Capitolo 6

SQL PER L'USO INTERATTIVO DI BASI DI DATI

Il linguaggio SQL (*Structured Query Language*), sviluppato nel 1973 da ricercatori dell'IBM per il sistema relazionale System/R[1], è il linguaggio universale per la definizione e l'uso delle basi di dati relazionali.

La versione originaria si è differenziata in una serie di dialetti agli inizi degli anni '80, quindi il linguaggio è stato sottoposto ad una standardizzazione da parte dei comitati X/OPEN, ANSI e ISO. Dopo il primo standard ANSI e ISO del 1984 (SQL-84), rivisto poi nel 1989 per introdurre principalmente il vincolo d'integrità referenziale (SQL-89), si è giunti allo standard di fatto oggi molto diffuso proposto dal comitato X/OPEN per i sistemi operativi UNIX, mentre i comitati ANSI e ISO hanno proseguito il loro lavoro, con una serie di estensioni significative a SQL-89. Il primo risultato è lo standard SQL-92 (o SQL2), concluso nel 1992, e l'ultimo è SQL:2003 per trattare basi di dati relazionali ad oggetti.

Attualmente, lo standard SQL-92 è seguito solo ai livelli più semplici (in pratica nel DML) dai principali DBMS relazionali (come DB2, Oracle, e Microsoft SQL Server), mentre per aspetti relativi al DDL, anche significativi, vi sono differenze importanti.

Lo standard prevede tre diversi livelli di linguaggio, di complessità crescente: *Entry SQL*, *Intermediate SQL* e *Full SQL*. Inoltre, ogni sistema che aderisce allo standard deve fornire almeno i seguenti modi di usare SQL (detti anche *binding styles*):

- *Direct* SQL, per l'uso interattivo.
- *Embedded* SQL, per l'uso con linguaggi di programmazione per lo sviluppo di applicazioni.

La trattazione di SQL verrà suddivisa in tre parti.

- In questo capitolo vengono trattati gli aspetti del linguaggio per ricerche e modifiche fatte interattivamente (*DML*), chiamate generalmente interrogazioni o *query*.
- Nel prossimo capitolo verranno mostrati gli aspetti del linguaggio per la definizione dello schema e l'amministrazione della base di dati (*DDL*).

1. Il linguaggio si chiamava SEQUEL, acronimo delle parole "Structured English QUEry Language."

– Nel Capitolo 8 verrà discusso l'uso di SQL all'interno di linguaggi di programmazione.

6.1 Algebra relazionale su multinsiemi

Nella terminologia SQL una relazione è pensata come una *tabella* con tante colonne quanti sono gli attributi delle ennuple (dette anche *record*) e tante righe quante sono le ennuple della relazione. L'ordine degli attributi è *significativo*. Un componente di un'ennupla è detto *campo*.

Si noti che in generale una tabella nel linguaggio SQL non è un insieme, come visto nel capitolo precedente quando si è discusso il modello relazionale, ma un multinsieme.[2]

Come l'algebra relazionale, il linguaggio SQL prevede solo operatori su tabelle e una query esprime in forma dichiarativa un'espressione di operatori dell'algebra relazionale estesa su multinsiemi come segue:

– ***Proiezione con duplicati***: $\pi^{b}_{X}(O)$

  con X attributi di O. Il risultato è un multinsieme.
– ***Eliminazione di duplicati***: $\delta(O)$

  Il risultato è un insieme.
– ***Ordinamento***: $\tau_X(O)$

  con X attributi di O. Il risultato è un *multinsieme ordinato*, valore che non appartiene al dominio dell'algebra su multinsiemi e quindi τ può essere usato *solo* come radice di un albero logico.
– ***Unione, intersezione e differenza***: $O_1 \cup^{b} O_2$, $O_1 \cap^{b} O_2$, $O_1 -^{b} O_2$

  Se un elemento t appare n volte in O_1 e m volte in O_2, allora

  – t appare $n + m$ volte nel multinsieme $O_1 \cup^{b} O_2$:

$$\{1,1,2,3\} \cup^{b} \{2,2,3,4\} = \{1,1,2,3,2,2,3,4\}$$

  – t appare $min(m,n)$ volte nel multinsieme $O_1 \cap^{b} O_2$:

$$\{1,1,2,3\} \cap^{b} \{2,2,3,4\} = \{2,3\}$$

  – t appare $max(0, n-m)$ volte nel multinsieme $O_1 -^{b} O_2$:

$$\{1,1,2,3\} -^{b} \{1,2,3,4\} = \{1\}$$

2. In effetti una tabella è un insieme solo se fra i suoi attributi vi è una chiave.

Alcune proprietà dell'algebra relazionale su insiemi non valgono nel caso dell'algebra relazionale su multinsiemi. Per esempio, la proprietà distributiva dell'unione e dell'intersezione su insiemi non vale nel caso di multinsiemi:

$$\{1\} \cap^{\mathbf{b}} (\{1\} \cup^{\mathbf{b}} \{1\}) = \{1\} \cap^{\mathbf{b}} \{1,1\} = \{1\}$$

mentre

$$(\{1\} \cap^{\mathbf{b}} \{1\}) \cup^{\mathbf{b}} (\{1\} \cap^{\mathbf{b}} \{1\}) = \{1\} \cup^{\mathbf{b}} \{1\} = \{1,1\}$$

Gli altri operatori dell'algebra relazionale su insiemi (selezione, raggruppamento, prodotto e giunzione) si generalizzano in modo ovvio al caso di multinsiemi.

6.2 Valori nulli

I sistemi commerciali prevedono l'uso di uno speciale valore, **NULL**, che fa parte di ogni tipo di dato e quindi assegnabile ad ogni attributo, a meno che non si imponga il vincolo d'integrità **NOT NULL** nella sua definizione, come vedremo nel prossimo capitolo.

Nel linguaggio SQL tutti gli operatori sono stati definiti in modo da poter operare anche in presenza di valori nulli e, quindi, la loro semantica presenta alcune complicazioni che ne rendono più difficile la comprensione.

Per questo motivo la presentazione degli operatori SQL per la ricerca dei dati è organizzata in due parti.

Nella prima parte, la prossima sezione, essi verranno presentati assumendo di operare su tabelle senza valori nulli e con operatori che non producono valori nulli.

Nella seconda parte, invece, descriveremo quali modifiche agli operatori descritti nella prima parte e quali nuovi operatori sono introdotti a causa di questi valori, insieme ad alcune indicazioni su come utilizzarli.

6.3 Operatori per la ricerca di dati

I comandi per la definizione delle tabelle verranno mostrati nel prossimo capitolo. In seguito negli esempi di interrogazioni faremo riferimento allo schema della base di dati di Figura 6.1.

Il comando principale dell'SQL è **SELECT** che, in una forma semplificata, ha la seguente struttura:

SELECT [**DISTINCT**] *Attributi*
FROM *GiunzioneDiTabelle*
[**WHERE** *Condizione*];

dove

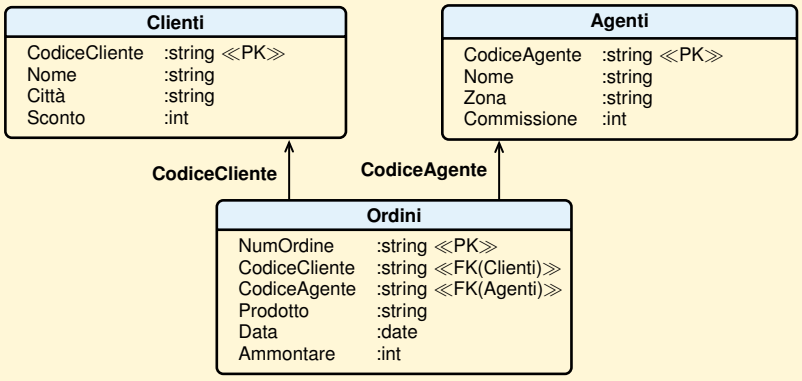


Figura 6.1: Rappresentazione grafica dello schema esempio

Attributi ::= $*$ | *Attributo* { , *Attributo* }
GiunzioneDiTabelle ::= *Tabella* [*Ide*] { **JOIN** *Tabella* [*Ide*] **ON** *CondizioneDiGiunzione* }

Il significato del comando **SELECT** può essere dato con le seguenti equivalenze:

SELECT $*$
FROM R_1 **JOIN** R_2 **ON** C_2
... **JOIN** R_n **ON** C_n
WHERE C; $\equiv \sigma_C (R_1 \underset{C_2}{\bowtie} R_2 \cdots \underset{C_n}{\bowtie} R_n)$

SELECT **DISTINCT** $A_1, \ldots, A_n$
FROM R_1 **JOIN** R_2 **ON** C_2
... **JOIN** R_n **ON** C_n
WHERE C ; $\equiv \delta(\pi^{\mathbf{b}}_{A_1,\ldots,A_n}(\sigma_C (R_1 \underset{C_2}{\bowtie} R_2 \cdots \underset{C_n}{\bowtie} R_n)))$

Il comando **SELECT** è quindi una combinazione di una giunzione, una restrizione e di una proiezione:

- con *Attributi* si specificano gli attributi del risultato; "$*$" sta per tutti gli attributi e **DISTINCT** è un'opzione per escludere i duplicati dal risultato;
- con *GiunzioneDiTabelle* si specificano le tabelle che intervengono nella giunzione con le relative condizioni di giunzione;
- con *Condizione* si specifica la condizione che deve essere soddisfatta dalle ennuple del risultato.

Le ricerche più semplici si fanno usando solo alcune clausole, come negli esempi seguenti.

- Singola tabella:

```
SELECT  *
FROM    Agenti;
```

– Restrizione:

```
SELECT  *
FROM    Ordini
WHERE   Ammontare > 1000;
```

– Proiezione:

```
SELECT  DISTINCT Nome, Città
FROM    Clienti;
```

– Giunzione:

```
SELECT  *
FROM    Clienti
        JOIN Ordini ON Clienti.CodiceCliente = Ordini.CodiceCliente;
```

Si noti che:

1. l'operatore **SELECT** può restituire un multinsieme se non viene specificata l'opzione **DISTINCT**, come nell'esempio seguente, dove il risultato è una tabella con righe duplicate se almeno un agente ha fatto più ordini per lo stesso ammontare:

   ```
   SELECT  CodiceAgente, Ammontare
   FROM    Ordini;
   ```

   Per non avere duplicati nel risultato occorre porre **SELECT DISTINCT**.
2. Per evitare ambiguità quando si opera sulla giunzione di tabelle con gli stessi attributi, si usa la notazione con il punto Tabella.Attributo, come nel seguente esempio, dove è usata uniformemente per trovare i codici dei clienti e l'ammontare degli ordini fatti:

   ```
   SELECT  Clienti.CodiceCliente, Ordini.Ammontare
   FROM    Clienti
           JOIN Ordini ON Clienti.CodiceCliente = Ordini.CodiceCliente;
   ```

   La condizione che segue la parola chiave **ON** è detta *condizione di giunzione*. Si noti che una sintassi datata in SQL prevedeva di elencare le tabelle separate da virgola nella clausola **FROM** e di mettere le condizioni di giunzioni in **AND** nella clausola **WHERE**. La virgola rappresenta il prodotto, che nella sintassi moderna viene indicato con **CROSS JOIN**. Vedremo più avanti questo e altri operatori per vari tipi di giunzione.
3. Una notazione alternativa alla precedente prevede l'associazione di identificatori alle tabelle, da usare per specificare gli attributi nella notazione Identificatore.Attributo. Questi identificatori sono detti *variabili di correlazione*, *pseudonimi* o *alias*. Così l'esempio precedente può essere scritto come:

```
SELECT   C.CodiceCliente, O.Ammontare
FROM     Clienti C
         JOIN Ordini O ON C.CodiceCliente = O.CodiceCliente;
```

Questa seconda notazione è indispensabile nei casi in cui si debba fare la giunzione di una tabella con se stessa, come nel seguente esempio:

```
SELECT   A2.Nome
FROM     Agenti A1
         JOIN Agenti A2 ON A2.Zona = A1.Zona
WHERE    A1.CodiceAgente = 'A01';
```

che restituisce i nomi degli agenti della stessa zona dell'agente con codice 'A01'.

Vediamo adesso in dettaglio le varie componenti del comando.

6.3.1 La clausola SELECT

Il risultato di un'espressione **SELECT** *Attributi* **FROM** ... è una tabella (a cui è possibile dare un nome con il comando **CREATE TABLE**, come vedremo nel prossimo capitolo), i cui nomi di colonna sono quelli indicati in *Attributi*. La sintassi completa della clausola *Attributi* è la seguente:[3]

Attributi ::= $*$ | *Espr* [[**AS**] *NuovoNome*] {, *Espr* [[**AS**] *NuovoNome*] }
Espr ::= [*Ide*.]*Attributo* |
Costante |
"(" *Espr* ")" |
[−] *Espr* [ρ *Espr*] |
(**SUM** | **COUNT** | **AVG** | **MAX** | **MIN**)
"(" [**DISTINCT**] [*Ide*.]*Attributo* ")" |
COUNT "(" $*$ ")"

ρ ::= (+ | − | $*$ | /)

NuovoNome ::= *Identificatore*

[**AS**] *NuovoNome* si usa per cambiare il nome delle colonne del risultato, come in:

```
SELECT   CodiceCliente AS Codice, Nome AS NomeECognome
FROM     Clienti;
```

che restituisce una tabella ottenuta proiettando Clienti su CodiceCliente e Nome e ridenominando le due colonne con Codice e CognomeENome rispettivamente.

3. In realtà i sistemi commerciali prevedono espressioni con un numero maggiore di operatori.

Con il comando **SELECT** si possono calcolare tabelle le cui colonne non corrispondono a colonne delle tabelle selezionate, ma sono ottenute come espressioni a partire da attributi e costanti.

Ad esempio:

```
SELECT   CodiceAgente, Nome,
         Commissione / 100 AS Commissione,
         'Italia' AS Paese
FROM     Agenti;
```

restituisce una tabella con tante righe quanti sono gli agenti, e con quattro colonne, due ottenute per proiezione dalla tabella Agenti, una con valori calcolati da un'altra colonna, ed una formata da valori costanti.

6.3.2 La clausola FROM

Come già detto, nella parte che segue il **FROM** vi può essere una tabella, su cui fare una restrizione o una proiezione, oppure una serie di tabelle combinate attraverso un operatore di giunzione sul cui risultato viene fatto una restrizione o una proiezione.

Al posto di una tabella si può usare un'espressione **SELECT**, ma questa possibilità non verrà considerata per semplicità. Un'espressione **SELECT** può invece apparire nella condizione, come vedremo, e viene detta *SottoSelect* o **SELECT** *annidata*. Quando daremo la sintassi completa del **SELECT**, si vedrà che in una *SottoSelect* non si possono usare tutte le opzioni previste per una **SELECT**, ma per ora queste limitazioni si possono ignorare.

Nello standard SQL-92 sono stati introdotti vari tipi di giunzione, comprese le giunzioni esterne, oltre al prodotto. La sintassi dell'argomento *TabelleDiGiunzione* della clausola **FROM** diventa così la seguente:

TabelleDiGiunzione ::= Tabella [*Ide*]
| "(" *TabelleDiGiunzione* ")"
| Tabella [*Ide*] { *Giunzione Tabella* [*Ide*]
[**USING** "(" *Attributo* {, *Attributo* } ")" | **ON** *Condizione*] }

Giunzione ::= , | [**CROSS** | **NATURAL**] **JOIN**

dove le clausole **USING** e **ON** si possono usare solo con l'operatore **JOIN**. Il significato è il seguente:

1. **CROSS JOIN** e "," corrispondono all'operatore di prodotto cartesiano; nella notazione moderna si preferisce l'uso dell'operatore **CROSS JOIN**;
2. **NATURAL JOIN** è la giunzione naturale;
3. **JOIN** ... **USING** è la giunzione sui valori uguali degli attributi specificati e presenti in entrambe le tabelle, in generale un sottoinsieme degli attributi con lo stesso nome presenti in entrambe le tabelle;

4. **JOIN** ... **ON** è la giunzione sui valori degli attributi che soddisfano una generica condizione;

Vediamo degli esempi d'uso dei nuovi operatori:

– *Giunzione naturale*: per trovare i codici dei clienti e l'ammontare degli ordini fatti, si pone:

```
SELECT  Clienti.CodiceCliente, Ordini.Ammontare
FROM    Clienti
        NATURAL JOIN Ordini;
```

in alternativa, si può usare la forma **USING**:

```
SELECT  Clienti.CodiceCliente, Ordini.Ammontare
FROM    Clienti
        JOIN Ordini USING (CodiceCliente);
```

– *Giunzione*: per trovare gli ordini degli agenti, si pone:

```
SELECT  Agenti.CodiceAgente, Ordini.Ammontare
FROM    Agenti
        JOIN Ordini ON Agenti.CodiceAgente = Ordini.CodiceAgente;
```

6.3.3 La clausola WHERE

Come già detto, nella parte che segue il **WHERE** viene specificata la condizione che deve essere soddisfatta dalle ennuple da elaborare.

Una condizione di ricerca è definita ricorsivamente come segue:

Condizione ::= *Predicato* |
"(" *Condizione* ")" |
NOT *Condizione* |
Condizione (**AND** | **OR**) *Condizione*

con **AND**, **OR** e **NOT** i classici connettivi logici per esprimere condizioni complesse.

SQL prevede anche i seguenti predicati:

1. *Espr* θ (*Espr* | "(" *SottoSelect* ")")

   con θ un operatore di confronto dell'insieme $\{=, <>, >, >=, <, <=\}$. La *SottoSelect* a destra è ammessa solo se la tabella risultante contiene un solo valore. Ad esempio, per trovare il codice degli agenti con commissione uguale a quella dell'agente con codice 'A01', evitando che il suo codice appaia nel risultato, si pone:

```
SELECT   CodiceAgente
FROM     Agenti
WHERE    NOT (CodiceAgente = 'A01')
         AND Commissione = (
             SELECT   Commissione
             FROM     Agenti
             WHERE    CodiceAgente = 'A01');
```

2. *Espr*$_1$ **BETWEEN** *Espr*$_2$ **AND** *Espr*$_3$

   Il predicato è equivalente alla condizione:

   (*Espr*$_2$ $<=$ *Espr*$_1$ **AND** *Espr*$_1$ $<=$ *Espr*$_3$)

3. *Attributo* **LIKE** *Stringa*

   È un predicato per confrontare stringhe. *Stringa* può contenere i seguenti caratteri speciali:

   a) "_" che sta per qualsiasi carattere;
   b) "%" che sta per una qualsiasi sequenza di caratteri, anche vuota.

4. **EXISTS** "(" *SottoSelect* ")"

   Il predicato **EXISTS** è vero se la *SottoSelect* ritorna una relazione con almeno un'ennupla, falso se la relazione restituita è la relazione vuota.
   Ad esempio, per trovare i nomi dei clienti che hanno fatto acquisti dall'agente con codice 'A01', si pone:

```
SELECT   C.Nome
FROM     Clienti C
WHERE    EXISTS (
             SELECT   *
             FROM     Ordini O
                      JOIN Agenti A ON O.CodiceAgente = A.CodiceAgente
             WHERE    O.CodiceCliente = C.CodiceCliente
                      AND A.CodiceAgente = 'A01');
```

   Si noti che in una *SottoSelect* si può usare la variabile di correlazione della **SELECT** più esterna, ma non si può fare il contrario.
   Il predicato **EXISTS** è bene usarlo quando è strettamente necessario e non in espressioni che possono usare la giunzione, sfruttando la seguente equivalenza:

```
SELECT   R1.A1,..., R1.An
FROM     R1
WHERE    [Condizione C1 su R1 AND ]
         EXISTS ( SELECT   *
                  FROM     R2
                  WHERE    Condizione C2 su R2 e R1 [AND Condizione C3 su R2] );
```

è equivalente a[4]:

```
SELECT   DISTINCT R1.A1,..., R1.An
FROM     R1 JOIN R2 ON Condizione C2 su R1 e R2
WHERE    [Condizione C1 su R1 AND]
         [Condizione C3 su R2];
```

Ad esempio, per la query precedente si può scrivere:

```
SELECT   DISTINCT C.Nome
FROM     Clienti C
         JOIN Ordini O ON O.CodiceCliente = C.CodiceCliente
         JOIN Agenti A ON O.CodiceAgente = A.CodiceAgente
WHERE    A.CodiceAgente = 'A01');
```

Questa seconda formulazione è preferibile perché in generale i sistemi relazionali prevedono un *ottimizzatore* delle interrogazioni che ha più possibilità di ottimizzare l'esecuzione di una giunzione che un'espressione con *SottoSelect*. Solo in alcuni sistemi invece l'ottimizzatore è anche in grado di riconoscere che l'interrogazione con il predicato **EXISTS** può essere trasformata innanzitutto nell'espressione con la giunzione prima di procedere con gli altri passi di ottimizzazione.

5. *Espr* **IN** ("(" *Valore* {, *Valore* } ")" | "(" *SottoSelect* ")")

È un predicato per controllare se un valore di *Espr* appartiene ad un insieme definito per enumerazione o come risultato di una *SottoSelect*. *Espr* è un valore numerico o una stringa. Il predicato vale **TRUE** se il valore è fra quelli elencati o quelli prodotti dalla *SottoSelect* e falso altrimenti. Ad esempio, per trovare i codici degli agenti della zona di Pisa, Firenze o Siena si pone:

```
SELECT   CodiceAgente
FROM     Agenti
WHERE    Zona IN ('Pisa', 'Firenze', 'Siena');
```

Si noti che per questo operatore, come per **BETWEEN**, **LIKE** e **EXISTS** la negazione, oltre che con la sintassi **NOT** (*Predicato*), si ottiene anche premettendo **NOT** al nome dell'operatore. Ad esempio per trovare i codici di tutti gli agenti tranne quelli della zona di Pisa e Firenze si pone:

```
SELECT   CodiceAgente
FROM     Agenti
WHERE    Zona NOT IN ('Pisa', 'Firenze');
```

4. L'equivalenza vale sole se R1.A1,..., R1.An sono una superchiave per R1.

Nel caso dell'uso con una *SottoSelect*, il predicato **IN** è equivalente ad **EXISTS** come si può vedere dal seguente esempio, in cui si cercano i nomi dei clienti con ordini per più di 1000 euro:

```
SELECT   Nome
FROM     Clienti
WHERE    CodiceCliente IN (
             SELECT   CodiceCliente
             FROM     Ordini
             WHERE    Ammontare > 1000);
```

equivalente a:

```
SELECT   Nome
FROM     Clienti C
WHERE    EXISTS (
             SELECT   *
             FROM     Ordini O
             WHERE    Ammontare > 1000
                      AND C.CodiceCliente = O.CodiceCliente);
```

Una trasformazione equivalente con **NOT EXISTS** vale per il **NOT IN**.
Per le considerazioni precedenti su **EXISTS** anche in questo caso è bene riscrivere l'interrogazione usando una giunzione.

6. *Espr* θ(**ALL** | **ANY**) "(" *SottoSelect* ")"

Il predicato "θ **ALL**" (con θ un operatore di confronto), è vero se il primo operando sta nella relazione specificata con ogni elemento del secondo operando, che deve essere un insieme risultato di una **SELECT**. Se l'insieme è vuoto il predicato è vero.
Il predicato "θ **ANY**", è vero se il primo operando sta nella relazione specificata con almeno un elemento del secondo operando. Se l'insieme è vuoto, il predicato è falso.
Si noti che "*Espr* **IN** (*SottoSelect*)" equivale a "*Espr* =**ANY** (*SottoSelect*)", mentre "*Espr* **NOT IN** (*SottoSelect*)" equivale a "*Espr* <>**ALL** (*SottoSelect*)".
Ad esempio, per trovare il codice degli agenti la cui commissione supera quella di ogni agente della zona di Pisa, si pone:

```
SELECT   CodiceAgente
FROM     Agenti
WHERE    Commissione >ALL(
             SELECT   Commissione
             FROM     Agenti
             WHERE    Zona = 'Pisa');
```

Condizioni con quantificatore esistenziale

Immaginiamo di voler trovare i nomi dei clienti che hanno fatto *almeno* un ordine per più di 1000 euro. Se esistesse in SQL il quantificatore esistenziale, come previsto nell'SQL:2003, estensione dell'SQL per sistemi relazionali ad oggetti, e definito come segue:

FOR SOME x **IN** S [**WHERE** C1(x)] : C2(x)

che è vero se esiste almeno un elemento di S [**WHERE** C1(x)] che soddisfa C2(x), si porrebbe:

```
SELECT  Nome
FROM    Clienti C
WHERE   FOR SOME O IN Ordini WHERE O.CodiceCliente = C.CodiceCliente :
                Ammontare > 1000;
```

Questo tipo di interrogazione si può esprimere in SQL in vari modi, vediamo quelli più comuni.

– si usa una giunzione:

```
SELECT  DISTINCT Nome
FROM    Clienti C
        JOIN Ordini O ON O.CodiceCliente = C.CodiceCliente
WHERE   Ammontare > 1000;
```

– si usa il predicato **EXISTS**:

```
SELECT  Nome
FROM    Clienti C
WHERE   EXISTS (
            SELECT  *
            FROM    Ordini O
            WHERE   O.CodiceCliente = C.CodiceCliente
                    AND Ammontare > 1000);
```

Condizioni con quantificatore universale

Immaginiamo di voler trovare il codice dei clienti che hanno fatto ordini da *tutti* gli agenti della zona Pisa. Questo tipo di interrogazioni sono fra quelle più difficili da scrivere in SQL e si mostra un metodo per farlo, anche se il risultato finale non è l'unico possibile. Se esistesse in SQL il quantificatore universale definito come segue:

FOR ALL x **IN** S [**WHERE** C1(x)] : C2(x)

che è vero se ogni elemento di S [**WHERE** C1(x)] soddisfa C2(x), l'interrogazione si potrebbe esprimere nella forma "trovare il codice dei clienti C tali che per ogni agente A della zona Pisa esiste un ordine che riguarda C e A" ponendo:

```
SELECT  CodiceCliente
FROM    Clienti C
WHERE   FOR ALL A IN Agenti WHERE Zona = 'Pisa' :
            FOR SOME O IN Ordini WHERE A.CodiceAgente = O.CodiceAgente :
                O.CodiceCliente = C.CodiceCliente;
```

Purtroppo in SQL non esiste l'operatore **FOR ALL**, e i predicati del tipo =**ALL** o =**ANY** non consentono in generale di formulare interrogazioni di questo tipo.

Per risolvere il problema, una soluzione può essere trovata ricordando che vale la tautologia:

$$\forall x \in P.Q(x) \Leftrightarrow \neg\exists x \in P.\neg Q(x)$$

da cui discende:

$$\forall x \in A(\exists y \in B(x).p(x,y)) \Leftrightarrow \neg\exists x \in A\neg(\exists y \in B(x).p(x,y))$$

In parole, la relazione significa che sono equivalenti le due affermazioni: (1) per ogni x esiste un y tale che $p(x,y)$ è vero e (2) non esiste un x tale che non esiste un y con $p(x,y)$ vero.

Pertanto, l'interrogazione iniziale si può formulare in modo equivalente con una doppia negazione usando il quantificatore esistenziale: trovare il codice dei clienti C per i quali non esiste un agente A della zona Pisa per il quale non esiste un ordine che riguarda C e A:

```
SELECT  CodiceCliente
FROM    Clienti C
WHERE   NOT FOR SOME A IN Agenti WHERE Zona = 'Pisa' :
            NOT FOR SOME O IN Ordini WHERE A.CodiceAgente = O.CodiceAgente :
                O.CodiceCliente = C.CodiceCliente;
```

A questo punto l'interrogazione si può esprimere in SQL usando il predicato **EXISTS**:

```
SELECT  C.CodiceCliente
FROM    Clienti C
WHERE   NOT EXISTS (
            SELECT  *
            FROM    Agenti A
            WHERE   A.Zona = 'Pisa'
                    AND NOT EXISTS (
                        SELECT  *
                        FROM    Ordini O
                        WHERE   A.CodiceAgente = O.CodiceAgente
                                AND O.CodiceCliente = C.CodiceCliente));
```

Si noti che se non ci sono agenti della zona Pisa, l'interrogazione ritorna i codici di tutti i clienti, perché la quantificazione universale **FOR ALL** x **IN** S [**WHERE** C1(x)]: C2(x)

è vera se l'insieme S è vuoto. Per escludere questo caso occorre completare l'interrogazione con una quantificazione esistenziale come segue:

```
SELECT  C.CodiceCliente
FROM    Clienti C
WHERE   NOT EXISTS(
            SELECT  *
            FROM    Agenti A
            WHERE   A.Zona = 'Pisa'
                    AND NOT EXISTS (
                        SELECT  *
                        FROM    Ordini O
                        WHERE   A.CodiceAgente = O.CodiceAgente
                                AND O.CodiceCliente = C.CodiceCliente))
        AND EXISTS (
            SELECT  *
            FROM    Agenti A
            WHERE   A.Zona = 'Pisa');
```

6.3.4 Clausola di ordinamento

ORDER BY *Attributo* [**DESC**] {, *Attributo* [**DESC**]}

Questa specifica va posta dopo l'eventuale **WHERE**. Il risultato del **SELECT** viene ordinato in base al valore di certi attributi, in ordine discendente specificando **DESC**, altrimenti in ordine ascendente. Ad esempio:

```
SELECT      CodiceAgente, Commissione
FROM        Agenti
WHERE       Zona = 'Pisa'
ORDER BY    Commissione DESC;
```

6.3.5 Funzioni di aggregazione

Nella clausola **SELECT** si possono usare anche le funzioni predefinite **MAX**, **MIN**, **COUNT**, **AVG**, **SUM**, dette *funzioni di aggregazione* o *funzioni statistiche*. Esse operano su un insieme di valori restituendo rispettivamente il massimo valore, il minimo, o il numero dei valori, il valore medio e la somma (solo per valori numerici). Per semplicità, assumeremo che i valori siano quelli di una colonna, ma in generale possono essere calcolati con un'espressione aritmetica da quelli di più colonne.

COUNT(**DISTINCT** *Attributo*) restituisce il numero dei valori diversi di una colonna, mentre **COUNT**(∗) restituisce il numero di ennuple del risultato.

```
SELECT   MIN(Ammontare), MAX(Ammontare), AVG(Ammontare)
FROM     Ordini
WHERE    Data = '01032019';
```

restituisce il valore minimo, massimo e medio dell'ammontare degli ordini effettuati in data 1/3/2019, mentre

```
SELECT   COUNT(*)
FROM     Ordini
WHERE    CodiceAgente = 'A01';
```

restituisce il numero degli ordini dell'agente con codice 'A01'.

Quando si usano le funzioni di aggregazione nella **SELECT**, il risultato deve essere un'unica ennupla di valori e quindi si possono usare più funzioni di aggregazione, ma non si possono usare funzioni di aggregazione e attributi, se non nel caso descritto nella prossima sezione. Ad esempio, è scorretto scrivere:

```
SELECT   NumOrdine, MIN(Ammontare), MAX(Ammontare)
FROM     Ordini
WHERE    Data = '01032019';
```

6.3.6 Operatore di raggruppamento

Il comando **SELECT** con il **GROUP BY** ha la seguente struttura:

```
SELECT      DISTINCT S_A, S_FA
FROM        T
WHERE       W_C
GROUP BY    G_A
HAVING      H_C
ORDER BY    O_A;
```

dove

- S_A sono gli attributi e S_{FA} sono le funzioni di aggregazione della clausola **SELECT**;
- T sono le tabelle della clausola **FROM**;
- W_C è la condizione della clausola **WHERE**;
- G_A sono gli attributi di raggruppamento della clausola **GROUP BY**, con $S_A \subseteq G_A$;
- H_C è la condizione della clausola **HAVING** con le funzioni di aggregazione H_{FA};
- O_A sono gli attributi di ordinamento della clausola **ORDER BY**;
- le clausole **DISTINCT**, **WHERE**, **HAVING** e **ORDER BY** sono opzionali.

In Figura 6.2 è mostrato il significato del comando **SELECT** con il **GROUP BY**, e l'uso di due tabelle R e S in giunzione, con un albero logico dell'algebra relazionale su multinsiemi.

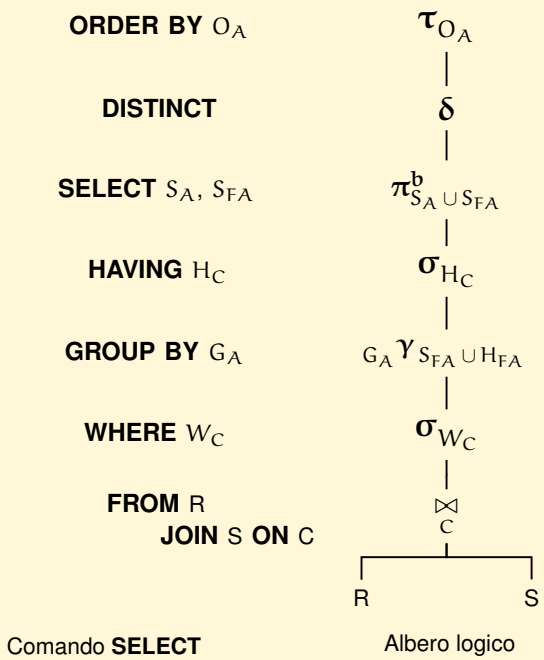


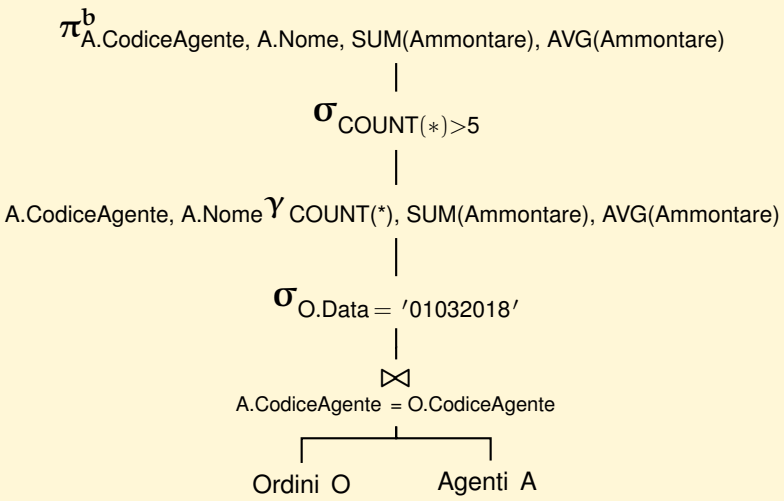
Figura 6.2: Significato del comando **SELECT** con il **GROUP BY**

Ad esempio, per trovare il codice e nome degli agenti con più di cinque ordini in data 1/3/2019 e, degli ordini fatti, il totale e la media dell'ammontare, si pone:

```
SELECT    A.CodiceAgente, A.Nome,
          SUM(Ammontare), AVG(Ammontare)
FROM      Ordini O
          JOIN Agenti A ON A.CodiceAgente = O.CodiceAgente
WHERE     O.Data = '01032019'
GROUP BY  O.CodiceAgente, A.Nome
HAVING    COUNT(*) > 5;
```

Si osservi che due agenti con lo stesso codice hanno anche lo stesso nome, per cui raggruppare rispetto a CodiceAgente, Nome equivale a raggruppare rispetto al solo codice. Tuttavia, se si raggruppasse solo rispetto al codice, il sistema non permetterebbe poi di selezionare il campo Nome.

Il significato del comando **SELECT** è dato dal seguente albero logico:



6.3.7 Operatori insiemistici

UNION, **INTERSECT** ed **EXCEPT** sono gli operatori insiemistici dell'algebra relazionale, che ritornano *insiemi* anche se gli argomenti contengono duplicati. Gli argomenti sono espressioni **SELECT** con risultato tabelle dello stesso tipo, ovvero hanno uguale gli attributi e il tipo degli attributi con lo stesso nome. L'ordine degli attributi è *significativo*.

Per esempio, per trovare i nomi diversi dei clienti e degli agenti, si pone:

```
SELECT  Nome
FROM    Clienti
    UNION
SELECT  Nome
FROM    Agenti;
```

Un operatori insiemistico seguito dalla parola chiave **ALL** (ad esempio **UNION ALL**) non elimina i duplicati dal risultato, con la semantica degli operatori unione, intersezione e differenza di multinsiemi dell'algebra su multinsiemi definiti in Sezione 6.1.

6.3.8 Sintassi completa del SELECT

Come riepilogo, diamo la sintassi del **SELECT** per come è stato presentato finora:

Select ::= *SelectEspr*
[**ORDER BY** *Attributo* [**DESC**] {, *Attributo* [**DESC**]}]

SelectEspr ::= *BloccoSelect* |
SelectEspr OperatoreInsiemistico SelectEspr |
"(" *SelectEspr* ")"

dove *BloccoSelect* è definita come:

BloccoSelect ::=
SELECT [**DISTINCT**]
($*$ | *Espr* [[**AS**] *NuovoNome*] {, *Espr* [[**AS**] *NuovoNome*] })
FROM *Tabella* [*Ide*] { **JOIN** *Tabella* [*Ide*] **ON** *Condizione* }
[**WHERE** *Condizione*]
[**GROUP BY** *Attributo* {, *Attributo*} [**HAVING** *Condizione*]]

Si noti che in un *BloccoSelect* non si può usare **ORDER BY**.
Una condizione ha la seguente sintassi:

Condizione ::= *Predicato* |
"(" *Condizione* ")" |
NOT *Condizione* |
Condizione (**AND** | **OR**) *Condizione*

Predicato ::= *Espr* [**NOT**] **IN** "(" *SottoSelect* ")" |
Espr [**NOT**] **IN** "(" *Valore* {,*Valore*} ")" |
Espr θ (*Espr* | "(" *SelectEspr* ")") |
Espr **IS** [**NOT**] NULL |
Espr θ(**ANY** | **ALL**) "(" *SelectEspr* ")" |
[**NOT**] **EXISTS**"(" *SelectEspr* ")" |
Espr [**NOT**] **BETWEEN** *Espr* **AND** *Espr* |
Espr [**NOT**] **LIKE** *Stringa*

θ ::= $< \mid <= \mid = \mid <> \mid > \mid >=$

Un'espressione *Espr* è definita dalla seguente grammatica:

Espr ::= [*Ide.*]*Attributo* |
Costante |
"(" *Espr* ")" |
[$-$] *Espr* [ρ *Espr*] |
(**SUM** | **COUNT** | **AVG** | **MAX** | **MIN**)
"(" [**DISTINCT**] [*Ide.*]*Attributo* ")" |
COUNT "(" $*$ ")"

ρ ::= ($+ \mid - \mid * \mid /$)

Infine, una tabella è data dalla seguente grammatica:

Tabella ::= *Ide* |
Tabella Giunzione Tabella
[**USING** "(" *Attributo* { , *Attributo* } ")" | **ON** *Condizione*]

Giunzione ::= "," | [(**CROSS** | **NATURAL**)] **JOIN**

OperatoreInsiemistico ::=
(**UNION** [**ALL**] | **INTERSECT** [**ALL**] | **EXCEPT** [**ALL**])

Si noti che la sintassi permette di scrivere delle **SELECT** che non sono ammesse nel linguaggio. Un'interrogazione valida deve infatti soddisfare almeno i seguenti vincoli:

- le funzioni di aggregazione non possono essere usate nei predicati della clausola **WHERE**;
- in un'interrogazione senza **GROUP BY**, tutti gli attributi nella clausola **SELECT** sono argomento di una funzione di aggregazione oppure nessuna funzione di aggregazione può apparire nella clausola;
- ricordiamo infine che in un'interrogazione con **GROUP BY**, tutti gli attributi che appaiono nelle clausole **HAVING** e **SELECT**, non come argomento di una funzione di aggregazione, sono attributi di raggruppamento.

6.4 Il valore NULL

In SQL i valori nulli sono rappresentati dal valore speciale **NULL** che può essere il valore di un attributo di qualunque tipo, a meno che non sia presente il vincolo **NOT NULL** nella definizione dell'attributo.

La presenza di questo valore complica il linguaggio SQL, e limita alcune importanti ottimizzazioni. In questa sezione vediamo le principali conseguenze della presenza del valore **NULL**, rimandando ad un manuale del linguaggio per approfondimenti sull'argomento.

In seguito negli esempi di interrogazioni faremo riferimento allo schema della base di dati di Figura 6.3 in cui l'attributo Supervisore di Agenti rappresenta una relazione gerarchica su Agenti con diretta univoca e parziale. Si noti che, come descritto nel Capitolo 4, la freccia che rappresenta il collegamento attraverso la chiave esterna ha un taglio per indicare che l'attributo può avere valore **NULL** .

6.4.1 Operazioni con valori NULL

Operatori aritmetici e di confronto su dati elementari

Gli operatori aritmetici e di confronto sui dati elementari quando applicati ad almeno un operando nullo non provocano errore, ma restituiscono il valore **NULL**. Si noti che questo è vero anche per l'operatore di uguaglianza, quindi anche se si confronta **NULL** con sé stesso il risultato è **NULL**.

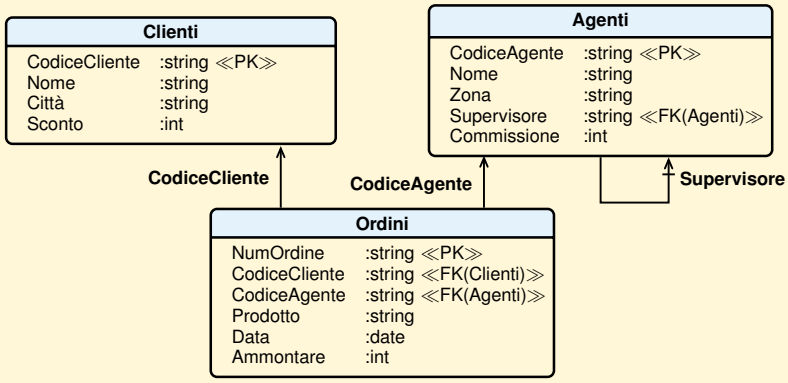


Figura 6.3: Rappresentazione grafica dello schema esempio con valori nulli

Si noti che nello standard, in questo caso seguito solo da alcuni DBMS, gli operatori di confronto in presenza di valore **NULL** producono in realtà lo speciale valore booleano **UNKNOWN**, che può essere considerato come un tipo particolare di valore nullo. Infatti, quando tale valore è il risultato di una espressione usata nella clausola **SELECT**, viene convertito a **NULL**. Per questo motivo, e dato che questo valore non è presente in tutti i sistemi, in questo testo lo consideriamo equivalente al valore **NULL**.

Operatori booleani

Gli operatori booleani seguono la regola precedente con le seguenti eccezioni:

- L'operatore **OR** quando ha un operando **TRUE** restituisce sempre **TRUE**;
- **AND** con un operando **FALSE** restituisce **FALSE**;
- gli operatori logici che accettano un insieme come operando, come gli operatori **IN**, "θ **ALL**" e "θ **ANY**", se l'insieme contiene almeno un valore **NULL**, hanno un comportamento analogo a quello di **AND** e **OR**: se l'operatore può restituire un risultato certo, come **IN** o "θ **ANY**" quando trovano un valore che soddisfa la condizione restituiscono **TRUE**, altrimenti restituiscono il valore **NULL** (o, come già detto, **UNKNOWN** per i sistemi che lo prevedono).

In ogni caso la cosa importante da notare è che ai fini del recupero delle informazioni, cioè quano la condizione è usata nelle clausole **WHERE** and **HAVING**, il valore **NULL** è equivalente al valore **FALSE** (e quindi i dati non vengono recuperati).

Operatori di aggregazione

Le funzioni di aggregazione **MAX**(A), **MIN**(A), **AVG**(A), **SUM**(A) e **COUNT**(A) ignorano i valori **NULL**. Se una colonna A contiene solo valori **NULL** le funzioni **MAX**(A), **MIN**(A), **AVG**(A), **SUM**(A) resituiscono **NULL**, mentre **COUNT**(A) vale zero.

COUNT(**DISTINCT** A) restituisce il numero dei valori diversi di una colonna, esclusi i valori **NULL**, mentre **COUNT**(∗) restituisce il numero di ennuple del risultato. Si noti quindi che in presenza di valori **NULL**, **AVG**(A) è uguale a **SUM**(A)/**COUNT**(A) ma è diverso da **SUM**(A)/**COUNT**(∗).

GROUP BY e operatori insiemistici

Se negli attributi di raggruppamento esistono valori **NULL**, questi sono considerati indistinti ai fini del raggruppamento e quindi costituiscono un unico gruppo.

Analogamente, gli operatori insiemistici che eliminano i duplicati eliminano anche le eventuali duplicazioni del valore **NULL**.

Operatori specifici per i valori NULL

1. *Espr* **IS NULL**

   per controllare se un'espressione ha valore **NULL**, ritorna sempre **TRUE** oppure **FALSE**. Ad esempio, assumendo che alcuni agenti non hanno supervisore, quindi che per loro l'attributo Supervisore è **NULL**, per trovare i loro codici si pone:

   ```
   SELECT  CodiceAgente AS CodiceSupervisoreAltoLivello
   FROM    Agenti
   WHERE   Supervisore IS NULL;
   ```

2. $Espr_1$ **IS DISTINCT FROM** $Expr_2$

   Questo predicato, diversamente da <>, ritorna **FALSE** non solo quando i due valori sono uguali, ma anche quando sono entrambi **NULL**. Se i due valori sono diversi, o uno solo dei due è **NULL**, ritorna **TRUE**[5].

   Si possono usare **IS NOT NULL**, e **IS NOT DISTINCT FROM** come abbreviazioni della forma negativa.

3. **COALESCE** "(" $Espr_1, \ldots Espr_n$ ")" .

   Viene usato per trasformare un valore **NULL** in un valore non nullo. Valuta le espressioni in sequenza, da sinistra verso destra. Viene restituito il primo valore trovato diverso da **NULL**. L'operatore ritorna **NULL** se tutte le espressioni hanno valore **NULL**.

Giunzioni esterne

Oltre alle forme di giunzione riportate precedentemente, sono disponibili anche le giunzioni **esterne** (**outer join**), che possono restituire ennuple con valori **NULL**.

5. Questo operatore, anche se presente nello standard, non è implementato da tutti i DBMS commerciali.

La sintassi completa della giunzione diventa la seguente:

Giunzione ::= "," | [**CROSS** | **NATURAL**] [[**LEFT** | **RIGHT** | **FULL**] [**OUTER**]] **JOIN**

In particolare, solo per **NATURAL JOIN** e **JOIN**, se è presente anche una delle specifiche **LEFT** , **RIGHT** o **FULL** , allora viene effettuata la giunzione esterna come segue:

a) con **LEFT** si aggiungono al risultato della giunzione le ennuple della tabella a sinistra che non fanno parte della giunzione, con valori **NULL** per i campi della tabella a destra:

SELECT $*$
FROM R **NATURAL LEFT JOIN** S; $\equiv R \overleftarrow{\bowtie} S$

b) con **RIGHT** si aggiungono al risultato della giunzione le ennuple della tabella a destra che non fanno parte della giunzione, con valori **NULL** per i campi della tabella a sinistra:

SELECT $*$
FROM R **NATURAL RIGHT JOIN** S; $\equiv R \overrightarrow{\bowtie} S$

c) con **FULL** si combinano gli effetti di **LEFT** e **RIGHT** aggiungendo al risultato della giunzione le ennuple delle due tabelle che non fanno parte della giunzione, estese opportunamente con valori **NULL**.

SELECT $*$
FROM R **NATURAL FULL JOIN** S; $\equiv R \overleftrightarrow{\bowtie} S$

Esempio 6.1
Siano $R(A, B, C)$ e $S(A, D)$, due schemi di relazioni con attributi definiti sui domini: $dom(A) = \{a1, a2, a3\}$, $dom(B) = \{b1, b2, b3\}$, $dom(C) = \{c1, c2\}$ e $dom(D) = \{d1, d2, d4\}$. Si mostra il risultato della *giunzione naturale, esterna, esterna destra* ed *esterna sinistra* con le relazioni R e S:

R

A	B	C
a1	b1	c1
a1	b1	c2
a2	b1	c1
a3	b1	c1

S

A	D
a1	d1
a2	d2
a4	d4

$R \bowtie S =$

A	B	C	D
a1	b1	c1	d1
a1	b1	c2	d1
a2	b1	c1	d2

$R \overleftrightarrow{\bowtie} S =$

A	B	C	D
a1	b1	c1	d1
a1	b1	c2	d1
a2	b1	c1	d2
a3	b1	c1	null
a4	null	null	d4

$R \overleftarrow{\bowtie} S =$

A	B	C	D
a1	b1	c1	d1
a1	b1	c2	d1
a2	b1	c1	d2
a3	b1	c1	null

$R \overrightarrow{\bowtie} S =$

A	B	C	D
a1	b1	c1	d1
a1	b1	c2	d1
a2	b1	c1	d2
a4	null	null	d4

6.4.2 Uso dei valori NULL

I valori **NULL** sono stati oggetto di critiche e di opinioni contrastanti sia negli aspetti teorici, che in quelli pratici, in particolare per come gli operatori SQL li trattano, in alcuni casi in maniera non omogenea, e per le complicazioni che introducono nell'ottimizzazione delle query.

Di seguito diamo alcune indicazioni sul loro uso, ricordando che il loro trattamento può differire nei sistemi commerciali che adottano il linguaggio SQL.

Ridurre l'uso dei valori NULL

Esistono varie interpretazioni di **NULL**: ad esempio come valore presente nella realtà ma non conosciuto nella base di dati, oppure come valore non presente (e quindi, in pratica, non un valore).

Data la natura problematica del valore **NULL**, alcuni suggeriscono di limitarne l'uso. Ad esempio, nella traduzione di uno schema concettuale in uno schema relazionale, in presenza di un'associazione con diretta parziale, al posto del valore **NULL** possiamo usare una tabella ausiliaria. Ad esempio, se alcuni agenti, ma non tutti, hanno un supervisore, possiamo definire, al posto della chiave esterna Supervisore nella tabella Agenti, una tabella ausiliaria che contiene una ennupla per ogni agente che ha un supervisore, con le due chiavi esterne verso l'agente e il suo supervisore. Analogamente può essere fatto per gli attributi opzionali: se non tutti gli agenti hanno il numero di telefono, potremmo definire, al posto dell'attributo, una tabella ausiliaria dei telefoni con la chiave esterna per l'agente e il numero di telefono. Ovviamente questo approccio richiede la definizione di molte tabelle ausiliare, e la corrispondente esecuzione di un gran numero di giunzioni per la ricostruzione dell'intera entità.

Escludere i valori NULL dalle operazioni di ricerca

Se dobbiamo scrivere delle query per per alcuni attributi che possono avere valore **NULL**, si usino opportunamente i predicati **IS NULL** e la sua negazione per escludere le ennuple con valori **NULL** dalla query.

Ad esempio, se la relazione Agenti ha l'attributo Supervisore che può avere valori **NULL** e vogliamo trovare tutti gli agenti con un codice minore di quello del proprio supervisore, si scriva:

```
SELECT   *
FROM     Agenti
WHERE    Supervisore IS NOT NULL AND Codice < Supervisore;
```

Convertire i valori NULL in valori significativi

Talvolta si devono produrre dei dati di riepilogo, che devono essere usati da un utente finale. In questi casi si può usare l'operatore **COALESCE** nella clausola **SELECT** per tradurre gli eventuali valori **NULL**, anche generati da giunzioni esterne, in valori significativi.

Ad esempio, per produrre una lista completa di tutti gli agenti, specificando il nome del supervisore, quando presente, si può scrivere:

```
SELECT   A1.Nome, COALESCE (A2.Nome, 'NON HA SUPERVISORE')
FROM     Agenti A1
         LEFT JOIN Agenti A2 ON A1.Supervisore = A2.CodiceAgente
WHERE    A1.Zona = 'Pisa';
```

Si noti che il **LEFT JOIN** produce un risultato per tutti gli agenti, anche per quelli che hanno il valore Supervisore uguale a **NULL**.

6.5 Operatori per la modifica dei dati

I comandi per modificare i dati sono i seguenti:

INSERT INTO *Tabella* ["(" *Attributo* {, *Attributo* } ")"]
VALUES "(" *Valore* {,*Valore*}")" ;

per effettuare l'inserzione di un'ennupla in una tabella. La lista valori deve rispettare l'ordine degli attributi o, in loro assenza, l'ordine specificato nella definizione della tabella. Ad esempio, per inserire un nuovo cliente:

```
INSERT INTO Clienti
    VALUES ('A03', 'Rossi Mario', 'Roma', 10);
```

La modifica si effettua con il comando **UPDATE**:

UPDATE *Tabella*
SET *Attributo = Espr* {, *Attributo = Espr*}
WHERE *Condizione*;

Con un solo comando **UPDATE** si possono modificare uno o più attributi di un insieme di ennuple che soddisfano una condizione. Si noti che l'ennupla è la più piccola unità di inserzione e l'attributo la più piccola unità di aggiornamento.

Ad esempio, se vogliamo assegnare a tutti gli agenti con supervisore 's1' il nuovo supervisore 's2', possiamo scrivere:

UPDATE Agenti
SET Supervisore = 's2'
WHERE Supervisore = 's1';

Infine, il comando di cancellazione è il seguente:

DELETE FROM *Tabella*
WHERE *Condizione*;

Come **UPDATE**, il comando **DELETE** può coinvolgere una o più ennuple della tabella come determinato dalla clausola **WHERE**.

Ad esempio, se vogliamo cancellare tutti gli ordini precedenti al 2018, per cui l'agente corrispondente non esiste più nella relativa tabella, possiamo scrivere:

DELETE FROM Ordini
WHERE Data $<$ '01012018';

6.6 Il potere espressivo di SQL

Come già accennato, il linguaggio SQL non ha la potenza computazionale delle Macchine di Turing, e quindi non permette di scrivere espressioni equivalenti a tutte le funzioni calcolabili. Una tale limitazione non è un grosso problema pratico perché le interrogazioni che non si possono esprimere non sono molto comuni. Quando si presentano occorre usare l'SQL all'interno di un linguaggio di programmazione "Turing-equivalente".

La lista che segue mostra alcuni esempi di interrogazioni non esprimibili in SQL.

1. Le funzioni di aggregazione di solito disponibili non sono tutte quelle interessanti: se si vuole calcolare ad esempio la *moda*, o la *varianza*, di una colonna di valori, come altre importanti funzioni di tipo statistico, siamo impossibilitati a farlo. Infatti, se queste operazioni non sono fornite come primitive (né potrebbero esserlo tutte), non è possibile esprimerle come semplici espressioni sugli attributi (richiedono infatti uno o più "cicli" di calcolo sui valori di una colonna).
2. Le funzioni di aggregazione non si possono applicare ad altre funzioni. Ad esempio, non si può calcolare il totale di tutte le medie dell'ammontare degli ordini degli agenti, o il massimo di tutti i loro totali di vendita.[6]
3. La possibilità di creare "report" con la clausola **GROUP BY** è molto limitata rispetto a quelle offerte da un linguaggio di tipo generale. Ad esempio, non possiamo creare una tabella che contenga per certi intervalli di ammontare di ordini il totale delle vendite relative ad ogni intervallo.

6. Per risolvere questi problemi si può ricorrere all'espediente di definire tabelle derivate, come sarà mostrato nel prossimo capitolo.

4. Le condizioni permesse impediscono alcuni tipi di interrogazioni. Ad esempio, supponiamo di avere le seguenti tabelle per trattare documenti e le parole chiavi che ne descrivono il contenuto:

   Documenti(CodiceDocumento, TitoloDocumento, Autore)
   ParoleChiave(CodiceDocumento, ParolaChiave)

   È semplice trovare i titoli dei documenti con una certa parola chiave oppure che contengono tutte le parole chiave di un certo insieme K, ma non è possibile scrivere un'interrogazione per trovare tutti i documenti che contengono almeno k delle chiavi di K (problema tipico di *information retrieval*). Ad esempio, per trovare i documenti che contengono quattro chiavi di un insieme di sei chiavi occorre invece formulare $(6 \times 5)/2 = 15$ interrogazioni.
5. La *chiusura transitiva* di una relazione binaria rappresentata in forma di tabella non può essere calcolata (ma alcune varianti di SQL prevedono un operatore apposito). Ad esempio, supponiamo che non solo gli agenti ma anche i loro supervisori possano avere un supervisore. Con gli operatori visti non è possibile trovare tutti i supervisori, di qualsiasi livello, di un agente.

Come si vede da questi esempi, quindi, il linguaggio SQL non è sufficiente per esprimere interrogazioni di qualsiasi natura, ma è stato definito prevedendo gli operatori più utili per le interrogazioni più comuni.

Diverso è il caso del suo uso in un linguaggio di programmazione, come vedremo nel capitolo sullo sviluppo di applicazioni: lì è usato come un insieme di operatori "primitivi" (anche se non certo elementari) per operare sulla base di dati, mentre il linguaggio di programmazione utilizza i risultati delle interrogazioni per esprimere computazioni qualsiasi.

6.7 QBE: un esempio di linguaggio basato sulla grafica

Il linguaggio QBE (*Query by Example*), sviluppato alla IBM negli anni '70, è un esempio di linguaggio per il calcolo relazionale di domini, basato su un'interfaccia grafica. Le interrogazioni sono formulate usando la seguente rappresentazione grafica delle tabelle:

Clienti	CodiceCliente	Nome	Città	Sconto

Ordini	NumOrdine	CodiceCliente	CodiceAgente	Prodotto	Data	Ammontare

Questo modo di visualizzare le tabelle è quello originariamente previsto per i terminali a caratteri. Con i moderni sistemi provvisti di interfacce grafiche le cose cambiano molto e un esempio molto noto è Microsoft Access.

Per formulare un'interrogazione, l'utente riempie le righe con esempi di valori che desidera nel risultato e con variabili che assumono valori nei domini di certe colonne. Per distinguere costanti da variabili, queste ultime iniziano con il carattere "_". I campi che si vogliono vedere nel risultato sono quelli in cui si pongono i caratteri "P.". Un valore può essere preceduto da un operatore di confronto "$>$" o "$\geqslant$". Tutte le condizioni espresse nella stessa riga sono in **AND** e quelle su righe diverse sono in **OR**.

Ad esempio, per trovare il codice e la commissione degli agenti di Pisa, lo scheletro della tabella si riempie nel modo seguente:

Agenti	CodiceAgente	Nome	Zona	Supervisore	Commissione
	P._x		Pisa		P._y

A differenza dell'SQL, QBE restituisce sempre insiemi, mentre per mantenere nel risultato ennuple uguali si usa "P. all".

Condizioni più complesse si danno separatamente in una *condition box*, senza usare l'operatore di negazione "$\neg$", che va eliminato quindi usando le classiche equivalenze:

$\neg\neg C_1 \equiv C_1$
$\neg(C_1 \wedge C_2) \equiv \neg C_1 \vee \neg C_2$
$\neg(C_1 \vee C_2) \equiv \neg C_1 \wedge \neg C_2$

Si "spinge" poi la negazione verso l'interno di qualunque condizione, fino ad accostarla alle condizioni atomiche, che a loro volta sono in grado di assorbirla, grazie alle equivalenze:

$\neg x = y \equiv x \neq y$
$\neg x \neq y \equiv x = y$
$\neg x < y \equiv x \geqslant y$ ecc.

Ad esempio, per trovare il nome dei clienti che hanno fatto ordini con ammontare superiore a 10 000 euro, si pone:

Clienti	CodiceCliente	Nome	Città	Sconto
	_x	P.		

Ordini	NumOrdine	CodiceCliente	CodiceAgente	Prodotto	Data	Ammontare
		_y				$\geqslant$10 000

Conditions
_x = _y

6.8 Conclusioni

Sono state presentate le caratteristiche del linguaggio SQL per il recupero dei dati. SQL è ormai uno standard ed è offerto da tutti i sistemi, anche se continuano ad esistere differenze fra le versioni dei sistemi commerciali più diffusi. Molti sistemi prevedono interfacce grafiche per facilitarne l'uso interattivo, nello stile di Access, un prodotto della Microsoft molto popolare in ambiente Windows.

Nel prossimo capitolo si vedranno gli altri comandi dell'SQL per definire e amministrare basi di dati e poi come si possa usare in un linguaggio di programmazione per sviluppare applicazioni.

Per fare pratica con l'SQL è possibile utilizzare il sistema didattico gratuito multipiattaforma **JRS** (Java Relational System)[7]. Oltre a permettere di eseguire comandi SQL, l'applicazione permette di vedere gli alberi logici e fisici e i risultati delle fasi di ottimizzazione delle interrogazioni.

Esercizi

1. Dare un'espressione **SELECT** per stabilire se i valori di A in una relazione con schema R(A, B, C) siano tutti diversi. L'espressione deve essere diversa da **SELECT** A **FROM** R.
2. Si ricorda che il predicato **IN** è equivalente a =**ANY**. Spiegare perché il predicato **NOT IN** non è equivalente a <>**ANY** ma a <>**ALL**.
3. È importante sapere quando una **SELECT** ritorna una tabella con righe diverse per evitare di usare inutilmente la clausola **DISTINCT**, che comporta un costo addizionale per l'esecuzione dell'interrogazione (perché?).

   a) È vero che con una **SELECT** con una tabella nella parte **FROM**, e senza **GROUP BY**, non vi saranno righe duplicate nel risultato se gli attributi della parte **SELECT** sono una superchiave della tabella?

   b) È vero che con una **SELECT** con più tabelle nella parte **FROM**, e senza **GROUP BY**, non vi saranno righe duplicate nel risultato se gli attributi della parte **SELECT** sono una superchiave di ogni tabella?

   c) È vero che nessuna interrogazione con un **GROUP BY** può avere duplicati nel risultato?

7. **JRS** è stato sviluppato in Java per il supporto allo studio dei Sistemi di Gestione di Basi di Dati Relazionali presso il Dipartimento di Informatica dell'Università di Pisa dagli autori del presente testo e dagli studenti Lorenzo Brandimarte, Leonardo Candela, Giovanna Colucci, Patrizia Dedato, Stefano Fantechi, Stefano Dinelli, Martina Filippeschi, Simone Marchi, Cinzia Partigliani, Marco Sbaffi e Ciro Valisena. Può essere scaricato, insieme con la documentazione, dal sito del libro, http://fondamentidibasididati.it