisce la chiave primaria;
2. l’attributo ‘**`RagioneSociale`**’ è di tipo ‘**`VARCHAR`**’ a 120 caratteri e non ammette il valore nullo;
3. l’attributo ‘**`Paese`**’ è di tipo ‘**`CHAR`**’ a 30 caratteri e il suo valore predefinito è ‘**`ITALIA`**’;
4. l’attributo ‘**`Indirizzo`**’ è di tipo ‘**`VARCHAR`**’ a 120 caratteri e non ammette il valore nullo;
5. l’attributo ‘**`CAP`**’ è di tipo ‘**`CHAR`**’ a 10 caratteri;
6. l’attributo ‘**`Citta`**’ è di tipo ‘**`VARCHAR`**’ a 120 caratteri e non ammette il valore nullo;
7. l’attributo ‘**`Prov`**’ è di tipo ‘**`CHAR`**’ a 2 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla;
8. l’attributo ‘**`Telefono`**’ è di tipo ‘**`CHAR`**’ a 20 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla;
9. l’attributo ‘**`Fax`**’ è di tipo ‘**`CHAR`**’ a 20 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla;
10. l’attributo ‘**`CFPI`**’ (codice fiscale o partita IVA) è di tipo ‘**`CHAR`**’ a 30 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla.

Figura 79.13. Relazione **`Clienti (Cliente, RagioneSociale, Paese, Indirizzo, CAP, Citta, Prov, Telefono, Fax, CFPI)`**.

Cliente	RagioneSociale	Paese	Indirizzo	CAP	Citta	Prov	Telefono	Fax	CFPI
1	Mevio Mevi	ITALIA	via Mare, 11	31050	Morgano	TV	0422,444444	0422,555555	45678901234
2	Filano Filani	ITALIA	via Farfalle, 22	31032	Feltre	BL	0439,555555	0439,666666	56789012345
3	Martino Martini	ITALIA	via Marte, 33	31010	Mareno di Piave	TV	0438,666666	0438,777777	67890123456

Figura 79.14. Scheletro del file ‘`creazione-clienti.sql`’, da completare.

```
-- Creazione della relazione "Clienti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: creazione-clienti.sql

CREATE TABLE Clienti …

INSERT INTO Clienti …
…
```

Una volta completato e salvato il file ‘`creazione-clienti.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < creazione-clienti.sql [Invio]
```

Se si ottengono degli errori, si deve eliminare la relazione ‘**`Clienti`**’ dalla base di dati contenuta nel file ‘`mag.db`’, utilizzando il programma ‘**`sqlite3`**’ in modo interattivo, quindi, dopo le correzioni, si deve riprovare.

Una volta eseguita l’operazione con successo, si stampi il file ‘`creazione-clienti.sql`’ e lo si consegni per la correzione all’insegnante.

Nella valutazione viene controllata la correttezza del contenuto del file e la coerenza estetica nella scrittura delle istruzioni SQL.

79.1.7 Verifica sulla creazione e popolazione della relazione «Movimenti»

«

Prima di svolgere questa verifica, è necessario avere svolto quelle precedenti, delle quali valgono anche gli stessi requisiti.

Si crei il file '`creazione-movimenti.sql`', inserendo le istruzioni necessarie a creare la relazione '**`Movimenti`**', con il contenuto che si vede nella figura 79.15, tenendo conto che:

1. l'attributo '**`Movimento`**' è di tipo '**`INTEGER`**', non ammette il valore nullo e costituisce la chiave primaria;
2. l'attributo '**`Articolo`**' è di tipo '**`INTEGER`**' e non ammette il valore nullo;
3. l'attributo '**`Causale`**' è di tipo '**`INTEGER`**' e non ammette il valore nullo;
4. l'attributo '**`Data`**' è di tipo '**`DATE`**' e non ammette il valore nullo;
5. l'attributo '**`Cliente`**' è di tipo '**`INTEGER`**';
6. l'attributo '**`Fornitore`**' è di tipo '**`INTEGER`**';
7. l'attributo '**`Quantita`**' è di tipo '**`NUMERIC`**' a 15 cifre, di cui cinque sono usate per i decimali, e non ammette il valore nullo;
8. l'attributo '**`Valore`**' è di tipo '**`NUMERIC`**' a 14 cifre, di cui due sono usate per i decimali, e non ammette il valore nullo.

Figura 79.15. Relazione **`Movimenti (Movimento, Articolo, Causale, Data, Cliente, Fornitore, Quantita, Valore)`**.

Movimento	Articolo	Causale	Data	Cliente	Fornitore	Quantita	Valore
1	2	1	2012-01-15	NULL	3	10000	100,00
2	2	2	2012-01-16	2	NULL	1000	10,00
3	102	1	2012-01-17	NULL	2	1000	200,00
4	102	2	2012-01-18	1	NULL	100	20,00
5	401	1	2012-01-19	NULL	1	1000	200,00
6	401	2	2012-01-20	3	NULL	200	40,00
7	401	4	2012-01-20	NULL	1	100	20,00
8	102	4	2012-01-20	NULL	2	100	20,00
9	601	1	2012-01-21	NULL	3	2000	1000,00
10	601	2	2012-01-25	1	NULL	1000	500,00

Figura 79.16. Scheletro del file ‘`creazione-movimenti.sql`’, da completare.

```
-- Creazione della relazione "Movimenti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: creazione-movimenti.sql

CREATE TABLE Movimenti …

INSERT INTO Movimenti …
…
```

Si ricorda che i valori numerici vanno indicati come sono, con l’accortezza di usare il punto per la separazione dei decimali; le date, come le stringhe (i valori testuali) vanno delimitate con apici singoli. In questo caso, quando viene a mancare il valore per il cliente o il fornitore, si inserisce il valore «nullo», che si scrive con la parola chiave ‘**`NULL`**’, come appare nella figura 79.15, senza usare apici.

Una volta completato e salvato il file ‘`creazione-movimenti.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < creazione-movimenti.sql[Invio]
```

Se si ottengono degli errori, si deve eliminare la relazione '**Movimenti**' dalla base di dati contenuta nel file 'mag.db', utilizzando il programma '**sqlite3**' in modo interattivo, quindi, dopo le correzioni, si deve riprovare.

Una volta eseguita l'operazione con successo, si stampi il file 'creazione-movimenti.sql' e lo si consegni per la correzione all'insegnante.

Nella valutazione viene controllata la correttezza del contenuto del file e la coerenza estetica nella scrittura delle istruzioni SQL.

79.1.8 Conclusione

«

Prima di passare alla sezione successiva, si deve realizzare un file contenente le istruzioni con cui creare e popolare le relazioni descritte in questa. In pratica, si tratta di copiare il contenuto dei file 'creazione-articoli.sql', 'creazione-causali.sql', 'creazione-fornitori.sql', 'creazione-clienti.sql' e 'creazione-movimenti.sql', in un file completo, denominato 'magazzino.sql'.

Una volta realizzato il file 'magazzino.sql', si deve cancellare il file 'mag.db' e ricreare a partire dalle istruzioni contenute nel file 'magazzino.sql':

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

Se vengono segnalati degli errori, occorre correggere il file 'magazzino.sql', cancellare nuovamente il file 'mag.db', quindi si deve ripetere l'operazione. La base di dati contenuta nel file 'mag.db', viene usata ancora e non si può proseguire se non si riesce a ricrearla correttamente.

All'inizio della sezione è stato creato il file '`prova-creazione-articoli.sql`' che a questo punto non serve più e va eliminato.

79.2 Interrogazione semplice di una relazione

Attraverso l'istruzione '**`SELECT`**' è possibile estrarre il contenuto di una o più relazioni simultaneamente. In questa sezione si mostrano alcune situazioni riferite a una sola relazione.

79.2.1 Interrogazione completa

Si ottiene l'elenco completo di una relazione utilizzando l'istruzione seguente:

```
SELECT * FROM nome_relazione
```

Si eseguano i passaggi seguenti, per ottenere la visualizzazione del contenuto complessivo della relazione '**`Articoli`**' e della relazione '**`Causali`**', così come dovrebbero essere contenute nella base di dati del file '**`mag.db`**':

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Articoli;[Invio]
```

```
Articolo    Descrizione                  UM          Listino     ScortaMin
----------  ---------------------------  ----------  ----------  ----------
1           Dischetti da 9 cm 1440 Kibyte  pz          0.2         500
2           Dischetti da 9 cm 1440 Kibyte  pz          0.25        500
101         CD-R 16x                     pz          0.5         500
102         CD-R 52x                     pz          1           500
201         CD-RW 4x                     pz          1           200
202         CD-RW 8x                     pz          1.5         200
301         DVD-R 8x                     pz          1           200
302         DVD-R 16x                    pz          2           200
401         DVD+R 8x                     pz          1           200
402         DVD+R 16x                    pz          2           200
501         DVD-RW 8x                    pz          2           200
601         DVD+RW 8x                    pz          2           200
```

sqlite> **SELECT * FROM Causali;** [*Invio*]

```
Causale     Descrizione          Variazione
----------  -------------------  ----------
1           Carico per acquisto  1
2           Scarico per vendita  -1
3           Reso da cliente      1
4           Reso a fornitore     -1
5           Rettifica aumento a  1
6           Rettifica aumento v  -1
7           Rettifica diminuzio  1
8           Rettifica diminuzio  -1
9           Carico da produzion  1
10          Scarico a produzion  -1
11          Carico da altro mag  1
12          Scarico ad altro ma  -1
13          Saldo iniziale       1
```

sqlite> **.quit** [*Invio*]

Si osservi che i comandi '**.headers on**' e '**.mode column**' servono a ottenere un elenco incolonnato con le intestazioni, altrimenti, il

risultato sarebbe poco gradevole esteticamente.

79.2.2 Interrogazione con selezione di alcuni attributi

Si ottiene l'elenco di tutte le tuple di una relazione, limitatamente a un certo gruppo di attributi, mettendo, al posto dell'asterisco, i nomi degli attributi desiderati:

```
SELECT attributo [, attributo] ... FROM nome_relazione
```

Si eseguano i passaggi seguenti, per ottenere la visualizzazione del contenuto di tutte le tuple della relazione '**Articoli**', limitatamente agli attributi '**Articolo**', '**Descrizione**' e '**Listino**', così come dovrebbero essere contenute nella base di dati del file '**mag.db**':

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT Articolo, Descrizione, Listino[Invio]

      >         FROM Articoli; [Invio]
```

```
Articolo    Descrizione                    Listino
----------  -----------------------------  ----------
1           Dischetti da 9 cm 1440 Kibyte  0.2
2           Dischetti da 9 cm 1440 Kibyte  0.25
101         CD-R 16x                       0.5
102         CD-R 52x                       1
201         CD-RW 4x                       1
202         CD-RW 8x                       1.5
301         DVD-R 8x                       1
302         DVD-R 16x                      2
401         DVD+R 8x                       1
402         DVD+R 16x                      2
501         DVD-RW 8x                      2
601         DVD+RW 8x                      2
```

Intuitivamente, si comprende che si può anche cambiare l'ordine di visualizzazione degli attributi:

```
sqlite> SELECT Descrizione, Articolo, Listino[Invio]

      >         FROM Articoli;[Invio]
```

```
Descrizione                    Articolo    Listino
-----------------------------  ----------  ----------
Dischetti da 9 cm 1440 Kibyte  1           0.2
Dischetti da 9 cm 1440 Kibyte  2           0.25
CD-R 16x                       101         0.5
CD-R 52x                       102         1
CD-RW 4x                       201         1
CD-RW 8x                       202         1.5
DVD-R 8x                       301         1
DVD-R 16x                      302         2
DVD+R 8x                       401         1
DVD+R 16x                      402         2
DVD-RW 8x                      501         2
DVD+RW 8x                      601         2
```

Si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit[Invio]
```

79.2.3 Stampa del contenuto di una relazione

Per ottenere la stampa del contenuto di una o di più relazioni, conviene scrivere le istruzioni necessarie in un file di testo, come già fatto in precedenza. Si proceda con la creazione del file '`prova-stampa-artico-caus.sql`', con il contenuto seguente, che ricalca quanto già mostrato nelle sezioni precedenti:

```
-- Stampa del contenuto delle relazioni "Articoli" e "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-stampa-artico-caus.sql

.headers on
.mode column
```

```
SELECT * FROM Articoli;
SELECT * FROM Causali;
```

Per verificare il funzionamento delle istruzioni contenute nel file ‘`stampa-artico-caus.sql`’, si può utilizzare il comando seguente, che interviene nella base di dati contenuta nel file ‘`mag.db`’, limitandosi a visualizzare il risultato:

```
$ sqlite3 mag.db < prova-stampa-artico-caus.sql [Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Articolo    Descrizione                  UM          Listino     ScortaMin
----------  ---------------------------  ----------  ----------  ----------
1           Dischetti da 9 cm 1440 Kibyte  pz          0.2         500
2           Dischetti da 9 cm 1440 Kibyte  pz          0.25        500
101         CD-R 16x                     pz          0.5         500
102         CD-R 52x                     pz          1           500
201         CD-RW 4x                     pz          1           200
202         CD-RW 8x                     pz          1.5         200
301         DVD-R 8x                     pz          1           200
302         DVD-R 16x                    pz          2           200
401         DVD+R 8x                     pz          1           200
402         DVD+R 16x                    pz          2           200
501         DVD-RW 8x                    pz          2           200
601         DVD+RW 8x                    pz          2           200
Causale     Descrizione          Variazione
----------  -------------------  ----------
1           Carico per acquisto  1
2           Scarico per vendita  -1
3           Reso da cliente      1
4           Reso a fornitore     -1
5           Rettifica aumento a  1
6           Rettifica aumento v  -1
7           Rettifica diminuzio  1
8           Rettifica diminuzio  -1
9           Carico da produzion  1
10          Scarico a produzion  -1
11          Carico da altro mag  1
```

```
12          Scarico ad altro ma  -1
13          Saldo iniziale       1
```

Per ottenere il risultato stampato su carta, basta modificare leggermente il comando:

```
$ sqlite3 mag.db < prova-stampa-artico-caus.sql ↩
↪| lpr[Invio]
```

79.2.4 Verifica sull'interrogazione della relazione «Articoli»

Si prepari il file '`interr-artico-01.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco di tutte le tuple della relazione '**`Articoli`**', ordinando gli attributi in questo modo: '**`Descrizione`**', '**`Articolo`**', '**`UM`**', '**`ScortaMin`**' e '**`Listino`**'.

Figura 79.25. Scheletro del file '`interr-artico-01.sql`', da completare.

```
-- Interrogazione della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-artico-01.sql

.headers on
.mode column

SELECT …
```

Una volta completato e salvato il file '`interr-artico-01.sql`', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-artico-01.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Descrizione                  Articolo    UM          ScortaMin   Listino
---------------------------  ----------  ----------  ----------  ----------
Dischetti da 9 cm 1440 Kibyte  1          pz          500         0.2
Dischetti da 9 cm 1440 Kibyte  2          pz          500         0.25
CD-R 16x                     101         pz          500         0.5
CD-R 52x                     102         pz          500         1
CD-RW 4x                     201         pz          200         1
CD-RW 8x                     202         pz          200         1.5
DVD-R 8x                     301         pz          200         1
DVD-R 16x                    302         pz          200         2
DVD+R 8x                     401         pz          200         1
DVD+R 16x                    402         pz          200         2
DVD-RW 8x                    501         pz          200         2
DVD+RW 8x                    601         pz          200         2
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-artico-01.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-artico-01.sql`’.

79.2.5 Verifica sull’interrogazione delle relazioni «Fornitori» e «Clienti»

«

Si prepari il file ‘`interr-forn-clie-01.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l’elenco di tutte le tuple delle relazioni ‘**`Fornitori`**’ e ‘**`Clienti`**’, limitatamente agli attributi: ‘**`Fornitore`**’ (nel caso della relazione ‘**`Fornitori`**’) o ‘**`Cliente`**’ (nel caso della relazione ‘**`Clienti`**’), ‘**`RagioneSociale`**’, ‘**`Telefono`**’ e ‘**`Fax`**’.

Figura 79.27. Scheletro del file ‘`interr-forn-clie-01.sql`’, da completare.

```
-- Interrogazione delle relazioni "Fornitori" e "Clienti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-forn-clie-01.sql

.headers on
.mode column

SELECT …
SELECT …
```

Una volta completato e salvato il file ‘`interr-forn-clie-01.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-forn-clie-01.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Fornitore   RagioneSociale  Telefono     Fax
----------  --------------  -----------  -----------
1           Tizio Tizi      0422,111111  0422,222222
2           Caio Cai        0423,222222  0423,333333
3           Sempronio Semp  0422,333333  0422,444444
Cliente     RagioneSociale  Telefono     Fax
----------  --------------  -----------  -----------
1           Mevio Mevi      0422,444444  0422,555555
2           Filano Filani   0439,555555  0439,666666
3           Martino Martin  0438,666666  0438,777777
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-forn-clie-01.sql ↩
↪         | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-forn-clie-01.sql`’.

79.2.6 Conclusione

«

Il file ‘`prova-stampa-artico-caus.sql`’, non serve più nelle sezioni successive, pertanto va eliminato.

79.3 Interrogazione ordinata di una relazione

«

Attraverso l’istruzione ‘**`SELECT`**’, aggiungendo l’opzione ‘**`ORDERED BY`**’, è possibile specificare gli attributi secondo i quali ordinare il risultato. In mancanza dell’indicazione di questa opzione, l’elenco delle tuple si ottiene secondo un ordine «casuale», che solitamente coincide con la sequenza di inserimento.

79.3.1 Interrogazione ordinata

«

A titolo di esempio, si vuole ottenere l’elenco delle tuple della relazione ‘**`Articoli`**’, in ordine di descrizione. Si può utilizzare il programma ‘**`sqlite3`**’ in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Articoli ↩
↪                 ORDER BY Descrizione;[Invio]
```

```
Articolo    Descrizione  UM          Listino     ScortaMin
----------  -----------  ----------  ----------  ----------
101         CD-R 16x     pz          0.5         500
102         CD-R 52x     pz          1           500
201         CD-RW 4x     pz          1           200
202         CD-RW 8x     pz          1.5         200
402         DVD+R 16x    pz          2           200
401         DVD+R 8x     pz          1           200
601         DVD+RW 8x    pz          2           200
302         DVD-R 16x    pz          2           200
301         DVD-R 8x     pz          1           200
501         DVD-RW 8x    pz          2           200
1           Dischetti d  pz          0.2         500
2           Dischetti d  pz          0.25        500
```

Con la relazione '**Articoli**', potrebbe essere interessante un ordinamento per listino, ma in questo caso si aggiunge anche la descrizione, quando il prezzo di listino risulta uguale:

```
sqlite> SELECT * FROM Articoli ORDER BY Listino, Descrizione;
```
[*Invio*]

```
Articolo    Descrizione                    UM          Listino     ScortaMin
----------  -----------------------------  ----------  ----------  ----------
1           Dischetti da 9 cm 1440 Kibyte  pz          0.2         500
2           Dischetti da 9 cm 1440 Kibyte  pz          0.25        500
101         CD-R 16x                       pz          0.5         500
102         CD-R 52x                       pz          1           500
201         CD-RW 4x                       pz          1           200
401         DVD+R 8x                       pz          1           200
301         DVD-R 8x                       pz          1           200
202         CD-RW 8x                       pz          1.5         200
402         DVD+R 16x                      pz          2           200
601         DVD+RW 8x                      pz          2           200
302         DVD-R 16x                      pz          2           200
501         DVD-RW 8x                      pz          2           200
```

```
sqlite> .quit
```
[*Invio*]

Si osservi che l'ordinamento dipende dal tipo di informazione che l'attributo relativo può contenere. Per esempio, nel caso della relazione '**Articoli**', il riordino per descrizione avviene in modo lessicografico, mentre il riordino per listino avviene in base al valore numerico.

79.3.2 Verifica sull'interrogazione ordinata della relazione «Articoli»

«

Si prepari il file '`interr-artico-02.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco di tutte le tuple della relazione '**Articoli**', ordinate in base al livello di scorta minima e di descrizione; inoltre, si vogliono ottenere solo alcuni attributi, secondo la sequenza: '**ScortaMin**', '**Descrizione**', '**Articolo**'.

Figura 79.32. Scheletro del file '`interr-artico-02.sql`', da completare.

```
-- Interrogazione della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-artico-02.sql

.headers on
.mode column

SELECT …
```

Una volta completato e salvato il file '`interr-artico-02.sql`', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-artico-02.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
ScortaMin   Descrizione  Articolo
----------  -----------  ----------
200         CD-RW 4x     201
200         CD-RW 8x     202
200         DVD+R 16x    402
200         DVD+R 8x     401
200         DVD+RW 8x    601
200         DVD-R 16x    302
200         DVD-R 8x     301
200         DVD-RW 8x    501
500         CD-R 16x     101
500         CD-R 52x     102
500         Dischetti d  1
500         Dischetti d  2
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-artico-02.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-artico-02.sql`’.

79.3.3 Verifica sull’interrogazione ordinata della relazione «Clienti»

Si prepari il file ‘`interr-clie-01.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l’elenco di tutte le tuple della relazione ‘**`Clienti`**’, ordinate in base alla denominazione della ragione sociale, limitatamente agli attributi ‘**`Cliente`**’ e ‘**`RagioneSociale`**’.

Figura 79.34. Scheletro del file ‘`interr-clie-01.sql`’, da completare.

```
-- Interrogazione della relazione "Clienti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-clie-01.sql

.headers on
.mode column

SELECT …
```

Una volta completato e salvato il file ‘`interr-clie-01.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-clie-01.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Cliente     RagioneSociale
----------  --------------
2           Filano Filani
3           Martino Martin
1           Mevio Mevi
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-clie-01.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-clie-01.sql`’.

79.3.4 Verifica sull'interrogazione ordinata della relazione «Causali»

Si prepari il file '`interr-caus-01.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco di tutte le tuple della relazione '**`Causali`**', ordinate in base al fatto che si tratti di movimenti in diminuzione o in aumento (l'attributo '**`Variazione`**').

Figura 79.36. Scheletro del file '`interr-caus-01.sql`', da completare.

```
-- Interrogazione della relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-caus-01.sql

.headers on
.mode column

SELECT …
```

Una volta completato e salvato il file '`interr-caus-01.sql`', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-clie-01.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Causale     Descrizione                Variazione
----------  -------------------------  ----------
12          Scarico ad altro magazzino  -1
10          Scarico a produzione        -1
8           Rettifica diminuzione acqu  -1
6           Rettifica aumento vendite   -1
4           Reso a fornitore            -1
2           Scarico per vendita         -1
```

```
13          Saldo iniziale              1
11          Carico da altro magazzino   1
9           Carico da produzione        1
7           Rettifica diminuzione vend  1
5           Rettifica aumento acquisto  1
3           Reso da cliente             1
1           Carico per acquisto         1
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-caus-01.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-caus-01.sql`’.

79.4 Interrogazione selettiva di una relazione

«

Attraverso l’istruzione ‘**`SELECT`**’, aggiungendo l’opzione ‘**`WHERE`**’, è possibile specificare una condizione per la selezione delle tuple desiderate. In mancanza dell’indicazione di questa opzione, l’elenco delle tuple è sempre completo. La parola chiave ‘**`WHERE`**’ precede un’espressione logica, che viene valutata per ogni tupla: se l’espressione risulta valida (*Vero*), allora la tupla viene presa in considerazione.

In queste lezioni non viene descritto in modo dettagliato come scrivere delle espressioni logiche; tuttavia, vengono raccolte qui delle tabelle riassuntive per la loro realizzazione. Possono essere usate in modo intuitivo, ma nelle verifiche non si richiede altro che utilizzare o modificare leggermente degli esempi già mostrati.

Tabella 79.38. Operatori di confronto.

Operatore e operandi	Descrizione
op1 `=` ***op2***	*Vero* se gli operandi si equivalgono.
op1 `<>` ***op2***	*Vero* se gli operandi sono differenti.
op1 `<` ***op2***	*Vero* se il primo operando è minore del secondo.
op1 `>` ***op2***	*Vero* se il primo operando è maggiore del secondo.
op1 `<=` ***op2***	*Vero* se il primo operando è minore o uguale al secondo.
op1 `>=` ***op2***	*Vero* se il primo operando è maggiore o uguale al secondo.

Tabella 79.39. Operatori logici.

Operatore e operandi	Descrizione
`NOT` ***op***	Inverte il risultato logico dell'operando.
op1 `AND` ***op2***	*Vero* se entrambi gli operandi restituiscono il valore *Vero*.
op1 `OR` ***op2***	*Vero* se almeno uno degli operandi restituisce il valore *Vero*.

Tabella 79.40. Espressioni sulle stringhe di caratteri.

Espressioni e modelli	Descrizione
stringa `LIKE` ***modello***	Restituisce *Vero* se il modello corrisponde alla stringa. Si osservi che SQLite non accetta la forma '**`IS LIKE`**'.

Espressioni e modelli	Descrizione
stringa `NOT LIKE` ***modello***	Restituisce *Vero* se il modello non corrisponde alla stringa. Si osservi che SQLite non accetta la forma '**`IS NOT LIKE`**'.
_	Rappresenta un carattere qualsiasi.
%	Rappresenta una sequenza indeterminata di caratteri.

Tabella 79.41. Espressioni di verifica dei valori indeterminati.

Operatori	Descrizione
espressione `IS NULL`	Restituisce *Vero* se l'espressione genera un risultato indeterminato.
espressione `IS NOT NULL`	Restituisce *Vero* se l'espressione non genera un risultato indeterminato.

Tabella 79.42. Espressioni per la verifica dell'appartenenza di un valore a un intervallo o a un elenco.

Operatori e operandi	Descrizione
op1 `IN (`***elenco***`)`	*Vero* se il primo operando è contenuto nell'elenco.
op1 `NOT IN (`***elenco***`)`	*Vero* se il primo operando non è contenuto nell'elenco.
op1 `BETWEEN` ***op2*** `AND` ***op3***	*Vero* se il primo operando è compreso tra il secondo e il terzo.
op1 `NOT BETWEEN` ***op2*** `AND` ***op3***	*Vero* se il primo operando non è compreso nell'intervallo.

79.4.1 Interrogazione selettiva

A titolo di esempio, si vuole ottenere l'elenco delle tuple della relazione '**Articoli**', selezionando solo quelle che riportano un prezzo di listino maggiore o uguale a 1,00 €. Si può utilizzare il programma '**sqlite3**' in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Articoli WHERE Listino >= 1;[Invio]

  Articolo    Descrizione  UM          Listino     ScortaMin
  ----------  -----------  ----------  ----------  ----------
  102         CD-R 52x     pz          1           500
  201         CD-RW 4x     pz          1           200
  202         CD-RW 8x     pz          1.5         200
  301         DVD-R 8x     pz          1           200
  302         DVD-R 16x    pz          2           200
  401         DVD+R 8x     pz          1           200
  402         DVD+R 16x    pz          2           200
  501         DVD-RW 8x    pz          2           200
  601         DVD+RW 8x    pz          2           200
```

La condizione di selezione potrebbe essere più articolata; per esempio si potrebbe decidere di selezionare gli articoli che hanno un prezzo di listino maggiore o uguale a 1,00 € e che hanno una descrizione che inizia con «DVD»:

```
sqlite> SELECT * FROM Articoli[Invio]
```

```
...>          WHERE Listino >= 1 [Invio]

...>            AND Descrizione LIKE 'DVD%'; [Invio]

Articolo    Descrizione  UM          Listino     ScortaMin
----------  -----------  ----------  ----------  ----------
301         DVD-R 8x     pz          1           200
302         DVD-R 16x    pz          2           200
401         DVD+R 8x     pz          1           200
402         DVD+R 16x    pz          2           200
501         DVD-RW 8x    pz          2           200
601         DVD+RW 8x    pz          2           200
```

Come sempre, si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit [Invio]
```

79.4.2 Verifica sull'interrogazione selettiva della relazione «Articoli»

«

Si prepari il file '`interr-artico-03.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco delle tuple della relazione '**Articoli**', corrispondenti a dei «DVD», che abbiano un prezzo minore o uguale a 1,00 € (l'operatore da usare per rappresentare il confronto «minore o uguale» è '**<=**').

Figura 79.46. Scheletro del file ‘`interr-artico-03.sql`’, da completare.

```
-- Interrogazione della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-artico-03.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
```

Una volta completato e salvato il file ‘`interr-artico-03.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-artico-03.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Descrizione  UM          Listino     ScortaMin
----------  -----------  ----------  ----------  ----------
301         DVD-R 8x     pz          1           200
401         DVD+R 8x     pz          1           200
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-artico-03.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-artico-03.sql`’.

79.4.3 Verifica sull'interrogazione selettiva e ordinata della relazione «Articoli»

«

Si prepari il file '`interr-artico-04.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco ordinato per livello di scorta minima delle tuple della relazione '**`Articoli`**', che corrispondono a dei «CD».

Figura 79.48. Scheletro del file '`interr-artico-04.sql`', da completare.

```
-- Interrogazione della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-artico-04.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
       ORDER BY …
```

Una volta completato e salvato il file '`interr-artico-04.sql`', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-artico-04.sql [Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Descrizione  UM          Listino     ScortaMin
----------  -----------  ----------  ----------  ----------
202         CD-RW 8x     pz          1.5         200
201         CD-RW 4x     pz          1           200
102         CD-R 52x     pz          1           500
```

```
101         CD-R 16x     pz          0.5         500
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-artico-04.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-artico-04.sql`’.

79.4.4 Verifica sull’interrogazione selettiva della relazione «Causali»

Si prepari il file ‘`interr-caus-02.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l’elenco delle tuple della relazione ‘**`Causali`**’ che comportano un aumento (contabile) della quantità di un articolo in magazzino. Le causali che rappresentano un aumento della quantità sono quelle che, nell’attributo ‘**`Variazione`**’ hanno il valore 1 (ovvero +1); pertanto, per selezionare le tuple in questione, è sufficiente verificare che questo valore sia esattamente pari a uno (utilizzando l’operatore ‘**=**’).

Figura 79.50. Scheletro del file ‘`interr-caus-02.sql`’, da completare.

```
-- Interrogazione della relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-caus-02.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
```

Una volta completato e salvato il file ‘`interr-caus-02.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-caus-02.sql [Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Causale     Descrizione          Variazione
----------  -------------------  ----------
1           Carico per acquisto  1
3           Reso da cliente      1
5           Rettifica aumento a  1
7           Rettifica diminuzio  1
9           Carico da produzion  1
11          Carico da altro mag  1
13          Saldo iniziale       1
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-caus-02.sql | lpr [Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo,

assieme alla stampa del file ‘`interr-caus-02.sql`’.

79.5 Interrogazioni simultanee di più relazioni

Quando si realizzano delle relazioni, spesso si considerano dei collegamenti tra queste, per evitare di ripetere le stesse informazioni in relazioni differenti. La relazione ‘**`Movimenti`**’, creata all’inizio di queste lezioni, contiene diversi attributi che, in pratica, fanno riferimento a tuple di altre relazioni.

Figura 79.52. La relazione ‘**`Movimenti`**’, già apparsa nella figura 79.15.

Movimento	Articolo	Causale	Data	Cliente	Fornitore	Quantita	Valore
1	2	1	2012-01-15	NULL	3	10000	100,00
2	2	2	2012-01-16	2	NULL	1000	10,00
3	102	1	2012-01-17	NULL	2	1000	200,00
4	102	2	2012-01-18	1	NULL	100	20,00
5	401	1	2012-01-19	NULL	1	1000	200,00
6	401	2	2012-01-20	3	NULL	200	40,00
7	401	4	2012-01-20	NULL	1	100	20,00
8	102	4	2012-01-20	NULL	2	100	20,00
9	601	1	2012-01-21	NULL	3	2000	1000,00
10	601	2	2012-01-25	1	NULL	1000	500,00

Intuitivamente si comprende che i dati usati per creare il collegamento con un’altra relazione, devono essere sufficienti a individuare le tuple in modo univoco. Quindi, sulla base di questa univocità, si possono collegare effettivamente i dati attraverso delle interrogazioni che coinvolgono tutte le relazioni interessate, per generare un listato con le informazioni desiderate.

79.5.1 Interrogazione simultanea delle relazioni «Movimenti» e «Articoli»

Con l’ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file ‘`prova-interr-movi-arti.sql`’, conte-

nente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Interrogazione delle relazioni "Movimenti" e "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-interr-movi-arti.sql

.headers on
.mode column

SELECT Movimenti.Data, Articoli.Descrizione,
       Movimenti.Causale, Movimenti.Quantita
       FROM Movimenti, Articoli
       WHERE Movimenti.Articolo = Articoli.Articolo;
```

Come si può vedere, per evitare ambiguità, i nomi degli attributi sono preceduti dal nome della relazione a cui appartengono, separati da un punto.

Si controlli di avere scritto il file '`prova-interr-movi-arti.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-interr-movi-arti.sql[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Data        Descrizione                              Causale     Quantita
----------  ---------------------------------------  ----------  ----------
2012-01-15  Dischetti da 9 cm 1440 Kibyte colorati   1           10000
2012-01-16  Dischetti da 9 cm 1440 Kibyte colorati   2           1000
2012-01-17  CD-R 52x                                 1           1000
2012-01-18  CD-R 52x                                 2           100
2012-01-19  DVD+R 8x                                 1           1000
2012-01-20  DVD+R 8x                                 2           200
```

```
2012-01-20  DVD+R 8x                    4           100
2012-01-20  CD-R 52x                    4           100
2012-01-21  DVD+RW 8x                   1           2000
2012-01-25  DVD+RW 8x                   2           1000
```

79.5.2 Interrogazione simultanea delle relazioni «Movimenti», «Articoli» e «Causali»

Si riprenda il file ‘`prova-interr-movi-arti.sql`’ e lo si modifichi in modo da avere il contenuto seguente:

```
-- Interrogazione delle relazioni "Movimenti", "Articoli"
--   e "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-interr-movi-arti.sql

.headers on
.mode column

SELECT Movimenti.Data, Articoli.Descrizione,
       Causali.Descrizione
       FROM Movimenti, Articoli, Causali
       WHERE Movimenti.Articolo = Articoli.Articolo
         AND Movimenti.Causale = Causali.Causale;
```

Si controlli di avere modificato il file ‘`prova-interr-movi-arti.sql`’ in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-interr-movi-arti.sql [Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Data        Descrizione                                 Descrizione
----------  ------------------------------------------  ------------------
2012-01-15  Dischetti da 9 cm 1440 Kibyte colorati      Carico per acquisto
2012-01-16  Dischetti da 9 cm 1440 Kibyte colorati      Scarico per vendita
2012-01-17  CD-R 52x                                    Carico per acquisto
2012-01-18  CD-R 52x                                    Scarico per vendita
2012-01-19  DVD+R 8x                                    Carico per acquisto
2012-01-20  DVD+R 8x                                    Scarico per vendita
2012-01-20  DVD+R 8x                                    Reso a fornitore
2012-01-20  CD-R 52x                                    Reso a fornitore
2012-01-21  DVD+RW 8x                                   Carico per acquisto
2012-01-25  DVD+RW 8x                                   Scarico per vendita
```

79.5.3 Verifica sull'interrogazione simultanea delle relazioni «Movimenti» e «Causali»

«

Si prepari il file '`interr-movi-caus-01.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco di tutte le tuple per le quali si possa stabilire un abbinamento in base al codice della causale. Precisamente, si vuole ottenere l'attributo '**`Articolo`**' dalla relazione '**`Movimenti`**'; l'attributo '**`Descrizione`**' dalla relazione '**`Causali`**'; l'attributo '**`Data`**' dalla relazione '**`Movimenti`**'.

Figura 79.57. Scheletro del file ‘`interr-movi-caus-01.sql`’, da completare.

```
-- Interrogazione delle relazioni "Movimenti" e "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-movi-caus-01.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
```

Una volta completato e salvato il file ‘`interr-movi-caus-01.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-movi-caus-01.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Descrizione          Data
----------  -------------------  ----------
2           Carico per acquisto  2012-01-15
2           Scarico per vendita  2012-01-16
102         Carico per acquisto  2012-01-17
102         Scarico per vendita  2012-01-18
401         Carico per acquisto  2012-01-19
401         Scarico per vendita  2012-01-20
401         Reso a fornitore     2012-01-20
102         Reso a fornitore     2012-01-20
601         Carico per acquisto  2012-01-21
601         Scarico per vendita  2012-01-25
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-movi-caus-01.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-movi-caus-01.sql`’.

79.5.4 Verifica sull’interrogazione simultanea delle relazioni «Movimenti», «Causali» e «Clienti»

Si prepari il file ‘`interr-movi-caus-clienti-01.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l’elenco di tutte le tuple per le quali si possa stabilire un abbinamento in base al codice della causale e in base al codice del cliente. Precisamente, si vuole ottenere l’attributo ‘**`Articolo`**’ dalla relazione ‘**`Movimenti`**’; l’attributo ‘**`Descrizione`**’ dalla relazione ‘**`Causali`**’; l’attributo ‘**`Data`**’ dalla relazione ‘**`Movimenti`**’; l’attributo ‘**`RagioneSociale`**’ dalla relazione ‘**`Clienti`**’.

Figura 79.59. Scheletro del file ‘`interr-movi-caus-clienti-01.sql`’, da completare.

```
-- Interrogazione delle relazioni "Movimenti", "Causali"
-- e "Clienti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-movi-caus-clienti-01.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
```

Una volta completato e salvato il file ‘`interr-movi-caus-clienti-01.sql`’, se ne controlli il funzionamento con la

base di dati:

```
$ sqlite3 mag.db < interr-movi-caus-clienti-01.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Descrizione          Data        RagioneSociale
----------  -------------------  ----------  --------------
2           Scarico per vendita  2012-01-16  Filano Filani
102         Scarico per vendita  2012-01-18  Mevio Mevi
401         Scarico per vendita  2012-01-20  Martino Martin
601         Scarico per vendita  2012-01-25  Mevio Mevi
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-movi-caus-clienti-01.sql ↩
↪         | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-movi-caus-clienti-01.sql`’.

79.5.5 Verifica sull’interrogazione ordinata e simultanea delle relazioni «Movimenti», «Causali» e «Clienti»

Si prepari il file ‘`interr-movi-caus-clienti-02.sql`’, che deve avere gli stessi requisiti della verifica precedente, facendo in modo, però, che il risultato dell’interrogazione avvenga in modo ordinato, in base alla ragione sociale dei clienti.

Figura 79.61. Scheletro del file ‘interr-movi-caus-clienti-02.sql’, da completare.

```
-- Interrogazione delle relazioni "Movimenti", "Causali"
-- e "Clienti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-movi-caus-clienti-02.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
       ORDER BY …
```

Una volta completato e salvato il file ‘interr-movi-caus-clienti-02.sql’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-movi-caus-clienti-02.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Descrizione          Data        RagioneSociale
----------  -------------------  ----------  --------------
2           Scarico per vendita  2012-01-16  Filano Filani
401         Scarico per vendita  2012-01-20  Martino Martin
601         Scarico per vendita  2012-01-25  Mevio Mevi
102         Scarico per vendita  2012-01-18  Mevio Mevi
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-movi-caus-clienti-02.sql ↩
↪         | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-movi-caus-clienti-02.sql`’.

79.6 Interrogazioni simultanee di più relazioni e alias

Quando si interrogano simultaneamente più relazioni, può succedere che il risultato che si ottiene contenga degli attributi di relazioni differenti, ma con lo stesso nome, oppure potrebbe non essere abbastanza esplicito il suo contenuto. Nell’istruzione ‘**`SELECT`**’ con cui si esegue l’interrogazione, è possibile dichiarare dei nomi alternativi agli attributi, secondo le modalità descritte in questa sezione.

79.6.1 Interrogazione simultanea delle relazioni «Movimenti», «Articoli» e «Causali»

Si riprenda il file ‘`prova-interr-movi-arti.sql`’ e lo si modifichi in modo da avere il contenuto seguente:

```
-- Interrogazione delle relazioni "Movimenti", "Articoli"
-- e "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-interr-movi-arti.sql

.headers on
.mode column

SELECT Movimenti.Data,
       Articoli.Descrizione AS Articolo,
       Causali.Descrizione AS Causale
       FROM Movimenti, Articoli, Causali
       WHERE Movimenti.Articolo = Articoli.Articolo
         AND Movimenti.Causale = Causali.Causale;
```

Si controlli di avere modificato il file '`prova-interr-movi-arti.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-interr-movi-arti.sql[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Data        Articolo                                Causale
----------  --------------------------------------  ------------------
2012-01-15  Dischetti da 9 cm 1440 Kibyte colorati  Carico per acquisto
2012-01-16  Dischetti da 9 cm 1440 Kibyte colorati  Scarico per vendita
2012-01-17  CD-R 52x                                Carico per acquisto
2012-01-18  CD-R 52x                                Scarico per vendita
2012-01-19  DVD+R 8x                                Carico per acquisto
2012-01-20  DVD+R 8x                                Scarico per vendita
2012-01-20  DVD+R 8x                                Reso a fornitore
2012-01-20  CD-R 52x                                Reso a fornitore
2012-01-21  DVD+RW 8x                               Carico per acquisto
2012-01-25  DVD+RW 8x                               Scarico per vendita
```

Come si può osservare, l'attributo '**Descrizione**' della relazione '**Articoli**' appare con il nome '**Articolo**', mentre l'attributo '**Descrizione**' della relazione '**Causali**' appare con il nome '**Causale**'.

79.6.2 Verifica sull'interrogazione simultanea delle relazioni «Movimenti» e «Causali»

Si prepari il file '`interr-movi-caus-02.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco di tutte le tuple per le quali si possa stabilire un abbinamento in base al codice della causale. Precisamente, si vuole ottenere l'attributo '**Articolo**' dalla relazione '**Movimenti**'; l'attributo '**Descrizione**' dalla relazione '**Causali**'; l'attributo '**Data**' dalla relazione '**Movimenti**'. Inoltre, si vuole che l'attributo '**Descrizione**' della relazione '**Causali**', appaia con il nome '**Causale**'.

Figura 79.65. Scheletro del file '`interr-movi-caus-02.sql`', da completare.

```
-- Interrogazione delle relazioni "Movimenti" e "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-movi-caus-02.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
```

Una volta completato e salvato il file '`interr-movi-caus-02.sql`', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-movi-caus-02.sql [Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Causale              Data
----------  -------------------  ----------
2           Carico per acquisto  2012-01-15
2           Scarico per vendita  2012-01-16
102         Carico per acquisto  2012-01-17
102         Scarico per vendita  2012-01-18
401         Carico per acquisto  2012-01-19
401         Scarico per vendita  2012-01-20
401         Reso a fornitore     2012-01-20
102         Reso a fornitore     2012-01-20
601         Carico per acquisto  2012-01-21
601         Scarico per vendita  2012-01-25
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-movi-caus-02.sql | lpr [Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-movi-caus-02.sql`’.

79.6.3 Verifica sull’interrogazione simultanea delle relazioni «Movimenti», «Causali» e «Clienti»

«

Si prepari il file ‘`interr-movi-caus-clienti-03.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l’elenco di tutte le tuple per le quali si possa stabilire un abbinamento in base al codice della causale e in base al codice del cliente. Precisamente, si vuole ottenere l’attributo ‘**`Articolo`**’ dalla relazione ‘**`Movimenti`**’; l’attributo ‘**`Descrizione`**’ dalla relazione ‘**`Causali`**’; l’attributo ‘**`Data`**’ dalla relazione ‘**`Movimenti`**’; l’at-

tributo '**RagioneSociale**' dalla relazione '**Clienti**'. L'attributo '**Descrizione**' della relazione '**Causali**' deve apparire con il nome '**Causale**' e l'attributo '**RagioneSociale**' della relazione '**Clienti**' deve apparire con il nome '**Cliente**'.

Figura 79.67. Scheletro del file '`interr-movi-caus-clienti-03.sql`', da completare.

```
-- Interrogazione delle relazioni "Movimenti", "Causali"
-- e "Clienti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-movi-caus-clienti-03.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
```

Una volta completato e salvato il file '`interr-movi-caus-clienti-03.sql`', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-movi-caus-clienti-03.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Causale              Data        Cliente
----------  -------------------  ----------  --------------
2           Scarico per vendita  2012-01-16  Filano Filani
102         Scarico per vendita  2012-01-18  Mevio Mevi
401         Scarico per vendita  2012-01-20  Martino Martin
601         Scarico per vendita  2012-01-25  Mevio Mevi
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-movi-caus-clienti-03.sql ↩
↪         | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-movi-caus-clienti-03.sql`’.

79.6.4 Conclusione

«

Il file ‘`prova-interr-movi-arti.sql`’ non serve più e va cancellato.

79.7 Viste

«

È possibile trasformare l’interrogazione di una o più relazioni in una ***vista***, la quale diventa in pratica una relazione virtuale.

79.7.1 Creazione della vista «Listino»

«

Con l’ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file ‘`prova-vista-listino.sql`’, contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione della vista "Listino"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-vista-listino.sql

CREATE VIEW Listino AS
            SELECT Articolo AS Codice,
                   Descrizione AS Articolo,
                   Listino AS EUR
                   FROM Articoli;
```

In questo modo, si crea la vista '**Listino**', composta dagli attributi '**Codice**', '**Articolo**' e '**EUR**', utilizzando, rispettivamente, gli attributi '**Articolo**', '**Descrizione**' e '**Listino**' dalla relazione '**Articoli**'.

Si controlli di avere scritto il file '`prova-vista-listino.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-vista-listino.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione della vista '**Listino**' ha avuto successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si deve prima cancellare la vista, quindi si può ritentare l'inserimento del comando (ammesso che il file '`prova-vista-listino.sql`' sia stato corretto di conseguenza). I passaggi per eliminare la vista, in modo interattivo, sono quelli seguenti:

```
$ sqlite3 mag.db[Invio]
```

```
  SQLite version …
  Enter ".help" for instructions
sqlite> DROP VIEW Listino;[Invio]

sqlite> .quit[Invio]
```

Quando si è consapevoli di avere creato correttamente la vista '**Listino**', la si può interrogare come se fosse una relazione normale. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Listino;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
  Codice      Articolo                        EUR
  ----------  ------------------------------  ----------
  1           Dischetti da 9 cm 1440 Kibyte   0.2
  2           Dischetti da 9 cm 1440 Kibyte   0.25
  101         CD-R 16x                        0.5
  102         CD-R 52x                        1
  201         CD-RW 4x                        1
  202         CD-RW 8x                        1.5
  301         DVD-R 8x                        1
  302         DVD-R 16x                       2
  401         DVD+R 8x                        1
  402         DVD+R 16x                       2
```

```
501          DVD-RW 8x                     2
601          DVD+RW 8x                     2
```

Come sempre, si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit[Invio]
```

79.7.2 Creazione della vista «Resi»

Con l'ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file '`prova-vista-resi.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione della vista "Resi" (resi a fornitori)
-- Esercizio di: cognome  nome  classe
-- Data: data
-- File: prova-vista-resi.sql

CREATE VIEW Resi AS
            SELECT Articoli.Descrizione     AS Articolo,
                   Movimenti.Data           AS Data,
                   Fornitori.RagioneSociale AS Fornitore,
                   Movimenti.Quantita       AS Reso,
                   Movimenti.Valore         AS Valore
                   FROM Articoli, Movimenti, Fornitori
                   WHERE Movimenti.Causale   = 4
                     AND Movimenti.Articolo
                         = Articoli.Articolo
                     AND Movimenti.Fornitore
                         = Fornitori.Fornitore;
```

In questo modo, si crea la vista '**Resi**', utilizzando le relazioni '**Articoli**', '**Movimenti**' e '**Fornitori**', limitando la selezione

delle tuple della relazione '**Movimenti**' a quelle che riguardano un reso a fornitore, in quanto la causale corrisponde al numero quattro.

Si controlli di avere scritto il file '`prova-vista-resi.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-vista-resi.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione della vista '**Resi**' ha avuto successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si deve prima cancellare la vista, quindi si può ritentare l'inserimento del comando (ammesso che il file '`prova-vista-resi.sql`' sia stato corretto di conseguenza). I passaggi per eliminare la vista, in modo interattivo, sono quelli seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP VIEW Resi;[Invio]

sqlite> .quit[Invio]
```

Quando si è consapevoli di avere creato correttamente la vista '**Resi**', la si può interrogare come se fosse una relazione normale. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
```

```
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Resi;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Articolo    Data        Fornitore   Reso        Valore
----------  ----------  ----------  ----------  ----------
CD-R 52x    2012-01-20  Caio Cai    100         20
DVD+R 8x    2012-01-20  Tizio Tizi  100         20
```

Come sempre, si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit[Invio]
```

79.7.3 Verifica sulla creazione della vista «Acquisti»

Si prepari il file '`vista-acquisti.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere un elenco dei movimenti di magazzino che riguardano i carichi per acquisto (causale uno). La vista deve essere composta dagli attributi seguenti:

1. '**Articolo**', corrispondente alla descrizione dell'articolo acquistato;
2. '**Data**', corrispondente alla data di acquisto;
3. '**Fornitore**', corrispondente alla ragione sociale del fornitore dal quale l'articolo è stato acquistato;
4. '**Acquistato**', corrispondente alla quantità acquistata;
5. '**Valore**', corrispondente al valore complessivo caricato (pari all'attributo con lo stesso nome della relazione '**Movimenti**').

Figura 79.77. Scheletro del file ‘`vista-acquisti.sql`’, da completare.

```
-- Creazione della vista "Acquisti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-acquisti.sql

CREATE VIEW …
       SELECT …
              FROM …
              WHERE …
```

Una volta completato e salvato il file ‘`vista-acquisti.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < vista-acquisti.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione della vista ‘**`Acquisti`**’ ha avuto successo, altrimenti, è stato commesso un errore. Per rimediare all’errore, si deve prima cancellare la vista, quindi si può ritentare l’inserimento del comando (ammesso che il file ‘`vista-acquisti.sql`’ sia stato corretto di conseguenza).

Quando si è consapevoli di avere creato correttamente la vista ‘**`Acquisti`**’, la si può interrogare come se fosse una relazione normale. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]
```

```
sqlite> .mode column [Invio]

sqlite> SELECT * FROM Acquisti; [Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Articolo                              Data        Fornitore           Acquistato  Valore
------------------------------------  ----------  ------------------  ----------  ----------
Dischetti da 9 cm 1440 Kibyte colorati  2012-01-15  Sempronio Semproni  10000       100
CD-R 52x                              2012-01-17  Caio Cai            1000        200
DVD+R 8x                              2012-01-19  Tizio Tizi          1000        200
DVD+RW 8x                             2012-01-21  Sempronio Semproni  2000        1000
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`vista-acquisti.sql`’.

79.7.4 Verifica sulla creazione della vista «Vendite»

Si prepari il file ‘`vista-vendite.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole ottenere un elenco dei movimenti di magazzino che riguardano gli scarichi per vendita (causale due). La vista deve essere composta dagli attributi seguenti:

1. ‘**`Articolo`**’, corrispondente alla descrizione dell’articolo venduto;
2. ‘**`Data`**’, corrispondente alla data di vendita;
3. ‘**`Cliente`**’, corrispondente alla ragione sociale del cliente al quale l’articolo è stato venduto;
4. ‘**`Venduto`**’, corrispondente alla quantità venduta;
5. ‘**`Valore`**’, corrispondente al valore complessivo scaricato (pari all’attributo con lo stesso nome della relazione ‘**`Movimenti`**’).

Figura 79.80. Scheletro del file ‘`vista-vendite.sql`’, da completare.

```
-- Creazione della vista "Vendite"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-vendite.sql

CREATE VIEW …
       SELECT …
              FROM …
              WHERE …
```

Una volta completato e salvato il file ‘`vista-vendite.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < vista-vendite.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione della vista ‘**`Vendite`**’ ha avuto successo, altrimenti, è stato commesso un errore. Per rimediare all’errore, si deve prima cancellare la vista, quindi si può ritentare l’inserimento del comando (ammesso che il file ‘`vista-vendite.sql`’ sia stato corretto di conseguenza).

Quando si è consapevoli di avere creato correttamente la vista ‘**`Vendite`**’, la si può interrogare come se fosse una relazione normale. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions

sqlite> .headers on[Invio]
```

```
sqlite> .mode column [Invio]

sqlite> SELECT * FROM Vendite; [Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Articolo                              Data        Cliente         Venduto     Valore
------------------------------------  ----------  -------------   ----------  ----------
Dischetti da 9 cm 1440 Kibyte colorati  2012-01-16  Filano Filani  1000        10
CD-R 52x                              2012-01-18  Mevio Mevi      100         20
DVD+R 8x                              2012-01-20  Martino Marti   200         20
DVD+RW 8x                             2012-01-25  Mevio Mevi      1000        500
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`vista-vendite.sql`’.

79.7.5 Conclusione

Prima di proseguire, si deve riprendere il file ‘`magazzino.sql`’ e vi si devono aggiungere le istruzioni per la creazione delle viste ‘**Acquisti**’ e ‘**Vendite**’, come contenuto nei file ‘`vista-acquisti.sql`’ e ‘`vista-vendite.sql`’.

Una volta aggiornato il file ‘`magazzino.sql`’ come descritto, si deve cancellare il file ‘`mag.db`’ e ricreare a partire dalle istruzioni contenute nel file ‘`magazzino.sql`’:

```
$ sqlite3 mag.db < magazzino.sql [Invio]
```

Se vengono segnalati degli errori, occorre correggere il file ‘`magazzino.sql`’, cancellare nuovamente il file ‘`mag.db`’, quindi si deve ripetere l’operazione. La base di dati contenuta nel file ‘`mag.db`’, viene usata ancora e non si può proseguire se non si riesce a ricrearla correttamente.

In precedenza sono stati creati i file ‘`prova-vista-listino.sql`’ e ‘`prova-vista-resi.sql`’, che a questo punto non servono più e vanno cancellati.

79.8 Modifica del contenuto delle tuple

Una volta inserita una tupla in una relazione, si può modificare il suo contenuto con l’istruzione ‘**`UPDATE`**’, la quale si applica a tutte le tuple che soddisfano una certa condizione.

79.8.1 Modifica di una causale di magazzino

Con l’ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file ‘`prova-modifica-causali.sql`’, contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Modifica della relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-modifica-causali.sql

UPDATE Causali
       SET Descrizione = 'car x acq'
       WHERE Causale = 1;
```

In questo modo, si vuole modificare la tupla della relazione ‘**`Causali`**’, con il codice causale uno, in modo che la descrizione risulti molto più breve.

Si controlli di avere scritto il file ‘`prova-modifica-causali.sql`’ in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-modifica-causali.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la modifica della tupla dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, dovrebbe essere sufficiente modificare il file 'prova-modifica-causali.sql' e riprovare.

Quando si è consapevoli di avere modificato correttamente la tupla in questione, si può interrogare la relazione per verificare i cambiamenti apportati. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Causali;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
  Causale     Descrizione  Variazione
  ----------  -----------  ----------
  1           car x acq    1
  2           Scarico per  -1
  3           Reso da cli  1
  4           Reso a forn  -1
  5           Rettifica a  1
  6           Rettifica a  -1
  7           Rettifica d  1
  8           Rettifica d  -1
```

```
9           Carico da p  1
10          Scarico a p  -1
11          Carico da a  1
12          Scarico ad   -1
13          Saldo inizi  1
```

Come sempre, si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit[Invio]
```

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file 'mag.db' e poi ricrearlo con il comando seguente:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

79.8.2 Modifica di diverse causali di magazzino

«

Si riprenda il file 'prova-modifica-causali.sql' e lo si modifichi secondo la forma seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Modifica della relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-modifica-causali.sql

UPDATE Causali
       SET Descrizione = UPPER (Descrizione)
       WHERE Variazione = 1;

UPDATE Causali
       SET Descrizione = LOWER (Descrizione)
       WHERE Variazione = -1;
```

In questo modo, si vuole modificare ogni tupla della relazione '**Causali**' che corrisponde a un aumento di quantità in magazzino (in quanto nell'attributo '**Variazione**' ha il valore +1), in modo da avere una descrizione con tutte lettere maiuscole. Nel contempo, si vuole che le descrizione associate a movimenti in diminuzione, siano scritte utilizzando soltanto caratteri minuscoli.

Si controlli di avere scritto il file '`prova-modifica-causali.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-modifica-causali.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la modifica delle tuple dovrebbe essere stata eseguita con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, dovrebbe essere sufficiente modificare il file '`prova-modifica-causali.sql`' e riprovare.

Quando si è consapevoli di avere modificato correttamente le tuple, si può interrogare la relazione per verificare i cambiamenti apportati. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Causali;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Causale     Descrizione          Variazione
----------  -------------------  ----------
1           CARICO PER ACQUISTO  1
2           scarico per vendita  -1
3           RESO DA CLIENTE      1
4           reso a fornitore     -1
5           RETTIFICA AUMENTO A  1
6           rettifica aumento v  -1
7           RETTIFICA DIMINUZIO  1
8           rettifica diminuzio  -1
9           CARICO DA PRODUZION  1
10          scarico a produzion  -1
11          CARICO DA ALTRO MAG  1
12          scarico ad altro ma  -1
13          SALDO INIZIALE       1
```

Come sempre, si conclude il funzionamento interattivo di '**`sqlite3`**' con il comando '**`.quit`**':

`sqlite>` **`.quit`** [*Invio*]

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file '`mag.db`' e poi ricrearlo con il comando seguente:

`$` **`sqlite3 mag.db < magazzino.sql`** [*Invio*]

79.8.3 Verifica sulla modifica della relazione «Articoli»

«

Si prepari il file '`modifica-articoli.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole cambiare la descrizione del primo e del secondo articolo, in modo da avere rispettivamen-

te: «Floppy 1.4» e «Floppy 1.4 C». Per ottenere questo risultato è necessario utilizzare due volte l'istruzione '**UPDATE**'.

Figura 79.89. Scheletro del file 'modifica-articoli.sql', da completare.

```
-- Modifica della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: modifica-articoli.sql

UPDATE Articoli
       SET …
       WHERE Articolo = 1;

UPDATE Articoli
       SET …
       WHERE Articolo = 2;
```

Una volta completato e salvato il file 'modifica-articoli.sql', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < modifica-articoli.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la modifica delle tuple dovrebbe essere avvenuta con successo, altrimenti è stato commesso un errore. Per rimediare all'errore dovrebbe essere sufficiente correggere il file 'modifica-articoli.sql' e riprovare. Quando si ritiene di avere eseguito l'operazione correttamente, si può interrogare la relazione '**Articoli**' per verificarne il risultato. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]
```

```
  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Articoli;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
  Articolo    Descrizione  UM          Listino     ScortaMin
  ----------  -----------  ----------  ----------  ----------
  1           Floppy 1.4   pz          0.2         500
  2           Floppy 1.4   pz          0.25        500
  101         CD-R 16x     pz          0.5         500
  102         CD-R 52x     pz          1           500
  201         CD-RW 4x     pz          1           200
  202         CD-RW 8x     pz          1.5         200
  301         DVD-R 8x     pz          1           200
  302         DVD-R 16x    pz          2           200
  401         DVD+R 8x     pz          1           200
  402         DVD+R 16x    pz          2           200
  501         DVD-RW 8x    pz          2           200
  601         DVD+RW 8x    pz          2           200
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`modifica-articoli.sql`’.

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file ‘`mag.db`’ e poi ricrearlo con il comando seguente:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

79.8.4 Verifica sulla modifica delle relazioni «Clienti» e «Fornitori»

Si prepari il file ‘`modifica-clienti-fornitori.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole cambiare la ragione sociale delle relazioni ‘**`Clienti`**’ e ‘**`Fornitori`**’, in modo che sia costituita da caratteri maiuscoli. Pertanto, la sostituzione riguarda tutte le tuple in entrambe le relazioni.

Figura 79.92. Scheletro del file ‘`modifica-clienti-fornitori.sql`’, da completare.

```
-- Modifica delle relazioni "Clienti" e "Fornitori"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: modifica-clienti-fornitori.sql

UPDATE Clienti
       SET …

UPDATE Fornitori
       SET …
```

Una volta completato e salvato il file ‘`modifica-clienti-fornitori.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < modifica-clienti-fornitori.sql [Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la modifica delle tuple dovrebbe essere avvenuta con successo, altrimenti è stato commesso un errore. Per rimediare all’errore dovrebbe essere sufficiente correggere il file ‘`modifica-articoli.sql`’ e riprovare. Quando si ritiene di avere eseguito l’operazione correttamente, si

possono interrogare le due relazioni per verificarne il contenuto. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Clienti;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Cliente RagioneSociale Paese  Indirizzo    CAP   Citta     Prov Telefono    Fax          CFPI
------- -------------- ------ ------------ ----- --------- ---- ----------- ------------ -----------
1       MEVIO VEVI     ITALIA via Mare, 11 31050 Morgano   TV   0422,444444 0422,555555 45678901234
2       FILANO FILANI  ITALIA via Farfalle 31032 Feltre    BL   0439,555555 0439,666666 56789012345
3       MARTINO MARTIN ITALIA via Marte, 3 31010 Mareno di TV   0438,666666 0438,777777 67890123456
```

```
sqlite> SELECT * FROM Fornitori;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Fornitore RagioneSociale Paese  Indirizzo     CAP   Citta      Prov Telefono    Fax          CFPI
--------- -------------- ------ ------------- ----- ---------- ---- ----------- ------------ -----------
1         TIZIO TIZI     ITALIA via Tazio, 11 31100 Treviso    TV   0422,111111 0422,222222 12345678901
2         CAIO CAI       ITALIA via Caino, 22 31033 Castelfran TV   0423,222222 0423,333333 23456789012
3         SEMPRONIO SEMP ITALIA via Salina, 3 31057 Silea      TV   0422,333333 0422,444444 34567890123
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`modifica-clienti-fornitori.sql`’.

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file ‘`mag.db`’ e poi ricrearlo con il comando seguente:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

79.8.5 Conclusione

Il file ‘`prova-modifica-causali.sql`’ non serve più e va cancellato.

79.9 Eliminazione delle tuple

La cancellazione delle tuple avviene attraverso l’istruzione ‘**`DELETE FROM`**’, con un procedimento simile a quello della modifica, in quanto va specificata la condizione di cancellazione, altrimenti si ottiene l’eliminazione di tutte le tuple della relazione.

79.9.1 Cancellazione di una causale di magazzino

Con l’ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file ‘`prova-cancella-causali.sql`’, contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Cancellazione nella relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-cancella-causali.sql

DELETE FROM Causali
       WHERE Causale = 1;
```

In questo modo, si vuole eliminare la tupla della relazione ‘**`Causali`**’, con il codice causale uno (quella che ha la descrizione «Carico per acquisto»).

Si controlli di avere scritto il file ‘`prova-cancella-causali.sql`’ in modo corretto, rispettando anche la punteggiatura; si con-

trolli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-cancella-causali.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la cancellazione della tupla dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, dovrebbe essere sufficiente modificare il file '`prova-cancella-causali.sql`' e riprovare. Quando si ritiene di avere cancellato la tupla in questione, si può interrogare la relazione per verificarne lo stato. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Causali;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
  Causale     Descrizione          Variazione
  ----------  -------------------  ----------
  2           Scarico per vendita  -1
  3           Reso da cliente      1
  4           Reso a fornitore     -1
  5           Rettifica aumento a  1
  6           Rettifica aumento v  -1
  7           Rettifica diminuzio  1
  8           Rettifica diminuzio  -1
  9           Carico da produzion  1
```

```
10          Scarico a produzion  -1
11          Carico da altro mag  1
12          Scarico ad altro ma  -1
13          Saldo iniziale       1
```

Come sempre, si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit[Invio]
```

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file 'mag.db' e poi ricrearlo con il comando seguente:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

79.9.2 Cancellazione di diverse causali di magazzino

Si riprenda il file 'prova-cancella-causali.sql' e lo si modifichi secondo la forma seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Cancellazione nella relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-cancella-causali.sql

DELETE FROM Causali
       WHERE Variazione = -1;
```

In questo modo, si vogliono eliminare le tuple corrispondenti a una riduzione della quantità in magazzino, (in quanto nell'attributo '**Variazione**' ha il valore −1).

Si controlli di avere scritto il file ‘`prova-cancella-causali.sql`’ in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-cancella-causali.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la cancellazione dovrebbe avere avuto successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, dovrebbe essere sufficiente modificare il file ‘`prova-cancella-causali.sql`’ e riprovare. Quando si ritiene di avere eseguito l'operazione con successo, si può interrogare la relazione per verificare i cambiamenti apportati. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Causali;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
  Causale     Descrizione          Variazione
  ----------  -------------------  ----------
  1           Carico per acquisto  1
  3           Reso da cliente      1
  5           Rettifica aumento a  1
  7           Rettifica diminuzio  1
  9           Carico da produzion  1
```

```
11          Carico da altro mag  1
13          Saldo iniziale       1
```

Come sempre, si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit[Invio]
```

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file 'mag.db' e poi ricrearlo con il comando seguente:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

79.9.3 Verifica sulla cancellazione di alcuni articoli

Si prepari il file 'cancella-articoli.sql', seguendo lo scheletro seguente, tenendo conto che si vogliono eliminare i dischetti (i primi due).

Figura 79.102. Scheletro del file 'cancella-articoli.sql', da completare.

```
-- Cancellazione di alcune tuple della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: cancella-articoli.sql

DELETE FROM Articoli
       WHERE …

DELETE FROM Articoli
       WHERE …
```

Una volta completato e salvato il file 'cancella-articoli.sql', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < cancella-articoli.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la cancellazione delle tuple dovrebbe essere avvenuta con successo, altrimenti è stato commesso un errore. Per rimediare all'errore dovrebbe essere sufficiente correggere il file '`modifica-articoli.sql`' e riprovare. Quando si ritiene di avere eseguito l'operazione correttamente, si può interrogare la relazione '**`Articoli`**' per verificarne il risultato. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Articoli;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
  Articolo    Descrizione  UM          Listino     ScortaMin
  ----------  -----------  ----------  ----------  ----------
  101         CD-R 16x     pz          0.5         500
  102         CD-R 52x     pz          1           500
  201         CD-RW 4x     pz          1           200
  202         CD-RW 8x     pz          1.5         200
  301         DVD-R 8x     pz          1           200
  302         DVD-R 16x    pz          2           200
  401         DVD+R 8x     pz          1           200
  402         DVD+R 16x    pz          2           200
  501         DVD-RW 8x    pz          2           200
  601         DVD+RW 8x    pz          2           200
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`cancella-articoli.sql`’.

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file ‘`mag.db`’ e poi ricrearlo con il comando seguente:

```
$ sqlite3 mag.db < magazzino.sql [Invio]
```

79.9.4 Conclusione

Il file ‘`prova-cancella-causali.sql`’ non serve più e va cancellato.

79.10 Grilletti per il controllo del dominio degli attributi

Nel momento in cui si inseriscono o si modificano i valori per una tupla di una certa relazione, può essere importante fare in modo di rifiutare i valori impossibili, in quanto non facenti parte del dominio previsto per gli attributi della stessa. Di solito, questo tipo di controllo può essere dichiarato in fase di creazione della relazione; tuttavia, un DBMS limitato potrebbe ignorare tali dichiarazioni.

I ***grilletti*** sono delle funzioni che «scattano», in quanto vengono eseguite, quando si verificano certi eventi. Attraverso i grilletti è possibile impedire l’inserimento di valori errati all’interno degli attributi e questo è l’obiettivo della sezione.

79.10.1 Creazione dei grilletti «Causali_ins» e «Causali_upd»

«

Con l'ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file '`grilletti-causali.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione dei grilletti "Causali_ins" e "Causali_upd"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-causali.sql

CREATE TRIGGER Causali_ins
               BEFORE INSERT ON Causali
               FOR EACH ROW
               BEGIN
                    SELECT CASE
                    WHEN (NEW.Variazione > 1)
                    THEN
                        RAISE (ABORT, 'L''attributo "Variazione" non può essere superiore a 1!')
                    WHEN (NEW.Variazione < -1)
                    THEN
                        RAISE (ABORT, 'L''attributo "Variazione" non può essere inferiore a -1!')
                    WHEN (NEW.Variazione = 0)
                    THEN
                        RAISE (ABORT, 'L''attributo "Variazione" non può essere pari a 0!')
                    END;
                END;

CREATE TRIGGER Causali_upd
               BEFORE UPDATE ON Causali
               FOR EACH ROW
               BEGIN
                    SELECT CASE
                    WHEN (NEW.Variazione > 1)
                    THEN
                        RAISE (ABORT, 'L''attributo "Variazione" non può essere superiore a 1!')
                    WHEN (NEW.Variazione < -1)
                    THEN
                        RAISE (ABORT, 'L''attributo "Variazione" non può essere inferiore a -1!')
                    WHEN (NEW.Variazione = 0)
                    THEN
                        RAISE (ABORT, 'L''attributo "Variazione" non può essere pari a 0!')
                    END;
                END;
```

In questo modo, si creano i grilletti '**Causali_ins**' e '**Causali_upd**', con lo scopo di avvisare in caso di inserimento di un valore impossibile nell'attributo '**Variazione**' della relazione '**Causali**' (sia nel caso di inserimento di una tupla nuova, sia quando si cerca di modificare quell'attributo in una tupla già esistente). Si osservi che, all'interno dei messaggi di errore, l'apostrofo è stato raddoppiato, per evitare che possa essere interpretato come la conclusione della stringa.

Si controlli di avere scritto il file '`grilletti-causali.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < grilletti-causali.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si devono prima cancellare i grilletti, quindi si può ritentare l'inserimento del comando (ammesso che il file '`grilletti-causali.sql`' sia stato corretto di conseguenza). I passaggi per eliminare i grilletti, in modo interattivo, sono quelli seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP TRIGGER Causali_ins;[Invio]

sqlite> DROP TRIGGER Causali_upd;[Invio]
```

```
sqlite> .quit[Invio]
```

Quando si ritiene di avere creato correttamente i grilletti, si può tentare l'inserimento o la modifica di tuple con valori errati nella relazione '**Causali**', per verificare se queste vengono rifiutate come dovrebbero. Si proceda con i passaggi seguenti, utilizzando '**sqlite3**' in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> INSERT INTO Causali VALUES (100, 'Doppio carico', +2);
[Invio]

  INSERT INTO Causali VALUES (100, 'Doppio carico', +2);
  SQL error: L'attributo "Variazione" non può essere ↩
 ↪superiore a 1!
sqlite> INSERT INTO Causali VALUES (101, 'Doppio scarico',
-2);[Invio]

  INSERT INTO Causali VALUES (101, 'Doppio scarico', -2);
  SQL error: L'attributo "Variazione" non può essere ↩
 ↪inferiore a -1!
sqlite> INSERT INTO Causali VALUES (102, 'Movimento nullo',
0);[Invio]

  INSERT INTO Causali VALUES (102, 'Movimento nullo', 0);
  SQL error: L'attributo "Variazione" non può essere pari a 0!
sqlite> UPDATE Causali SET Variazione = +2 WHERE Causale = 1;
[Invio]
```

```
  UPDATE Causali SET Variazione = +2 WHERE Causale = 1;
  SQL error: L'attributo "Variazione" non può essere ↩
 ↪superiore a 1!
```

```
sqlite> UPDATE Causali SET Variazione = -2 WHERE Causale = 2;
```
[*Invio*]

```
  UPDATE Causali SET Variazione = -2 WHERE Causale = 2;
  SQL error: L'attributo "Variazione" non può essere ↩
 ↪inferiore a -1!
```

```
sqlite> UPDATE Causali SET Variazione = 0 WHERE Causale = 3;
```
[*Invio*]

```
  UPDATE Causali SET Variazione = 0 WHERE Causale = 3;
  SQL error: L'attributo "Variazione" non può essere pari a 0!
```

```
sqlite> .quit
```
[*Invio*]

79.10.2 Creazione del grilletto «Articoli_ins» e «Articoli_upd»

Con l'ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file '`grilletti-articoli.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione dei grilletti "Articoli_ins" e "Articoli_upd"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-articoli.sql

CREATE TRIGGER Articoli_ins
               BEFORE INSERT ON Articoli
               FOR EACH ROW
               BEGIN
                    SELECT CASE
                    WHEN (NEW.Listino <= 0)
                    THEN
                        RAISE (ABORT, 'Il prezzo non può essere inferiore o uguale a zero!')
                    WHEN (NEW.ScortaMin < 0)
                    THEN
```

```
                    RAISE (ABORT, 'La scorta minima non può essere inferiore a zero!')
                END;
            END;

CREATE TRIGGER Articoli_upd
            BEFORE UPDATE ON Articoli
            FOR EACH ROW
            BEGIN
                SELECT CASE
                WHEN (NEW.Listino <= 0)
                THEN
                    RAISE (ABORT, 'Il prezzo non può essere inferiore o uguale a zero!')
                WHEN (NEW.ScortaMin < 0)
                THEN
                    RAISE (ABORT, 'La scorta minima non può essere inferiore a zero!')
                END;
            END;
```

In questo modo, si creano i grilletti '**Articoli_ins**' e '**Articoli_upd**', con lo scopo di impedire l'inserimento di valori impossibili per il prezzo di listino e per la scorta minima (sia con le istruzioni '**INSERT**', sia con '**UPDATE**').

Si controlli di avere scritto il file '`grilletti-articoli.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

`$` **`sqlite3 mag.db < grilletti-articoli.sql`** [*Invio*]

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si devono prima cancellare i grilletti, quindi si può ritentare l'inserimento del comando (ammesso che il file '`grilletti-articoli.sql`' sia stato corretto di conseguenza). I passaggi per eliminare i grilletti, in modo interattivo, sono quelli seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP TRIGGER Articoli_ins;[Invio]

sqlite> DROP TRIGGER Articoli_upd;[Invio]

sqlite> .quit[Invio]
```

Quando si ritiene di avere creato correttamente i grilletti in questione, si può tentare l'inserimento di tuple con valori errati nella relazione '**Articoli**', per verificare se queste vengono rifiutate come dovrebbero. Si proceda con i passaggi seguenti, utilizzando '**sqlite3**' in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> INSERT INTO Articoli[Invio]

   ...>     VALUES (660, 'DVD gratis','pz', 0, 200);[Invio]

  INSERT INTO Articoli VALUES (660, 'DVD gratis','pz', 0, 200);
  SQL error: Il prezzo non può essere inferiore o uguale a zero!
sqlite> INSERT INTO Articoli[Invio]

   ...>     VALUES (661, 'DVD ti paghiamo noi','pz', -2.00,
200);[Invio]

  INSERT INTO Articoli VALUES (661, 'DVD ti paghiamo noi','pz', -2.00, 200);
  SQL error: Il prezzo non può essere inferiore o uguale a zero!
```

```
sqlite> INSERT INTO Articoli [Invio]

   ...>     VALUES (662, 'DVD virtuale','pz', 2.00, -200);
[Invio]

  INSERT INTO Articoli VALUES (662, 'DVD virtuale','pz', ↩
 ↪2.00, -200);
  SQL error: La scorta minima non può essere inferiore a zero!

sqlite> UPDATE Articoli SET Listino = 0 WHERE Articolo = 1;
[Invio]

  UPDATE Articoli SET Listino = 0 WHERE Articolo = 1;
  SQL error: Il prezzo non può essere inferiore o uguale a zero!

sqlite> UPDATE Articoli SET Listino = -2.00 WHERE Articolo =
2; [Invio]

  UPDATE Articoli SET Listino = -2.00 WHERE Articolo = 2;
  SQL error: Il prezzo non può essere inferiore o uguale a zero!

sqlite> UPDATE Articoli SET ScortaMin = -200 WHERE Articolo =
101; [Invio]

  UPDATE Articoli SET ScortaMin = -200 WHERE Articolo = 101;
  SQL error: La scorta minima non può essere inferiore a zero!

sqlite> .quit [Invio]
```

79.10.3 Verifica sulla creazione dei grilletti «Movimenti_ins» e «Movimenti_upd»

«

Si prepari il file ‘`grilletti-movimenti.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole impedire l’inserimento nella relazione ‘**Movimenti**’ di quantità inferiori o uguali a zero e di valori inferiori a zero.

Figura 79.123. Scheletro del file ‘`grilletto-movimenti.sql`’, da completare.

```
-- Creazione dei grilletti "Movimenti_ins" e "Movimenti_upd"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-movimenti.sql

CREATE TRIGGER Movimenti_ins
               BEFORE INSERT …
               FOR EACH ROW
               BEGIN
                    …
                    …
                    …
               END;

CREATE TRIGGER Movimenti_upd
               BEFORE UPDATE …
               FOR EACH ROW
               BEGIN
                    …
                    …
                    …
               END;
```

Una volta completato e salvato il file ‘`grilletti-movimenti`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < grilletti-movimenti.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all’errore, si devono prima cancellare i grilletti, quindi si può ritentare l’inserimento del

comando (ammesso che il file ‘`grilletti-movimenti.sql`’ sia stato corretto di conseguenza).

Si consegni per la valutazione la stampa del file ‘`grilletti-movimenti.sql`’.

79.10.4 Conclusione

«

Prima di passare alla sezione successiva, si deve riprendere il file ‘`magazzino.sql`’ e vi si devono aggiungere le istruzioni per la creazione dei grilletti ‘**`Causali_ins`**’, ‘**`Causali_upd`**’, ‘**`Articoli_ins`**’, ‘**`Articoli_upd`**’, ‘**`Movimenti_ins`**’ e ‘**`Movimenti_upd`**’, come contenuto nei file ‘`grilletti-causali.sql`’, ‘`grilletti-articoli.sql`’ e ‘`grilletti-movimenti.sql`’.

Si osservi che la dichiarazione dei grilletti va collocata immediatamente dopo la creazione della relazione a cui fanno riferimento e immediatamente prima delle istruzioni che inseriscono delle tuple.

Una volta aggiornato il file ‘`magazzino.sql`’ come descritto, si deve cancellare il file ‘`mag.db`’ e ricreare a partire dalle istruzioni contenute nel file ‘`magazzino.sql`’:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

Se vengono segnalati degli errori, occorre correggere il file ‘`magazzino.sql`’, cancellare nuovamente il file ‘`mag.db`’, quindi si deve ripetere l’operazione. La base di dati contenuta nel file ‘`mag.db`’, viene usata nella sezione successiva e non si può proseguire se non si riesce a ricrearla correttamente.

79.11 Grilletti per il controllo della validità esterna

Nel momento in cui si inseriscono, modificano o eliminano dei valori per una certa relazione, può essere importante fare in modo di rifiutare le azioni che non sono valide, in base al contenuto di altre relazioni. Di solito, questo tipo di controllo può essere dichiarato in fase di creazione della relazione; tuttavia, un DBMS limitato potrebbe ignorare tali dichiarazioni.

Qui si mostra l'uso dei grilletti per imporre dei vincoli di validità dipendenti dal contenuto di altre relazioni.

79.11.1 Controllo del codice articolo tra la relazione «Movimenti» e la relazione «Articoli»

In precedenza sono stati creati due grilletti, denominati '**`Movimenti_ins`**' e '**`Movimenti_upd`**', con lo scopo di impedire l'inserimento (o la modifica) di valori impossibili per la quantità e per il valore del movimento. Questi due grilletti vengono ripresi ed estesi, allo scopo di impedire che possano essere inseriti movimenti riferiti ad articoli inesistenti, in quanto non ancora dichiarati nella relazione '**`Articoli`**'; inoltre ne viene aggiunto un altro, per impedire che un articolo possa essere eliminato dalla relazione '**`Articoli`**', se questo risulta essere ancora utilizzato nella relazione '**`Movimenti`**'.

Pertanto, si crei il file '`grilletti-movimenti-articoli.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione dei grilletti "Movimenti_ins", "Movimenti_upd" e "Articoli_del"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-movimenti-articoli.sql
```

```
CREATE TRIGGER Movimenti_ins
       BEFORE INSERT ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            END;
       END;

CREATE TRIGGER Movimenti_upd
       BEFORE UPDATE ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            END;
       END;

CREATE TRIGGER Articoli_del
       BEFORE DELETE ON Articoli
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN ((SELECT Articolo FROM Movimenti WHERE Articolo = OLD.Articolo) IS NOT NULL)
            THEN
                RAISE (ABORT, 'L''articolo non può essere rimosso, perché è utilizzato nella relazione Movimenti!')
            END;
       END;
```

Una volta completato e salvato il file '`grilletti-movimenti-articoli`', se ne deve controllare il funzionamento con la base di dati, ma prima vanno rimossi i grilletti '**`Movimenti_ins`**' e '**`Movimenti_upd`**', che qui vengono ricreati. Basta eseguire i passaggi seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP TRIGGER Articoli_ins;[Invio]

sqlite> DROP TRIGGER Articoli_upd;[Invio]

sqlite> .quit[Invio]
```

Quando i grilletti preesistenti sono stati rimossi, si può eseguire il file ‘`grilletti-movimenti-articoli.sql`’ nella base di dati:

```
$ sqlite3 mag.db < grilletti-movimenti-articoli.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all’errore, si devono prima cancellare i grilletti (questa volta sono tre: ‘**Movimenti_ins**’, ‘**Movimenti_upd**’ e ‘**Articoli_del**’), quindi si può ritentare l’inserimento del comando (ammesso che il file ‘`grilletti-movimenti-articoli.sql`’ sia stato corretto di conseguenza).

Per verificare che i vincoli dichiarati funzionino come previsto, si può provare a inserire un movimento che fa riferimento a un articolo inesistente; quindi, si può provare a cancellare un articolo che risulta invece movimentato:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> INSERT INTO Movimenti[Invio]
```

```
   ...>     VALUES (11, 777, 2, '2012-01-25', [Invio]

   ...>     1, NULL, 1000, 500.00); [Invio]

  INSERT INTO Movimenti VALUES (11, 777, 2, '2012-01-25', 1, NULL, 1000, 500.00);
  SQL error: Il codice articolo non è presente nella relazione Articoli!

sqlite> UPDATE Movimenti SET Articolo = 777 [Invio]

   ...>     WHERE Movimento = 2; [Invio]

  UPDATE Movimenti SET Articolo = 777 WHERE Movimento = 2;
  SQL error: Il codice articolo non è presente nella ↩
 ↪relazione Articoli!

sqlite> DELETE FROM Articoli WHERE Articolo = 2; [Invio]

  DELETE FROM Articoli WHERE Articolo = 2;
  SQL error: L'articolo non può essere rimosso, ↩
 ↪perché è utilizzato nella relazione Movimenti!

sqlite> .quit [Invio]
```

79.11.2 Controllo del codice cliente tra la relazione «Movimenti» e la relazione «Clienti»

«

Vengono qui ripresi i grilletti '**Movimenti_ins**' e '**Movimenti_upd**', aggiungendo il grilletto '**Clienti_del**', con lo scopo di impedire che possano essere inseriti movimenti riferiti a clienti inesistenti (in quanto non ancora dichiarati nella relazione '**Clienti**') e di impedire la cancellazione di un cliente quando questo risulta essere ancora utilizzato nella relazione '**Movimenti**'.

Pertanto, si crei il file '`grilletti-movimenti-clienti.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione dei grilletti "Movimenti_ins", "Movimenti_upd" e "Clienti_del"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-movimenti-clienti.sql

CREATE TRIGGER Movimenti_ins
       BEFORE INSERT ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            WHEN ((NEW.Cliente IS NOT NULL)
                  AND ((SELECT Cliente FROM Clienti WHERE Cliente = NEW.Cliente) IS NULL))
            THEN
                RAISE (ABORT, 'Il codice cliente non è presente nella relazione Clienti!')
            END;
       END;

CREATE TRIGGER Movimenti_upd
       BEFORE UPDATE ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            WHEN ((NEW.Cliente IS NOT NULL)
                  AND ((SELECT Cliente FROM Clienti WHERE Cliente = NEW.Cliente) IS NULL))
            THEN
                RAISE (ABORT, 'Il codice cliente non è presente nella relazione Clienti!')
            END;
       END;

CREATE TRIGGER Clienti_del
       BEFORE DELETE ON Clienti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN ((SELECT Cliente FROM Movimenti WHERE Cliente = OLD.Cliente) IS NOT NULL)
            THEN
                RAISE (ABORT, 'Il cliente non può essere rimosso, perché è utilizzato nella relazione Movimenti!')
            END;
       END;
```

A differenza dell'esempio che appare nella sezione precedente, l'attributo '**Cliente**' della relazione '**Movimenti**' può contenere il valore nullo ('**NULL**'). Per questa ragione, il grilletto verifica prima che il valore inserito non sia nullo, poi che il codice cliente esista nella relazione '**Clienti**'.

Una volta completato e salvato il file '`grilletti-movimenti-clienti`', se ne deve controllare il funzionamento con la base di dati, ma prima vanno rimossi i grilletti '**Movimenti_ins**' e '**Movimenti_upd**', che qui vengono ricreati. Basta eseguire i passaggi seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP TRIGGER Articoli_ins;[Invio]

sqlite> DROP TRIGGER Articoli_upd;[Invio]

sqlite> .quit[Invio]
```

Quando i grilletti preesistenti, associati alla relazione '**Movimenti**', sono stati rimossi, si può eseguire il file '`grilletti-movimenti-clienti.sql`' nella base di dati:

```
$ sqlite3 mag.db < grilletti-movimenti-clienti.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si devono prima cancellare i grilletti (questa volta sono tre: '**Movimenti_ins**', '**Movimenti_upd**' e '**Clienti_del**'), quin-

di si può ritentare l'inserimento del comando (ammesso che il file ‘grilletti-movimenti-clienti.sql’ sia stato corretto di conseguenza).

Per verificare che i vincoli dichiarati funzionino come previsto, si può provare a inserire un movimento che fa riferimento a un cliente inesistente; quindi, si può provare a cancellare un articolo che risulta invece movimentato:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> INSERT INTO Movimenti[Invio]
   ...>     VALUES (11, 101, 2, '2012-01-25',[Invio]
   ...>     999, NULL, 1000, 500.00);[Invio]

  INSERT INTO Movimenti VALUES (11, 101, 2, '2012-01-25', ↩
 ↪999, NULL, 1000, 500.00);
  SQL error: Il codice cliente non è presente nella ↩
 ↪relazione Clienti!
sqlite> UPDATE Movimenti SET Cliente = 999 WHERE Movimento =
2;[Invio]

  UPDATE Movimenti SET Cliente = 999 WHERE Movimento = 2;
  SQL error: Il codice cliente non è presente nella ↩
 ↪relazione Clienti!
sqlite> DELETE FROM Clienti WHERE Cliente = 2;[Invio]

  DELETE FROM Clienti WHERE Cliente = 2;
  SQL error: Il cliente non può essere rimosso, perché ↩
 ↪è utilizzato nella relazione Movimenti!
```

`sqlite> .quit` [*Invio*]

79.11.3 Verifica sulla creazione dei grilletti «Movimenti_ins», «Movimenti_upd» e «Causali_del»

«

Si prepari il file ‘`grilletti-movimenti-causali.sql`’, modificando il file ‘`grilletti-movimenti-clienti.sql`’, in modo da riutilizzare quanto già scritto nei grilletti ‘**`Movimenti_ins`**’ e ‘**`Movimenti_upd`**’. Si segua lo scheletro seguente, tenendo conto che si vuole impedire l’inserimento nella relazione ‘**`Movimenti`**’ di causali inesistenti e che si vuole impedire la cancellazione di una causale, dalla relazione ‘**`Causali`**’, se questa risulta utilizzata nella relazione ‘**`Movimenti`**’ (in pratica, per questa funzione ulteriore, si deve aggiungere il grilletto ‘**`Causali_del`**’).

Figura 79.136. Scheletro del file ‘`grilletto-movimenti-causali.sql`’, da completare.

```
-- Creazione dei grilletti "Movimenti_ins", "Movimenti_upd" e "Causali_del"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-movimenti-causali.sql

CREATE TRIGGER Movimenti_ins
       BEFORE INSERT ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            WHEN ((NEW.Cliente IS NOT NULL)
                  AND ((SELECT Cliente FROM Clienti WHERE Cliente = NEW.Cliente) IS NULL))
            THEN
```

```
                RAISE (ABORT, 'Il codice cliente non è presente nella relazione Clienti!')
            WHEN …
            THEN
                …
            END;
        END;

CREATE TRIGGER Movimenti_upd
        BEFORE UPDATE ON Movimenti
        FOR EACH ROW
        BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            WHEN ((NEW.Cliente IS NOT NULL)
                  AND ((SELECT Cliente FROM Clienti WHERE Cliente = NEW.Cliente) IS NULL))
            THEN
                RAISE (ABORT, 'Il codice cliente non è presente nella relazione Clienti!')
            WHEN …
            THEN
                …
            END;
        END;

CREATE TRIGGER Causali_del
        BEFORE DELETE ON Causali
        FOR EACH ROW
        BEGIN
            SELECT CASE
            WHEN …
            THEN
                …
            END;
        END;
```

Una volta completato e salvato il file '`grilletti-movimenti-causali`', se ne deve controllare il funzionamento con la base di dati, ma prima vanno rimossi i grilletti '**`Movimenti_ins`**' e '**`Movimenti_upd`**', che qui vengono ricreati. Basta eseguire i

passaggi seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP TRIGGER Articoli_ins;[Invio]

sqlite> DROP TRIGGER Articoli_upd;[Invio]

sqlite> .quit[Invio]
```

Quando i grilletti preesistenti, associati alla relazione '**Movimenti**', sono stati rimossi, si può eseguire il file '`grilletti-movimenti-causali.sql`' nella base di dati:

```
$ sqlite3 mag.db < grilletti-movimenti-causali.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si devono prima cancellare i grilletti (tutti), quindi si può ritentare l'inserimento del comando (ammesso che il file '`grilletti-movimenti-causali.sql`' sia stato corretto di conseguenza).

Si consegni per la valutazione la stampa del file '`grilletti-movimenti-causali.sql`'.

79.11.4 Verifica sulla creazione dei grilletti «Movimenti_ins», «Movimenti_upd» e «Fornitori_del»

«

Si prepari il file '`grilletti-movimenti-fornitori.sql`', modificando il file '`grilletti-movimenti-causali.sql`', in modo da riutilizzare quanto già scritto nei grilletti '**Movimenti_ins**'

e '**Movimenti_upd**'. Si segua lo scheletro seguente, tenendo conto che si vuole impedire l'inserimento nella relazione '**Movimenti**' di fornitori inesistenti e che si vuole impedire la cancellazione di un fornitore, dalla relazione '**Fornitori**', se questo risulta utilizzato nella relazione '**Movimenti**' (in pratica, per questa funzione ulteriore, si deve aggiungere il grilletto '**Fornitori_del**').

Si osservi che nella relazione '**Movimenti**', l'attributo '**Fornitore**' può avere un valore nullo.

Figura 79.138. Scheletro del file '`grilletto-movimenti-fornitori.sql`', da completare.

```
-- Creazione dei grilletti "Movimenti_ins", "Movimenti_upd" e "Fornitori_del"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-movimenti-fornitori.sql

CREATE TRIGGER Movimenti_ins
       BEFORE INSERT ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            WHEN ((NEW.Cliente IS NOT NULL)
                  AND ((SELECT Cliente FROM Clienti WHERE Cliente = NEW.Cliente) IS NULL))
            THEN
                RAISE (ABORT, 'Il codice cliente non è presente nella relazione Clienti!')
            WHEN …
            THEN
                …
            WHEN …
                  AND …
            THEN
                …
            END;
```

```
        END;

CREATE TRIGGER Movimenti_upd
        BEFORE UPDATE ON Movimenti
        FOR EACH ROW
        BEGIN
             SELECT CASE
             WHEN (NEW.Quantita <= 0)
             THEN
                 RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
             WHEN (NEW.Valore < 0)
             THEN
                 RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
             WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
             THEN
                 RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
             WHEN ((NEW.Cliente IS NOT NULL)
                   AND ((SELECT Cliente FROM Clienti WHERE Cliente = NEW.Cliente) IS NULL))
             THEN
                 RAISE (ABORT, 'Il codice cliente non è presente nella relazione Clienti!')
             WHEN …
             THEN
                 …
             WHEN …
                   AND …
             THEN
                 …
             END;
        END;

CREATE TRIGGER Fornitori_del
        BEFORE DELETE ON Fornitori
        FOR EACH ROW
        BEGIN
             SELECT CASE
             WHEN …
             THEN
                 …
             END;
        END;
```

Una volta completato e salvato il file '`grilletti-movimenti-fornitori`', se ne deve controllare il funzionamento con la base di dati, ma prima vanno rimossi i grilletti '**`Movimenti_ins`**' e '**`Movimenti_upd`**', che qui vengono ricreati. Basta eseguire i

passaggi seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP TRIGGER Articoli_ins;[Invio]

sqlite> DROP TRIGGER Articoli_upd;[Invio]

sqlite> .quit[Invio]
```

Quando i grilletti preesistenti, associati alla relazione '**Movimenti**', sono stati rimossi, si può eseguire il file '`grilletti-movimenti-fornitori.sql`' nella base di dati:

```
$ sqlite3 mag.db < grilletti-movimenti-fornitori.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si devono prima cancellare i grilletti (tutti), quindi si può ritentare l'inserimento del comando (ammesso che il file '`grilletti-movimenti-fornitori.sql`' sia stato corretto di conseguenza).

Si consegni per la valutazione la stampa del file '`grilletti-movimenti-fornitori.sql`'.

79.11.5 Conclusione

Prima di passare alla sezione successiva, si deve riprendere il file '`magazzino.sql`' e vi si devono sostituire le istruzioni per la creazione dei grilletti '**Movimenti_ins**' e '**Movimenti_upd**', come contenuto nel file '`grilletti-movimenti-fornitori.sql`';

inoltre vanno aggiunti i grilletti '**Articoli_del**', '**Causali_del**', '**Clienti_del**' e '**Fornitori_del**', come sono stati realizzati in questa sezione.

Si osservi che la dichiarazione dei grilletti va collocata dopo la creazione della relazione a cui fanno riferimento e prima delle istruzioni che inseriscono delle tuple nella stessa relazione.

Una volta aggiornato il file '`magazzino.sql`' come descritto, si deve cancellare il file '`mag.db`' e ricreare a partire dalle istruzioni contenute nel file '`magazzino.sql`':

```
$ sqlite3 mag.db < magazzino.sql [Invio]
```

Se vengono segnalati degli errori, occorre correggere il file '`magazzino.sql`', cancellare nuovamente il file '`mag.db`', quindi si deve ripetere l'operazione. La base di dati contenuta nel file '`mag.db`', viene usata ancora e non si può proseguire se non si riesce a ricrearla correttamente.

79.12 Selezione di attributi virtuali, ottenuti da un'espressione

«

Il linguaggio SQL consente di costruire delle espressioni elementari, attraverso operatori matematici e funzioni comuni; l'interrogazione di una relazione può essere realizzata anche attraverso l'uso di espressioni.

Tabella 79.140. Operatori aritmetici comuni.

Operatore e operandi	Descrizione
−op	Inverte il segno dell'operando.
op1 `+` ***op2***	Somma i due operandi.
op1 `-` ***op2***	Sottrae dal primo il secondo operando.
op1 `*` ***op2***	Moltiplica i due operandi.
op1 `/` ***op2***	Divide il primo operando per il secondo.
op1 `%` ***op2***	Modulo: il resto della divisione tra il primo e il secondo operando.

Tabella 79.141. Alcune funzioni riconosciute da SQLite.

Funzione	Descrizione
`ABS(`***n***`)`	Restituisce il valore assoluto di ***n***
`LENGTH(`***stringa***`)`	Restituisce la lunghezza in caratteri della stringa indicata come argomento.
`LOWER(`***stringa***`)` `UPPER(`***stringa***`)`	La prima funzione restituisce la stringa indicata come argomento, con lettere minuscole; la seconda con lettere maiuscole.
`MIN(`***x***`,`***y*** `[,…]` `)` `MAX(`***x***`,`***y*** `[,…]` `)`	La prima funzione restituisce il valore minimo tra quelli indicati come argomento; la seconda, invece, restituisce il valore massimo.

Funzione	Descrizione
`ROUND(`***n*** `[,` ***n*** `])`	Restituisce il valore di ***n*** arrotondato a ***m*** decimali. Se ***m*** viene omesso, si intende pari a zero.
`SUBSTR(`***stringa*** `,` ***n*** `,` ***m*** `)`	Estrae la stringa che inizia dalla posizione ***n***, lunga ***m*** caratteri.

79.12.1 Interrogazione della relazione «Movimenti» in modo da ottenere il valore unitario

«

Nella relazione '**`Movimenti`**' appare un attributo denominato '**`Valore`**'. Si tratta del valore dell'articolo, determinato in base al **costo di acquisto** (da non confondere con il prezzo di listino), con il quale si determina il valore delle merci in magazzino. Pre ogni tupla della relazione, si vuole ottenere il valore unitario, che si calcola dividendo il valore per la quantità movimentata corrispondente.

Si crei il file '`prova-interrogazione-movimenti-vu.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Interrogazione della relazione "Movimenti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-interrogazione-movimenti-vu.sql

.mode columns
.headers on

SELECT Articolo,
       Causale,
       Data,
       Quantita,
```

```
        (Valore/Quantita) AS ValoreUnitario
        FROM Movimenti;
```

Si osservi che, nell'ultima colonna del listato che si vuole ottenere, viene indicata l'espressione '**`(Valore/Quantita)`**', associata a un alias, in modo da mostrare una descrizione appropriata.

Una volta completato e salvato il file '`prova-interrogazione-movimenti-vu.sql`', se ne deve controllare il funzionamento con la base di dati:

```
$ sqlite3 mag.db < prova-interrogazione-movimenti-vu.sql [Invio]
```

Si dovrebbe ottenere un listato simile a quello seguente:

```
Articolo    Causale     Data        Quantita    ValoreUnitario
----------  ----------  ----------  ----------  --------------
2           1           2012-01-15  10000       0.01
2           2           2012-01-16  1000        0.01
102         1           2012-01-17  1000        0.2
102         2           2012-01-18  100         0.2
401         1           2012-01-19  1000        0.2
401         2           2012-01-20  200         0.2
401         4           2012-01-20  100         0.2
102         4           2012-01-20  100         0.2
601         1           2012-01-21  2000        0.5
601         2           2012-01-25  1000        0.5
```

Se invece si ottengono degli errori, dovrebbe essere sufficiente correggere il file '`prova-interrogazione-movimenti-vu.sql`' e poi riprovare.

79.12.2 Vista della relazione «Movimenti» in modo da ottenere il valore unitario

«

Così come è possibile scrivere un'interrogazione a una relazione indicando delle espressioni, se ne può realizzare una vista, così da semplificare gli accessi a queste informazioni generate attraverso dei calcoli.

Si crei il file '`vista-movimenti-extra.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Vista "MovimentiExtra"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-movimenti-extra.sql

CREATE VIEW MovimentiExtra AS
    SELECT Movimento,
           Articolo,
           Causale,
           Data,
           Cliente,
           Fornitore,
           Quantita,
           Valore,
           (Valore/Quantita) AS ValoreUnitario
           FROM Movimenti;
```

La vista '**`MovimentiExtra`**' che si ottiene in questo modo, include tutti gli attributi della relazione '**`Movimenti`**', aggiungendo l'attributo virtuale '**`ValoreUnitario`**', ottenuto dividendo il valore complessivo per la quantità movimentata.

Una volta completato e salvato il file ‘`vista-movimenti-extra.sql`’, se ne deve controllare il funzionamento con la base di dati:

```
$ sqlite3 mag.db < vista-movimenti-extra.sql[Invio]
```

Se non vengono generati dei messaggi, l’operazione dovrebbe essere stata completata con successo, altrimenti, se la vista è stata creata, ma in modo errato, è necessario eliminarla, quindi si può correggere il file ‘`vista-movimenti-extra.sql`’ e riprovare. Per eliminare la vista creata in modo errato, si può utilizzare il programma ‘**`sqlite3`**’ in modo interattivo, come già mostrato in altri capitoli (istruzione ‘**`DROP VIEW`**’).

Quando si ritiene di avere creato la vista in modo corretto, è bene verificare di avere ottenuto il risultato desiderato:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode columns[Invio]

sqlite> SELECT * FROM MovimentiExtra[Invio]
```

```
Movimento Articolo Causale Data       Cliente Fornitore Quantita Valore ValoreUnitario
--------- -------- ------- ---------- ------- --------- -------- ------ --------------
1         2        1       2012-01-15         3         10000    100    0.01
2         2        2       2012-01-16 2                 1000     10     0.01
3         102      1       2012-01-17         2         1000     200    0.2
4         102      2       2012-01-18 1                 100      20     0.2
5         401      1       2012-01-19         1         1000     200    0.2
6         401      2       2012-01-20 3                 200      20     0.2
7         401      4       2012-01-20         1         100      20     0.2
8         102      4       2012-01-20         2         100      20     0.2
9         601      1       2012-01-21         3         2000     1000   0.5
10        601      2       2012-01-25 1                 1000     500    0.5
```

```
sqlite> .quit[Invio]
```

79.12.3 Verifica sulla creazione della vista «MovimentiExtra»

In questa verifica si deve riprendere il file ‘`vista-movimenti-extra.sql`’, per modificarlo, in modo da aggiungere un attributo virtuale ulteriore, contenente la quantità in forma algebrica: valori positivi per i carichi e valori negativi per gli scarichi. Dal momento che l’informazione se trattasi di carico o scarico è contenuta nella relazione ‘**`Causali`**’, anche questa va utilizzata nella costruzione della vista.

Si modifichi il file ‘`vista-movimenti-extra.sql`’, seguendo lo scheletro che viene proposto, per far sì che la vista ‘**`MovimentiExtra`**’ contenga gli attributi seguenti:

1. ‘**`Movimento`**’, corrispondente al numero di sequenza assegnato a ogni movimento nella relazione ‘**`Movimenti`**’;
2. ‘**`Articolo`**’, corrispondente al codice articolo della relazione ‘**`Movimenti`**’;
3. ‘**`Causale`**’, corrispondente al codice causale della relazione ‘**`Movimenti`**’;

4. '**Data**', corrispondente alla data del movimento nella relazione '**Movimenti**';
5. '**Cliente**', corrispondente al codice cliente della relazione '**Movimenti**';
6. '**Fornitore**', corrispondente al codice fornitore della relazione '**Movimenti**';
7. '**Quantità**', corrispondente alla quantità movimentata nella relazione '**Movimenti**';
8. '**Valore**', corrispondente al valore del movimento, nella relazione '**Movimenti**';
9. '**ValoreUnitario**', corrispondente al valore unitario del movimento, ottenuto dividendo il valore complessivo per la quantità (dalla relazione '**Movimenti**');
10. '**QuantitaAlgebrica**', corrispondente alla quantità movimentata, con segno, ottenuta moltiplicando l'attributo '**Variazione**' della relazione '**Causali**' all'attributo '**Quantita**' della relazione '**Movimenti**'.

Figura 79.147. Scheletro del file ‘`vista-movimenti-extra.sql`’, da completare.

```
-- Vista "MovimentiExtra"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-movimenti-extra.sql

CREATE VIEW MovimentiExtra AS
    SELECT Movimenti.Movimento          AS Movimento,
           Movimenti.Articolo           AS Articolo,
           Movimenti.Causale            AS Causale,
           Movimenti.Data               AS Data,
           Movimenti.Cliente            AS Cliente,
           Movimenti.Fornitore          AS Fornitore,
           Movimenti.Quantita           AS Quantita,
           Movimenti.Valore             AS Valore,
           (Movimenti.Valore/Movimenti.Quantita)
                                        AS ValoreUnitario,
           …
           FROM …
           WHERE Movimenti.Causale = Causali.causale;
```

Prima di poter eseguire questo file con la base di dati, occorre eliminare la vista ‘**MovimentiExtra**’, che già dovrebbe esistere. Si ricorda che per eliminare una vista si utilizza l’istruzione ‘**DROP VIEW**’ e che conviene intervenire con il programma ‘**sqlite3**’ in modo interattivo.

Per eseguire il file ‘`vista-movimenti-extra.sql`’, si agisce come sempre:

```
$ sqlite3 mag.db < vista-movimenti-extra.sql [Invio]
```

Se la creazione della vista produce degli errori, occorre elimi-

nare nuovamente la vista e, dopo la correzione del file ‘`vista-movimenti-extra.sql`’, si può ritentare.

Quando si è consapevoli di avere creato correttamente la vista ‘**`MovimentiExtra`**’, la si può interrogare come se fosse una relazione normale:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM MovimentiExtra;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Movimento Articolo Causale Data        Cliente Fornitore Quantita Valore ValoreUnitario QuantitaAlgebrica
--------- -------- ------- ----------  ------- --------- -------- ------ -------------- -----------------
1         2        1       2012-01-15          3         10000    100    0.01           10000
2         2        2       2012-01-16  2                 1000     10     0.01           -1000
3         102      1       2012-01-17          2         1000     200    0.2            1000
4         102      2       2012-01-18  1                 100      20     0.2            -100
5         401      1       2012-01-19          1         1000     200    0.2            1000
6         401      2       2012-01-20  3                 200      20     0.2            -200
7         401      4       2012-01-20          1         100      20     0.2            -100
8         102      4       2012-01-20          2         100      20     0.2            -100
9         601      1       2012-01-21          3         2000     1000   0.5            2000
10        601      2       2012-01-25  1                 1000     500    0.5            -1000
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`vista-movimenti-extra.sql`’.

79.12.4 Conclusione

«

Prima di passare alla sezione successiva, si deve riprendere il file ‘`magazzino.sql`’ e vi si deve aggiungere l’istruzione per la creazione della vista ‘**`MovimentiExtra`**’, come realizzato nella verifica appena conclusa.

Una volta aggiornato il file ‘`magazzino.sql`’ come descritto, si deve cancellare il file ‘`mag.db`’ e ricreare a partire dalle istruzioni contenute nel file ‘`magazzino.sql`’:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

Se vengono segnalati degli errori, occorre correggere il file ‘`magazzino.sql`’, cancellare nuovamente il file ‘`mag.db`’, quindi si deve ripetere l’operazione. La base di dati contenuta nel file ‘`mag.db`’, viene usata nella sezione successiva e non si può proseguire se non si riesce a ricrearla correttamente.

79.13 Aggregazioni

«

L’aggregazione è una forma di interrogazione attraverso cui si ottengono risultati riepilogativi del contenuto di una relazione, nel suo complesso o a gruppi di tuple. Per questo si utilizzano delle funzioni speciali al posto dell’espressione che esprime gli attributi del risultato. Queste funzioni restituiscono un solo valore e come tali concorrono a creare un’unica tupla.

Tabella 79.150. Alcune funzioni aggreganti riconosciute da SQLite.

Funzione	Descrizione
`COUNT(`***x***`)`	Restituisce il numero di tuple, nel gruppo, per le quali l'espressione ***x*** restituisce un valore diverso da '**`NULL`**'.
`COUNT(*)`	Restituisce il numero di tuple esistenti nel gruppo.
`AVG(`***x***`)`	Restituisce la media, nel gruppo di tuple, dei valori che ottiene l'espressione ***x***, escludendo '**`NULL`**' e considerando i valori non numerici pari a zero.
`MIN(`***x***`)` `MAX(`***x***`)`	Restituisce il valore minimo o massimo, nel gruppo di tuple, dei valori che ottiene l'espressione ***x***, escludendo '**`NULL`**' e considerando i valori non numerici pari a zero.
`SUM(`***x***`)`	Restituisce la somma, nel gruppo di tuple, dei valori che ottiene l'espressione ***x***, escludendo '**`NULL`**' e considerando i valori non numerici pari a zero.

La forma che assume l'istruzione '**`SELECT`**' quando si usano le aggregazioni e tipicamente quella seguente:

```
SELECT specificazione_dell'attributo_1 [ , ... specificazione_dell'attributo_n ]
    FROM specificazione_della_relazione_1 [ , ... specificazione_della_relazione_n ]
    [ WHERE condizione ]
    [ GROUP BY attributo_1 [ , ... ] ]
```

In pratica, le funzioni aggreganti vanno usate nell'elenco che descri-

ve gli attributi. Se non si usa l'opzione '**GROUP BY**', il gruppo di tuple di riferimento comprende tutte le tuple della relazione o della congiunzione (di relazioni). Se si specifica l'opzione '**GROUP BY**', le tuple vengono raggruppate in base all'uguaglianza degli attributi indicati come argomento di tale opzione. In pratica:

1. la relazione ottenuta dall'istruzione '**SELECT**...**FROM**' viene filtrata dalla condizione '**WHERE**';
2. la relazione risultante viene riordinata in modo da raggruppare le tuple in cui i contenuti degli attributi elencati dopo l'opzione '**GROUP BY**' sono uguali;
3. su questi gruppi di tuple vengono valutate le funzioni di aggregazione.

79.13.1 Aggregazioni banali

«

Per prendere un po' di dimestichezza con le aggregazioni, conviene usare il programma '**sqlite3**' in modo interattivo e fare qualche piccolo esperimento:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]
```

Si vogliono contare le tuple della relazione '**Movimenti**':

```
sqlite> SELECT COUNT(*) FROM Movimenti;[Invio]
```

```
10
```

Si vogliono contare i movimenti per ogni tipo di articolo:

```
sqlite> SELECT Articolo, COUNT(*) FROM Movimenti GROUP BY
Articolo; [Invio]

Articolo    COUNT(*)
----------  ----------
2           2
102         3
401         3
601         2
```

Si vuole conoscere la quantità esistente di ogni articolo (si usa la vista ***MovimentiExtra***, che offre l'attributo '**`QuantitaAlgebrica`**'):

```
sqlite> SELECT Articolo, SUM(QuantitaAlgebrica) [Invio]
   ...>     FROM MovimentiExtra GROUP BY Articolo; [Invio]

Articolo    SUM(QuantitaAlgebrica)
----------  ----------------------
2           9000
102         800
401         700
601         1000
```

Si vuole conoscere la quantità esistente di ogni articolo in magazzino e il valore (il valore viene calcolato a partire da quello medio, moltiplicato per la quantità algebrica):

```
sqlite> SELECT Articolo, SUM(QuantitaAlgebrica), [Invio]
    ...>       SUM(QuantitaAlgebrica*ValoreUnitario) [Invio]
    ...>     FROM MovimentiExtra GROUP BY Articolo; [Invio]
```

```
Articolo    SUM(QuantitaAlgebrica)  SUM(QuantitaAlgebrica*ValoreUnitario)
----------  ----------------------  -------------------------------------
2           9000                    90
102         800                     160
401         700                     140
601         1000                    500
```

Si vuole conoscere la quantità esistente di ogni articolo in magazzino e il costo medio, determinato dividendo il valore complessivo per la quantità esistente:

```
sqlite> SELECT Articolo, [Invio]

sqlite>        SUM(QuantitaAlgebrica), [Invio]

sqlite>        SUM(QuantitaAlgebrica*ValoreUnitario)/↩
↪SUM(QuantitaAlgebrica) [Invio]

sqlite>        FROM MovimentiExtra GROUP BY Articolo; [Invio]
```

```
Articolo    SUM(QuantitaAlgebrica)  SUM(QuantitaAlgebrica*ValoreUnitario)/SUM(QuantitaAlgebrica)
----------  ----------------------  ------------------------------------------------------------
2           9000                    0.01
102         800                     0.2
401         700                     0.2
601         1000                    0.5
```

```
sqlite> .quit [Invio]
```

79.13.2 Verifica sulla creazione della vista «SituazioneMagazzino»

«

Si vuole realizzare la vista '**SituazioneMagazzino**' che, in questa verifica, si limiti a mostrare poche informazioni riepilogative sullo stato del magazzino.

Si realizzi il file '`vista-situazione-magazzino-1.sql`', seguendo lo scheletro che viene proposto, per far sì che la vista

‘**SituazioneMagazzino**’ contenga gli attributi seguenti:

1. ‘**Codice**’, corrispondente al codice articolo della relazione ‘**Movimenti**’ o della vista ‘**MovimentiExtra**’;
2. ‘**Articolo**’, corrispondente alla descrizione dell’articolo, come indicato nella relazione ‘**Articoli**’;
3. ‘**Esistenza**’, corrispondente alla somma algebrica dei carichi, come si ottiene dalla vista ‘**MovimentiExtra**’.

Figura 79.157. Scheletro del file ‘`vista-situazione-magazzino-1.sql`’, da completare.

```
-- Vista "SituazioneMagazzino"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-situazione-magazzino-1.sql

CREATE VIEW SituazioneMagazzino AS
    SELECT MovimentiExtra.Articolo        AS Codice,
           …
           …
           FROM …
           WHERE MovimentiExtra.Articolo = Articoli.Articolo
           GROUP BY MovimentiExtra.Articolo;
```

Per eseguire il file ‘`vista-situazione-magazzino-1.sql`’, si agisce come sempre:

```
$ sqlite3 mag.db < vista-situazione-magazzino-1.sql[Invio]
```

Se la creazione della vista produce degli errori, occorre eliminare la vista e, dopo la correzione del file ‘`vista-situazione-magazzino-1.sql`’, si può ritentare.

Quando si è consapevoli di avere creato correttamente la vista '**SituazioneMagazzino**', la si può interrogare come se fosse una relazione normale:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM SituazioneMagazzino;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Codice      Articolo                                 Esistenza
----------  ---------------------------------------  ----------
2           Dischetti da 9 cm 1440 Kibyte colorati   9000
102         CD-R 52x                                 800
401         DVD+R 8x                                 700
601         DVD+RW 8x                                1000
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file '`vista-situazione-magazzino-1.sql`'.

79.13.3 Verifica sulla creazione della vista «SituazioneMagazzino»

«

Si vuole estendere la vista '**SituazioneMagazzino**', già realizzata in parte nella verifica precedente; pertanto, in questa verifica si modifica il file '`vista-situazione-magazzino-1.sql`' Salvandolo con il nome '`vista-situazione-magazzino-2`'. Si vogliono ottenere gli attributi seguenti:

1. '**Codice**', corrispondente al codice articolo della relazione '**Movimenti**' o della vista '**MovimentiExtra**';
2. '**Articolo**', corrispondente alla descrizione dell'articolo, come indicato nella relazione '**Articoli**';
3. '**ScortaMin**', corrispondente alla scorta minima, come contenuto nella relazione '**Articoli**';
4. '**Esistenza**', corrispondente alla somma algebrica dei carichi, come si ottiene dalla vista '**MovimentiExtra**';
5. '**Valore**', corrispondente al valore complessivo di ogni articolo (come mostrato negli esempi prima di queste verifiche).

Figura 79.160. Scheletro del file '`vista-situazione-magazzino-2.sql`', da completare.

```
-- Vista "SituazioneMagazzino"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-situazione-magazzino-2.sql

CREATE VIEW SituazioneMagazzino AS
    SELECT MovimentiExtra.Articolo            AS Codice,
           …
           …
           FROM …
           WHERE MovimentiExtra.Articolo = Articoli.Articolo
           GROUP BY MovimentiExtra.Articolo;
```

Prima di poter eseguire questo file con la base di dati, occorre eliminare la vista '**SituazioneMagazzino**', che già dovrebbe esistere. Si ricorda che per eliminare una vista si utilizza l'istruzione '**DROP VIEW**' e che conviene intervenire con il programma '**sqlite3**' in modo interattivo.

Per eseguire il file ‘`vista-situazione-magazzino-2.sql`’, si agisce come sempre:

```
$ sqlite3 mag.db < vista-situazione-magazzino-2.sql[Invio]
```

Se la creazione della vista produce degli errori, occorre eliminare la vista e, dopo la correzione del file ‘`vista-situazione-magazzino-2.sql`’, si può ritentare.

Quando si è consapevoli di avere creato correttamente la vista ‘**`SituazioneMagazzino`**’, la si può interrogare come se fosse una relazione normale:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM SituazioneMagazzino;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Codice      Articolo                                ScortaMin   Esistenza   Valore
----------  --------------------------------------  ----------  ----------  ----------
2           Dischetti da 9 cm 1440 Kibyte colorati  500         9000        90
102         CD-R 52x                                500         800         160
401         DVD+R 8x                                200         700         140
601         DVD+RW 8x                               200         1000        500
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`vista-situazione-magazzino-2.sql`’.

79.13.4 Verifica sulla creazione della vista «SituazioneMagazzino»

Si vuole estendere la vista '**SituazioneMagazzino**', già realizzata in parte nella verifica precedente, in modo ottenere anche il costo medio; pertanto, in questa verifica si modifica il file '`vista-situazione-magazzino-2.sql`' salvandolo con il nome '`vista-situazione-magazzino-3.sql`'. Si vogliono ottenere gli attributi seguenti:

1. '**Codice**', corrispondente al codice articolo della relazione '**Movimenti**' o della vista '**MovimentiExtra**';
2. '**Articolo**', corrispondente alla descrizione dell'articolo, come indicato nella relazione '**Articoli**';
3. '**ScortaMin**', corrispondente alla scorta minima, come contenuto nella relazione '**Articoli**';
4. '**Esistenza**', corrispondente alla somma algebrica dei carichi, come si ottiene dalla vista '**MovimentiExtra**';
5. '**Valore**', corrispondente al valore complessivo di ogni articolo (come mostrato negli esempi prima di queste verifiche);
6. '**CostoMedio**', corrispondente al valore complessivo di ogni articolo, diviso la quantità esistente (come mostrato negli esempi prima di queste verifiche).

Figura 79.163. Scheletro del file ‘`vista-situazione-magazzino-3.sql`’, da completare.

```
-- Vista "SituazioneMagazzino"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-situazione-magazzino-3.sql

CREATE VIEW SituazioneMagazzino AS
    SELECT MovimentiExtra.Articolo         AS Codice,
           …
           …
           FROM …
           WHERE MovimentiExtra.Articolo = Articoli.Articolo
           GROUP BY MovimentiExtra.Articolo;
```

Prima di poter eseguire questo file con la base di dati, occorre eliminare la vista ‘**SituazioneMagazzino**’, che già dovrebbe esistere. Si ricorda che per eliminare una vista si utilizza l’istruzione ‘**DROP VIEW**’ e che conviene intervenire con il programma ‘**sqlite3**’ in modo interattivo.

Per eseguire il file ‘`vista-situazione-magazzino-3.sql`’, si agisce come sempre:

```
$ sqlite3 mag.db < vista-situazione-magazzino-3.sql[Invio]
```

Se la creazione della vista produce degli errori, occorre eliminare la vista e, dopo la correzione del file ‘`vista-situazione-magazzino-3.sql`’, si può ritentare.

Quando si è consapevoli di avere creato correttamente la vista ‘**SituazioneMagazzino**’, la si può interrogare come se fosse una relazione normale:

```
$ sqlite3 mag.db [Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on [Invio]

sqlite> .mode column [Invio]

sqlite> SELECT * FROM SituazioneMagazzino; [Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Codice Articolo                                ScortaMin Esistenza Valore CostoMedio
------ --------------------------------------  --------- --------- ------ ----------
2      Dischetti da 9 cm 1440 Kibyte colorati  500       9000      90     0.01
102    CD-R 52x                                500       800       160    0.2
401    DVD+R 8x                                200       700       140    0.2
601    DVD+RW 8x                               200       1000      500    0.5
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`vista-situazione-magazzino-3.sql`’.

79.13.5 Conclusione

Prima di passare alla sezione successiva, si deve riprendere il file ‘`magazzino.sql`’ e vi si deve aggiungere l’istruzione per la creazione della vista ‘**`SituazioneMagazzino`**’, come realizzato nel’ultima verifica appena conclusa.

Una volta aggiornato il file ‘`magazzino.sql`’ come descritto, si deve cancellare il file ‘`mag.db`’ e ricreare a partire dalle istruzioni contenute nel file ‘`magazzino.sql`’:

```
$ sqlite3 mag.db < magazzino.sql [Invio]
```

Se vengono segnalati degli errori, occorre correggere il file ‘`magazzino.sql`’, cancellare nuovamente il file ‘`mag.db`’, quindi

si deve ripetere l'operazione. La base di dati contenuta nel file '`mag.db`', viene usata ancora e non si può proseguire se non si riesce a ricrearla correttamente.

79.14 Inserimento automatico del costo medio

A conclusione di queste lezioni sul linguaggio SQL, viene mostrata la soluzione di un problema, senza richiedere altre verifiche.

Quando si inserisce un movimento nella relazione '**`Movimenti`**', l'utente deve indicare il valore del movimento. Si determina facilmente questo valore quando il bene viene acquistato, in quanto corrisponde al costo complessivo (IVA esclusa). Quando l'articolo viene scaricato per perché reso al fornitore, il valore deve essere lo stesso della fattura a cui si riferisce (in proporzione alla quantità resa), ma quando viene scaricato per la vendita, occorre decidere come attribuire questo valore.

Il modo più semplice per definire il valore del bene che viene scaricato per la vendita, o comunque per scopi diversi dal reso, è quello di calcolare il costo medio ponderato per movimento. In pratica, si tratterebbe di consultare la vista '**`SituazioneMagazzino`**', prima di procedere all'inserimento, in modo da conoscere il costo medio unitario, ottenuto in base ai movimenti esistenti.

Quello che si vuol fare qui è di costruire un grilletto che inserisca automaticamente il valore, determinandolo in base al costo medio ponderato per movimento, quando si inserisce un movimento e si omette l'indicazione del valore stesso.

79.14.1 Vista «CostoMedioValido»

«

La vista '**SituazioneMagazzino**' calcola il costo medio tenendo conto di tutte le tuple, anche quelle che contengono un valore indeterminato del movimento ('**NULL**'). Per lo scopo che si vuole raggiungere, è necessario calcolare il costo medio escludendo i valori indeterminati; pertanto, si realizza una vista apposita:

```
-- Vista "CostoMedioValido"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-costo-medio-valido.sql

CREATE VIEW CostoMedioValido AS
    SELECT Articolo,
           SUM(QuantitaAlgebrica*ValoreUnitario)        AS Valore,
           (SUM(QuantitaAlgebrica*ValoreUnitario)/SUM(QuantitaAlgebrica))
                                                        AS CostoMedio
           FROM MovimentiExtra
           WHERE Valore NOT NULL
           GROUP BY Articolo;
```

79.14.2 Grilletto «ValorizzazioneScarichi»

«

Si può creare un grilletto, che aggiorni automaticamente tutte le tuple della relazione '**Movimenti**', che hanno un valore movimentato indeterminato, traendo il costo medio dalla vista '**CostoMedioValido**':

```
-- Grilletto "ValorizzazioneScarichi"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletto-valorizzazione-scarichi.sql

CREATE TRIGGER ValorizzazioneScarichi
       AFTER INSERT ON Movimenti
       BEGIN
```

```
            UPDATE Movimenti
                   SET Valore =
                       (SELECT CostoMedio * NEW.Quantita FROM CostoMedioValido
                        WHERE Articolo = NEW.articolo)
                   WHERE Valore IS NULL;
        END;
```

Naturalmente, i movimenti che vengono presi in considerazione dal grilletto, sono solo quelli che vengono inseriti **dopo** la sua creazione.

Si osservi, comunque, che occorre anche impedire la sostituzione del valore con qualcosa di indeterminato. In pratica, occorre estendere il grilletto associato alla modifica delle tuple della relazione '**`Movimenti`**', in modo da non accettare valori indeterminati per l'attributo del valore:

```
CREATE TRIGGER Movimenti_upd
       BEFORE UPDATE ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            …
            …
            …
            WHEN (NEW.Valore IS NULL)
            THEN
                RAISE (ABORT, 'In fase di variazione, il valore non può essere indeterminato!')
            END;
       END;
```