Interrogazione della base di dati

SQL è un linguaggio di definizione e di manipolazione dei dati. In quanto linguaggio di manipolazione, SQL permette di **selezionare dati di interesse** dalla base e di aggiornarne il contenuto. In questa sezione vedremo la parte più interessante e usata del linguaggio SQL, vale a dire le interrogazioni per selezionare i dati. Tali interrogazioni sono usate sia nei costrutti SQL di definizione dei dati che in quelli di aggiornamento della base di dati. E' bene quindi iniziare lo studio di SQL dalle interrogazioni di selezione di dati.

SQL è un **linguaggio dichiarativo**: esso permette di specificare *cosa* cercare senza dire *come*. Quando una interrogazione (*query*) viene eseguita dall'elaboratore di interrogazioni (*query processor*), essa viene tradotta in un **linguaggio procedurale interno** al sistema il quale permette di specificare come accedere ai dati. Esistono in generale più possibili traduzioni di una interrogazione SQL nel linguaggio procedurale. Il compito dell'ottimizzatore delle interrogazioni (*query optimizer*) è scegliere il piano di esecuzione più efficiente.

E' quindi bene che l'utente si concentri sull'obiettivo della propria ricerca, cercando di scrivere interrogazioni leggibili e modificabili, delegando tutti gli aspetti procedurali e di efficienza al DBMS. Il vantaggio di poter scrivere interrogazioni ad alto livello (pensando allo schema logico e non alla sua realizzazione fisica) è una conseguenza dell'**indipendenza fisica dei dati** dei DBMS relazionali.

Una interrogazione SQL viene eseguita su una base di dati, quindi su un insieme di tabelle collegate tra loro mediante il meccanismo delle chiavi esterne. Il risultato di una interrogazione è una tabella. E' bene chiarire subito che esistono due differenze importanti tra relazioni del modello relazionale e tabelle SQL. Una tabella SQL può contenere **righe duplicate** e **colonne omonime**. Le colonne vengono identificate univocamente dalla loro posizione. Questo è vero per le **tabelle risultato** dalle interrogazioni (vedremo degli esempi). Le **tabelle di base**, cioè quelle che fanno parte della base di dati, non possono avere colonne omonime. Inoltre, se esiste una chiave per la tabella di base (questa è la norma), non possono esistere righe duplicate nella tabella. Dunque tabelle di base dotate di una chiave corrispondono a relazioni del modello relazionale. Come vedremo, esiste un modo per far si che una tabella non contenga duplicati. La presenza di duplicati nelle tabelle risultato è motivata come segue:

- in certe occasioni rimuovere le righe duplicare corrisponde ad una perdita di informazione. Si pensi ad esempio ad una tabella che contiene una sola colonna con dei dati numerici. Supponiamo di voler calcolare la media sui dati di questa colonna. L'eliminazione dei duplicati falserebbe il risultato;
- l'eliminazione dei duplicati dal risultato è in generale una operazione costosa (un modo per implementarla consiste nell'ordinare le righe e poi eliminare i duplicati).

Introdurremo SQL *by example*, cioè mostrando esempi sempre più ricchi e complessi di interrogazione.

Interrogazioni di base

Consideriamo la seguente tabella teatro:

teatro

nome	**città**	**email**
CSS	Udine	css@gmail.com

Litta	Milano	litta@gmail.com
Piccolo	Milano	piccolo@gmail.com
Eliseo	Roma	eliseo@gmail.com

L'interrogazione più semplice che si possa scrivere è la seguente:

```
select *
from teatro
```

Il risultato è l'intera tabella teatro. La prima riga dell'interrogazione è detta **clausola select** e serve per selezionare le colonne della tabella che ci interessano. L'operatore `*` permette di selezionare tutte le colonne. La seconda riga dell'interrogazione è detta **clausola from** e serve per indicare quali tabelle usare. La clausola select e quella from sono obbligatorie in una interrogazione.

Se siamo interessati solo al nome e all'email dei teatri, possiamo selezionarli in questo modo:

```
select nome, email
from teatro
```

Il risultato è la seguente tabella:

nome	email
CSS	css@gmail.com
Litta	litta@gmail.com
Piccolo	piccolo@gmail.com
Eliseo	eliseo@gmail.com

Per chiarire la differenza tra relazione del modello realzionale e tabella SQL, vediamo una semplice interrogazione che genera una tabella con due colonne con lo stesso nome:

```
select nome, nome
from teatro
```

Il risultato è la seguente tabella:

nome	nome
CSS	CSS
Litta	Litta
Piccolo	Piccolo
Eliseo	Eliseo

Inoltre, mostriamo una semplice interrogazione che genera una tabella con due righe uguali:

```
select città
from teatro
```

Il risultato è la seguente tabella:

città
Udine
Milano
Milano
Roma

E' possibile specificare la parola chiave `distinct` dopo la parola chiave `select` per eliminare i duplicati.

Introduciamo ora la clausola `where`:

```
select nome
from teatro
where città = 'Milano'
```

Il risultato è la seguente tabella:

nome
Litta
Piccolo

La clausola `where` definisce un **predicato sui valori degli attributi** delle tabelle selezionate. Le righe che soddisfano il predicato, cioè per le quali in predicato è vero, vengono inserite nella tabella risultato.

Un predicato è formato combinando predicati atomici con gli operatori Booleani `and`, `or` e `not`. Il `not` ha precedenza sugli altri due operatori ma non è stata definita una precedenza tra `or` e `and`. Un predicato atomico è ottenuto confrontando due espressioni sui valori degli attributi mediante i seguenti operatori di confronto: `=, <>, <, >, <=, >=`. Esempi di espressioni sono un attributo e una costante. Le costanti di tipo stringa, tempo e data si scrivono tra apici, come nell'esempio di sopra. Non servono gli apici per i numeri. Inoltre, l'operatore `like` permette un confronto con stringhe che contengono i valori speciali `_` (un carattere qualsiasi) e `%` (una sequenza arbitraria, eventualmente vuota, di caratteri). E' possibile confrontare il valore di un'espressione con il valore nullo con i predicati `is null` e `is not null`.

Facciamo qualche esempio sulla seguente tabella:

dipendente

cf	**nome**	**cognome**	**dataDiNascita**	**stipendio**
ELSDLL72	Elisa	D'Allarche	1972-04-29	2500
FRNDPP76	Fernanda	D'Ippoliti	1976-03-11	2100
MRCDLL70	Marco	Dall'Aglio	1970-01-09	2700

Di seguito scriveremo prima la query in linguaggio naturale, poi la sua traduzione in SQL e infine il suo risultato.

Il nome, il cognome e lo stipendio dei dipendenti con uno stipendio di almeno 2500 Euro.

```
select nome, cognome, stipendio
from dipendenti
where stipendio >= 2500
```

nome	**cognome**	**stipendio**
Elisa	D'Allarche	2500
Marco	Dall'Aglio	2700

Il nome e il cognome dei dipendenti nati dopo il 1975.

```
select nome, cognome
from dipendenti
where dataDiNascita > '1975-12-31'
```

nome	**cognome**
Fernanda	D'Ippoliti

Il nome e il cognome dei dipendenti con il nome che finisce con la lettera 'a' e con uno stipendio di almeno 2500 Euro.

```
select nome, cognome
from dipendente
where (stipendio >= 2500) and (nome like '%a')
```

nome	cognome
Elisa	D'Allarche

Il codice fiscale e lo stipendio annuale di Elisa D'Allarche.

```
select cf as codiceFiscale, stipendio * 12 as stipendioAnnuale
from dipendente
where nome = 'Elisa' and cognome = 'D\'Allarche'
```

codiceFiscale	stipendioAnnuale
ELSDLL72	30000

Alcune osservazioni sull'ultima query: nelle espressioni possiamo usare le 4 operazioni *, /, +, -. Possiamo inoltre rinominare le colonne. Infine occorre far precedere il carattere apice (') da una barra (\) all'interno delle stringhe come in 'D\'Allarche'.

A partire da SQL-2, la logica dei predicati in SQL è a **tre valori**: vero (V), falso (F) e sconosciuto (*unknown*, indicato con U). Le regole per usare i tre valori di verità sono le seguenti:

- ogni confronto in cui almeno una componente ha valore sconosciuto ha come risultato il valore sconosciuto. Fanno eccezione i predicati `is null` e `is not null`, il cui valore è sempre o vero o falso, anche se il valore di confronto è sconosciuto;
- gli operabori Booleani vengono estesi al valore U nel seguente modo:

  not U = U

  V and U = U, F and U = F, U and U = U

  V or U = V, F or U = U, U or U = U

  In sostanza, not A è vero se e solo se A è falso, A and B è vero se e solo se entrambi A e B sono veri e A or B è vero se e solo almeno uno tra A e B è vero;
- una tupla soddisfa un predicato se il valore del predicato per quella tupla è *vero*.

Ad esempio, la query:

```
select *
from dipendente
where (età >= 30)
```

seleziona i dipendenti la cui età *è nota* (non nulla) e il suo valore è maggiore o uguale a 30. Inoltre:

```
select *
from dipendente
where (età < 30) or (età >= 30)
```

seleziona i dipendenti la cui età *è nota*, qualsiasi sia in suo valore. Si noti che il risultato non contiene tutti i dipendenti, ma, *giustamente*, vengono esclusi quelli che hanno valore sconosciuto per l'attributo età. In realtà questo risultato sembra contrario all'intuizione, in quanto *noi sappiamo* che ogni persona ha un'età e qualsiasi valore abbia è sicuramente un valore che soddisfa il predicato dato. L'intuizione è però fuorviante in tal caso in quanto assume che l'attributo età abbia un valore, anche se sconosciuto. Invece, il valore nullo per un attributo significa che: (i) il valore dell'attributo non esiste, oppure (ii) il valore dell'attributo esiste ma non è noto, oppure

(iii) non è noto se il valore dell'attributo esista. L'interprete SQL non assume nessuno di questi tre casi. Per capire meglio, consideriamo la stessa query riferita all'attributo opzionale email:

```
select *
from dipendente
where (email like '%@gmail.com') or not (email like '%@gmail.com')
```

Un impiegato senza indirizzo di posta elettronica viene giustamente escluso dal risultato, in quanto è falso che il suo indirizzo appartiene al dominio gmail.com ed è falso pure che il suo indirizzo non appartiene a tale dominio. Semplicemente, il suo indirizzo non esiste.

Interrogazioni di congiunzione (join)

Finora abbiamo visto interrogazioni su una singola tabella. Nel modello relazionale, ogni concetto indipendente viene rappresentato con una tabella e le corrispondenze tra i dati in tabelle diverse vengono realizzate mediante il confronto tra i valori degli attributi (generalmente tra una chiave esterna e una chiave primaria). In particolare, le interrogazioni di congiunzione (più comunemente note con l'espressione inglese *join*) sfruttano il **modello di corrispondenza basato su valori**, tipico del modello relazionale, per recuperare dati correlati ma distribuiti in più di una tabella. Si considerino le seguenti tre tabelle:

teatro

nome	**città**	**email**
CSS	Udine	css@gmail.com
Litta	Milano	litta@gmail.com
Eliseo	Roma	eliseo@gmail.com

dipendente

cf	**nome**	**cognome**	**dataDiNascita**	**stipendio**
ELSDLL72	Elisa	D'Allarche	29/04/1972	2500
FRNDPP76	Fernanda	D'Ippoliti	11/03/1976	2100
MRCDLL70	Marco	Dall'Aglio	09/01/1970	2700

lavoro

teatro	**dipendente**	**ruolo**
CSS	ELSDLL72	relazioni
Litta	FRNDPP76	finanza
Eliseo	FRNDPP76	controllo
Eliseo	MRCDLL70	direzione

Supponiamo di voler recuperare il nome e cognome di tutti i dipendenti del teatro CSS. Queste informazioni sono distribuite in due tabelle: dipendente e lavoro. Occorre dunque *congiungere* queste due tabelle mediante la chiave esterna dipendente della tabella lavoro:

```
select nome, cognome
from lavoro, dipendente
where (teatro = 'CSS') and (dipendente = cf)
```

Per aumentare la leggibilità, possiamo riscrivere la query facendo precedere ad ogni attributo il nome della corrispondente tabella:

```
select dipendente.nome, dipendente.cognome
from lavoro, dipendente
```

```
where (lavoro.teatro = 'CSS') and (lavoro.dipendente = dipendente.cf)
```

Similmente, possiamo recuperare tutte le città in cui lavora il dipendente identificato dal codice FRNDPP76 nel seguente modo:

```
select teatro.città
from lavoro, teatro
where (lavoro.dipendente = 'FRNDPP76') and (lavoro.teatro = teatro.nome)
```

L'aspetto procedurale di interrogazioni che coinvolgono più tabelle è utile per meglio capirne il significato. Consideriamo l'ultimo join tra lavoro e teatro. L'esecuzione procede in questo modo: la tabella **prodotto cartesiano** delle tabelle lavoro e teatro viene calcolata. Tale tabella contiene righe formate giustapponendo ogni riga di lavoro ad ogni riga di teatro:

lavoro x teatro

teatro	dipendente	ruolo	nome	città	email
CSS	ELSDLL72	relazioni	CSS	Udine	css@gmail.com
CSS	ELSDLL72	relazioni	Litta	Milano	litta@gmail.com
CSS	ELSDLL72	relazioni	Eliseo	Roma	eliseo@gmail.com
Litta	FRNDPP76	finanza	CSS	Udine	css@gmail.com
Litta	FRNDPP76	finanza	Litta	Milano	litta@gmail.com
Litta	FRNDPP76	finanza	Eliseo	Roma	eliseo@gmail.com
Eliseo	FRNDPP76	controllo	CSS	Udine	css@gmail.com
Eliseo	FRNDPP76	controllo	Litta	Milano	litta@gmail.com
Eliseo	FRNDPP76	controllo	Eliseo	Roma	eliseo@gmail.com
Eliseo	MRCDLL70	direzione	CSS	Udine	css@gmail.com
Eliseo	MRCDLL70	direzione	Litta	Milano	litta@gmail.com
Eliseo	MRCDLL70	direzione	Eliseo	Roma	eliseo@gmail.com

Ad ogni riga della tabella prodotto viene applicato il predicato della clausola `where` e le righe che lo soddisfano vengono selezionate. In particolare la condizione di join lavoro.teatro = teatro.nome permette di associare righe di lavoro a corrispondenti righe di teatro. Tali righe sono colorate in blu nella tabella di sopra. Di queste, solo le righe con lavoro.dipendente = 'FRNDPP76' vengono trattenute:

teatro	dipendente	ruolo	nome	città	email
Litta	FRNDPP76	finanza	Litta	Milano	litta@gmail.com
Eliseo	FRNDPP76	controllo	Eliseo	Roma	eliseo@gmail.com

Infine viene proiettata solo la colonna città specificata nella clausola `select`:

città
Milano
Roma

Una visione operazionale alternativa del join è la seguente. Supponiamo di dover fare un join tra due tabelle R e S con condizione di join theta. Se dovessimo programmare questa operazione, dovremmo scrivere **due cicli for annidati**. Il ciclo esterno scandisce le righe di R. Per ogni riga r di R, il ciclo interno scandisce le righe s di S e verifica se la riga congiunta rs soddisfa theta. In tal caso la riga viene selezionata.

E' possibile fare il join di più di due tabelle. Supponiamo di voler selezionare il nome, il cognome e la città di lavoro di ogni dipendente. Queste informazioni sono sparse su due tabelle (dipedente e teatro) che devono essere collegate attraverso una terza tabella, lavoro. Dunque tre tabelle sono coinvolte nel join:

```
select dipendente.nome, dipendente.cognome, teatro.città
from lavoro, dipendente, teatro
where (lavoro.dipendente = dipendente.cf) and (lavoro.teatro = teatro.nome)
```

Il risultato è la seguente tabella:

nome	cognome	città
Elisa	D'Allarche	Udine
Fernanda	D'Ippoliti	Roma
Fernanda	D'Ippoliti	Milano
Marco	Dall'Aglio	Roma

Abbiamo visto che tecnica di denominazione degli attributi facendoli precedere la nome della tabella è utile per aumentare la leggibilità dell'interrogazione. Inoltre, questa tecnica è indispensabile per individuare un attributo senza ambiguità se vi sono attributi con lo stesso nome in tabelle diverse. Si noti che l'ambiguità rimane se le tabelle hanno lo stesso nome, cioè sono istanze diverse della stessa tabella. E' infatti possibile concatenare nella clausola `from` più istanze della stessa tabella. Per eliminare l'ambiguità in questo caso è possibile rinominare le istanze della stessa tabella con il costrutto `as`.

Vediamo un esempio. Supponiamo di aggiungere alla tabella dipendente un attributo capo che contenga il codice fiscale del capo del dipendente. In particolare, capo è una chiave esterna e si riferisce alla chiave primaria cf della tabella dipendente stessa. Supponiamo che un dipendente possa avere o meno un capo e che un capo possa dirigere zero o più dipendenti:

dipendente

cf	**nome**	**cognome**	**capo**
ELSDLL72	Elisa	D'Allarche	NULL
FRNDPP76	Fernanda	D'Ippoliti	ELSDLL72
MRCDLL70	Marco	Dall'Aglio	ELSDLL72

Cerchiamo il nome e cognome dei capi di tutti i dipendenti:

```
select d1.nome, d1.cognome, d2.nome as nomeCapo, d2.cognome as cognomeCapo
from dipendente as d1, dipendente as d2
where d1.capo = d2.cf
```

Abbiamo usato la parola chiave `as`, che può essere omessa, per rinominare sia gli attributi che le tabelle. Il risultato del join è il seguente:

dipendente

cf	nome	cognome	capo	cf	nome	cognome	capo
ELSDLL72	Elisa	D'Allarche	NULL	ELSDLL72	Elisa	D'Allarche	NULL
FRNDPP76	Fernanda	D'Ippoliti	ELSDLL72	FRNDPP76	Fernanda	D'Ippoliti	ELSDLL72
MRCDLL70	Marco	Dall'Aglio	ELSDLL72	MRCDLL70	Marco	Dall'Aglio	ELSDLL72

Il risultato dell'interrogazione è il seguente:

nome	cognome	nomeCapo	cognomeCapo
Fernanda	D'Ippoliti	Elisa	D'Allarche
Marco	Dall'Aglio	Elisa	D'Allarche

In SQL-2 è stata definita una sintassi particolare per i join e sono stati distinti 4 diversi tipi di join. La sintassi del join è la seguente:

```
select d1.nome, d1.cognome, d2.nome as nomeCapo, d2.cognome as cognomeCapo
from dipendente d1 join dipendente d2 on d1.capo = d2.cf
```

La tabella risultato dell'operazione di join è detta **tabella congiunta** (*joined table*). Le tabelle argomento dell'operazione di join possono essere lore stesse tabelle congiunte. Ad esempio, il seguente join recupera i dipendenti con stipendio inferiore a 1000 e i capi dei loro capi:

```
select d1.nome, d1.cognome, d3.nome as nomeSuperCapo, d3.cognome as cognomeSuperCapo
from (dipendente d1 join dipendente d2 on d1.capo = d2.cf)
      join dipendente d3 on d2.capo = d3.cf
where d1.stipendio < 1000
```

Un vantaggio di questa sintassi è che la condizione di join viene isolata dalle altre condizioni di selezione delle righe, che possono essere aggiunte mediante la clausola `where`. Precisamente, una **condizione di join** tra due tabelle R e S è una congiunzione (secondo l'operatore Booleano `and`) di condizioni atomiche di tipo A theta B, dove A è un attributo di R, B è un attributo di S, e theta è un operatore di confronto. Di solito, ma non necessariamente, A è una chiave primaria, B è una chiave esterna riferita ad A e theta è l'operatore di uguaglianza.

Un secondo vantaggio di questa sintassi per il join è che ci permette di distinguere diverse forme di join. Si noti che nell'ultima interrogazione fatta Elisa non viene selezionato nel risultato come dipendente. Questo perchè ella non ha un capo e dunque la sua riga nell'istanza d1 di dipendente non incontra nessuna riga nell'istanza d2 di dipendente che verifichi la condizione di join d1.capo = d2.cf e dunque non viene inserita nel risultato. Questa situazione è spiacevole perchè non ci informa che Elisa non ha capi. E' possibile risolvere questo problema usando un **left join**:

```
select d1.nome, d1.cognome, d2.nome as nomeCapo, d2.cognome as cognomeCapo
from dipendente d1 left join dipendente d2 on d1.capo = d2.cf
```

che produce il seguente risultato:

nome	cognome	nomeCapo	cognomeCapo
Elisa	D'Allarche	NULL	NULL
Fernanda	D'Ippoliti	Elisa	D'Allarche
Marco	Dall'Aglio	Elisa	D'Allarche

Similmente, i dipendenti che non sono a capo di alcun dipendente (Fernanda e Marco) non fanno parte del risultato come capi. E' possibile risolvere questo problema usando un **right join**:

```
select d1.nome, d1.cognome, d2.nome as nomeCapo, d2.cognome as cognomeCapo
from dipendente d1 right join dipendente d2 on d1.capo = d2.cf
```

nome	cognome	nomeCapo	cognomeCapo
Fernanda	D'Ippoliti	Elisa	D'Allarche
Marco	Dall'Aglio	Elisa	D'Allarche
NULL	NULL	Fernanda	D'Ippoliti
NULL	NULL	Marco	Dall'Aglio

Si noti che lo stesso risultato si avrebbe con il seguente left join a tabelle invertite:

```
select d1.nome, d1.cognome, d2.nome as nomeCapo, d2.cognome as cognomeCapo
from dipendente d2 left join dipendente d1 on d1.capo = d2.cf
```

Un **full join** unisce un left join e un right join:

```
select d1.nome, d1.cognome, d2.nome as nomeCapo, d2.cognome as cognomeCapo
from dipendente d1 full join dipendente d2 on d1.capo = d2.cf
```


nome	cognome	nomeCapo	cognomeCapo
Elisa	D'Allarche	NULL	NULL
Fernanda	D'Ippoliti	Elisa	D'Allarche
Marco	Dall'Aglio	Elisa	D'Allarche
NULL	NULL	Fernanda	D'Ippoliti
NULL	NULL	Marco	Dall'Aglio

Ordinamento delle tuple

Una tabella contiene tuple non ordinate. E' possibile ordinare le tuple secondo determinati criterio con il costrutto `order by`. Ad esempio, la seguente interrogazione ordina le tuple della tabella dipendente in ordine crescente per nome e cognome, e in ordine decrescente per stipendio:

```
select *
from dipendente
order by nome, cognome, stipendio desc
```

Le tuple vengono ordinate in ordine crescente secondo i valori dell'attributo nome. Due dipendenti con lo stesso nome vengono ordinati secondo il loro cognome in senso crescente. Infine due dipendenti con lo stesso nome e cognome vengono ordinati secondo lo stipendio in ordine discendente.

Operatori aggregati

Un operatore aggregato è una funzione che si applica ad un insieme di tuple di una tabella e ha come risultato un valore atomico. Lo standard SQL prevede i seguenti operatori aggregati:

count

Questo operatore serve per contare le tuple di una tabella. Può essere usato nei seguenti tre modi:

```
select count(*)
from dipendente
```

Il risultato è il numero di tuple della tabella dipendente.

```
select count(all nome)
from dipendente
```

Il risultato è il numero di valori **non nulli** dell'attributo nome della tabella dipendente. La parola chiave `all` può essere omessa ottenendo lo stesso risutato.

```
select count(distinct nome)
from dipendente
```

Il risultato è il numero di valori **distinti e non nulli** dell'attributo nome della tabella dipendente.

min e **max**

Restituiscono rispettivamente il minimo e il massimo di una espressione valutata sulle tuple di una tabella. L'espressione deve restituire valori su cui è definito un ordinamento (numeri, stringhe, istanti temporali). Ad esempio, la seguente interrogazione restituisce lo stipendio massimo di un dipendente:

```
select max(stipendio)
from dipendente
```

La seguente interrogazione restituisce la data di nascita più remota di un dipendente:

```
select min(dataDiNascita)
from dipendente
```

sum e **avg**

Restituiscono rispettivamente la somma e la media di una espressione valutata sulle tuple di una tabella. L'espressione deve restituire valori su cui è definita la somma (numeri). Ad esempio, la seguente interrogazione restituisce la somma degli stipendi dei dipendenti:

```
select sum(stipendio)
from dipendente
```

Le seguenti due interrogazioni restituiscono entrambe la media degli stipendi dei dipendenti:

```
select avg(stipendio)
from dipendente

select sum(stipendio)/count(stipendio)
from dipendente
```

E' possibile usare la parole chiave `distinct` per rimuovere i valori duplicati dal conteggio effettuato dagli operatori `sum` e `avg`.

Si noti che non è possibile mescolare nella clausola `select` espressioni aggregate, cioè espressioni che restituiscono un valore per un insieme di tuple, e espressioni di tupla, cioè espressioni che restituiscono un valore per ogni singola tuple. Dunque la sequente query è scorretta:

```
select nome, max(stipendio)
from dipendente
```

Raggruppamenti

Gli esempi di operatori aggregati visti fin ora operano sull'insieme di *tutte* le righe di una tabella. La clausola `group by` permette di partizionare le righe di una tabella in sottoinsiemi e applicare gli operatori aggregati ai singoli sottoinsiemi. Tale clausola ha come argomento un insieme di attributi e raggruppa le righe che posseggono lo stesso valore per gli attributi dell'insieme argomento. Nella clausola `select` posso usare operatori aggregati e attributi ma quest'ultimi devono essere inclusi nella clausola `group by`. Si consideri la seguente tabella lavoro:

lavoro

teatro	**dipendente**	**ruolo**
CSS	ELSDLL72	finanza
CSS	ABCDEF74	direzione
Litta	FRNDPP76	finanza
Litta	GHILMN77	finanza
Eliseo	FRNDPP76	controllo
Eliseo	MRCDLL70	direzione

Supponiamo di voler calcolare il numero di dipendenti per ogni ruolo. Possiamo scrivere la seguente interrogazione:

```
select ruolo, count(*) as numero
from lavoro
group by ruolo
```

La query raggruppa le tuple di lavoro secondo i valori dell'attributo ruolo. Vengono creati tre insiemi corrispondenti ai valori di finanza (3 righe), direzione (2 righe) e controllo (1 riga). Su ogni insieme viene selezionato il valore di ruolo e il numero di righe mediante l'operatore aggretato `count`. Si noti che, per ogni sottoinsieme identificato, esiste un unico valore di ruolo, che quindi può essere resituito. Non è così per gli attributi teatro e dipendente, che non possono essere usati nella clausola `select` in questo caso. Il risultato è la seguente tabella:

ruolo	**numero**

finanza	3
direzione	2
controllo	1

Se vogliamo calcolare il numero di dipendenti per ogni ruolo del teatro CSS possiamo scrivere la seguente interrogazione:

```
select ruolo, count(*) as numero
from lavoro
where teatro = 'CSS'
group by ruolo
```

ruolo	numero
finanza	1
direzione	1

Si noti che la tabella su cui opera la clausola `group by` può essere il risultato di una query, ad esempio una tabella risultato di un join come nel seguente esempio:

```
select ruolo, count(*), sum(stipendio)
from lavoro, dipendente
where lavoro.dipendente = dipendente.cf
group by ruolo
```

Se vogliamo selezionare solo i gruppi che soddisfano un certo predicato possiamo usare la clausola `having`. Un predicato sui gruppi può usare operatori aggregati e attributi della clausola che compaiono in `group by`. Ad esempio, se vogliamo calcolare il numero di dipendenti per ogni ruolo selezionando solo i gruppi di almeno due dipendenti possiamo scrivere nel seguente modo:

```
select ruolo, count(*) as numero
from lavoro
group by ruolo
having count(*) >= 2
```

Si badi bene alla differenza tra le clausole `where` e `having`: la clausola `where` contiene **predicati di tupla** e serve per filtrare le tuple delle tabelle, la clausola `having` contiene **predicati di gruppo** e serve per filtrare i gruppi ottenuti mediante la clausola `group by`. Concettualmente, prima si applica il filtro `where` e poi quello `having`.

Le clausole `select`, `from`, `where`, `group by`, `having`, `order by` debbono essere usate in quest'ordine e sono tutte le clausole che si possono usare in SQL. Solo le clausole `select` e `from` sono obbligatorie. La clausola `having` prevede l'uso di `group by`. Come esempio riepilogativo selezioniamo il numero di dipendenti per ogni ruolo del teatro CSS per gruppi maggiori di 1 ordinando il risultato per ruolo:

```
select ruolo, count(*) as numero
from lavoro
where teatro = 'CSS'
group by ruolo
having count(*) > 1
order by ruolo
```

Concettualmente la query viene eseguita nel seguente modo:

1. vengono selezionate le righe della tabella lavoro (`from`);
2. vengono filtrate le righe che corrispondono al teatro CSS (`where`);
3. le righe vengono suddivise in gruppi che hanno lo stesso valore di ruolo (`group by`);
4. vengono filtrati i gruppi di cardinalità maggiore di 1 (`having`);
5. i gruppi risultati vengono ordinati lessicograficamente secondo il valore di ruolo (`order by`);
6. per ogni gruppo viene selezionato il valore di ruolo e calcolato il numero di righe (`select`);

Si noti che la clausola `select` viene scritta per prima ma concettualmente eseguita per ultima. Inoltre, non è detto che la sequenza concettuale descritta corrisponda alla sequenza operazionale implementata dal DBMS il quale, per ragioni di efficienza, può eseguire la query in un ordine differente.

Operatori insiemistici

SQL permette di fare l'unione (`union`), l'intersezione (`intersect`) e la differenza (`except`) tra insiemi di righe di tabelle. Questi operatori richiedono che gli schemi su cui operano abbiano lo stesso numero di attributi e che i loro domini siano compatibili. Per default, questi operatori eliminano i duplicati dal risultato, restituendo quindi un insieme nel senso matematico del termine. E' possibile recuperare i duplicati facendo seguire la parola chiave `all` al nome dell'operatore. Vediamo alcuni esempi sulle seguenti tabelle:

attore

nome	**dataDiNascita**	**luogoDiNascita**
Elisa Bottega	1972-04-29	Treviso
Lorenzo Vignando	1970-05-29	Venezia
Barbara Altissimo	1982-03-01	Torino

autore

nome	**dataDiNascita**	**luogoDiNascita**
Vittorio Cortellessa	1969-11-29	Napoli
Lorenzo Vignando	1970-05-29	Venezia
Antinisca Di Marco	1972-08-01	L'Aquila

Selezioniamo tutte le persone che sono attori oppure autori:

```
select nome
from attore
   union
select nome
from autore
```

nome
Elisa Bottega
Lorenzo Vignando
Barbara Altissimo
Vittorio Cortellessa
Antinisca Di Marco

Si noti che l'autore e attore Lorenzo Vignando compare solo una volta nel risultato.

Selezioniamo tutte le persone che sono attori e autori:

```
select nome
from attore
   intersect
select nome
from autore
```

nome
Lorenzo Vignando

Selezioniamo tutte le persone che sono attori ma non sono autori:

```
select nome
from attore
   except
select nome
from autore
```

nome
Elisa Bottega
Barbara Altissimo

Interrogazioni nidificate

Una **interrogazione nidificata** è una interrogazione che contiene un'altra interrogazione. SQL non pone limiti al livello di annidamento ma solitamente più di due annidamenti rendono incomprensibile una query ad un umano (una macchina, invece, non ha problemi di comprensione in questo caso).

E' possibile nidificare una interrogazione nella clausola `where` (vedremo un'altra forma di annidamento parlando delle viste). In particolare una espressione può essere confrontata mediante gli usuali operatori di confronto con una interrogazione. L'operatore di confronto è seguito dalla parola chiave `any` oppure `all`. Nel primo caso, il predicato è vero se ha successo il confronto (secondo l'operatore usato) tra il valore dell'espressione e **almeno uno** tra i valori risultato della query. Nel secondo caso, il predicato è vero se ha successo il confronto (secondo l'operatore usato) tra il valore dell'espressione e **tutti** i valori risultato della query. L'espressione e il risultato dell'interrogazione devono essere compatibili, cioè avere lo stesso numero di attributi e gli attributi devono evere domini compatibili.

Vediamo alcuni esempi. Consideriamo le tabelle attore a autore usate nelle interrogazioni insiemistiche. La seguente interrogazione seleziona tutti gli attori che sono anche autori. Si noti che abbiamo già scritto la stessa query usando l'operatore `intersect`:

```
select nome
from attore
where nome = any (select nome
                  from autore)
```

Si noti che la query interna è scritta tra parentesi tonde. La seguente interrogazione seleziona tutti gli attori che non sono autori. Si noti che abbiamo già scritto la stessa query usando l'operatore `except`:

```
select nome
from attore
where nome <> all (select nome
                   from autore)
```

Selezionare tutti gli attori o autori è invece possibile solo usando l'operatore `union`. Le combinazioni `= any` e `<> all` sono molto frequenti e hanno un nome: nel primo caso posso usare la parola chiave `in` e nel secondo `not in`. Ad esempio:

```
select nome
from attore
where nome not in (select nome
                   from autore)
```

Se il nome e il cognome di un attore fosse stato specificanto usando due attributi invece che uno soltato, avremmo potuto usare il costruttore di tupla `(nome, cognome)` in questo modo:

```
select nome, cognome
from attore
where (nome, cognome) not in (select nome, cognome
                              from autore)
```

Il nome del dipendente con lo stipendio massimo può essere estratto nei seguenti due modi:

```
select nome, stipendio
from dipendente
where stipendio >= all (select stipendio
                        from dipendente)

select nome, stipendio
from dipendente
where stipendio = (select max(stipendio)
                   from dipendente)
```

Si noti che nel secondo caso la parola chiave `all` o `any` può essere omessa in quanto la query interna restituisce un solo elemento.

L'interrogazione interna può essere sostituita con un **insieme esplicito**. L'interrogazione seguente recupera tutti i nomi dei dipendenti in un certo insieme esplicito:

```
select nome
from dipendente
where nome in ("Elisa Bottega", "Lorenzo Vignando", "Barbara Altissimo")
```

Le interrogazioni nidificate viste fin ora possono essere risolte indipendentemente dall'interrogazione che le contiene. Ad esempio, nell'ultima interrogazione scritta, è possibile prima di tutto risolvere la query interna, cioè calcolare il massimo stipendio, e poi elaborare la query esterna confrontando il massimo stipendio con il valore dello stipendio di ogni dipendente. Questa soluzione è più efficiente rispetto a quella che valuta la query interna per ogni tupla della query esterna. Questa soluzione non può però sempre essere seguita. Consideriamo una interrogazione che restituisce gli attori per i quali esiste un altro attore con la medesima data di nascita. Osserviamo innanzitutto che è possibile risolvere l'interrogazione con il seguente join:

```
select distinct A1.nome
from attore A1 join attore A2 on
(A1.dataDiNascita = A2.dataDiNascita) and
(A1.nome <> A2.nome)
```

Vediamo ora una soluzione che usa una query nidificata e l'operatore `exists`. Questo operatore ha come argomento una query interna e restituisce vero se e soltanto se il risultato della query argomento contiene qualche elemento:

```
select A1.nome
from attore A1
where exists (select A2.nome
              from attore A2
              where (A1.dataDiNascita = A2.dataDiNascita) and
                    (A1.nome <> A2.nome))
```

Questo tipo di query si dicono **query nidificate con passaggio di variabili**. In particolare, la variabile A1, creata nella query esterna per la prima istanza della tabella attore, viene passata alla query interna che ne fa uso. In tal caso la query interna non può essere valutata indipendentemente da quella esterna in quanto si serve di variabili definite a livello di query esterna. Dunque l'unico modo di risolvere questa query consiste nel valutare la query interna per ogni tupla della query esterna. La **visibilità delle variabili** in SQL segue la seguente semplice regola: una variabile è visibile nella query che l'ha definita o in una query nidificata in essa (ad un qualsiasi livello).

Vediamo un esempio che usa la negazione dell'operatore `exists`: gli attori per i quali non esiste un altro attore con la medesima data di nascita:

```
select A1.nome
from attore A1
where not exists (select A2.nome
                  from attore A2
                  where (A1.dataDiNascita = A2.dataDiNascita) and
                        (A1.nome <> A2.nome))
```

Una formulazione che non usa le query nidificate è la seguente:

```
select nome
from attore
  except
select A1.nome
from attore A1 join attore A2 on
(A1.dataDiNascita = A2.dataDiNascita) and
(A1.nome <> A2.nome)
```

Avanti Indietro Indice