Appunti di SQL

Corso di Laboratorio di Basi di Dati

Giuseppe Della Penna

9 giugno 2005

Indice

Capitolo 1

Tipi di dato

I tipi di dato in SQL:1999 si suddividono in:

- tipi predefiniti
- tipi strutturati
- tipi user-defined.

Ci concentreremo sui tipi predefiniti (i tipi strutturati e user-defined vengono considerati nelle caratteristiche object-relational di SQL: 1999) I tipi di dato predefiniti sono suddivisi in 5 categorie: tipi numerici tipi binari tipi carattere tipi temporali tipi booleani.

1.1 Tipi numerici

Si dividono in:

- Tipi numerici esatti: rappresentano valori interi e valori decimali in virgola fissa (es. 75, 4.5, -6.2)
- tipi numerici approssimati: rappresentano valori reali in virgola mobile (es. $1256e^{-4}$).

1.1.1 Tipi numerici esatti

INTEGER	Rappresenta i valori interi. La precisione (numero totale di cifre) di questo tipo di dato è espressa in numero di bit o cifre, a seconda della specifica implementazione di SQL.
SMALLINT	Rappresenta i valori interi. I valori di questo tipo sono usati per eventuali ottimizzazioni poichè richiedono minore spazio di memorizzazione. L' unico requisito è che la precisione di questo tipo di dato sia non maggiore della precisione del tipo di dato INTEGER.
NUMERIC	Rappresenta i valori decimali. È caratterizzato da una precisione e una scala (rispettivamente numero totale di cifre da rappresentare e numero di cifre della parte frazionaria). La specifica di questo tipo di dato ha la forma NUMERIC (p, s).
DECIMAL	È simile al tipo NUMERIC. La specifica di questo tipo di dato ha la forma DECIMAL (p, s). La differenza tra NUMERIC e DECIMAL è che il primo deve essere implementato esattamente con la precisione richiesta, mentre il secondo può avere una precisione maggiore (in pratica, il numero viene rappresentato internamente con una precisione possibilmente maggiore e approssimato alla precisione data solo per la visualizzazione).

1.1.2 Tipi numerici approssimati

REAL	Rappresenta valori a singola precisione in virgola mobile. La precisione dipende dalla specifica implementazione di SQL.
DOUBLE PRECISION	Rappresenta valori a doppia precisione in virgola mobile. La precisione dipende dalla specifica implementazione di SQL.
FLOAT	Permette di richiedere la precisione che si desidera (il parametro p indica il numero di cifre da rappresentare). Il formato è FLOAT (p).

1.2 Tipi binari

Questi tipi sono principalmente usati per inserire nella base di dati contenuto non testuale, ad esempio immagini o generici files. La manipolazione di questo tipo di dati non è sempre possibile tramite l'SQL interattivo.

BIT	Rappresenta stringhe di bit di lunghezza fissata. La specifica di questo tipo di dato ha la forma BIT(n) dove n è la lunghezza massima della stringa (se non viene specificata alcuna lunghezza, il default è 1).
BIT VARYING	Rappresenta stringhe di bit a lunghezza variabile con lunghezza massima predefinita. La specifica di questo tipo di dato ha la forma BIT VARYING(n) dove n è la lunghezza massima delle stringhe. La differenza con il tipo BIT è che con questo tipo lo spazio allocato per ogni stringa corrisponde sempre alla lunghezza massima predefinita, mentre per BIT VARYING si usano strategie diverse, allocando uno spazio proporzionale alla effettiva lunghezza della stringa inserita. Per specificare i valori di entrambi i tipi possono essere utilizzate le rappresentazioni binaria o esadecimale (es.01111 o 0x44).
BLOB (Binary Large OBject)	Questo tipo di dato permette di definire sequenze di bit di elevate dimensioni (tipicamente grossi files). Rispetto ai tipi BIT, un BLOB può arrivare alle dimensioni di magebytes o gigabytes. I dati associati vengono memorizzati in files separati, caricati solo se richiesti, e non inclusi nei record, in modo da rendere più rapido l'accesso alle tabelle.

1.3 Tipi carattere

Sono tipi usati per rappresentare caratteri, stringhe o interi blocchi di testo. È possibile associare ai tipi testo un `CHARACTER SET` di riferimento e la relativa `COLLATION` (ordine dei caratteri nel set). Per ognuno dei tipi carattere esiste inoltre la variante `NATIONAL`.

CHARACTER	Questo tipo di dato (spesso abbreviato in CHAR) permette di definire stringhe di caratteri di lunghezza fissata. La specifica di questo tipo di dato ha la forma CHAR(n) dove n è la lunghezza massima delle stringhe (se non viene specificata alcuna lunghezza, il default è 1).
CHARACTER VARYING	Questo tipo di dato (spesso abbreviato in VARCHAR) permette di definire stringhe di caratteri a lunghezza variabile con lunghezza massima predefinita. La specifica di questo tipo di dato ha la forma VARCHAR(n) dove n è la lunghezza massima delle stringhe. La differenza con il tipo CHAR è la stessa descritta per i tipi BIT e BIT VARYING.
CLOB (Character Large OBject)	Questo tipo di dato permette di definire sequenze di caratteri di elevate dimensioni (blocchi di testo).

1.4 Tipi temporali

Permettono di gestire date e ore. Tipicamente il formato usato per specificare queste due informazioni è quello anglosassone, quindi una data dovrà essere inserita come 'anno-mese-giorno' e un'ora come 'ore:minuti:secondi'. Inoltre, nella maggior parte dei DBMS date e ore sono inserite sotto forma di stringhe di caratteri, per cui vanno racchiuse tra apici.

DATE	Rappresenta le date espresse come anno (4 cifre), mese (2 cifre comprese tra 1 e 12), giorno (2 cifre comprese tra 1 e 31 con ulteriori restrizioni a seconda del mese).
TIME	Rappresenta i tempi espressi come ora (2 cifre), minuti (2 cifre) e secondi (2 cifre).
TIMESTAMP	Rappresenta una concatenazione dei due tipi di dato precedenti con una precisione al microsecondo, pertanto permette di rappresentare valori temporali che consistono di anno, mese, giorno, ora, minuto, secondo e microsecondo (di solito specificati nella forma 'anno-mese-giorno ore:minuti:secondi.microsecondi').

1.5 Tipi booleani

I tipi booleani non sono presenti in tutti i DBMS. Spesso sono sostituiti con un dominio definito dall'utente.

BOOLEAN	Rappresenta valori booleani. I valori di tale tipo sono TRUE, FALSE, UNKNOWN.

La logica a tre valori dell'SQL, che include anche il valore UNKNOWN, è utilizzata per trattare i casi in cui alcuni dati presenti in operazioni quali i confronti non siano disponibili (nulli). Le tabelle di verità delle tre operazioni fondamentali, estese con il valore UNKNOWN (?), sono le seguenti.

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

1.6 Valori *null*

Il valore *null* è una speciale costante presente un tutti i DBMS, e rappresenta "assenza di informazione". E' compatibile con ogni tipo di dato e si usa per dare "valore" ad attributi non applicabili o dal valore non noto.

Esempi. I tipi di dato in DB2:

Tipo di dato	Descrizione
SMALLINT	Intero a 16 bit
INTEGER	Intero a 32 bit
DECIMAL (P,S)	Numero decimale con precisione p e scala s (default p=5, s=0). NUMERIC (p,s) è sinonimo di DECIMAL (p,s)
DOUBLE	Numero in virgola mobile a 64 bit. FLOAT e DOUBLE PRECISION sono sinonimi di DOUBLE.
CHAR(N)	Stringa di lunghezza n, dove n $\leq$ 254 (default n=1)
VARCHAR(N)	Stringa di lunghezza variabile con lunghezza massima pari ad n, dove n $\leq$ 4000
DATE	Consiste di anno, mese e giorno
TIME	Consiste di ora, minuti e secondi TIMESTAMP Consiste di anno, mese, giorno, ora, minuti, secondi, microsecondi.

1.7 Domini

La definizione dei domini fa propriamente parte del DDL. Tuttavia, viene anticipata in questo capitolo in quanto concernente la definizione di dipi di dato.

SQL permette di definire dei domini e utilizzarli nella definizione di tabelle. Un dominio è un tipo di dato derivato, basato su uno dei tipi semplici visti finora, cui si possono associare particolari vincoli o informazioni.

```
CREATE DOMAIN d AS t [DEFAULT v] [CHECK(vincolo)];
```

definisce il dominio d (il nome deve essere una stringa alfanumerica), basandolo sul tipo t. Opzionalmente è possibile specificare un default d per i valori associati a questo dominio (il default verrà usato per inizializzare tutti gli attributi dichiarati usando il dominio come tipo) e/o un vincolo di tipo CHECK. Un vincolo check è definito come un predicato sullo speciale argomento VALUE, che rappresenta i possibili valori ammessi nel dominio. Il dominio conterrà i soli valori per cui la valutazione del predicato risulta vera.

Nella composizione del predicato, oltre alla parola chiave VALUE, si possono usare tutte le operazioni descritte nel capitolo operatori e funzioni comuni dell'SQL.

Esempi.

```
CREATE DOMAIN tipoEta AS INTEGER CHECK(VALUE BETWEEN 1 AND 120);
CREATE DOMAIN occupazione AS VARCHAR(30) DEFAULT 'disoccupato';
CREATE DOMAIN DominioMansione AS CHAR(10) CHECK (VALUE IN('
   DIRIGENTE','INGEGNERE','TECNICO','SEGRETARIA'));
```

1.7.1 Modificare un dominio

```
ALTER DOMAIN d SET DEFAULT v
```

modifica il valore di default del dominio d in v.

```
ALTER DOMAIN d DROP DEFAULT
```

elimina il valore di default del dominio d.

```
ALTER DOMAIN d DROP CONSTRAINT
```

rimuove il vincolo (check) dal dominio d.

```
ALTER DOMAIN d ADD CONSTRAINT CHECK(vincolo)
```

aggiunge il vincolo check specificato al dominio d.

1.7.2 Cancellare un dominio

```
DROP DOMAIN d {RESTRICT | CASCADE}
```

rimuove il dominio d.

- Se viene specificato RESTRICT: il dominio viene rimosso solo senessuna tabella lo utilizza
- Se viene specificato CASCADE: in ogni tabella che lo utilizza il nome del dominio viene sostituito dalla sua definizione (la modifica non influenza i dati presenti nella tabella)

Capitolo 2

Operatori e funzioni comuni dell'SQL

Il linguaggio SQL definisce una serie di operatori e funzioni di base che possono essere usati in diversi comandi, ad esempio per creare vincoli CHECK, per filtrare i record di una o più tabelle, o per derivare valori dai dati presenti nel database.

In questo capitolo presenteremo le funzioni e gli operatori più comuni, presenti in tutti i dialetti SQL

2.1 Funzioni aritmetiche di base

SQL mette a disposizione tutte le usuali funzioni aritmetiche, indicate con gli operatori +,-,* e /. Nel calcolo delle espressioni aritmetiche, il valore nullo viene considerato uguale al valore zero.

2.2 Funzioni scalari

Sebbene non siano fornite in tutte le implementazioni di SQL, le funzioni che seguono sono molto comuni.

- ABS(n) calcola il valore assoluto del valore numerico n.
- MOD(n,b) calcola il resto intero della divisione di n per b.
- SQRT(n) calcola la radice quadrata di n.

Alcuni DBMS supportano anche funzioni trigonometriche e funzioni per il calcolo della parte intera superiore ed inferiore.

2.3 Espressioni e Funzioni per Stringhe

Questi operatori si applicano ai tipi carattere. Anche in questo caso, diversi dialetti di SQL potrebbero offrire insiemi di funzioni diversi.

- LENGTH(s): calcola la lunghezza della stringa s.
- SUBSTR(s, m, [n]) (m ed n sono interi): estrae dalla stringa s la sottostringa dal carattere di posizione m per una lunghezza n (se n è specificato) oppure fino all'ultimo carattere.
- s1 || s2: restituisce la concatenazione delle due stringhe s1 e s2.

2.4 Funzioni e costanti per tipi temporali

Le funzioni temporali sono quelle più variabili tra DBMS diversi. Le costanti temporali sono invece di solito ampiamente supportate

- DATE(v), TIME(v), TIMESTAMP(v): convertono rispettivamente un valore scalare in una data, un tempo, un timestamp.
- CHAR(d, [f]): converte un valore di data/tempo d in una stringa di caratteri; può inoltre ricevere una specifica di formato f.
- DAYS(v): converte una data v in un intero che rappresenta il numero di giorni a partire dall'anno zero.
- CURRENT_DATE: rappresenta la data corrente
- CURRENT_TIME: rappresenta l'ora corrente
- CURRENT_TIMESTAMP: rappresenta il timestamp corrente

Esempi.

```
DATE('6/20/1997')
CHAR(DATA$\_$ASSUNZIONE, EUR)
DAYS('1/8/2001')
```

2.4.1 Espressioni aritmetiche con valori temporali

Date e tempi possono essere usati in espressioni aritmetiche; in tali espressioni è possibile usare diverse unità temporali quali YEAR[S], MONTH[S], DAY[S], HOUR[S], MINUTE[S], SECOND[S], da posporre ai numeri.

Esempi.

```
    DATA + 90 DAYS >= CURRENT_DATE;
```

l'espressione è vera se la data è al più tre mesi nel passato.

2.5 Operatori di confronto

SQL mette a disposizione tutte gli usuali operatori di confronto, cioè maggiore '>', minore '<', maggiore o uguale '>=', minore o uguale '<=', uguale '=', diverso '<>'. Nella valutazione di questi operatori, la presenza di almeno un operando nullo porta alla generazione di un valore logico UNKNOWN come risultato.

2.6 Intervalli di valori

L'operatore `BETWEEN` permette di determinare se un valore si trova in un intervallo dato.

```
c BETWEEN v1 AND v2
c NOT BETWEEN v1 AND v2
```

le due espressioni sono vere se, rispettivamente, il valore c è compreso o non compreso tra i valori v1 e v2.

2.7 Ricerca di valori in un insieme

L'operatore `IN` permette di determinare se un valore si trova in un insieme specificato.

```
c IN (v1, v2,$\ldots$ ,vn)
c NOT IN (v1, v2,$\ldots$ ,vn)

c IN q
c NOT IN q
```

le prime due espressioni sono vere se, rispettivamente, il valore c è compreso o non è compreso tra i valori v1, v2, . . . , vn.

Le due espressioni che seguono hanno lo stesso significato, ma in questo caso la lista di valori non è dichiarata esplicitamente, ma viene generata da una interrogazione q sulla base di dati. Questa interrogazione deve necessariamente restituire una tabella costituita da una sola colonna.

2.8 Confronto tra stringhe di caratteri

L'operatore `LIKE` permette di eseguire alcune semplici operazioni di pattern-matching su valori di tipo stringa.

```
c LIKE pattern
```

l'espressione è vera se il pattern fa match con la stringa c.

Un pattern è una stringa di caratteri che può contenere i caratteri speciali (metacaratteri o wildcards) % e _

- il carattere % denota una sequenza di caratteri arbitrari di lunghezza qualsiasi (anche zero);
- il carattere _ denota esattamente un carattere.

ogni altro carattere nel pattern fa match solo con un carattere identico nella stringa.

Esempi.

- 'M%A' fa match con 'MA, 'MAMMA', 'MINESTRA', ecc.
- '%MA' fa match con 'MAMMA', 'MA', 'PROGRAMMA', ecc.
- '%MA%' fa match con qualsiasi stringa che contenga la sottostringa 'MA'
- '___' (tre underscore) fa match con qualsiasi stringa di esattamente tre caratteri
- 'A_R%' fa match con 'ABRACADABRA', 'ARRIVO', ecc.

2.9 Operatori logici

SQL mette a disposizione i quattro usuali operatori logici, indicati con le parole chiave `AND`, `OR` e `NOT`. Questi operatori possono essere usati per costruire condizioni logiche complesse a partire dai test semplici visti finora.

2.10 Trattamento dei valori *null*

In SQL-89, la presenza di valori *null* rende ogni predicato falso.

Nelle versioni successive di SQL, si può fare uso uso del predicato **`IS [ NOT ] NULL`** per effettuare un test sulla presenza o assenza di valori nulli. Per trattare i valori *null*, nel caso l'utente non specifichi esplicitamente un predicato **`IS [ NOT ] NULL`**, SQL usa le logica a tre valori vista nel capitolo *tipi di dato*.

Capitolo 3

Definizione dei Dati (DDL)

3.1 Creazione di tabelle

La creazione di una tabella avviene tramite il comando `CREATE TABLE`, che ha la seguente sintassi:

```
CREATE TABLE nome_tabella(
  spec_col${_1}$,
    ...
  spec_col${_n}$,
  vincolo_tabella${_1}$,
    ...
  vincolo_tabella${_n}$
);
```

dove:

- nome_tabella è il nome della relazione che viene creata;
- spec_col$_i$ (i=1...n) è una specifica di colonna;
- vincolo_tabella$_i$ è la specifica di un vincolo per le tuple della relazione.

Il formato di una specifica di colonna è il seguente:

```
nome_colonna dominio [DEFAULT v] [vincolo_colonna${_1}$ ...
   vincolo_colonna${_m}$]
```

dove:

- nome_colonna è il nome della colonna che viene creata;
- dominio è un tipo base di SQL o il nome di un dominio creato dall'utente;
- `DEFAULT` v permette di dichiarare il valore di default per la colonna;

- vincolo_colonna$_i$ è la specifica di un vincolo per la colonna.

I vincoli verranno discussi più avanti.

Esempi.

```
CREATE TABLE IMPIEGATI(
 IDIMPIEGATO DECIMAL(4),
 NOME CHAR(20),
 MANSIONE CHAR( 10),
 DATA_A DATE,
 STIPENDIO DECIMAL(7 ,2),
 PREMIO_P DECIMAL(7,2) DEFAULT 0,
 IDREPARTO DECIMAL(2)
);
```

```
CREATE DOMAIN DominioMansione AS CHAR(10) DEFAULT 'IMPIEGATO';

CREATE TABLE IMPIEGATI (
  IDIMPIEGATO DECIMAL(4),
  NOME CHAR(20),
  MANSIONE DominioMansione,
  DATA_A DATE,
  STIPENDIO DECIMAL(7,2),
  PREMIO_P DECIMAL(7,2) DEFAULT 0,
  IDREPARTO DECIMAL(2)
);
```

3.1.1 Cancellare una tabella

```
DROP TABLE r
```

cancella la tabella r.

3.1.2 Rinominare una tabella

```
RENAME Rv TO Rn
```

rinomina la ravella Rv in Rn.

3.1.3 Modificare le colonne di una tabella

```
ALTER TABLE r ADD COLUMN spec_col
```

aggiunge una nuova colonna ad una relazione r.

```
ALTER TABLE r DROP COLUMN nome_colonna
```

rimuove la colonna nome_colonna dalla relazione r.

```
ALTER TABLE r ALTER COLUMN nome_colonna SET DEFAULT v
```

modifica il valore di default della colonna nome_colonna nella relazione r, impostandolo al valore v.

```
ALTER TABLE r ALTER COLUMN nome_colonna DROP DEFAULT
```

elimina il valore di default della colonna nome_colonna nella relazione r.

Esempi.

```
ALTER TABLE IMPIEGATI ADD COLUMN PROG# DECIMAL(3);
ALTER TABLE IMPIEGATI DROP COLUMN STIPENDIO;
ALTER TABLE IMPIEGATI ALTER COLUMN PROG# SET DEFAULT 0;
ALTER TABLE IMPIEGATI ALTER COLUMN PROG# DROP DEFAULT;
```

3.2 Vincoli di integrità

Un vincolo è una regola che specifica delle condizioni sui valori delle colonne in una relazione. Un vincolo può essere associato a una tabella o a un songolo attributo. In SQL è possibile specificare diversi tipi di vincoli:

- Chiavi (`UNIQUE` e `PRIMARY KEY`)
- Obbligatorietà di attributi (`NOT NULL`)
- Chiavi esterne (`FOREIGN KEY`)
- Vincoli generali (`CHECK`)

È possibile specificare se il controllo del vincolo debba essere compiuto non appena si esegue un'operazione che ne può causare la violazione (`NON DEFERRABLE`) o se possa essere rimandato alla fine della transazione (`DEFERRABLE`). Per i vincoli differibili si può specificare un check time iniziale: `INITIALLY DEFERRED` (default) o `INITIALLY IMMEDIATE`. I vincoli non possono contenere condizioni la cui valutazione può dare risultati differenti a seconda momento in cui sono esaminati (es. riferimenti al tempo di sistema).

3.2.1 Chiavi

La specifica delle chiavi si effettua in SQL mediante le parole chiave `UNIQUE` o `PRIMARY KEY`:

- `UNIQUE` garantisce che non esistano due tuple che condividono gli stessi valori non nulli per gli attributi specificati (colonne `UNIQUE` possono contenere valori nulli).
- `PRIMARY KEY` impone che per ogni tupla i valori degli attributi specificati siano non nulli e diversi da quelli di ogni altra tupla.

In una tabella è possibile specificare più chiavi `UNIQUE` ma una sola `PRIMARY KEY`. Se la chiave è formata da un solo attributo è sufficiente far seguire la specifica dell'attributo dalla parola chiave `UNIQUE` o `PRIMARY KEY` (vincolo su attributo). Alternativamente si può far seguire la definizione della tabella dala vincolo su tabella `PRIMARY KEY(lista_colonne)` o `UNIQUE(lista_colonne)`, dove lista_colonne è una lista, separata da virgole, dei nomi delle colonne coinvolte nella chiave.

Esempi.

```
CREATE TABLE IMPIEGATI (
 IDIMPIEGATO DECIMAL(4) PRIMARY KEY,
 CODICE_FISCALE CHAR(16) UNIQUE,
 NOME CHAR(20),
 MANSIONE DominioMansione,
 DATA_A DATE,
 STIPENDIO DECIMAL(7 ,2),
 PREMIO_P DECIMAL(7 ,2),
 IDREPARTO DECIMAL(2)
);
```

```
CREATE TABLE FILM (
 TITOLO CHAR(20),
 ANNO INTEGER,
 STUDIO CHAR(20),
 COLORE BOOLEAN,
 PRIMARY KEY (Titolo,Anno));
```

3.2.2 Valori *null*

Per indicare che un colonna non può assumere valore nullo (che è il default per tutti gli attributi di una tupla, se non specificato diversamente tramite la parola chiave `DEFAULT`) è sufficiente includere il vincolo `NOT NULL` nella specifica della colonna. Questo rende obbligatorio l'inserimento esplicito di un valore nella colonna per ogni record.

Esempi.

```
CREATE TABLE IMPIEGATI (
 IDIMPIEGATO DECIMAL(4) PRIMARY KEY,
 CODICE_FISCALE CHAR(16) UNIQUE,
 NOME CHAR(20) NOT NULL,
 MANSIONE DominiMansione,
 DATA_A DATE,
 STIPENDIO DECIMAL(7,2) NOT NULL,
 PREMIO_P DECIMAL(7,2),
 IDREPARTO DECIMAL(2) NOT NULL
);
```

3.2.3 Chiavi esterne

Una chiave esterna specifica che i valori di un particolare insieme di attributi (chiave esterna) di ogni tupla devono necessariamente corrispondere a quelli presenti in un corrispondente insieme di attributi che sono una chiave per le tuple di un'altra relazione. Una relazione può avere zero o più chiavi esterne. La specifica di chiavi esterne avviene mediante la clausola `FOREIGN KEY ... REFERENCES`:

```
FOREIGN KEY (lista_colonne1) REFERENCES nome_tabella (
   lista_colonne2)
[MATCH{FULL|PARTIAL|SIMPLE}]
[ON DELETE{NO ACTION|RESTRICT|CASCADE|SET \textit{null}|SET DEFAULT
   }]
[ON UPDATE{NO ACTION|RESTRICT|CASCADE|SET \textit{null}|SET DEFAULT
   }]
```

dove lista_colonne1 è un sottoinsieme delle colonne della tabella corrente, ch dovranno corrispondere alle colonne lista_colonne2 che costituiscono una chiave della tabella nome_tabella. Non è necessario che i nomi siano gli stessi, ma i domini degli attributi corrispondenti devono essere compatibili. Nel caso di chiave esterna costituita da un solo attributo si può far semplicemente seguire la specifica dell'attributo da `REFERENCES nome_tabella(nome_colonna)` (vincolo su attributo).

Tipo di Match

Il tipo di match è significativo nel caso di chiavi esterne costituite da più di un attributo e in presenza di valori nulli.

- `MATCH SIMPLE`: il vincolo di integrità referenziale è soddisfatto se per ogni tupla della tabella referente
  - almeno una delle colonne della chiave esterna è *null*, oppure

  - nessuna di tali colonne è *null* ed esiste una tupla nella tabella riferita in cui i valori delle corrispondenti colonne concidono con i valori delle colonne della chiave esterna.

- `MATCH FULL`: il vincolo di integrità referenziale è soddisfatto se per ogni tupla della tabella referente

  - tutte le colonne della chiave esterna sono *null*, oppure
  - nessuna di tali colonne è *null* ed esiste una tupla nella tabella riferita in cui i valori delle corrispondenti colonne concidono con i valori delle colonne della chiave esterna.

- `MATCH PARTIAL`: il vincolo di integrità referenziale è soddisfatto se per ogni tupla della tabella referente i valori delle colonne non nulle della chiave esterna corrispondono ai valori delle corrispondenti colonne in una tupla della tabella riferita.

Il default è `MATCH SIMPLE`.

3.2.4 Azioni

Le clausole opzionali `ON DELETE` e `ON UPDATE` indicano al DBMS cosa fare nel caso una tupla cui si fa riferimento nella tabella referente venga cancellata o modificata.

Le opzioni riguardanti le azioni da eseguire nel caso di cancellazione di una tupla riferita:

- `NO ACTION`: la cancellazione di una tupla dalla tabella riferita è eseguita solo se non esiste alcuna tupla nella tabella referente che che vi fa riferimento. In altre parole, l'operazione di cancellazione viene respinta se la tupla da cancellare è in relazione con qualche tupla della tabella referente.

- `RESTRICT`: come per `NO ACTION`, con la differenza che questa condizione viene controllata subito, mentre `NO ACTION` viene considerata dopo che sono state esaminate tutte le altre specifiche relative all'integrità referenziale.

- `CASCADE`: la cancellazione di una tupla dalla tabella riferita implica la cancellazione di tutte le tuple della tabella referente che vi si riferiscono.

- `SET null`: la cancellazione di una tupla dalla tabella riferita implica che, in tutte le tuple della tabella referente che vi si riferiscono, la chiave esterna viene posta al valore *null* (se ammesso).

- `SET DEFAULT`: la cancellazione di una tupla dalla tabella riferita implica che, in tutte le tuple della tabella referente che vi si riferiscono, il valore della chiave viene posto uguale al valore di default specificato per le colonne che costituiscono la chiave esterna.

Le opzioni riguardanti le azioni da eseguire nel caso di modifica della chiave della tupla riferita tramite chiave esterna hanno lo stesso significato delle opzioni viste per la cancellazione, con l'eccezione di `CASCADE`, che ha l'effetto di assegnare alla chiave esterna il nuovo valore della chiave della tupla riferita.

Il default è `NO ACTION` sia per la cancellazione sia per la modifica.

L'ordine in cui vengono considerate le varie opzioni (nel caso di più riferimenti) è `RESTRICT`, `CASCADE`, `SET null`, `SET DEFAULT`, `NO ACTION`.

Esempi.

```
CREATE TABLE DOCENTE (
 Dno CHAR(7),
 DNOME VARCHAR(20),
 RESIDENZA VARCHAR(15),
 PRIMARY KEY(Dno)
);
```

```
CREATE TABLE STUDENTE (
 Sno CHAR(6),
 Sname VARCHAR(20),
 RESIDENZA VARCHAR( 15),
 BIRTHDATE DATE,
 PRIMARY KEY(Sno)
);
```

```
CREATE TABLE RELATORE (
 Dno CHAR(7) REFERENCES DOCENTE
  ON DELETE RESTRICT
  ON UPDATE CASCADE,
 Sno CHAR(6),
 PRIMARY KEY (Sno),
 FOREIGN KEY (Sno) REFERENCES STUDENTE
  ON DELETE CASCADE
  ON UPDATE CASCADE
);
```

3.2.5 Vincoli `CHECK`

Quando si definisce una tabella è possibile aggiungere alla specifica di ciascuna colonna la parola chiave CHECK seguita da una condizione, cioè un predicato o una combinazione booleana di predicati, comprendenti anche sottointerrogazioini che fanno riferimento ad altre tabelle (componente `WHERE` di una query SQL). Il vincolo viene controllato solo quando viene inserita o modificata una tupla nella. I vincoli check possono essere anche scritti come vincoli su tabella (dopo la dichiarazione di tutte le colonne).

Esempi.

```
CREATE TABLE IMPIEGATO (
 PREMIOP DECIMAL(7 ,2),
 STIPENDIO DECIMAL(7 ,2),
CHECK (STIPENDIO > PREMIOP)
);
```

```
CREATE TABLE IMPIEGATO (
 STIPENDIO DECIMAL(7 ,2),
 CHECK (STIPENDIO < (SELECT MAX(STIPENDIO) FROM IMPIEGATO WHERE
    MANSIONE ='DIRIGENTE'))
);
```

```
CREATE TABLE IMPIEGATO (
  MANSIONE CHAR(10) CHECK (MANSIONE IN  ('DIRIGENTE','INGEGNERE','
     TECNICO','SEGRETARIA'))
);
```

3.2.6 Attribuire un nome ai vincoli

Ogni vincolo ha associato nel DBMS un descrittore costituito da:

- nome (se non specificato è assegnato automaticamente dal sistema)
- specifica di differibilità
- checking time iniziale.

È possibile assegnare un nome ai vincoli facendo precedere la specifica del vincolo stesso dalla parola chiave `CONSTRAINT` e dal nome. Specificare un nome per i vincoli è utile per potervisi riferire in seguito (ad esempio per modificarli o cancellarli).

Esempi.

```
IDIMPIEGATO Char(6) CONSTRAINT Chiave PRIMARY KEY
CONSTRAINT StipOK CHECK(Stipendio > PremioP)
```

```
CREATE TABLE Esame (
 Sno Char(6) NOT NULL,
 Cno Char(6) NOT NULL,
 Cnome Varchar (20) NOT NULL,
 Voto Integer NOT NULL,
 PRIMARY KEY (Sno,Cno),
 CONSTRAINT Rel_Stud FOREIGN KEY (Sno) REFERENCES Studente  ON
    DELETE CASCADE,
 CONSTRAINT Cname-constr CHECK Cname IN ('Documentazione$\
    $Automatica','SEI$\ $I', 'SEI$\ $II'),
 CONSTRAINT Voto-constr CHECK Voto > 18 AND Voto < 30
);
```

3.2.7 Valutazione dei vincoli

Nel valutare i vincoli, il DBMS si attiene alle seguenti regole:

- Un vincolo su una tupla è violato se la condizione valutata sulla tupla restituisce valore FALSE.
- Un vincolo la cui valutazione su una data tupla restituisce valore TRUE o UNKNOWN (a causa di valori nulli) non è violato.
- Durante l'inserimento o la modifica di un insieme di tuple, la violazione di un vincolo per una delle tuple causa la non esecuzione dell'intero insieme.

3.2.8 Modificare i vincoli di una tabella

Per modificare un vincolo è necessario conoscerne il nome.

```
ALTER TABLE r DROP CONSTRAINT c {RESTRICT | CASCADE}
```

cancella il vincolo chiamato c dalla tabella r.

```
ALTER TABLE r ADD [CONSTRAINT c] vincolo_colonna
```

crea un vincolo chiamato c o anonimo nella tabella r. vincolo_colonna può essere uno qualsiasi dei vincoli visti in precedenza.

```
SET CONSTRAINTS (lista_vincoli|ALL) {IMMEDIATE|DEFERRED}
```

modifica il checking time (solo per i vincoli `DEFERRABLE`).

Esempi.

```
ALTER TABLE IMPIEGATI ADD UNIQUE(NOME);
ALTER TABLE IMPIEGATI ADD COSTRAINT STIPOK CHECK (STIPENDIO <
   PREMIO_P);
```

Capitolo 4

Modifica dei Dati (DML)

In SQL esistono tipi di query che permettono di modificare i dati nelle relazioni:

- Query di aggiornamento
- Query di inserimento
- Query di cancellazione

Questi tre tipi di query, che saranno descritte più in dettaglio nelle sezioni che seguono, integrano nella loro sintassi quella delle più generali query di selezione, che verranno descritte nel capitolo successivo. Le query di modifica vengono comunque trattate prima di quelle di selezione, poichè il loro uso è necessario per popolare la base di dati prima di effettuare interrogazioni.

4.1 Query di inserimento

Le query di inserimento permettono di inserire nuove tuple in una relazione. L'origine delle tuple può essere una serie di valori costanti forniti dall'utente o un insieme di tuple restituito da una query di selezione. E' possibile specificare quali attributi verranno inseriti nella tupla e in che ordine. Se non vengono specificati tutti gli attributi, gli altri avranno il loro valore di default (sempre che ciò non violi qualche vincolo).

```
INSERT INTO nome_tabella [(lista_colonne)] VALUES (lista_valori)
```

inserisce una nuova tupla nella tabella nome_tabella.

Nella nuova tupla, il valore di ciascuna colonna *c*, contenuta nella *lista_colonne*, viene impostato al corrispondente valore *v* inserito nella *lista_valori* che segue la parola chiave `VALUES` (i tipi devono ovviamente essere compatibili). Se non si specifica una lista di nomi di colonne, allora la lista dei valori dovrà contenere un valore per ciascuna colonna della tabella, esattamente nell'ordine con cui le colonne sono state create (tramite il comando `CREATE`)

```
INSERT INTO nome_tabella [(lista_colonne)] query
```

inserisce nella tabella nome_tabella tante tuple quante sono le righe restituite dalla *query* di selezione fornita. La *query* deve restituire una tabella formata da tante colonne quante sono quelle specificate nella *lista_colonne*, con tipi corrispondente compatibili. Se si omette la *lista_colonne*, la *query* dovrà restituire una tabella compatibile (per colonne, ordine delle stesse e tipi) con *nome_tabella*.

Esempi.

```
INSERT INTO IMPIEGATI(IMP\#,NOME,MANSIONE) VALUES (234,'ROSSI','
   INGEGNERE')
```

inserisce la tupla indicata nella tabella IMPIEGATI.

```
INSERT INTO MANSIONI SELECT DISTINCT MANSIONE FROM IMPIEGATI
```

inserisce tutte le tuple derivanti dalla query `SELECT DISTINCT MANSIONE FROM IMPIEGATI` (vedi capitolo successivo) nella tabella MANSIONI.

4.2 Query di aggiornamento

Le query di aggiornamento permettono di variare il contenuto di tuple già presenti in una particolare relazione del database. Le tuple da modificare vengono selezionate tramite una clausola `WHERE`, e a tutte le tuple selezionate vengono applicati degli aggiornamenti che possono essere costanti o far riferimento agli attributi della stessa tupla.

```
UPDATE nome_tabella SET nome_colonna = espressione, nome_colonna =
   espressione, ... WHERE condizione
```

dove

- *nome_colonna* è il nome di una colonna delle tuple della tabella specificata
- *espressione* è un'espressione costante, o che fa riferimento agli altri attributi della tabella specificata, in cui si possono utilizzare tutti gli operatori (aritmetici, stringa, ...) messi a disposizione dal DBMS.
- *condizione* è una condizione `WHERE` standard (vedi capitolo successivo).

Questa query aggiorna la tabella *nome_tabella* impostando, per ogni tupla selezionata dalla *condizione*, ciascuna *nome_colonna* specificata nella lista che segue la parola chiave `SET` al valore calcolato dalla corrispondente *espressione*.

Esempi.

```
UPDATE IMPIEGATI
SET STIPENDIO = STIPENDIO + STIPENDIO$\*$0.1, PREMIO$\_$P = 1000
WHERE MANSIONE = 'AMM.NE'
```

incrementa lo stipendio del 10% e imposta il premio a 1000 in tutti i record della tabella impiegati la cui mansione è 'AMM.NE'.

4.3 Query di cancellazione

Con una query di cancellazione si possono cancellare da una tabella le tuple che rispondono a un certo criterio (espresso con una clausola `WHERE`), o tutte le tuple.

```
DELETE FROM nome_tabella [ WHERE condizione ]
```

cancella dalla tabella *nome_tabella* le tuple che corrispondono alla *condizione* data (*condizione* è una clausola `WHERE` standard, vedi il capitolo successivo). Omettendo la *condizione*, l'intero contenuto della tabella verrà eliminato.

Esempi.

```
DELETE FROM IMPIEGATI WHERE DATA_A > DATE('1/1/2001')
```

elimina dalla tabella impiegati tutti i record in cui DATA_A è successiva al 1/1/2001.

```
DELETE FROM IMPIEGATI
```

svuota la tabella impiegati.

Capitolo 5

Interrogazioni (QL)

Le interrogazioni, o query di selezione, sono la funzionalità principale di SQL. Tramite le interrogazioni è possibile estrarre dati dal database secondo criteri anche molto complessi, generando in output

- **valori singoli**: questo tipo di interrogazioni, dette *singleton query*, restituisce una tabella formata da un solo record con una sola colonna. Il valore così resittuito può essere usato in molte espressioni SQL valide come il risultato di una speciale chiamata di funzione.
- **liste**: questo tipo di interrogazioni restituisce una tabella formata da una singola colonna. In alcune espressioni SQL questo tipo di risultato può essere utilizzato come una lista di valori (ad esempio, come secondo operando dell'istruzione `IN`).
- **tabelle**: si tratta della forma di risultato più generale e conosciuta. Le tabelle restituite da una interrogazione sono temporanee, cioè non sono inserite nella base di dati, e possono anche essere usate per creare *viste* dinamiche sui dati.

5.1 Interrogazioni di base

La sintassi di base di una interrogazione SQL è la seguente:

```
SELECT lista_attributi FROM lista_tabelle [ WHERE condizione ]
```

- la clausola `SELECT` dichiara le colonne da estrarre, il loro ordine e il loro nome.
- la clausola `FROM` dichiara le tabelle dalle quali verranno estratti i dati.
- la clausola opzionale `WHERE` definisce una condizione di filtro per i record da estrarre.

La lista_attributi ha la forma

```
nome_colonna [[AS] [alias]],nome_colonna [[AS] [alias]], ...
```

Tramite gli alias è possibile ridefinire il nome delle colonne estratte, o fornire un nome alle *colonne calcolate*, che non ne hanno uno predefinito. Ogni colonna è indicata semplicemente col suo nome nel caso questo non sia ambiguo (cioè se nell'interrogazione non sono presenti più tabelle che hanno colonne con lo stesso nome). In caso di ambiguità, è possibile usare la notazione `nome_tabella.nome_colonna` per indicare la tabella specifica contenente la colonna desiderata. Il simbolo speciale `*` usato in questa clausola indica ad SQL di estrarre tutte le colonne; è possiblle anche scrivere `tabella.*` per estrarre tutte le colonne di una particolare tabella.
La lista_tabelle ha la forma

```
nome_tabella [[AS] [alias]],nome_tabella [[AS] [alias]], ...
```

Anche in questo caso gli alias possono essere usati per ridenominare una tabella localmente alla query. Questo è indispensabile nel caso si importino più istanze della stessa tabella all'interno della query. Sia la condizione di filtro che le colonne indicate nella clausola `SELECT` devono far riferimento a tabelle importate tramite la clausola `FROM`, utilizzando eventualmente l'alias dichiarato come loro nome.

La condizione espressa dopo la parola chiave `WHERE` deve essere un'espressione con valore booleano (true/false/unknown). Per la composizione di questa espressione si possono usare tutti gli operatori e le funzioni visti in in precedenza. I record selezionati sono tutti e soli quelli per cui la condizione è true.

Il significato di una query SQL può essere espresso per mezzo dell'espressione algebrica

$$\Pi(colonna_1, colonna_2, \ldots)(\sigma_{condizione}(tabella_1 \times tabella_2 \times \ldots))$$

Dal punto di vasta del DBMS, la query viene eseguita socondo la seguente procedura:

1. Si genera il prodotto cartesiano di tutte le tabelle specificate dalla clausola `FROM`, nell'ordine dato.
2. Si applica la condizione di ricerca specificata nella clausola `WHERE` alle tuple della relazione generata al passo precedente.
3. Si restituiscono in output i valori delle colonne spacificate dalla clausola `SELECT`.

Esempi.
Siano date le seguenti definizioni di relazioni:

```
IMPIEGATO (IDIMPIEGATO, NOME, COGNOME, MANSIONE, DATA_A, UFFICIO,
   STIPENDIO, PREMIO_P, IDREPARTO)
REPARTO (IDREPARTO, NOME,INDIRIZZO,ÀCITT)
```

I dati inseriti nelle relazioni sono i seguenti:

Nome	Cognome	IDREPARTO	Ufficio	Stipendio
Mario	Rossi	Amm.ne	10	45
Carlo	Bianchi	Prod.ne	20	36
Giuseppe	Verdi	Amm.ne.	20	40
Franco	Neri	Distr.ne	16	45
Carlo	Rossi	Direz.ne	14	80
Lorenzo	Lanzi	Direz.ne	7	73
Paola	Borroni	Amm.ne	75	40
Marco	Franco	Prod.ne	20	46

Nome	Indirizzo	Città
Amm.ne	via Tito Livio, 27	Milano
Prod.ne	p.le Lavater, 3	Torino
Distr.ne	via Segre,9	Roma
Direz.ne	via tito Livio, 27	Milano
Ricerca	via Morone, 6	Milano

Query 1 *Reperire i cognomi degli impiegati del reparto 12.*

```
SELECT COGNOME
FROM IMPIEGATO
WHERE IDREPARTO = 12
```

Risultato:

Cognome
Bianchi
Franco

Query 2 *Reperire i dati degli impiegati del reparto 1*

```
SELECT *
FROM IMPIEGATO
WHERE IDREPARTO = 1
```

Risultato:

Nome	Cognome	IDReparto	Ufficio	Stipendio
Mario	Rossi	Amm.ne	10	45
Giuseppe	Verdi	Amm.ne	20	40
Paola	Borroni	Amm.ne	75	40

non sono state riportate tutte le colonne per motivi di spazio

Query 3 *Reperire lo stipendio mensile degli impiegati di cognome Bianchi*

```
SELECT STIPENDIO /12 AS STIPNDIOMENSILE
FROM IMPIEGATO
WHERE COGNOME = 'BIANCHI'
```

Risultato:

StipendioMensile
3,00

Query 4 *Reperire i cognomi degli impiegati del reparto 1 o 12.*

```
SELECT COGNOME
FROM IMPIEGATO
WHERE IDREPARTO= 1 OR IDREPARTO=12
```

Risultato:

Cognome
Rossi
Bianchi
Verdi
Borroni
Franco

Query 5 *Lista di tutti i manager*

```
SELECT *
FROM IMPIEGATI
WHERE MANSIONE = 'MANAGER'
```

Query 6 *Lista gli id di tutti gli impiegati*

```
SELECT IDIMPIEGATO
FROM IMPIEGATI
```

Query 7 *Lista di tutti i reparti con numero di codice maggiore o uguale a 30*

```
SELECT *
FROM REPARTO
WHERE IDREPARTO $\geq$30
```

Query 8 *Listare gli impiegati che hanno premio di produzione maggiore dello stipendio*

```
SELECT *
FROM IMPIEGATI
WHERE PREMIO_P >STIPENDIO
```

Query 9 *Selezionare gli impiegati che hanno uno stipendio maggiore di 2000*

```
SELECT *
FROM IMPIEGATI
WHERE STIPENDIO > 2000;
```

Risultato:

Imp#	Nome	Mansione	Data_A	Stipendio	Premio_P	Dip
7566	Rosi	dirigentem	02-Apr-81	2975,00	?	20
7698	Biacchi	dirigente	01-Mag-81	2850,00	?	30
7782	Neri	ingegnere	01-Giu-81	2450,00	200,00	10
7839	Dare	ingegnere	17-Nov-81	2600,00	300,00	10
7977	Verdi	dirigente	10-Dic-80	3000,00	?	10

non sono state riportate tutte le colonne per motivi di spazio

Query 10 *Selezionare il nome e il numero di dipartimento degli impiegati che hanno uno stipendio maggiore di 2000 e hanno mansione di ingegnere*

```
SELECT NOME, IDREPARTO
FROM IMPIEGATI
WHERE STIPENDIO > 2000 AND MANSIONE ='INGEGNERE';
```

Risultato:

Nome	IDREPARTO
Neri	10
Dare	10

Query 11 *Selezionare il numero di matricola degli impiegati che lavorano nel dipartimento 30 e sono ingegneri o tecnici*

```
SELECT IMP$\#$
FROM IMPIEGATI
WHERE IDREPARTO=30 AND(MANSIONE = 'INGEGNERE' OR MANSIONE = '
   TECNICO');
```

Risultato:

Imp#
7499
7521
7844
7900

Query 12 *Lista degli impiegati dei reparto 10 che guadagnano più di 2000*

```
SELECT NOME, STIPENDIO
FROM IMPIEGATI
WHERE STIPENDIO > 2000 AND IDREPARTO=10
```

Query 13 *Trovare il nome, lo stipendio e il premio di produzione di tutti gli ingegneri per cui la somma dello stipendio e dei premio di produzione è maggiore di 2000*

```
SELECT NOME, STIPENDIO, PREMIO_P
FROM IMPIEGATI
WHERE MANSIONE ='INGEGNERE'AND STIPENDIO+PREMIO_P > 2000;
```

Risultato:

Nome	Stipendio	Premio_P
Rossi	1600,00	500,00
Neri	2450,00	200,00
Dare	2000,00	300,00

Query 14 *Lista degli impiegati (con nome e stipendio) aventi stipendio compreso tra 1100 e 1400*

```
SELECT NOME,STIPENDIO
FROM IMPIEGATI
WHERE STIPENDIO BETWEEN 1100 AND 1400
```

Risultato:

Nome	Stipendio
Adami	1100,00
Muli	1300,00

Query 15 *Tutti i dati dei dipartimenti 10 e 30*

```
SELECT *
FROM REPARTO
WHERE IDREPARTO IN (10,30);
```

Risultato:

IDREPARTO	Nome_Dip	Ufficio	Divisione	Dirigente
10	Edilizia Civile	1100	Dl	7977
30	Edilizia Stradale	5100	D2	7698

non sono state riportate tutte le colonne per motivi di spazio

Query 16 *Lista degli impiegati che non sono nè analisti nè programmatori*

```
SELECT NOME, STIPENDIO, DIP, QUALIFICA
FROM IMPIEGATI
WHERE QUALIFICA NOT IN ('ANALISTA','PROGRAMMATORE')
```

Query 17 *Determinare tutti gli impiegati che hanno 'R' come terza lettera del cognome.*

```
SELECT NOME
FROM IMPIEGATI
WHERE NOME LIKE '__R%'
```

Risultato:

Nome
Martini
Neri
Dare
Turni
Fordi
Verdi

5.2 Join tra tabelle

L'operazione di join è molto importante in quanto permette di correlare dati rappresentati da relazioni diverse. Il join può essere espresso in SQL tramite un prodotto cartesiano a cui sono applicati uno o più *predicati di join*. Un predicato di join esprime una relazione che deve essere verificata dalle tuple risultato dell'interrogazione.

Query 18 *Determinare il nome dei dipartimento in cui lavora l'impiegato Rossi*

```
SELECT REPARTO.NOME
FROM IMPIEGATO, REPARTO
WHERE NOME = 'ROSSI' AND IMPIEGATO.IDREPARTO = REPARTO.IDREPARTO;
```

Il predicato di join è `IMPIEGATO.IDREPARTO=REPARTO.IDREPARTO`.

Query 19 *Lista di tutti gli impiegati che guadagnano più di Rossi (theta-join)*

```
SELECT ROSSI.NOME, ROSSI.STIPENDIO, ROSSI.QUALIFICA, ALTRI.NOME,
   ALTRI.STIPENDIO, ALTRI.QUALIFICA
FROM IMPIEGATO ROSSI, IMPIEGATO ALTRI
WHERE ALTRI.STIPENDIO > ROSSI.STIPENDIO AND ROSSI.NOME='ROSSI'
```

questa interrogazione mostra anche come è possibile importare più di una istanza della stessa tabella in una query usando il costrutto degli *alias*.

Query 20 *Lista degli impiegati con i rispettivi capi (self-join)*

```
SELECT SUBORDINATO.NOME, CAPO.NOME
FROM IMPIEGATO SUBORDINATO, IMPIEGATO CAPO
WHERE SUBORDINATO.CAPO=CAPO.IDIMPIEGATO
```

5.3 L'operatore JOIN

Abbiamo visto come è possibile esprimere join mediante predicati di join nella clausola `WHERE`. In realtà SQL:1999 prevede diversi tipi di *operatori di join*. Questi operatori producono relazioni e quindi possono essere usati nella clausola `FROM`.

L'operatore `JOIN` può essere usato per eseguire esplicitamente join tra tabelle (anzichè inserirne le condizioni nella clausola `WHERE`) e per realizzare tipi di join altrimenti impossibili (come i join esterni).

5.3.1 Join interni (inner join)

- `tabella1` **`CROSS JOIN`** `tabella2`: genera il cross join tra *tabella1* e *tabella2*, cioè il loro prodotto cartesiano
- `tabella1 [` **`INNER`** `] JOIN tabella2` **`ON`** `condizione`: genera il theta join tra *tabella1* e *tabella2* secondo la *condizione* specificata.
- `tabella1` **`NATURAL JOIN`** `tabella2`: genera il join naturale tra *tabella1* e *tabella2*. Le due tabelle devono avere una o più colonne con lo stesso nome.
- `tabella1 [` **`INNER`** `] JOIN tabella2` **`USING`** `(lista_colonne)`: genera il join naturale tra *tabella1* e *tabella2* usando le colonne specificate nella *lista_colonne* (che devono essere presenti in entrambe le tabelle).

Query 21 *Selezionare gli impiegati e la città in cui lavorano.*

```
SELECT IMPEGATO.NOME, IMPEGATO.COGNOME, REPARTO.CITTA
FROM IMPIEGATO INNER JOIN REPARTO ON
IMPIEGATO.IDREPARTO=REPARTO.IDREPARTO
```

5.3.2 Join esterni (outer join)

Nel normale join tra due relazioni R e S non si ha traccia delle tuple di R che non corrispondono ad alcuna tupla di S. Questo non sempre è quello che si desidera.

L'operatore di `OUTER JOIN` aggiunge al risultato le tuple di R e S che non hanno partecipato al join, completandole con colonne *null*. Esistono diverse varianti dell'outer join:

- `tabella1` **`LEFT`** `[` **`OUTER`** `] JOIN tabella2` **`ON`** `condizione`: fornisce come risultato il theta join interno tra *tabella1* e *tabella2* esteso con le righe della relazione che compare a sinistra del join (*tabella1*) per le quali non esiste una corrispondente riga nella tabella di destra.

- `tabella1` **`RIGHT`** `[` **`OUTER`** `]` **`JOIN`** `tabella2` **`ON`** `condizione`: fornisce come risultato il theta join interno tra *tabella1* e *tabella2* esteso con le righe della relazione che compare a destra del join (*tabella2*) per le quali non esiste una corrispondente riga nella tabella di sinistra.

- `tabella1` **`FULL`** `[` **`OUTER`** `]` **`JOIN`** `tabella2` **`ON`** `condizione`: fornisce come risultato il theta join interno tra *tabella1* e *tabella2* esteso con le righe delle due relazioni che non hanno un corrispondente secondo la *condizione* di join.

La variante OUTER può essere utilizzata anche per il join naturale, escludento la clausola **`ON`** `espressione`.

Esempi. Siano date le seguenti definizioni di relazioni:

```
GUODATORE (NOME, COGNOME, NPATENTE)
AUTOMOBILE (TARGA, MARCA, NPATENTE)
```

I dati inseriti nelle relazioni sono i seguenti:

Nome	Cognome	Npatente
Mario	Rossi	VR 7777
Carlo	Bianchi	PZ 8888
Marco	Neri	AP9999

Targa	Marca	NPatente
AB 574 WW	Fiat	VR 7777
AA642FF	Fiat	VR 7777
BJ 741 XX	Lancia	PZ 8888
BB 421 JJ	Fiat	Ml 4444

le query che seguono restituiscono i risultati mostrati.

```
SELECT *
FROM GUIDATORE LEFT JOIN AUTOMOBILE ON (GUIDATORE.NPATENTE =
   AUTOMOBILE.NPATENTE)
```

Risultato:

Nome	Cognome	Npatente	Targa	Marca
Mario	Rossi	VR 7777	AB 574 WW	Fiat
Mario	Rossi	VR 7777	AA642FF	Fiat
Carlo	Bianchi	PZ 8888	BJ 741 XX	Lancia
Marco	Neri	AP 9999	*null*	*null*

```
SELECT *
FROM GUIDATORE RIGHT JOIN AUTOMOBILE ON (GUIDATORE.NPATENTE =
   AUTOMOBILE.NPATENTE)
```

Risultato:

Nome	Cognome	Npatente	Targa	Marca
Mario	Rossi	VR 7777	AB 574 WW	Fiat
Mario	Rossi	VR 7777	AA642FF	Fiat
Carlo	Bianchi	PZ 8888	BJ 741 XX	Lancia
null	*null*	MI 4444	BB 421 JJ	fiat

```
SELECT *
FROM GUIDATORE FULL JOIN AUTOMOBILE ON (GUIDATORE.NPATENTE =
   AUTOMOBILE.NPATENTE)
```

Risultato:

Nome	Cognome	Npatente	Targa	Marca
Mario	Rossi	VR 7777	AB 574 WW	Fiat
Mario	Rossi	VR 7777	AA642FF	Fiat
Carlo	Bianchi	PZ 8888	BJ 741 XX	Lancia
Marco	Neri	AP 9999	*null*	*null*
null	*null*	MI 4444	BB 421 JJ	fiat

5.4 Ordinamento del risultato di una query

Negli esempi visti, l'ordine delle tuple risultato di una interrogazione è determinato dal sistema (dipende dalla strategia usata per eseguire l'interrogazione). È possibile specificare un ordinamento diverso aggiungendo alla fine dell'interrogazione la clausola `ORDER BY` con la seguente sintassi:

```
SELECT lista_attributi
FROM lista_tabelle
[ WHERE condizione ]
[ ORDER BY colonna [ASC | DESC], colonna [ASC | DESC], ... ]}
```

I dati saranno ordinati in base al contenuto delle colonne citate. Se si specifica più di una colonna, le successive saranno usate, nell'ordine, per ordinare le righe che risultino uguali secondo le colonne precedenti (ordinamento secondario). Il metodo di ordinamento dipende dal tipo di dato della colonna, e di default l'ordine è discendente. E' possibile specificare le parole chiave `ASC` (ascendente) o `DESC` (discendente) per forzare un particolare ordinamento su ciascuna colonna indicata.

Dal punto di vasta del DBMS, la query viene eseguita socondo la seguente procedura:

1. Si genera il prodotto cartesiano di tutte le tabelle specificate dalla clausola `FROM`, nell'ordine dato.
2. Si applica la condizione di ricerca specificata nella clausola `WHERE` alle tuple della relazione generata al passo precedente.
3. Si ordinano le tuple della tabella filtrata ottenuta al passo precedente usando, nell'ordine, i criteri di ordinamento specificati con la clausola `ORDER BY`.
4. Si restituiscono in output i valori delle colonne spacificate dalla clausola `SELECT`.

Query 22 *Elencare lo stipendio, la mansione e il nome di tutti gli impiegati del dipartimento 30, ordinando le tuple in ordine crescente in base allo stipendio*

```
SELECT STIPENDIO, MANSIONE,NOME
FROM IMPIEGATI
WHERE IDREPARTO = 30
ORDER BY STIPENDIO;
```

Risultato:

Stipendio	Mansione	Nome
800,00	tecnico	Andrei
800,00	tecnico	Bianchi
800,00	segretaria	Martini
1500,00	tecnico	Turni
1950,00	ingegnere	Gianni
2850,00	dirigente	Biacchi

Query 23 *Si vogliono elencare mansione, nome e stipendio di tutti gli impiegati ordinando le tuple in base alla mansione in ordine crescente, ed in base allo stipendio in ordine decrescente*

```
SELECT MANSIONE, STIPENDIO, NOME
FROM IMPIEGATI
ORDER BY MANSIONE, STIPENDIO DESC;
```

Risultato:

Mansione	Stipendio	Nome
dirigente	3000,00	Verdi
dirigente	2975,00	Rosi
dirigente	2850,00	Biacchi
ingegnere	2450,00	Neri
ingegnere	2000,00	Dare
ingegnere	1950,00	Gianni
ingegnere	1600,00	Rossi
ingegnere	1300,00	Muli
ingegnere	1100,00	Adami
segretari	1000,00	Fordi
segretari	800,00	Martini
segretari	800,00	Scotti
tecnico	1500,00	Turni
tecnico	800,00	Andrei
tecnico	800,00	Bianchi

5.5 Eliminazione dei duplicati

Supponiamo di voler recuperare la lista di tutte le mansioni presenti nella relazione IMPIEGATI:

```
SELECT MANSIONE
FROM IMPIEGATI;
```

Risultato:

Mansione
ingegnere
tecnico
tecnico
dirigente
segretaria
dirigente
ingegnere
segretaria
ingegnere
tecnico
ingegnere
ingegnere
segretaria
ingegnere
dirigente

In questo caso, è possibile richiedere l'eliminazione dei duplicati tramite la parola chiave `DISTINCT`:

```
SELECT DISTINCT MANSIONE
FROM IMPIEGATI;
```

Risultato:

Mansione
ingegnere
tecnico
dirigente
segretaria

La parola chiave `DISTINCT` forza l'interrogazione arestituire record distinti, cioè privi di duplicati. Nel caso siano restituite righe con più colonne, l'intera riga viene utilizzata per determinare l'unicità.

Query 24 *Determinare tutti gli indirizzi dei reparti ubicati a Milano*

```
SELECT INDIRIZZO
FROM REPARTO
WHERE CITTA'='MILANO'
```

Risultato:

Indirizzo
via Tito Livio, 27
via Tito Livio, 27
via Morone, 6

```
SELECT DISTINCT INDIRIZZO
FROM REPARTO
WHERE CITTA'='MILANO'
```

Risultato:

Indirizzo
via Tito Livio, 27
via Morone, 6

Query 25 *Trovare i cognomi più lunghi di tre caratteri nella tabella PERSONE*

```
SELECT DISTINCT COGNOME
FROM PERSONE
WHERE LENGTH(COGNOME) > 3
```

la funzione LENGTH non è disponibile in tutti i DBMS.

5.6 Colonne calcolate

All'interno della clausola SELECT è possibile specificare anche espressioni complesse che generano colonne colacolate (cioè non mostrano colonne estratte da altre tabelle, ma colonne con valori calcolati a partire dai dati presenti nelle tabelle stesse)

Query 26 *Trovare il nome, lo stipendio, il premio di produzione, e la somma dello stipendio e dei premio di produzione di tutti gli ingegneri.*

```
SELECT NOME, STIPENDIO, PREMIO_P, STIPENDIO+PREMIO_P
FROM IMPIEGATI
WHERE MANSIONE ='INGEGNERE';
```

Risultato:

Nome	Stipendio	Premio_P	Stipendio+Premio_P
Rossi	1600,00	500,00	2100,00
Neri	2450,00	200,00	2650,00
Dare	2000,00	300,00	2300,00
Adami	1100,00	500,00	1600,00
Gianni	1950,00	?	1950,00
Muli	1300,00	150,00	1450,00

Query 27 *Generare una tabella che contenga lo stipendio corrente di ciascun impiegato, indicando la data in cui la tabella è stata generata.*

```
SELECT CURRENT_DATE, NOME, STIPENDIO
FROM IMPIEGATI;
```

la prima colonna contiene una costante, che verrà ripetuta su ogni riga. E' un particolare tipo di colonna calcolata, utile in casi particolari, ad esempio quando si creano tabelle tramite unione.

Query 28 *Restituire una lista di stringhe di testo nel formato cognome nome indirizzo per ogni persona nella tabella PERSONE.*

```
SELECT COGNOME || ' ' || NOME || ' ' || INDIRIZZO
FROM PERSONE
```

Query 29 *Si vuole avere un colloquio con tutti i nuovi impiegati del dipartimento 10 dopo 90 giorni dalla loro assunzione. In particolare, per tutti gli impiegati interessati, si vogliono determinare il nome, la data di assunzione, la data del colloquio e, inoltre, la data corrente di esecuzione dell'interrogazione;*

```
SELECT NOME, CHAR(DATA_A,EUR), CHAR(DATA_A + 90 DAYS,
EUR), CHAR(CURRENT_DATE, EUR)
FROM IMPIEGATI
WHERE DATA_A + 90 DAYS = CURRENT_DATE AND IDREPARTO=10;
```

Risultato:

Nome	Data_A	Data_A + 90	CURRENT_DATE
Rossi	23/01/82	04/03/82	25/03/82

si ricorda che le funzioni di manipolazione delle date possono variare tra DBMS diversi.

5.7 Operatori Aggregati

In algebra relazionale, le condizioni sono predicati valutati su singole tuple indipendentemente dalle altre. Spesso è invece necessario valutare proprietà che dipendono da insiemi di tuple. Ad esempio, il numero di impiegati del re parto Produzione non è una proprietà di una singola tupla.

Gli operatori aggregati di SQL permettono di calcolare valori aggregando più tuple secondo particolari criteri:

- `COUNT (*|[DISTINCT | ALL] lista_attributi)`: conta gli attributi non-nulli di tutte tuple selezionate.
- **SUM** `([DISTINCT | ALL ] espressione)`: somma i valori dell'espressione calcolata su tutte tuple selezionate.

- **MAX** ([**DISTINCT** | **ALL**] espressione): calcola il massimo tra i valori dell'espressione calcolata su tutte tuple selezionate.
- **MIN** ([**DISTINCT** | **ALL**] espressione): calcola il minimo tra i valori dell'espressione calcolata su tutte tuple selezionate.
- **AVG** ([**DISTINCT** | **ALL**] espressione): calcola la media tra i valori dell'espressione calcolata su tutte tuple selezionate.

I modificatori ALL e DISTINCT possono essere specificati per richiedere che l'operatore aggregato sia applicato rispettivamente a tutti i valori o a tutti i valori distinti dell'*espressione* richiesta.

Per la funzione COUNT, l'argomento speciale $\star$ permette di calcolare il numero totale di tuple selezionate. In SQL2, se COUNT ha come argomento il nome di una singola colonna, il modificatore DISTINCT è obbligatorio.

Le funzioni di gruppo possono anche apparire in espressioni aritmetiche complesse.

Query 30 *Calcolare il numero di valori distinti dell'attributo Stipendio fra tutte le righe di Impiegato.*

```
SELECT COUNT (DISTINCT STIPENDIO)
FROM IMPIEGATO
```

Query 31 *Calcolare il numero di righe che possiedono un valore non nullo per entrambi gli attributi Nome e Cognome.*

```
SELECT COUNT (ALL NOME, COGNOME)
FROM IMPIEGATO
```

Query 32 *Determinare il numero di impiegati del reparto 15*

```
SELECT COUNT(*)
FROM IMPIEGATO
WHERE REPARTO=15
```

Query 33 *Calcolare la somma degli stipendi del reparto Amministrazione*

```
SELECT SUM(STIPENDIO)
FROM IMPIEGATO, REPARTO
WHERE REPARTO.NOME='AMM.NE' AND IMPIEGATO.IDREPARTO=REPARTO.
   IDREPARTO
```

Query 34 *Calcolare la somma degli stipendi e dei premi produzione del reparto Amministrazione*

```
SELECT SUM(STIPENDIO) + SUM(PREMIO$\_$P)
FROM IMPIEGATO, REPARTO
WHERE REPARTO.NOME='AMM.NE' AND IMPIEGATO.IDREPARTO=REPARTO.
   IDREPARTO
```

Query 35 *Determinare il massimo stipendio tra quelli degli Impiegati che lavorano in un reparto di Milano*

```
SELECT MAX(STIPENDIO)
FROM IMPIEGATO, REPARTO
WHERE IMPIEGATO.IDREPARTO=REPARTO.IDREPARTO AND REPARTO.ÀCITT='
   MILANO'
```

Query 36 *Calcolare lo stipendio massimo, minimo e medio tra gli impiegati*

```
SELECT MAX(STIPENDIO), AVG(STIPENDIO), MIN(STIPENDIO)
FROM IMPIEGATO
```

Query 37 *Determinare gli impiegati con il massimo stipendio*

```
SELECT NOME
FROM IMPIEGATO
WHERE STIPENDIO = (
  SELECT MAX(STIPENDIO)
  FROM IMPIEGATO
);
```

questo esempio usa una query nidificata. Questo tipo di sintassi verrà discussa più avanti.

5.8 Raggruppamento di tuple

Finora abbiamo usato gli operatori aggregati per calcolare funzioni su tutte le tuple selezionate da una query. Spesso è invece necessario calcolare funzioni aggregate su sotto-gruppi di tuple. Ad esempio, fornire la somma degli stipendi degli impiegati in ciascun dipartimento. La seguente query **non è corretta**:

```
SELECT IDREPARTO, SUM(STIPENDIO)
FROM IMPIEGATO;
```

Infatti, se è presente una funzione di gruppo nella clausola `SELECT`, SQL non accetta che vengano specificate altre colonne che non siano risultati di funzioni di gruppo. Per ottenere il risultato desiderato, dobbiamo prima forzare un raggruppamento di tuple tramite la clausola `GROUP BY`.

```
SELECT lista_attributi
FROM lista_tabelle
[ WHERE condizione ]
[ GROUP BY colonna, colonna, ... ]
[ ORDER BY colonna [ASC | DESC], colonna [ASC | DESC], ... ]}
```

Se è presente una clausola `GROUP BY` nell'interrogazione, le funzioni aggregate saranno calcolate su ogni singolo sotto-gruppo, secondo la seguente procedura:

1. Si genera il prodotto cartesiano di tutte le tabelle specificate dalla clausola `FROM`, nell'ordine dato.
2. Si applica la condizione di ricerca specificata nella clausola `WHERE` alle tuple della relazione generata al passo precedente.
3. Si partizionano le tuple della tabella filtrata ottenuta al passo precedente in base al valore di una o più colonne, come specificato dalla clausola `GROUP BY`.
4. Si ordinano le tuple della tabella filtrata ottenuta al passo precedente usando, nell'ordine, i criteri di ordinamento specificati con la clausola `ORDER BY`.
5. Si restituiscono in output tante tuple quanti sono i gruppi ottenuti dal partizionamento, con i valori associati calcolati dalle espressioni inserite nella clausola `SELECT`.

Le colonne usate per il raggruppamento sono le sole che possono apparire anche nella clausola `SELECT` e `ORDER BY`, in modo da caratterizzare i sotto-gruppi su cui sono state calcolate le funzioni aggregate.

Query 38 *Calcolare la somma degli stipendi degli impiegati in ciascun dipartimento.*

```
SELECT IDREPARTO, SUM(STIPENDIO)
FROM IMPIEGATO
GROUP BY IDREPARTO;
```

Risultato:

Dip	SUM(Stipendio)
10	125
20	82
30	45
40	153

Query 39 *Determinare il massimo stipendio in ogni dipartimento*

```
SELECT IDREPARTO, MAX(STIPENDIO)
FROM IMPIEGATI
GROUP BY IDREPARTO
```

Risultato:

IDREPARTO	MAX(Stipendio)
10	3000,00
20	2975,00
30	2850,00

In una query contenente una clausola GROUP BY, ogni tupla della relazione risultato rappresenta un gruppo di tuple della relazione su cui la query è eseguita. Nell'esempio visto i gruppi sono tre: uno per ogni valore di IDREPARTO. Ad ognuno di questi gruppi è applicata la funzione MAX sulla colonna Stipendio.

Query 40 *Determinare la somma degli stipendi dei vari reparti (di cui vogliamo conoscere il nome)*

```
SELECT REPARTO.NOME, SUM(IMPIEGATO.STIPENDIO)
FROM IMPIEGATO, REPARTO
WHERE REPARTO.IDREPARTO = IMPIEGATO.IDREPARTO
GROUP BY REPARTO.IDREPARTO
```

È una interrogazione ERRATA! Tutte le colonne non risultanti di una operazione di aggregazione presenti nella clausola `SELECT` devono essere anche presenti in quella `GROUP BY`! Le due seguenti interrogazioni sono invece corrette:

```
SELECT REPARTO.NOME, SUM(IMPIEGATO.STIPENDIO)
FROM IMPIEGATO, REPARTO
WHERE REPARTO.IDREPARTO = IMPIEGATO.IDREPARTO
GROUP BY REPARTO.NOME

SELECT REPARTO.NOME, SUM(IMPIEGATO.STIPENDIO)
FROM IMPIEGATO, REPARTO
WHERE REPARTO.IDREPARTO = IMPIEGATO.IDREPARTO
GROUP REPARTO.IDREPARTO, REPARTO.NOMIE
```

Query 41 *Supponiamo di voler raggruppare gli impiegati sulla base del dipartimento e della mansione; per ogni gruppo si vogliono determinare il nome del dipartimento, la somma degli stipendi, quanti impiegati appartengono al gruppo e la media degli stipendi*

```
SELECT REPARTO.NOME, MANSIONE, SUM(STIPENDIO), COUNT(*), AVG(
   STIPENDIO)
FROM IMPIEGATO, REPARTO
WHERE REPARTO.IDREPARTO= IMPIEGATO.IDREPARTO
GROUP BY REPARTO.NOME, MANSIONE;
```

Risultato:

Reparto.Nome	Mansione	SUM(Stipendio)	COUNT(*)	AVG(Stipendio)
Edilizia Civile	dirigente	3000,00	1	3000,00
Edilizia Civile	ingegnere	5750,00	3	1916,66
Edilizia Stradale	dirigente	2850,00	1	100,00
Edilizia Stradale	ingegnere	1950,00	1	1950,00
Edilizia Stradale	segretaria	800,00	1	800,00
Edilizia Stradale	tecnico	3100,00	3	1033,33
Ricerche	dirigente	2975,00	1	2975,00
Ricerche	ingegnere	2700,00	2	1350,00
Ricerche	segretaria	1800,00	2	900,00

5.8.1 La Clausola HAVING

SQL permette anche di specificare condizioni sui gruppi di tuple generati dalla clausola `GROUP BY`, in modo da selezionare solo i gruppi che rispettano certi criteri (espressi tramite funzioni aggregate). Questo si ottiene usando la clausola `HAVING` dopo la clausola `GROUP BY`:

```
SELECT lista_attributi
FROM lista_tabelle
[ WHERE condizione ]
[ GROUP BY colonna, colonna, ... ]
[ HAVING espressione ]
[ ORDER BY colonna [ASC | DESC], colonna [ASC | DESC], ... ]}
```

dove l'*espressione* è specificata come nella clausola `WHERE`, ma può fare riferimento solo a funzioni aggregate.

Il modello di esecuzione di una query di questo tipo è il seguente:

1. Si genera il prodotto cartesiano di tutte le tabelle specificate dalla clausola `FROM`, nell'ordine dato.
2. Si applica la condizione di ricerca specificata nella clausola `WHERE` alle tuple della relazione generata al passo precedente.
3. Si partizionano le tuple della tabella filtrata ottenuta al passo precedente in base al valore di una o più colonne, come specificato dalla clausola `GROUP BY`.
4. Se selezionano i soli sotto-gruppi che rendono vera la condizione espressa con la clausola `HAVING`.

5. Si ordinano le tuple della tabella filtrata ottenuta al passo precedente usando, nell'ordine, i criteri di ordinamento specificati con la clausola `ORDER BY`.

6. Si restituiscono in output tante tuple quanti sono i gruppi ottenuti dal partizionamento, con i valori associati calcolati dalle espressioni inserite nella clausola `SELECT`.

Query 42 *Supponiamo di voler raggruppare gli impiegati sulla base del dipartimento e della mansione; per ogni gruppo si vogliono determinare il nome del dipartimento, la somma degli stipendi, quanti impiegati appartengono al gruppo e la media degli stipendi. Inoltre, siamo interessati solo ai gruppi che contengono almeno due impiegati.*

```
SELECT REPARTO.NOME, MANSIONE, SUM(STIPENDIO), COUNT(*), AVG(
   STIPENDIO)
FROM IMPIEGATO, REPARTO
WHERE REPARTO.IDREPARTO = IMPIEGATO.IDREPARTO
GROUP BY REPARTO.NOME, MANSIONE;
HAVING COUNT(*) >= 2;
```

Risultato:

Reparto.Nome	Mansione	SUM(Stipendio)	COUNT(*)	AVG(Stipendio)
Edilizia Civile	ingegnere	5750,00	3	1916,66
Edilizia Stradale	tecnico	3100,00	3	1033,33
Ricerche	ingegnere	2700,00	2	1350,00
Ricerche	segretaria	1800,00	2	900,00

Query 43 *Determinare i reparti che spendono più di 100 in stipendi*

```
SELECT IDREPARTO, SUM(STIPENDIO) AS SOMMASTIPENDI
FROM IMPIEGATO
GROUP BY IDREPARTO
HAVING SUM(STIPENDIO) > 100
```

Risultato:

Dip	SommaStipendi
Amm.ne	125
Direz.ne	153

Query 44 *Determinare i reparti per cui la media degli stipendi degli impiegati dell'ufficio 20 è superiore a 25*

```
SELECT IDREPARTO
FROM IMPIEGATO
WHERE UFFICIO=20
GROUP BY IDREPARTO
HAVING AVG(STIPENDIO) > 25
```

5.8.2 Valori *null* e funzioni aggregate

Le tuple con valori *null* in un attributo di raggruppamento vengono poste nello stesso gruppo.

Le funzioni aggregate scartano (non usano nei calcoli) i valori *null*. Con l'unica eccezione è costituita dalla funzione `COUNT(*)`. Su una tabella vuota tutte le funzioni di gruppo hanno valore *null* tranne `COUNT`, che vale zero.

Esempi. Data la seguente relazione R:

A	B	C
a	*null*	c1
a1	b	c2
a2	*null*	*null*

le query che seguono restituiscono i risultati mostrati.

```
SELECT *
FROM R
WHERE A=a OR B=b;
```

Risultato:

A	B	C
a	*null*	c1
a1	b	c2

```
SELECT *
FROM R
WHERE A=a AND B=b;
```

Risultato:

nessuna tupla verifica la query

```
SELECT *
FROM R
WHERE C<>c1;
```

Risultato:

A	B	C
a1	b	c2

```
SELECT *
FROM R
WHERE B IS NULL;
```

Risultato:

A	B	C
a	*null*	c1
a2	*null*	*null*

```
SELECT *
FROM R
WHERE B IS NULL AND C IS NULL;
```

Risultato:

A	B	C
a2	*null*	*null*

```
SELECT *
FROM R
WHERE B IS NULL OR C IS NULL;
```

Risultato:

A	B	C
a	*null*	c1
a2	*null*	*null*

```
SELECT *
FROM R
WHERE B IS NOT NULL;
```

Risultato:

A	B	C
a1	b	c2

5.9 Subquery

Una delle ragioni che rendono SQL un linguaggio potente, è la possibilità di esprimere query complesse in termini di query più semplici, tramite il meccanismo delle subquery (sottointerrogazioni).

La clausola `WHERE` di una query (detta query esterna) può infatti contenere altre query (dette subquery). Le subquery vengono usate per determinare uno o più valori da utilizzare come valori di confronto in un predicato della query esterna. È anche possibile per una subquery avere al suo interno altre subquery.

Le subquery possono essere usate anche all'interno dei comandi di manipolazione dei dati (`INSERT`, `DELETE`, `UPDATE`).

Query 45 *Si vogliono elencare tutti gli impiegati che hanno la stessa mansione dell'impiegato di nome Gianni*

```
SELECT NOME,MANSIONE
FROM IMPIEGATI
WHERE MANSIONE = (
  SELECT MANSIONE
  FROM IMPIEGATI
  WHERE NOME = 'GIANNI');
```

la subquery restituisce come valore ”ingegnere” ; la query esterna determina quindi tutti gli impiegati che sono ingegneri.

Query 46 *Trovare i reparti con gli stipendi più elevati*

```
SELECT IDREPARTO
FROM IMPIEGATO
WHERE STIPENDIO = (
  SELECT MAX(STIPENDIO)
  FROM IMPIEGATO
);
```

in questo caso è meglio usare un operatore aggregato: la query risulta più leggibile e in alcuni sistemi più efficiente.

Query 47 *Si vogliono elencare tutti gli impiegati che hanno uno stipendio superiore alla media*

```
SELECT NOME, STIPENDIO
FROM IMPIEGATI
WHERE STIPENDIO > (
  SELECT AVG(STIPENDIO)
  FROM IMPIEGATI);
```

Risultato:

Nome	Stipendio
Rosi	2975,00
Biacchi	2850,00
Neri	2450,00
Dare	2000,00
Gianni	1950,00
Verdi	3000,00

Alcune interrogazioni con sottoquery hanno un loro corrispondente che può essere espresso in algebra relazionale, quindi, con una query singola. Tuttavia, la versione con sottoquery è generalmente più leggibile:

Query 48 *Tutti gli impiegati che sono stati assunti nella stessa data dell'impiegato con id uguale a 1000*

```
SELECT i.*
FROM IMPIEGATO i, IMPIEGATO j
WHERE i.DATA_A=j.DATA_A AND j.IDIMPIEGATO=1000;
```

versione con singola query

```
SELECT *
FROM IMPIEGATO
WHERE DATA_A = (
 SELECT DATA_A
 FROM IMPIEGATO
 WHERE IDIMPIEGATO=1000);
```

versione con sottoquery

5.9.1 Quantificatori per subquery

Negli esempi visti finora, le subquery dovevano necessariamente essere singleton query, cioè restituire singoli valori. In caso contrario, SQL avrebbe generato un errore.

E' possibile però gestire anche casi in cui le subquery restituisca liste (cioè tabelle di una sola colonna). In questi casi è necessario specificare come i valori restituiti devono essere usati nella clausola `WHERE`. A tale scopo vengono usati i quantificatori `ANY` ed `ALL`, che sono inseriti tra l'operatore di confronto e la subquery.

- `ALL`: rende l'espressione in cui è contenuto vera solo se l'espressione è vera per ogni valore della lista (AND)
- `ANY`: rende l'espressione in cui è contenuto vera solo se l'espressione è vera per almeno un valore della lista (OR)

Query 49 *Trovare i reparti con gli stipendi più elevati (nuova sintassi)*

```
SELECT IDREPARTO
FROM IMPIEGATO
WHERE STIPENDIO >= ALL (
  SELECT STIPENDIO
  FROM IMPIEGATO);
```

Query 50 *Determinare lo stipendio, la mansione, il nome e il numero di dipartimento degli impiegati che guadagnano più di almeno un impiegato del dipartimento 30*

```
SELECT STIPENDIO, MANSIONE, NOME, IDREPARTO
FROM IMPIEGATI
WHERE STIPENDIO > ANY (
  SELECT STIPENDIO
  FROM IMPIEGATI
  WHERE IDREPARTO=30);
```

Risultato:

Stipendio	Mansione	Nome	Dip
3000,00	dirigente	Verdi	10
2975,00	dirigente	Rosi	20
2850,00	dirigente	Biacchi	30
2450,00	ingegnere	Neri	10
2000,00	ingegnere	Dare	10
1950,00	ingegnere	Gianni	30
1600,00	ingegnere	Rossi	20
1500,00	tecnico	Turni	30
1300,00	ingegnere	Milli	10
1100,00	ingegnere	Adarni	10

Query 51 *Determinare lo stipendio, la mansione, il nome e il numero di dipartimento degli impiegati che guadagnano più di tutti gli impiegati dei dipartimento 30*

```
SELECT STIPENDIO, MANSIONE, NOME, IDREPARTO
FROM IMPIEGATI
WHERE STIPENDIO > ALL (
  SELECT STIPENDIO
  FROM IMPIEGATI
  WHERE IDREPARTO=30);
```

Risultato:

Stipendio	Mansione	Nome	Dip
3000,00	dirigente	Verdi	10
2975,00	dirigente	Rosi	20

E' infine possibile selezionare più di una colonna usando una subquery: in tal caso è necessario porre tra parentesi la lista delle colonne a sinistra dell'operatore di confronto.

Query 52 *Si vogliono elencare gli impiegati con la stessa mansione e stipendio di Martini*

```
SELECT NOME
FROM IMPIEGATO
WHERE (MANSIONE,STIPENDIO) = (
  SELECT MANSIONE, STIPENDIO
  FROM IMPIEGATO
  WHERE NOME = 'MARTINI');
```

Anche l'operatore `IN` può essere usato con una subquery di tipo lista:

Query 53 *Tutti i dati dei dipartimenti per cui lavorano degli analisti*

```
SELECT *
FROM REPARTO
WHERE IDREPARTO IN (SELECT IDREPARTO FROM IMPIEGATI WHERE MANSIONE
    = 'ANALISTA');
```

Query 54 *Tutti i dati dei dipartimenti il cui nome contiene la parola 'Informatica'*

```
SELECT *
FROM IMPIEGATO
WHERE IDREPARTO IN (
 SELECT ID
 FROM REPARTI
 WHERE NOME LIKE '%Informatica%');
```

Lo speciale predicato `EXISTS`, infine, può essere usato con una subquery per verificare se questa restituisce un risultato non vuoto.

5.9.2 Visibilità degli identificatori

In una query nidificata è possibile fare riferimento alle tabelle (o agli alias delle stesse) definite nella clausola `FROM` della query esterna. In questo modo, si possono usare i valori della tupla attualmente in fase di valutazione da parte della `WHERE` esterna anche nella `WHERE` della query nidificata.

Se una query nidificata importa una tabella con lo stesso nome (o alias) della sua query esterna, la tabella più interna nasconde quella esterna.

Query 55 *Elencare tutti gli impiegati che hanno uno stipendio minore di altri impiegati assunti nella stessa data*

```
SELECT i.*
FROM IMPIEGATO i, IMPIEGATO j
WHERE i.DATA_A=j.DATA_A AND j.STIPENDIO>i.STIPENDIO;
```

versione con singola query

```
SELECT *
FROM IMPIEGATO i
WHERE i.STIPENDIO < ANY(
 SELECT STIPENDIO
 FROM IMPIEGATO j
 WHERE i.DATA_A=j.DATA_A);
```

versione con sottoquery e quantificatore

```
SELECT *
FROM IMPIEGATO i
WHERE EXISTS(
 SELECT *
 FROM IMPIEGATO j
 WHERE i.DATA_A=j.DATA_A AND j.STIPENDIO>i.STIPENDIO);
```

versione con sottoquery e predicato `EXISTS`

Capitolo 6

Procedure e Trigger

Procedure e trigger permettono di scrivere codice per implementare funzionalità e controlli sui dati direttamente nella base di dati. In questa maniera, è possibile fornire agli utenti della base di dati stessa un ulteriore strato di astrazione verso la struttura logica dei dati e assicurarsi che alcuni controlli essenziali siano sempre eseguiti e non delegati al programma client che si interfaccia con il DBMS.

6.1 Le Procedure

Una procedura è un frammento di codice composto da istruzioni SQL dichiarative e procedurali memorizzate nella base di dati. Questo codice può essere attivato richiamandolo da un programma, da un'altra procedura o da un trigger, e può avere parametri di input (argomenti) e output (valori di ritorno).

Creare procedure per manipolare i dati e associarle ai dati nel DBMS è di solito utile per fornire uno strato di astrazione più completo sulla rappresentazione delle informazioni nel database, in modo che le applicazioni client non debbano conoscerne troppo a fondo la struttura.

Ad esempio, grazie all'uso delle procedure è possibile

- evitare che un programmatore implementi scorrettamente le operazioni di aggiornamento dei dati (soprattutto nel caso coinvolgano più tabelle);
- migliorare le prestazioni del database, perchè l'interazione tra il programma client e la base di dati diminuisce sensibilmente se delle operazioni complesse vengono gestite completamente dal DBMS;

Una procedura può essere composta da due tipi di istruzioni:

- dichiarative: le istruzioni SQL viste finora (`CREATE`, `UPDATE`, `SELECT`), con speciali estensioni per gestire la memorizzazione dei valori estratti dalla base di dati in variabili;
- procedurali: costrutti comuni dei linguaggi di programmazione, come `IF-THEN-ELSE` e `WHILE-DO`.

La sintassi delle procedure non è oggetto di standardizzazione, per cui qui è possibile darne solo una versione generica, tipicamente accettata con piccole differenze da molti DBMS. Quella descritta di seguito è la sintassi Interbase/Firebird.

Per creare una procedura, si utilizza il seguente comando SQL:

```
CREATE PROCEDURE nome_procedura
 [( nome_parametro tipo_parametro, ... )]
 [RETURNS ( nome_risultato tipo_risultato, ... )] AS
  corpo_procedura
```

dove,

- nome_procedura è il nome univoco della procedura;
- l'elenco dei parametri della procedura, opzionale, è composto da coppie nome_parametro tipo_parametro, dove nome_parametro è un identificatore e tipo_parametro è uno dei tipi definiti da SQL.
- se la procedura deve ritornare dei valori (quindi, in realtà, è una funzione), la parola chiave `RETURNS` permette di specificare una lista di valori di ritorno composta da coppie nome_risultato tipo_risultato, dove nome_risultato è un identificatore e tipo_risultato è uno dei tipi definiti da SQL. Notare che le procedure SQL, oltre a singoli valori, possono quindi restituire vere e proprie *tuple*.

Il corpo di una procedura può opzionalmente iniziare con una o più *dichiarazioni di variabili locali*, che potranno essere poi usate all'interno del codice della procedura stessa. La sintassi della dichiarazione è la seguente:

```
DECLARE VARIABLE nome_variabile tipo_variabile;
```

dove nome_variabile è un identificatore e tipo_variabile è uno dei tipi definiti da SQL. *Notare che tutte le variabili corrispondenti agli argomenti e ai valori di ritorno sono implicatamente dichiarate.*

Le istruzioni vere e proprie della procedura sono poste, dopo le eventuali dichiarazioni di variabili, all'interno di un *blocco* delimitato dalle keyword `BEGIN` e `END`. Le singole istruzioni sono separate tra loro da un punto e virgola.

Di seguito vengono elencati i costrutti più comuni ammessi dalla sintassi Interbase/Firebird all'interno di una procedura.

`SELECT INTO` Si tratta di una variante della `SELECT` standard, che permette di memorizzare i risultati all'interno di variabili locali, per poi poterli riutilizzare nella procedura. Usando questa sintassi, è necessario accertarsi che la query restituisca *una sola riga di risultati*, altrimenti verrà generato un errore.

```
SELECT lista_attributi
FROM ...
[ WHERE ... ]
```

```
[ GROUP BY ... ]
[ HAVING ... ]
[ ORDER BY ... ]
INTO :variabile, :variabile, ...
```

Le variabili specificate dalla clausola `INTO` devono essere state precedentemente dichiarate, e devono essere tante quanti sono i valori estratti dalla clausola `SELECT`. E' inoltre possibile usare variabili all'interno della clausola `WHERE`, rendendone il predicato parametrico. *Notare che i nomi di variabile, quando sono usati all'interno di istruzioni SQL, devono essere preceduti dai due punti*.

`FOR SELECT INTO ... DO` Questa sintassi permette di iterare un blocco di codice su ogni riga restituita dalla query indicata.

```
FOR SELECT lista_attributi
FROM ...
INTO :variabile, :variabile, ...
DO
 BEGIN
  ...
 END
```

Ad ogni iterazione del loop, i valori delle colonne della riga in esame vengono assegnate alle variabili indicate dalla clausola `INTO`, quindi viene eseguito il codice associato all'istruzione.

`IF...THEN...ELSE` Si tratta del comune costrutto condizionale presente in tutti i linguaggi di programmazione.

```
IF (condizione) THEN
 BEGIN
  ...
 END
[ELSE
 BEGIN
  ...
 END]
```

`WHILE...DO` Si tratta del comune costrutto condizionale presente in tutti i linguaggi di programmazione.

```
WHILE (condizione) DO
 BEGIN
  ...
 END
```

Esempi.

```
SET TERM !! ;
CREATE PROCEDURE incrementa_stipendi AS
DECLARE VARIABLE ID INTEGER;
DECLARE VARIABLE STIPENDIO INTEGER;
DECLARE VARIABLE MEDIA INTEGER;
BEGIN
 SELECT AVG(STIPENDIO)
 FROM IMPIEGATO
 INTO :MEDIA;
 FOR SELECT IDIMPIEGATO,STIPENDIO
  FROM IMPIEGATO
  INTO :ID,:STIPENDIO
 DO
 BEGIN
  IF (STIPENDIO < MEDIA) THEN UPDATE IMPIEGATO SET STIPENDIO = :
     MEDIA WHERE IDIMPIEGATO = :ID;
 END
END!!
SET TERM ; !!
```

Questa procedura colcola la media degli stipendi degli impiegati e poi la assegna a tutti gli impiegati che hanno uno stipendio minore della media stessa.

Nota. Negli esempi riguardanti le precedure e i trigger si vedrà sempre usare lo speciale comando `SET TERM`. Questo comando è necessario quando si inserisce del codice nel DBMS. Infatti, poichè le istruzioni di cui è composto il codice sono separate dal punto e virgola, che è anche il terminatore dei comandi SQL, l'interprete confonderebbe il primo separatore di istruzioni nel corpo della procedura con un terminatore e, tentando di elaborare immediatamente la dichiarazione di procedura, genererebbe un errore.

Per ovviare a questo problema, si usa `SET TERM` per variare (momentaneamente) il terminatore dei comandi SQL, in modo che non coincida più con il punto e virgola. La sintassi è la seguente:

```
SET TERM nuovo_terminatore vecchio_terminatore
```

Negli esempi proposti il terminatore viene cambiato nella stringa !! (due punti esclamativi) fino alla fine della dichiarazione di procedura (infatti la dichiarazione è chiusa proprio da questo terminatore), quindi viene ripristinato al suo valore di default (il punto e virgola).

6.1.1 Restituire valori

Quando una procedura termina, i valori associati alle variabili dichiarate come valori di ritorno (tramite l'istruzione `RETURNS`) vengono restituiti al chiamante. Se si desidera concludere una procedura in un qualsiasi punto del suo codice, è possibile usare la keyword `EXIT`.

Tuttavia, in SQL una procedura può restituire anche insiemi di tuple, cioè tabelle. A questo scopo, la procedura dovrà assegnare alle variabili di ritorno i valori di ciascuna riga, e poi eseguire il comando SUSPEND.

Esempi.

```
SET TERM !! ;
CREATE PROCEDURE parametric_query(ID INTEGER) RETURNS(NOME VARCHAR
    (20), STIPENDIO INTEGER) AS
BEGIN
 FOR SELECT NOME,STIPENDIO
  FROM IMPIEGATO
  WHERE IDREPARTO = :ID
  INTO :NOME, :STIPENDIO
 DO
 BEGIN
  SUSPEND;
 END
END!!
SET TERM ; !!
```

Questa procedura accetta in input l'id di un reparto e restituisce i nomi e gli stipendi di tutti gli impiegati che vi lavorano. Notare l'uso del SUSPEND all'interno del ciclo FOR SELECT, che permette di restituire, uno dopo l'altro, gli insiemi di valori estratti dalla query, assegnati tramite il costrutto INTO alle variabili di ritorno NOME e STIPENDIO (distinte, nella query, dalle colonne NOME e STIPENDIO della tabella IMPIEGATO grazie ai due punti che le precedono).

6.1.2 Invocare una procedura

Una procedura può essere chiamata da un'altra procedura o trigger usando la seguente sintassi:

```
EXECUTE PROCEDURE nome_procedura [( nome_parametro, ... )]
 [RETURNING_VALUES ( nome_risultato, ... )]
```

Se la procedura restituisce dei valori, si utilizza la clausola RETURNING_VALUES per assegnarli ad altrettante variabili. Tuttavia, se la procedura ritorna più di una tupla (usando il comando SUSPEND), questa sintassi permette di catturarne solo la prima. Infatti, le procedure che restituiscono più tuple devono essere trattate come fossero vere e proprie tabelle dinamiche, quindi sono utilizzabili all'interno della clausola FROM di una interrogazione:

```
SELECT NOME
FROM parametric_query(12)
```

Questa query restituisce la colonna NOME estratta dalla tabella generata chiamando la procedura parametric_query con argomento 12. Notare che i nomi delle colonne generate da una procedura di questo tipo corrispondono ai nomi delle sue variabili di ritorno.

Nota: in generale, se una procedura non restituisce alcun valore, può essere invocata, con la sintassi appena vista, anche dall'interprete interattivo di SQL.

Esempi. Vogliamo creare una procedura che restituisce i dieci impiegati che prendono lo stipendio più alto e i dieci che prendono il più basso.

```
SET TERM !! ;
CREATE PROCEDURE topandbottom RETURNS(NOME VARCHAR(20), STIPENDIO
   INTEGER) AS
DECLARE VARIABLE INDICE INTEGER;
DECLARE VARIABLE NUMERO INTEGER;
DECLARE VARIABLE QUANTI INTEGER;
BEGIN
 SELECT count(*)
 FROM IMPIEGATO
 INTO :NUMERO;

 IF (NUMERO < 20) THEN QUANTI = NUMERO / 2;
 ELSE QUANTI = 10;

 INDICE =1;
 FOR SELECT NOME,STIPENDIO
  FROM IMPIEGATO
  ORDER BY STIPENDIO DESC
  INTO :NOME, :STIPENDIO
 DO
 BEGIN
  IF (INDICE<=QUANTI) THEN SUSPEND;
  INDICE = INDICE + 1;
 END

 INDICE =1;
 FOR SELECT NOME,STIPENDIO
  FROM IMPIEGATO
  ORDER BY STIPENDIO ASC
  INTO :NOME, :STIPENDIO
 DO
 BEGIN
  IF (INDICE<=QUANTI) THEN SUSPEND;
  INDICE = INDICE + 1;
 END
END!!
SET TERM ; !!
```

La procedura, in particolare, nel caso in cui i record della tabella IMPIEGATO siano meno di venti, diminuisce il numero di impiegati da restituire in modo che non ci siano sovrapposi-

zioni tra i due segmenti della classifica restituita. Notare l'uso dell'`IF` per restituire il record corrente (con `SUSPEND`) solo se si trova nell'intervallo desiderato.

A questo punto, per visualizzare la classifica così generata, dobbiamo semplicemente usare la query che segue

```
SELECT *
FROM topandbottom;
```

6.2 I Trigger

In molti DBMS è dispoibile un costrutto detto *trigger* che esegue un blocco di codice sul database quando si verifica un evento.

Dal punto di vista di SQL, i trigger sono molto simili alle procedure: si tratta di una sorta di procedura particolare che si attiva automaticamente dopo un determinato evento. Il codice di un trigger deve essere semplice e di rapida esecuzione, in quanto viene eseguito dal DBMS ad ogni manipolazione delle tabelle associate.

Gli eventi gestibili mediante trigger sono inserimenti, aggiornamenti e cancellazioni su una tabella.

I trigger sono utilizzati per diversi scopi nella progettazione di un database:

- mantenere **l'integrità referenziale** tra le varie tabelle
- mantenere **l'integrità dei dati** della singola tabella
- **monitorare** i campi di una tabella ed eventualmente generare eventi ad hoc
- **calcolare** i valori di campi particolari ad ogni modifica dei dati associati

La sintassi di un trigger, come quella di una procedura, non è completamente standardizzata. Di seguito usaremo sempre la sintassi Interbase/Firebird.

Per creare un trigger, si usa il seguente comando SQL:

```
CREATE TRIGGER nome_trigger FOR nome_tabella
 {BEFORE | AFTER} {INSERT | DELETE | UPDATE} AS
  corpo_trigger
```

dove,

- nome_trigger è il nome univoco del trigger;
- nome_tabella identifica la tabella monitorata dal trigger;
- le parole chiave `BEFORE` e `AFTER` specificano se attivare il trigger prima o dopo che l'evento indicato sia gestito dal DBMS;

- l'evento associato al trigger è specificato tramite una della parole chiave `INSERT`, `DELETE` , `UPDATE`.

In generale, i trigger che devono poter bloccare l'operazione monitorata vanno attivati *prima* (`BEFORE`) che l'evento stesso sia gestito dal DBMS. Tutti gli altri trigger possono essere attivati dopo l'evento.

All'interno del trigger, è disponibile la speciale sintassi `NEW.come_colonna` e `OLD.come_colonna` che serve a riferirsi, rispettivamente, alle colonne della riga appena inserita o aggiornata e alle colonne della riga appena cancellata o sostituita dall'aggiornamento. In particolare, manipolando i valori delle colonne della tupla `NEW` è possibile modificare direttamente il record di dati inserito/aggiornato nella tabella. Questo permette al trigger di apportare correzioni o integrazioni ai dati proposti dall'utente.

Il corpo di un trigger può contenere qualsiasi istruzione ammessa per le procedure (comprese le dichiarazioni di variabili), ma non può restituire valori. Se, all'interno del trigger, si *solleva un'eccezione* usando l'istruzione `EXCEPTION nome_eccezione`, l'evento gestito viene cancellato e non ha effetti sul DBMS. Le eccezioni, prima di essere utilizzate, devono essere create utilizzando il comando che segue, che permette di associare all'eccezione un messaggio di errore da mostrare all'utente:

```
CREATE EXCEPTION nome_eccezione 'messaggio di errore'.
```

Nota: sebbene nei trigger si possano usare istruzioni SQL come `UPDATE`, `DELETE` e `INSERT`, è necessario prestare molta attenzione alla possibilità che queste istruzioni attivino altri trigger in cascata. Potrebbe infatti accadere che, in uno o più passi, gli eventi attivati dal codice del trigger richiamino il trigger di partenza, generado un ciclo infinito.

Esempi.

Per inserire un id auto-incrementante nella tabella IMPIEGATO, da usare come chiave, è possibile, in Firebird, creare un *generatore* (cioè un contatore intero) con la sintassi

```
CREATE GENERATOR nome_generatore;
```

In seguito, è possibile usare la funzione `GEN_ID` in un trigger per riempire automaticamente il campo IDIMPIEGATO ogni volta che un nuovo record viene inserito nella tabella IMPIEGATO:

```
CREATE GENERATOR impkey;

SET TERM !! ;
CREATE TRIGGER autokey_rep FOR IMPIEGATO BEFORE INSERT AS
BEGIN
 NEW.IDIMPIEGATO = GEN_ID(impkey,1);
END!!
SET TERM ; !!
```

Abbiamo creato un trigger che usa il generatore impkey per impostare il valore della colonna IDIMPIEGATO nel record inserito, e l'abbiamo attivato *prima* del completamento dell'operazione di inserimento. Infatti, in caso contrario, avremmo ricevuto un messaggio di

errore in quanto non è possibile inserire un record di impiegato senza dare un valore alla sua chiave primaria IDIMPIEGATO.

Vogliamo adesso usare un trigger per assicurarci che il valore dello stipendio di un impiegato sia almeno di 1000 euro per ogni levello ottenuto.

```
SET TERM !! ;
CREATE TRIGGER checkMinStip FOR IMPIEGATO AFTER INSERT AS
BEGIN
 if (NEW.stipendio < NEW.livello*1000) then NEW.stipendio=NEW.
    livello*1000;
END!!
SET TERM ; !!
```

In questo caso il trigger è posto dopo l'inserimento (ma poteva anche essere posto prima), e reimposta il valore dello stipendio al suo minimo nel caso sia impostato a un valore monore. Tuttavia, questo trigger non è sufficiente: bisogna verificare che lo stipendio non sia abbassato sotto la soglia minima anche da un'operazione di aggiornamento! Per ottenere questo, si riscrive lo stesso trigger associandolo anche all'evento `UPDATE`:

```
SET TERM !! ;
CREATE TRIGGER checkMinStipUpd FOR IMPIEGATO AFTER UPDATE AS
BEGIN
 if (NEW.STIPENDIO < NEW.LIVELLO*1000) then NEW.STIPENDIO=NEW.
    LIVELLO*1000;
END!!
SET TERM ; !!
```

E' possibile, infine, usare un trigger per bloccare un'operazione non valida. A questo scopo, si dichiara prima una eccezione, che incorpora il messaggio di errore da mostrare all'utente, quindi si scrive un trigger come il seguente, che verifica che il livello di un impiegato non possa mai essere abbassato:

```
CREATE EXCEPTION illegale 'Operazione illegale!';

SET TERM !! ;
CREATE TRIGGER checkLevUpd FOR IMPIEGATO BEFORE UPDATE AS
BEGIN
 if (OLD.LIVELLO > NEW.LIVELLO) THEN EXCEPTION illegale;
END!!
SET TERM ; !!
```

Il trigger verifica se il livello *successivo* all'aggiornamento (tupla `NEW`) è minore di quello *precedente* allo stesso (tupla `OLD`). In caso affermativo, usa l'istruzione `EXCEPTION` per sollevare una eccezione che bloccherà l'operazione e ne impedirà il completamento (il trigger è posto infatti *prima* del completamento dell'operazione).

Un trigger può anche richiamare delle procedure. Ad esempio il seguente trigger utilizza la procedura controlla per decidere se il valore aggiornato di uno stipendio è valido:

```
SET TERM !! ;
CREATE TRIGGER controllo_avanzato FOR IMPIEGATO BEFORE UPDATE AS
DECLARE VARIABLE ERRORE INTEGER;
BEGIN
 EXECUTE PROCEDURE controlla(NEW.STIPENDIO) RETURNING_VALUES(ERRORE)
    ;
 IF (ERRORE<>0) NEW.STIPENDIO = OLD.STIPENDIO;
END!!
SET TERM ; !!
```

se il nuovo valore non è valido, viene ripristinato quello precedente.

Capitolo 7

Interfacciamento con i linguaggi di programmazione

Tipicamente una base di dati sarà utilizzata da una o più applicazioni client, che si interfacceranno ad essa per inserire, modificare ed estrarre informazioni (oltre che, in casi particolari, anche per modificare le struttura della base di dati stessa).

In generale, l'interazione tra DBMS e programma client utilizza SQL come lingua franca, ma necessita comunque di particolari strutture, diverse per ciascun linguaggio/piattaforma software, che servono ad adattare i valori restituiti da SQL (tabelle, informazioni di stato, eccezioni, ...) alle strutture dati utilizzabili all'interno del codice.

In questo capitolo mostreremo, molto brevemente, il sistema di interfacciamente con i DBMS utilizzato da due linguaggi largamente diffusi e multipiattaforma: PHP e Java.

7.1 I Cursori

Un concetto utile da comprendere, su cui si basano (anche se non esplicitamente) molte delle modalità di accesso ai DBMS è quello di *cursore*.

Poichè nella maggior parte dei linguaggi di programmazione non esiste il concetto di tabella dati e poichè, ad esempio, trasformare una tabella dati del DBMS in un array comporterebbe un enorme spreco di memoria nell'applicazione client, spesso i risultati di una query SQL vengono gestiti con l'aiuto di un cursore. Questo oggetto non è altro che un puntatore che può nuoversi all'interno di un insieme di tuple (il risultato dell'interrogazione), permettendo di leggere le tuple stesse *una alla volta*.

In pratica, un cursore viene creato associandovi una particolare query. A questo punto, esistono istruzioni SQL che permettono di *muovere il cursore* avanti e indietro sull'insieme di tuple risultanti. In ogni momento, è possibile usare il cursore per leggere i valori di ciascuna colonna della tupla corrente, cioè quella correntemente puntata dal cursore.

Sebbene molti DBMS mettano a disposizione comandi SQL per gestire i cursori (ad es. `CREATE CURSOR`), il loro uso spesso è implicito e gestito dallo strato di astrazione verso il DBMS fornito dal linguaggio di programmazione in uso.

7.2 PHP

Il linguaggio di scripting PHP è molto diffuso per la programmazione server-side di siti internet. Per questo motivo l'interfacciamento con i DBMS è una sua funzionalità nativa.

In PHP esistono set di istruzioni differenti per interfacciarsi con tutti i più noti DBMS. Queste istruzioni iniziano tutte con un prefisso che identifica il DBMS a cui sono dedicate.

Descriveremo qui di seguito una tipica interazione con un DBMS Interbase (che vale anche per Firebird), limitandoci alle istruzioni necessarie a eseguire delle query. Per gli altri DBMS le istruzioni e le procedure sono molto simili.

1. Per prima cosa, è necessario connettersi al DBMS e selezionare il database su cui operare. A questo scopo, si usa il comando ibase_connect:

   ```
   $dbhandle = ibase_connect($host, $username, $password)
   ```

   Al comando vengono forniti una username e una password valide, più la specifica di un database, secondo il formato di interbase, ad esempio `localhost:/home/databases/test.fdb`. Il valore restituito è l'handle di connessione, che andrà passato a tutte le successive istruzioni.

2. La query SQL viene passata al DBMS come una semplice stringa di testo usando il comando ibase_query:

   ```
   $queryhandle = ibase_query($dbhandle, $query);
   ```

   Al comando vengono passati l'handle di connessione ottenuto al passo precedente e la stringa rappresentante la query. Il valore restituito è l'handle dei risultati, tramite il quale è possibile scorrere le tuple restituite. Se la query non è di tipo `SELECT`, e quindi non restituisce risultati, il passo successivo viene saltato.

3. Si usa la funzione ibase_fetch_assoc:

   ```
   $datarow = ibase_fetch_assoc($queryhandle);
   ```

   Ad ogni chiamata a questa funzione, viene resituito un array associativo, del tipo nome_colonna = valore_colonna, contenente la successiva tupla nel set di risultati. Terminato l'insieme, la funzione restituisce false. E' inoltre possibile usare le funzioni ibase_fetch_row e ibase_fetch_object per prelevare le tuple in formati diversi. Una volta prelevati i risultati, si libera lo spazio a loro riservato chiamando ibase_free_result:

   ```
   ibase_free_result($queryhandle);
   ```

4. Terminato l'uso della base di dati, si può chiudere la connessione corrispondente chiamando ibase_close:

   ```
   ibase_close($dbhandle);
   ```

7.3 Java

L'accesso ai dati in Java si effettua usando il package JDBC (Java DataBase Connectivity). Al contrario di PHP, Java non dispone di metodi predefiniti per l'accesso a particolari DBMS. E' necessario quindi procurarsi prima di tutto il driver JDBC per il DBMS in uso ed installarlo nel sistema, dopodichè si potranno usare le classi del JDBC per accedere al DBMS in maniera astratta (cioè non dipendente dallo specifico DBMS).

1. Per prima cosa, è necessario caricare il driver del DBMS in uso. A questo scopo, si fa semplicemente riferimento alla classe che implementa il driver con la sintassi che segue:

   ```
   Class.forName (driverClassName);
   ```

   Ad esempio, per Firebird la classe del driver JDBC è `org.firebirdsql.jdbc.FBDriver`.

2. Si procede quindi con la creazione dell'oggetto Connenction tramite il metodo getConnetion della classe DriverManager:

   ```
   Connection con=DriverManager.getConnection(URL,user,pass);
   ```

   I tre parametri del metodo sono il nome utente e password da usare per l'accesso al DBMS e la stringa di connessione JDBC, che specifica l'indirizzo del DBMS e il database da selezionare. Questa stringa ha un formato che varia a seconda del DBMS in uso: ad esempio, una tipica stringa di connessione per Firebird è del tipo `jdbc:firebirdsql:localhost/3050:c:/database/azienda.gdb`.

3. Si crea un oggetto Statement sulla connessione, con la sintassi

   ```
   Statement stmt = con.createStatement();
   ```

4. Si invia la query SQL, sotto forma di stringa, al DBMS tramite lo stetment appena creato:

   ```
   ResultSet rs = stmt.executeQuery(SQLQuery);
   ```

   L'oggetto restituito, di tipo ResultSet, permette di navigare tra i risultati della query. Se invece si desidera inviare una query che non restituisce risultati, ad esempio di inserimento o aggiornamento, si usa il metodo executeUpdate:

   ```
   int rc = executeUpdate(SQLQuery);
   ```

   In questo caso, il valore restituito è un intero che rappresenta il numero di record interessati dalla query.

5. Tramite il ResultSet è possibile leggere le colonne di ciascuna tupla restituita, usando i metodi getX(nome_colonna), dove X è il tipo di dato da estrarre, ad esempio

```
String s = rs.getString("nome");
```

   Per spostarsi al record successivo, si usa il metodo next:

```
boolean end_of_set = !rs.next();
```

   Il metodo restituisce false quando i record sono esauriti. Una volta prelevati i risultati, si libera lo spazio a loro riservato chiamando il metodo close dello Statement:

```
stmt.close();
```

6. Terminato l'uso della base di dati, si può chiudere la connessione corrispondente chiamando il metodo close della Connection:

```
con.close();
```

Nota: tutte le istruzioni JDBC, in caso di errore, sollevano eccezioni derivate da SQLException.