Il Linguaggio SQL

SQL è l'acronimo di *Standard Query Language* e identifica un linguaggio di interrogazione (gestione) per basi di dati relazionali. Le sue origini risalgono alla fine degli anni '70 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 Description 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 detto anche SQL2; SQL3.
Il livello SQL3 è ancora in corso di definizione.

L'aderenza dei vari sistemi DBMS allo standard SQL2 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.

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 tabelle, colonne e altri oggetti. Solo quando si tratta di definire il contenuto di una variabile, allora le differenze contano.

La tradizione richiederebbe che, quando si fa riferimento a istruzioni SQL, queste vengano indicate utilizzando solo lettere maiuscole; occorre però dire che alcuni motori di database accettano istruzioni scritte anche con lettere minuscole.

I nomi degli oggetti (tabelle e altro) possono essere composti utilizzando lettere, numeri e il simbolo di sottolineatura; il primo carattere deve essere una lettera oppure il simbolo di sottolineato.

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. Alcuni sistemi SQL richiedono l'uso di un simbolo di terminazione delle righe, che potrebbe essere un 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.

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.

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.

CHARACTER | CHARACTER(*<dimensione>*)

CHAR | CHAR(*<dimensione>*)

Quelle appena mostrate 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 potrà avere. Il minimo è rappresentato dalla stringa nulla.

Costanti stringa

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.

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.

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.

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 potrà 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 precisione inferiore o uguale al tipo **INTEGER**.

FLOAT | FLOAT(<*precisione*>)

REAL

DOUBLE PRECISION

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).

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.

Costanti numeriche

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 attraverso il punto (**.**).

Valori Data-orario e intervalli di tempo

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.

DATE

TIME | TIME(*<precisione>*)

TIME WITH TIME ZONE | TIME(*<precisione>*) WITH TIME ZONE

TIMESTAMP | TIMESTAMP(*<precisione>*)

TIMESTAMP WITH TIME ZONE | TIMESTAMP(*<precisione>*) WITH TIME ZONE

Il tipo **DATE** permette di rappresentare delle date composte dall'informazione anno-mese-giorno.

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.

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), e il tempo universale corrispondente sarebbe invece 21:05:10+0:00.

Quanto mostrato fino a questo punto, 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 qui 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 elenca le parole chiave che rappresentano queste unità di misura.

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

Costanti data-orario

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.

```
'1999-12-31'
'12/31/1999'
'31.12.1999'
```

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.

```
'1999-12-31 12:30:50+1.00'
'1999-12-31 12:30:50.10'
'1999-12-31 12:30:50'
'1999-12-31 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.

Costanti che esprimono intervalli

Un'informazione che rappresenta un intervallo di tempo inizia sempre con la parola chiave **INTERVAL**, ed è seguita da una stringa che contiene l'indicazione di uno o più valori, seguiti ognuno dall'unità di misura relativi (ammesso che ciò sia necessario).
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, ma nel caso di unità più grandi, diventa impossibile.

```
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, e per questo 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.

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.

Operatori aritmetici

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

Operazione	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.

Operazioni con i valori data-orario e intervallo

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

Operazione	Risultato
<data-orario> - *<data-orario>*	Intervallo
<data-orario> +|- *<intervallo>*	Data-orario
<intervallo> + *<data-orario>*	Data-orario
<intervallo> +|- *<intervallo>*	Intervallo
<intervallo> *|/ *<numerico>*	Intervallo
<numerico> * *<intervallo>*	Intervallo

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.

Operazione	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.

Operazione	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 simbolo di sottolineato (_), che rappresenta un carattere qualsiasi, e dal simbolo di percentuale (**%**), che rappresenta una sequenza qualsiasi di caratteri.
La tabella riassume quanto detto.

Espressioni e modelli	Descrizione
<stringa> IS LIKE *<modello>*	*Vero* se il modello corrisponde alla stringa.
<stringa> IS NOT LIKE *<modello>*	*Vero* se il modello non corrisponde alla stringa.

[_]	Rappresenta un carattere qualsiasi.
[%]	Rappresenta una sequenza indeterminata di caratteri.

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

Operatori	Descrizione
<espress.> IS NULL	*Vero* se l'espressione genera un risultato indeterminato.
<espress.> IS NOT NULL	*Vero* se l'espressione non genera un risultato indeterm.

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 riassume gli operatori utilizzabili.

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.

Funzioni Numeriche

ABS(n)	valore assoluto di n.
ROUND(n[,m])	n arrotond. a m cifre decimali; m=0 di default; m puo' essere negativo.
TRUNC(n[,m])	n troncato a m cifre decimali; m=0 di default; m puo' essere negativo.
SIGN(n)	1 se n e' positivo; 0 se n e' 0; -1 se n e' negativo.
CEIL(n)	il piu' piccolo intero maggiore o uguale a n
FLOOR(n)	il piu' grande intero minore o uguale a n
MOD(n,m)	il resto della divisione di n per m
POWER(n,m)	n elevato alla m
SQRT(n)	radice quadrata di n

Funzioni su Stringhe

SUBSTR(char,m[,n])	una sottostringa di char, che inizia al carattere m, lunga n byte (se n manca, lunga fino alla fine della stringa char)
LENGTH(char)	lunghezza della stringa char in byte
CHR(n)	carattere con valore ASCII n
ASCII(char)	valore ASCII del primo carattere della stringa char
UPPER(char)	stringa char con tutte le lettere maiuscole
LOWER(char)	stringa char con tutte le lettere minuscole
INITCAP(char)	stringa char con l' iniziale di ogni parola maiuscola
REPLACE(char,str1[,str2])	char con ogni occorrenza di string1 sostituita da string2 (se manca string2, string1 viene cancellata)
TRANSLATE(char,from,to)	char con ogni carattere presente in from sostituito col corrispondente carattere di to
RPAD(char1,n[,char2])	char1, riempito a destra di char2 fino alla lunghezza n
LPAD(char1,n[,char2])	char1, riempito a sinistra di char2 fino alla lunghezza n
RTRIM(char[,set])	char, con i caratteri finali cancellati dopo l' ultimo car. non in set
LTRIM(char[,set])	char, con i car. iniziali cancellati prima del primo car. non in set

Tabelle

SQL tratta le «relazioni» attraverso il modello tabellare, e di conseguenza si adegua tutta la sua filosofia e il modo di esprimere i concetti nella sua documentazione. Le tabelle di SQL 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, e 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.

Creazione di una tabella

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

CREATE TABLE *<nome-tabella>* (*<specifiche>*)

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

CREATE TABLE *<nome-tabella>* (*<nome-colonna>* *<tipo>*[,...])

In questo modo, all'interno delle parentesi vengono semplicemente elencati i nomi delle colonne seguiti dal tipo di dati che in esse possono essere contenuti.

L'esempio seguente rappresenta l'istruzione necessaria a creare una tabella composta da cinque colonne, 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)
)
```

Valori predefiniti

Quando si inseriscono delle righe all'interno della tabella, in linea di principio è possibile che i valori corrispondenti a colonne particolari non siano inseriti esplicitamente.

Se si verifica questa situazione (purché ciò sia consentito dai vincoli), viene attribuito a questi elementi mancanti un valore predefinito. Questo può essere stabilito all'interno delle specifiche di creazione della tabella, e se questo non è stato fatto, viene attribuito **NULL**, corrispondente al valore indefinito.

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

```
CREATE TABLE <nome-tabella> (
        <nome-colonna> <tipo> [DEFAULT <espressione>] [,...]
)
```

L'esempio seguente crea la stessa tabella 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)
)
```

Vincoli interni alla tabella

Può darsi che in certe situazioni, determinati valori all'interno di una riga non siano ammissibili, a seconda del contesto a cui si riferisce la tabella.
I vincoli interni alla tabella sono quelli che possono essere risolti senza conoscere informazioni esterne alla tabella 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-tabella> (
        <nome-colonna> <tipo> [NOT NULL] [,...]
)
```

L'esempio seguente crea la stessa tabella 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 colonne deve rappresentare dati unici in ogni riga, cioè che non siano ammissibili righe che per quel gruppo di colonne abbiano dati uguali. Segue lo schema sintattico relativo.

```
CREATE TABLE <nome-tabella> (
        <nome-colonna> <tipo> [,...], UNIQUE ( <nome-colonna> [,...] ) [,...]
)
```

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

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

L'esempio seguente crea la stessa tabella già vista negli esempi precedenti, specificando che i dati della colonna del codice devono essere unici per ogni riga.

```
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 una colonna, o un gruppo di colonne, costituisce un riferimento importante per identificare le varie righe che compongono la tabella, 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-tabella> (
        <nome-colonna> <tipo> [,...], PRIMARY KEY ( <nome-colonna>[,...] )
)
```

L'esempio seguente crea la stessa tabella già vista negli esempi precedenti specificando che la colonna del codice deve essere considerata la chiave primaria.

```
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,
        PRIMARY KEY (Codice)
)
```

Vincoli esterni alla tabella

I vincoli esterni alla tabella riguardano principalmente la connessione con altre tabelle, 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 tabella esterna specificata.

Il problema nasce però nel momento in cui nella tabella esterna viene eliminata o modificata una riga che era oggetto di un riferimento da parte della prima. Si pongono le alternative seguenti.

CASCADE
Se nella tabella esterna il dato a cui si fa riferimento è stato cambiato, viene cambiato anche il riferimento nella tabella di partenza; se nella tabella esterna la riga corrispondente viene rimossa, viene rimossa anche la riga della tabella di partenza.

SET NULL
Se viene a mancare l'oggetto a cui si fa riferimento, viene modificato il dato attribuendo il valore indefinito.

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 tabella di partenza.

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

Modifica della struttura della tabella

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

```
ALTER TABLE <nome-tabella> (
        ADD [COLUMN] <nome-colonna> <tipo> [<altre caratteristiche>]
)

ALTER TABLE <nome-tabella> (
        DROP [COLUMN] <nome-colonna>
)
```

Nel primo caso si aggiunge una colonna, della quale si deve specificare il nome e il tipo, ed eventualmente si possono specificare i vincoli; nel secondo si tratta solo di indicare la colonna da eliminare.
A livello di singola colonna può essere eliminato o attribuito un valore predefinito.

```
ALTER TABLE <nome-tabella> (
        ALTER [COLUMN] <nome-colonna> DROP DEFAULT
)

ALTER TABLE <nome-tabella> (
        ALTER [COLUMN] <nome-colonna> SET DEFAULT <valore-predefinito>
)
```

Eliminazione di una tabella

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

```
DROP TABLE <nome-tabella>
```

Inserimento, eliminazione e modifica dei dati

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

Inserimento di righe

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

```
INSERT INTO <nome-tabella>
        VALUES (<espressione-1>[,...<espressione-N>])
```

Per esempio, l'inserimento di una riga nella tabella **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 delle colonne della tabella, i valori mancanti, in coda, ottengono quanto stabilito come valore predefinito, o **NULL** in sua mancanza (sempre che ciò sia concesso dai vincoli della tabella).

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

```
INSERT INTO <nome-tabella> (<colonna-1>[,...<colonna-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 alle colonne garantisce anche che eventuali modifiche di lieve entità nella struttura della tabella non debbano necessariamente riflettersi nei programmi.
L'esempio seguente mostra l'inserimento di alcuni degli elementi della riga, lasciando che gli altri ottengano l'assegnamento di un valore predefinito.

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

Aggiornamento delle righe

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

```
UPDATE <tabella>
        SET <colonna-1>=<espressione-1>[,...<colonna-N>=<espressione-N>]
 [WHERE <condizione>]
```

L'istruzione **UPDATE** esegue tutte le sostituzioni indicate dalle coppie *<colonna>=<espressione>*, per tutte le righe 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 righe della tabella.

L'esempio seguente aggiunge una colonna alla tabella 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 quella colonna.

Eliminazione di righe

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

```
DELETE FROM <tabella> [WHERE <condizione>]
```

N.B. Se la condizione non viene indicata, si cancellano tutte le righe!

Interrogazioni di tabelle

L'interrogazione di una tabella 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 tabelle differenti, in base a delle relazioni che possono intercorrere tra queste.

Interrogazioni elementari

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

SELECT *<espress-col-1>*[,...*<espress-col-N>*]
FROM *<tabella>*
[WHERE *<condizione>*]

In questo modo è possibile definire le colonne che si intendono utilizzare per il risultato, e le righe si specificano, eventualmente, con la condizione posta dopo la parola chiave **WHERE**.

L'esempio seguente mostra la proiezione delle colonne del cognome e nome della tabella di indirizzi già vista negli esempi delle altre sezioni, senza porre limiti alle righe.

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

Quando si vuole ottenere una selezione composta dalle stesse colonne della tabella originale, nel suo stesso ordine, si può utilizzare un carattere jolly particolare, l'asterisco (*****).
Questo rappresenta l'elenco di tutte le colonne della tabella indicata.

```
SELECT * FROM Indirizzi
```

È bene osservare che le colonne si esprimono attraverso un'espressione, questo significa che le colonne a cui si fa riferimento sono quelle del risultato finale, cioè della tabella che viene restituita come selezione o proiezione della tabella originale.

L'esempio seguente emette una sola colonna contenente un ipotetico prezzo scontato del 10%, in pratica viene moltiplicato il valore di una colonna 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 attribuire esplicitamente un nome alle colonne del risultato finale, come indicato dalla sintassi seguente:

SELECT *<espress-col-1>* AS *<nome-col-1>*
    [,...*<espress-col-N>* AS *<nome-col-N>*]
FROM *<tabella>*
[WHERE*<condizione>*]

In questo modo, l'esempio precedente può essere trasformato come segue, dando un nome alla colonna generata e chiarendone così il contenuto.

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

Finora è stata volutamente ignorata la condizione che controlla le righe 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 tabella da cui si attingono. Quindi, non è valida una condizione che utilizza un riferimento a un nome utilizzato dopo la parola chiave **AS** abbinata alle espressioni delle colonne.

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

```
SELECT <specificazione-della-colonna-1>
        [,...<specificazione-della-colonna-N>]
FROM <tabella> AS <alias>
[WHERE <condizione>]
```

Quando si vuole fare riferimento al nome di una colonna, se per qualche motivo questo nome dovesse risultare ambiguo, si può aggiungere anteriormente il nome della tabella a cui appartiene, separandolo attraverso l'operatore punto (**.**).
L'esempio seguente è la proiezione dei cognomi e dei nomi della solita tabella degli indirizzi. In questo caso, le espressioni delle colonne rappresentano solo le colonne corrispondenti della tabella originaria, con l'aggiunta dell'indicazione esplicita del nome della tabella stessa.

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

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

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

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 <tabella>
[WHERE <condizione>]
ORDER BY <espressione> [ASC|DESC] [,...]
```

L'espressione può essere il nome di una colonna, oppure un'espressione che genera un risultato da una o più colonne; 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 delle sole colonne **Cognome** e **Nome**, della tabella **Indirizzi**, ordinato per **Cognome**.

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

Quest'altro 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 )
```

Quest'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.

Interrogazioni simultanee di più tabelle

Se dopo la parola chiave **FROM** si indicano più tabelle (ciò vale anche se si indica più volte la stessa tabella), si intende fare riferimento a una tabella generata dal prodotto di queste.

Se per esempio si vogliono abbinare due tabelle, una di tre righe per due colonne e un'altra di due righe per due colonne, quello che si ottiene sarà una tabella di quattro colonne composta da sei righe. Infatti, ogni riga della prima tabella risulta abbinata con ogni riga della seconda.

SELECT *<specificazione-della-colonna-1>*[,...*<specificazione-della-colonna-N>*]

Vediamo un esempio molto semplice di gestione del magazzino.

Articoli

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

Movimenti

Codice	*Data*	*Carico*	*Scarico*
dado30	01/01/1999	1200	0
vite30	01/01/1999	0	800
vite30	03/01/1999	2000	0
rond50	03/01/1999	0	500

Da questa situazione si vuole ottenere il join della tabella **Movimenti** con tutte le informazioni corrispondenti della tabella **Articoli**, basando il riferimento sulla colonna **Codice**.

In pratica si vuole ottenere la seguente tabella.

Codice	*Data*	*Carico*	*Scarico*	*Descrizione*
dado30	01/01/1999	1200	0	Dado 3 mm
vite30	01/01/1999	0	800	Vite 3 mm
vite30	03/01/1999	2000	0	Vite 3 mm
rond50	03/01/1999	0	500	Rondella 5 mm

Considerato che da un'istruzione **SELECT** contenente il riferimento a più tabelle si genera il prodotto tra queste, si pone poi il problema di eseguire una proiezione delle colonne desiderate, e soprattutto di selezionare le righe.
In questo caso, la selezione deve essere basata sulla corrispondenza tra la colonna **Codice** della prima tabella, con la stessa colonna della seconda. Dovendo fare riferimento a due colonne di tabelle differenti, aventi però lo stesso nome, diviene indispensabile indicare i nomi delle colonne prefissandoli con i nomi delle tabelle 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 tabelle si presta anche per elaborazioni della stessa tabella 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 tabella 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 tabelle: l'***unione***.
Si tratta semplicemente di attaccare il risultato di un'interrogazione su una tabella con quello di un'altra tabella, quando le colonne finali appartengono allo stesso tipo di dati.

```
SELECT <specificazione-della-colonna-1>[,...<specificazione-della-colonna-N>]
FROM <specificazione-della-tabella-1>[,...<specificazione-della-tabella-N>]
[WHERE <condizione>]
        UNION
SELECT <specificazione-della-colonna-1>[,...<specificazione-della-colonna-N>]
FROM <specificazione-della-tabella-1>[,...<specificazione-della-tabella-N>]
[WHERE <condizione>]
```

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

Condizioni

La condizione che esprime la selezione delle righe può essere composta come si vuole, purché il risultato sia di tipo logico e i dati a cui si fa riferimento provengano dalle tabelle 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**.

Aggregazioni

L'aggregazione è una forma di interrogazione attraverso cui si ottengono risultati riepilogativi del contenuto di una tabella, in forma di tabella contenente una sola riga.
Per questo si utilizzano delle funzioni speciali al posto dell'espressione che esprime le colonne del risultato.

Queste funzioni restituiscono un solo valore, e come tali concorrono a creare un'unica riga.
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 righe della tabella **Movimenti** che soddisfano la condizione posta dopo la parola chiave **WHERE** (qui non è stata indicata).

L'asterisco posto come parametro della funzione **COUNT()** rappresenta effettivamente l'elenco di tutti i nomi delle colonne della tabella **Movimenti**.

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

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

```
COUNT( * )

COUNT( [DISTINCT|ALL] <lista-colonne>)
```

Utilizzando la forma già vista, quella dell'asterisco, si ottiene solo il numero delle righe della tabella.

L'opzione **DISTINCT**, seguita da una lista di nomi di colonne, fa in modo che vengano contate le righe contenenti valori differenti per quel gruppo di colonne.

L'opzione **ALL** è implicita quando non si usa **DISTINCT**, e indica semplicemente di contare tutte le righe.

Il conteggio delle righe esclude in ogni caso quelle in cui il contenuto di tutte le colonne selezionate è indefinito (**NULL**).

Le altre funzioni aggreganti non prevedono l'asterisco, perché fanno riferimento a un'espressione che genera un risultato per ogni riga 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()**, e la media **AVG()**.

L'esempio seguente calcola lo stipendio medio degli impiegati, ottenendo i dati da un'ipotetica tabella **Emolumenti**, limitandosi ad analizzare le righe 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 colonna normali.

— Esempio errato:

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

Raggruppamenti

Le aggregazioni possono essere effettuate in riferimento a gruppi di righe, distinguibili in base al contenuto di una o più colonne.
In questo tipo di interrogazione si può generare solo una tabella composta da tante colonne quante sono quelle prese in considerazione dalla clausola di raggruppamento, e da altre contenenti solo espressioni di aggregazione.
Alla sintassi normale già vista nelle sezioni precedenti, si aggiunge la clausola **GROUP BY**.

SELECT *<specificazione-della-colonna-1>*[,...*<specificazione-della-colonna-N>*]
FROM *<specificazione-della-tabella-1>*[,...*<specificazione-della-tabella-N>*]
[WHERE *<condizione>*]
GROUP BY *<colonna-1>*[,...]

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

la tabella ottenuta dall'istruzione **SELECT...FROM** viene filtrata dalla condizione **WHERE**;

la tabella risultante viene riordinata in modo da raggruppare le righe in cui i contenuti delle colonne elencate dopo la clausola **GROUP BY** sono uguali;

su questi gruppi di righe vengono valutate le funzioni di aggregazione.

Si osservi la tabella riportata in figura 3, mostra la solita sequenza di carichi e scarichi di magazzino.

Movimenti

Codice	*Data*	*Carico*	*Scarico*
vite40	01/01/1999	1200	0
vite30	01/01/1999	0	800
vite40	01/01/1999	1500	0
vite30	02/01/1999	0	1000
vite30	03/01/1999	2000	0
rond50	03/01/1999	0	500
vite40	04/01/1999	2200	0

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 seguente.

Codice	*SUM(Carico)*	*Sum(Scarico)*
vite40	4900	0
vite30	2000	1800
rond50	0	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
```

Si è detto che la condizione posta dopo la parola chiave **WHERE** serve a filtrare inizialmente le righe da considerare nel raggruppamento.

Se quello che si vuole è filtrare ulteriormente il risultato di un raggruppamento, occorre usare la clausola **HAVING**.

SELECT *<specificazione-della-colonna-1>*[,...*<specificazione-della-colonna-N>*]
FROM *<specificazione-della-tabella-1>*[,...*<specificazione-della-tabella-N>*]
[WHERE *<condizione>*]
GROUP BY *<colonna-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 precedente risulterebbe escluso l'articolo **rond50**.

Trasferimento di dati in un'altra tabella

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

Creazione di una nuova tabella a partire da altre

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

```
SELECT <specificazione-della-colonna-1>[,...<specificazione-della-colonna-N>]
INTO TABLE <tabella-da-generare>
FROM <specificazione-della-tabella-1>[,...<specificazione-della-tabella-N>]
[WHERE <condizione>]
```

L'esempio seguente crea la tabella **Mia_prova** come risultato della fusione delle tabelle **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;
```

Inserimento in una tabella esistente

L'inserimento di dati in una tabella 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-tabella> [(<colonna-1>...<colonna-N>)]
SELECT <espressione-1>, ... <espressione-N>
FROM <tabelle-di-origine>
[WHERE <condizione>]
```

L'esempio seguente aggiunge alla tabella 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';
```

Viste

Le viste sono delle tabelle virtuali ottenute a partire da tabelle 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 tabella. La vista crea una sorta di interrogazione permanente che acquista la personalità di una tabella normale.

CREATE VIEW *<nome-vista>* [(*<colonna-1>*[,...*<colonna-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
```

Controllare gli accessi

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 sistema Unix, il DBMS può riutilizzare la definizione degli utenti del sistema operativo, farvi riferimento, oppure astrarsi completamente.

Un DBMS SQL richiede la presenza di un DBA (*Data Base Administrator*) che in qualità di amministratore ha sempre tutti i privilegi necessari a intervenire come vuole nel DBMS.
Il nome simbolico predefinito per questo utente dal linguaggio SQL è **_SYSTEM**.

Il sistema di definizione degli utenti è esterno al linguaggio SQL, perché SQL si occupa solo di stabilire i privilegi legati alle tabelle.

Creatore

L'utente che crea una tabella, 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 tabella, o un'altra risorsa, può concedere o revocare i privilegi degli altri utenti su di essa.

Tipi di privilegi

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

SELECT — rappresenta l'operazione di lettura del valore di un oggetto della risorsa, per esempio dei valori di una riga da una tabella (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 riga in una tabella;

UPDATE — rappresenta l'operazione di aggiornamento del valore di un oggetto della risorsa, per esempio la modifica del contenuto di una riga di una tabella;

DELETE — rappresenta l'eliminazione di un oggetto dalla risorsa, come la cancellazione di una riga da una tabella;

ALL PRIVILEGES — rappresenta simultaneamente tutti i privilegi possibili riferiti a un oggetto.

Concedere i privilegi

I privilegi su una tabella, 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 tabella. L'esempio seguente permette all'utente **Pippo** di leggere il contenuto della tabella **Movimenti**.

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

L'esempio seguente, concede tutti i privilegi sulla tabella **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 tabella **Movimenti** e gli permette di concedere lo stesso privilegio ad altri.

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

Revocare i privilegi

I privilegi su una tabella, 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 tabella **Movimenti**.

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

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

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

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 delle istruzioni determinate, e termina 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:

```
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.

```
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 riga nella tabella **Indirizzi**, viene confermata immediatamente la transazione.

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

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

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 più di una sola riga.

Per poter scorrere un elenco ottenuto attraverso un'istruzione **SELECT**, riga per riga, 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 **COMMIT** o un **ROLLBACK**, si chiude anche il cursore.

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 righe 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 delle colonne composte dal collegamento di due tabelle, **Presenze** e **Indirizzi**, dove le righe devono avere lo stesso numero di codice.
Per attivare questo cursore, lo si deve aprire come nell'esempio seguente:

```
OPEN Mio_cursore
```

Scansione

La scansione di un'interrogazione inserita in un cursore, avviene attraverso l'istruzione **FETCH**. Il suo scopo è quello di estrarre una riga 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 riga successiva, quella precedente, la prima, e l'ultima.

Le parole chiave **ABSOLUTE** e **RELATIVE** sono seguite da un numero, corrispondente alla scelta della riga *n*-esima, rispetto all'inizio del gruppo per il quale è stato definito il cursore (**ABSOLUTE**), oppure della riga *n*-esima rispetto all'ultima riga 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 delle varie colonne della riga 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 riga 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 riga precedente.

```
FETCH PRIOR FROM Mio_cursore
```

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

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
```