

# **BASI DI DATI**

Prof. Stefano Ceri  
5 CFU

**Tommaso Fontana**  
**Valentina Deda**

Lecture Notes  
Year 2017/2018



Triennale Ingegneria Informatica  
Politecnico di Milano  
Italy  
4 gennaio 2018

# Indice

<b>1</b>	<b>Le query SQL</b>	<b>2</b>
1.1	Sintassi delle query SQL . . . . .	2
1.1.1	Attributi con valore null . . . . .	2
1.1.2	Tabelle e join . . . . .	3
1.1.3	Ridenominazione . . . . .	3
1.2	Modificare il database in SQL . . . . .	3
1.3	Query complesse . . . . .	4
1.3.1	Ordinamento . . . . .	4
1.3.2	Funzioni aggregate . . . . .	4
1.3.3	Raggruppamento . . . . .	4
1.3.4	Query binarie . . . . .	4
1.3.5	Query nidificate . . . . .	5
1.4	Permessi d'accesso . . . . .	5
1.5	Transazioni . . . . .	6
<b>2</b>	<b>I linguaggi formali</b>	<b>7</b>
2.1	Algebra . . . . .	7
2.2	Calcolo relazionale . . . . .	8
2.2.1	Proprietà del calcolo relazionale . . . . .	9
2.2.2	Formule corrette . . . . .	9
2.2.3	Relazione con l'algebra . . . . .	9
2.3	Datalog . . . . .	9
2.3.1	Negazione e query ricorsive . . . . .	9

# 1

## Le query SQL

### 1.1 Sintassi delle query SQL

Ogni query SQL si compone di tre clausole:

- `select`
- `from`
- `where`

Ogni query ha sintassi:

`select *attributo*`

`from *tabella*`

`[ where *condizione* ]`

Dunque *select* indica l'attributo che ci interessa estrarre, *from* la tabella da cui estrarre l'attributo, *where* la condizione che l'attributo deve rispettare affinché sia rilevante per la nostra query.

La condizione indicata dal *where* può essere espressa tramite un'espressione i cui operatori sono:

- **predicati semplici dell'algebra**: uguaglianza, operatori booleani ...
- **between**: indica gli estremi di un intervallo.
- **distinct**: impedisce che vi siano duplicati fra i risultati della query

**like**: indica parte del nome dell'attributo cercato, ad esempio

```
1 | nome like 'Mar\%o'
```

cerca i nomi che iniziano con “Mar” e finiscono con “o”. Con *like* si usano i simboli `_` e `%`, che indicano rispettivamente un carattere e una sottostringa (di lunghezza arbitraria) di cui non conosciamo (o non ci interessa) il valore; dunque qualunque valore essi abbiano nell'attributo, tale attributo sarà ritenuto valido e restituito dalla query.

#### 1.1.1 Attributi con valore null

Esiste un valore speciale **null**, che viene utilizzato se non si conosce un valore, se un valore non si può applicare a un determinato attributo, se non si sa se tale valore possa essere applicato a un determinato attributo. `/textitNull` può essere usato nelle query attraverso gli operatori

```
1 | attributo is null
```

e

```
1 | attributo is null
```

che restituiscono rispettivamente le righe con valore nullo per un determinato attributo e quelle con valore non nullo per un determinato attributo.

### 1.1.2 Tabelle e join

Le tabelle indicate dopo la clausola *where* indicano il dominio della query. Possono essere indicate più tabelle, separate da una virgola: la query ne farà automaticamente il prodotto cartesiano e cercherà l'attributo richiesto in tale prodotto cartesiano. Si può anche effettuare la join in modo esplicito, con l'espressione

```
1 | select attributo from tabella1 join tabella on condizione_su_cui_viene_effettuata_la_join
```

La condizione della join è espressa come in algebra.

La *join* può essere di due tipi differenti:

- interna
- esterna

Per indicare la *join* interna si scrive semplicemente “join”, per indicare una *join* esterna si scrive “left/right/full join”. Una *join* esterna restituisce anche le righe per cui la condizione espressa dall’“on” della join restituisca valori nulli. Dunque con una *left join* prenderò tutte le righe della tabella dichiarata a sinistra della join, anche se in quella di destra alcune di esse avranno attributi con valori nulli. La *right join* fa lo stesso ma con le righe della tabella a destra dell’operatore join. La *full join* prende tutte le righe di entrambe le tabelle.

### 1.1.3 Ridenominazione

È possibile “dare un nome” al risultato della query usando l’operatore “as”, ad esempio:

```
1 | select * as informatico from STUDENTE where ...
```

As si usa anche nell’espressione della *from* nel caso si debbano estrarre più variabili dalla stessa tabella. Ad esempio

```
1 | select * from STUDENTE as stud1, STUDENTE as stud2, ...
```

## 1.2 Modificare il database in SQL

È possibile:

- Effettuare inserimenti:

```
1 | insert into nome tabella values lista valori
```

vengono messi a null o al valore di default.

Se mancano dei valori nella lista Al posto di usare “values + lista valori” si può inserire il risultato di una query, scrivendo la query, come una normale query, alla fine del comando:

```
1 | insert into nome tabella query
```

- Effettuare cancellazioni:

```
1 | delete from nome tabella where condizione che identifica gli elementi da cancellare
```

- Modificare i valori degli attributi:

```
1 | update nome_tabella set attributo = valore where condizione
```

Cancellare intere tabelle: **drop table \*nome tabella\***

```
1 | update nome_tabella set attributo = valore where condizione
```

## 1.3 Query complesse

### 1.3.1 Ordinamento

È possibile riordinare i risultati di una query attraverso il comando **order by**. Si usa la sintassi:

```
1 | order by AttributoOrdinamento [ crescente | decrescente ]
```

### 1.3.2 Funzioni aggregate

Le funzioni aggregate sono funzioni, utilizzate all'interno delle espressioni della clausola *where*, che utilizzano operatori complessi, che operano su più elementi del database. Esistono cinque operatori SQL per realizzare funzioni aggregate: - **count**: restituisce il numero di righe per un certo attributo. Se si aggiunge l'operatore *distinct*, restituisce il numero di righe per l'attributo omettendo le righe duplicate.

```
1 | count( * | [ distinct | all ] ListaAttributi )
```

- **sum**: restituisce la somma dei valori dell'attributo passato come parametro. - **max**: restituisce il massimo fra i valori dell'attributo passato come parametro. - **min**: restituisce il minimo fra i valori dell'attributo passato come parametro. - **avg**: restituisce la media fra i valori dell'attributo passato come parametro.

```
1 | sum|max|min|avg ([distinct|all] Attributo )
```

### 1.3.3 Raggruppamento

È possibile che ci sia il bisogno di applicare gli operatori appena visti a un sottoinsieme di righe di una tabella, non all'intera tabella. Allora si usa l'operatore *group by*, che seleziona le righe che ci interessano.

```
1 | group by attributo1 having operatore_aggregato(attributo2)
```

### 1.3.4 Query binarie

Sono realizzate concatenando due query attraverso gli operatori insiemistici di unione, intersezione e differenza.

```
1 | Query1 union | intersect | except [ all ] Query2
```

Se si usa l'operatore "all" vengono inclusi anche eventuali duplicati, se si omette essi sono esclusi automaticamente.

### 1.3.5 Query nidificate

Le query nidificate sono query che contengono ulteriori query al proprio interno. In particolare, si tratta di query che hanno nella propria clausola *where* un predicato che effettua il confronto con il risultato di un'altra query. Questi predicati sono formati da due elementi:

- **L'operatore:** è un operatore logico, quindi `=`, `<`, `>`, `<=`, `>`, ...
- **any|all:** indicano la quantità di righe con cui fare il confronto.

*All* indica che il confronto effettuato dall'operatore deve valere per ciascuna riga della query espressa dopo l'*all*, *any* invece indica che il confronto deve valere per almeno una riga della query espressa dopo l'*any*. Ad esempio:

```
1 | select Nome
2 | from STUDENTE
3 | where Nome = any
4 |     select Nome
5 |     from STUDENTE
```

estrae gli studenti con almeno un omonimo.

Esistono anche altre predicati per effettuare le query nidificate. Questi predicati sono: **-in:** è vero se l'espressione è presente almeno in una riga restituita da Query2:

```
1 | | Espressione in Query2
```

Esiste anche il *not in*, che è il negato di *in* e ha la stessa sintassi. Il predicato *in* è equivalente a *= any*. **-exists:** è vero se Query2 ha almeno una tupla.

```
1 | | exists Query2
```

Il suo negato è *not exists*.

#### Visibilità delle variabili nelle query nidificate

Le variabili sono visibili **solo** nelle query in cui sono dichiarate o nelle query nidificate ad esse. Dunque due query nidificate alla stessa query principale ma non fra loro **non** vedono le variabili l'una dell'altra, ma entrambe vedono le variabili della query principale.

#### Query nidificate nelle modifiche

Le query nidificate possono essere utilizzate anche con il comando **update**. È sufficiente scrivere:

```
1 | update nome_riga
2 |     set attributo = Query2
```

## 1.4 Permessi d'accesso

È possibile controllare l'accesso di ciascun utente al database. In particolare si può consentire/limitare le operazioni che ogni utente può effettuare su di esso. Per concedere un permesso, si utilizza il comando **grant**. Per revocare un permesso, si utilizza il comando **revoke**. La sintassi è la seguente per la *grant*:

```
1 | | grant < nome_privilegi | all privileges > on nome_risorsa nome_utente [withgrantoption]
```

L'espressione *withgrantoption*, che è opzionale, indica che l'utente a sua volta è autorizzato a fornire il privilegio ad altri utenti.

La *revoke* invece ha sintassi:

```
1 | revoke nome_privilegi on nome_risorsa from nome_utente [ restrict | cascade ]
```

## 1.5 Transazioni

Le transazioni sono sequenze di comandi SQL che devono essere racchiuse fra l'espressione **begin transaction** e **end transaction**. Attraverso queste transazioni è possibile effettuare delle modifiche al database. La differenza fra l'utilizzare una transazione e non utilizzarla sta nel fatto che, utilizzando la transazione, è possibile decidere se rendere permanenti le modifiche oppure eliminarle e tornare allo stato iniziale. Infatti, prima di ogni *end transaction* è necessario decidere mantenere o no le modifiche attraverso, rispettivamente, i comandi **commit work** e **rollback work**. Il primo rende definitive le modifiche, il secondo le cancella e mantiene il database allo stato precedente la transazione.

# 2

## I linguaggi formali

### 2.1 Algebra

L'algebra è un linguaggio formale caratterizzato da cinque operazioni fondamentali e tre operazioni derivate. Le operazioni fondamentali sono:

1. Selezione  $\sigma$
2. Proiezione  $\pi$
3. Unione  $\cup$
4. Differenza  $-$
5. Prodotto cartesiano  $\times$

Le operazioni derivate sono:

1. Intersezione  $\cap$
2. Semijoin  $\ltimes$
3. join  $\bowtie$

Data una tabella di studenti chiamata "STUDENTE", contenente matricola, nome, città, definiamo le operazioni tramite i seguenti esempi.

#### Selezione

Esempio:  $\sigma_{Nome='Paola'} \text{STUDENTE}$

La selezione è un operatore che produce come risultato una tabella che abbia lo stesso schema (cioè le stesse colonne) di studente e come istanze le sole righe che soddisfano il predicate  $Nome = Paola$ .

#### Proiezione

Esempio:  $\pi_{Nome} \text{STUDENTE}$

La proiezione è un operatore che produce come risultato una tabella che abbia come schema soltanto la colonna *nome* e come istanze tutte le righe della tabella STUDENTE.

Da questo punto in poi per comodità considereremo due generiche tabelle, TABELLA1 e TABELLA2 per gli esempi successivi.



### Unione

Esempio:  $TABELLA1 \cup TABELLA2$ .

L'unione è un operatore che produce come risultato una tabella che abbia come schema le colonne di TABELLA1 e come righe le righe delle due tabelle concatenate. È possibile realizzare questa operazione **solo** se le tabelle sono compatibili.

### Differenza

Esempio:  $TABELLA1 - TABELLA2$ .

La differenza è un operatore che produce come risultato una tabella che abbia come schema lo schema di TABELLA1 e come righe la differenza delle tuple delle due tabelle (cioè elimina le righe che compaiono in entrambe le tabelle). È possibile realizzare questa operazione **solo** se le tabelle sono compatibili.

### Prodotto cartesiano

Esempio:  $TABELLA1 \times TABELLA2$

Il prodotto cartesiano è un operatore che produce come risultato una tabella che abbia come schema le colonne delle due tabelle e come tutte le possibili coppie degli elementi di TABELLA1 e TABELLA2.

### Unione

Esempio:  $TABELLA1 \cap TABELLA2$ .

L'intersezione è un operatore che produce come risultato una tabella che abbia come schema le colonne di TABELLA1 e come righe le righe che compaiono in entrambe le tabelle. È un'operazione derivata perché è equivalente a  $TABELLA1 - (TABELLA1 - TABELLA2)$ . È possibile realizzare questa operazione **solo** se le tabelle sono compatibili.

### Join

Esempio:  $TABELLA1 \bowtie_{TABELLA1.attribute1=TABELLA2.attribute1} TABELLA2$ .

La join è un operatore che produce come risultato una tabella che abbia come schema la concatenazione degli schemi delle due tabelle e come righe la concatenazione delle righe che soddisfano il predicato della join. È un'operazione derivata perché è equivalente a  $\sigma_{TABELLA1.attribute1=TABELLA2.attribute1} TABELLA1 \times TABELLA2$ .

La join che utilizza come operatore del predicato l'operatore di uguaglianza ( $=$ ) è detta *equi-join*. Un'equi-join che voglia selezionare solo gli attributi omonimi è detta *natural join* e viene espressa omettendo il predicato che confronta gli attributi (quindi  $TABELLA1 \bowtie TABELLA2$ ).

### Semi-join

Esempio:  $TABELLA1 \ltimes_{TABELLA1.attribute1=TABELLA2.attribute1} TABELLA2$ .

La semi-join è un operatore che produce come risultato una tabella che abbia come schema lo schema di TABELLA1 e come istanze le tuple ottenute proiettando su TABELLA1 la join di TABELLA1 e TABELLA2.

Esiste anche la semi-join naturale, che proietta su TABELLA1 la join naturale delle due tabelle.

## 2.2 Calcolo relazionale

Il calcolo relazionale è un linguaggio dichiarativo: esprime cosa si vuole ottenere ma non la "strada" per giungere al risultato. La forma standard di una qualunque query nel calcolo relazionale è  $t|p(t)$ , dove  $t$  è una variabile e  $p(t)$  un predicato in funzione di  $t$ , detto *formula*. Ogni formula è costituita da *atomi*, dove ogni atomo può essere:

- $t \in R$ , dove  $R$  è un dominio;
- espressione - comparatore - espressione, dove un comparatore è un operatore come  $=$ ,  $<$ ,  $>$ ,  $\geq$ , ... e un'espressione è un'insieme di costanti e di restrizioni. Una restrizione è espressa come  $t[A]$  e seleziona l'attributo  $A$  della variabile  $t$ .

### 2.2.1 Proprietà del calcolo relazionale

Per il calcolo relazionale valgono le seguenti proprietà:

**La legge di de Morgan:**  $p1 \wedge p2 \equiv \neg(\neg p1 \vee \neg p2)$

**La Corrispondenze tra quantificatori:**  $\forall t \in R(p(t)) \equiv \neg \exists t \in R(\neg p(t))$

**Definizione di implicazione:**  $p1 \Rightarrow p2 \equiv \neg p1 \vee p2$

### 2.2.2 Formule corrette

Una formula è considerata corretta **solo** se non dipende dal dominio. Cioè è indipendente dal dominio degli attributo, ma dipende soltanto dall'istanza del database che stiamo analizzando.

### 2.2.3 Relazione con l'algebra

È possibile esprimere attraverso il calcolo relazionale tutti gli operatori dell'algebra, e viceversa è possibile esprimere qualunque query del calcolo attraverso l'algebra.

## 2.3 Datalog

Datalog è un linguaggio formale basato sulle *regole*. Ogni regola è una “riga” datalog, formata da una head e da un body, separati dall'operatore :- e formati da predicati. Ad esempio, per la regola seguente il predicato P è la head, P1 e P2 sono il body: P :- P1, P2. Ogni predicato è formato da un nome e uno o più argomenti. Gli argomenti possono essere variabili, costanti oppure “don't care”. Il “don't care” è rappresentato dal simbolo \_ ed significa che qualunque attributo va bene. Non può essere usato nella head.

L'interpretazione delle regole è che la testa è vera se il corpo è vero.

Le query sono realizzate mettendo “?” nella testa: ?-Predicato(“Costante”, Variabile) Questa query cerca una regola che abbia Predicato come testa e cerca valori della Variabile per cui valga “Costante”.

Una query senza variabile, come ?-Predicato(“costante”) restituisce true o false.

### 2.3.1 Negazione e query ricorsive

È possibile negare predicati in datalog premettendo “not” al predicato. Introducendo la negazione, è possibile esprimere in datalog tutte le query dell'algebra relazionale e, in aggiunta, anche le query ricorsive (che nell'algebra non abbiamo). Una query ricorsiva è una query che presenta nel body il predicato della testa.

Perché la regola sia valida, la negazione deve essere *safe*. Una negazione è safe se:

- tutte le variabili di un letterale negato compaiono in un letterale positivo del body della regola.
- Il predicato positivo e il suo negato non dipendono l'uno dall'altro. Quindi  $P(X) :- R(X), \neg P(X)$  non è una negazione safe.