Capitolo 3 - SQL

Appunti di **Giuseppe Pitruzzella** - Corso di Database @ DMI, UniCt

[TODO] *Integra con gli appunti delle Slides.*

1. Introduzione

Il linguaggio SQL nasce alla IBM negli anni ’70 ed è oggi lo standard per tutti i sistemi commerciali ed open source. Con il passare degli anni si sono succeduti diversi standard, quali SQL2 negli anni ’90 ed SQL3 nei primi anni 2000. Noi ci concentreremo soprattutto su SQL2.

2. Sintassi

La sintassi generale per la scrittura di una query in SQL è la seguente:

```
SELECT [DISTINCT] ListaDiAttributi
FROM Tabelle
WHERE Condizione
```

In questo primo esempio notiamo tre parti, o meglio tre **clausole**, una per ogni istruzione.

- La **clausola SELECT**, legata all’istruzione omonima, si compone della lista di attributi (anche conosciuta con il nome *Target List*). Si noti che, utilizzando la keyword `DISTINCT` è possibile rimuovere eventuali duplicati all’interno del risultato, mentre attraverso il simbolo `*`, possiamo riferirci all’intero insieme di attributi della tabella.

  [NB] Utilizzare `DISTINCT` indica svolgere un’operazione aggiuntiva molto dispendiosa e non necessaria se stiamo operando con sole tabelle che posseggono una chiave primaria.

- La **clausola FROM** rappresenta le tabelle su cui sarà effettuata l'interrogazione; in particolare, si noti che SQL svolge un **prodotto cartesiano** sulle tabelle descritte.

- La **clausola WHERE** si compone essenzialmente di una **condizione C** sulla tabella, e crea quindi un filtro sul risultato. Quest'ultima condizione può essere costruita mediante **predicati semplici**, **operatore logici** (come AND, OR e NOT) ed **operatori di confronto** (come <, >, =, <>).

  Altri **operatori interessanti** e specifici per SQL sono i seguenti:

  - **LIKE**, che è possibile utilizzare mediante due simboli speciali: '_' e '*%*'. Essi indicano rispettivamente "*un singolo carattere*" ed "un numero qualsiasi di caratteri".

    [ES] Trovare tutti i cognomi cui seconda lettera è 'B' ed ultime due 'AB'.

    ```
    SELECT Cognome FROM Studenti WHERE Cognome LIKE _B%AB
    ```

  - **IN**, che è possibile utilizzare per notare se esiste almeno una corrispondenza tra il valore a sinistra e la lista di valori a destra.

    [ES] `SELECT Nome FROM Dipendenti WHERE Salario IN (2000, 2100, 2200)`

  - [**NOT**] **BETWEEN** X **AND** Y, per cui è possibile indicare se un valore n è compreso tra due valori specificati.

    [ES] `SELECT Nome FROM Dipendenti WHERE Salario BETWEEN 1000 AND 1500`

  - **IS** [**NOT**] **NULL**, quale risulta *true* se l'attributo possiede valore nullo. Il predicato IS NOT NULL è la sua negazione.

    Si noti che SQL-2 gestisce i valori nulli attraverso una logica a tre valori: *true*, *false* ed *unknown*. Quest'ultimo, in particolare, è restituito da ogni predicato che non sia `IS [NOT] NULL`.

  Esiste inoltre la possibilità di **ridenominare** le tabelle all'interno della clausola SELECT e FROM attraverso la keyword (opzionale) "AS". Questo processo diventa particolarmente importante nel caso in cui più attributi di più tabelle hanno lo stesso nome. In tal caso, infatti, utilizzeremo (tendenzialmente all'interno della clausola WHERE) la notazione NomeTabella.NomeAttributo .

  [NB] Si noti che i comandi SQL **non** sono **case-sensitive** e possono essere distribuiti su una o più righe per migliorare la leggibilità.

In definitiva, è interessante notare cosa avviene nello specifico mediante map di una generica query in *Algebra Relazionale*.

```
SELECT DISTINCT A1, A2
FROM R1, R2
WHERE C
```

$$\pi_{A_1,A_2}(\sigma_C(R_1 \times R_2))$$

2.1 Operazione di Join

Abbiamo notato come per svolgere un'operazione di JOIN tra due tabelle sia possibile semplicemente descrivere la corretta condizione C all'interno della **clausola WHERE**.

L'operatore di **Join** è utile per effettuare query su più tabelle e viene **specificata all'interno** di **WHERE**. Il tutto poichè, come abbiamo notato nel capitolo relativo all'Algebra Relazionale, vale la seguente uguaglianza:

$$\sigma_C(R_1 \times R_2) = R_1 \bowtie_C R_2$$

[ES] Trova gli esami sostenuti dallo studente Mario Rossi.

```
SELECT *
FROM Studente S, Corso C
WHERE S.Matricola = C.Matricola AND S.Nome = Mario Rossi
```

[NB] Si ricordi che è necessario l'utilizzo degli **alias** per query che prevedono più tabelle con uno o più **attributi uguali**.

Conseguentemente a ciò che è stato descritto sopra, intuiamo quindi che in assenza di una condizione di join all'interno del WHERE, la query basata su più tabelle calcolerà il prodotto cartesiano tra quest'ultime, generando quindi ogni possibile coppia.

Esiste comunque **un'alternativa** che prevede l'esplicitazione del JOIN. Non più all'interno della clausola WHERE, ma all'interno della clausola FROM.

```
SELECT *
FROM R [TYPE] JOIN S ON C
WHERE C
```

E' possibile specificare il tipo di Join sostituendo a `TYPE` dei termini come "**INNER**", "**LEFT OUTER**", "**RIGHT OUTER**" e "**FULL OUTER**". Può infatti capitare che alcune righe non vengano considerate in quanto non esiste una corrispondete riga nell'altra tabella per cui la condizione sia soffisfatta. E' possibile quindi mantenere quest'ultime righe, introducendo dei valori nulli per rappresentare le informazioni mancanti.

In particolare, l'`OUTER JOIN` (o join esterno) prevede di mantenere le righe senza corrispondenza dell'una o dell'altra tabella, in base all'utilizzo di `LEFT`, `RIGHT`, `FULL`, che estenderanno (con valori nulli) rispettivamente le righe della tabella a sinistra, le righe della tabella a destra ed entrambe.

[NB] Di default, quindi omettendo il parametro `TYPE`, verrebbe svolta una semplice **Theta Join**.

In definitiva (e per completezza) notiamo l'esistenza di un ultimo tipo di join, ovvero la `NATURAL JOIN`, la quale prevede di utilizzare una condizione *implicita* di uguaglianza su tutti gli attributi caratterizzati dallo stesso nome. Quest'ultima non è tipicamente consigliata poichè il suo comportamente può mutare al variare della tabella.

2.2 Operatori aritmetici

All'interno di SQL è possibile utilizzare i classici operatori aritmetici. Un esempio del loro utilizzo potrebbe essere il seguente:

SELECT Nome, Salario, Salario + 300 FROM Dipendenti

In tal caso, la query restituirà il nome del dipendente, il suo salario ed il risultato della somma tra il suo salario e la costante 300.

Si noti sia possibile utilizzare più operatori all'interno di un'espressione, i quali saranno soggetti alla precedenza che tutti noi conosciamo (per es. lo svolgimento del prodotto prima della somma).

2.3 Operatori logici

È anche possibile, all'interno di SQL, utilizzare gli operatori logici come **AND**, **OR** e **NOT**, rispettivamente per imporre che due condizioni devono essere soddisfatte, che almeno una delle due deve essere soddisfatta e che la condizione stessa deve essere falsa.

Riguardo quest'ultimi, è interessante notare le **regole di precedenza**. Infatti, supponendo che vi sia una condizione che li prevede tutti (compresi gli operatori di confronto), allora inizialmente (i) verranno notati gli operatori di confronto, per poi (ii) notare se vi siano delle negazioni (NOT), in seguito (iii) degli AND ed infine (iv) degli OR. In ogni caso, *l'ordine prestabilito può essere modificato* mediante utilizzo delle parentesi.

[ES] Trova tutti i dipendenti che coprono una carica di Software Developer o sono dei Manager ed il loro salario è maggiore di 4000.

```
SELECT *
FROM Dipendenti
WHERE Posizione = 'Software Dev' OR Posizione = 'Data Scientist' AND
Salario > 4000
--  Il predicato sopra è da intendersi come:  (Posizione = 'Software
Dev' OR (Posizione = 'Data Scientist' AND Salario > 4000))
```

2.4 Ordinamento all'interno di una tabella

Nell'uso di un database sorge spesso il bisogno di costruire un ordine sulle righe delle tabelle. Pensiamo al caso in cui un utente voglia sapere quali sono gli stipendi più elevati in un azienda.

In SQL è possibile svolgere ciò attraverso la keyword `ORDER BY`, il tutto attraverso la seguente sintassi:

```
ORDER BY Attributo [ASC | DESC]
```

[NB] E' possibile ordinare anche secondo un numero maggiore di attributi.

In questo modo si specificano gli attributi che devono essere usati per l'ordinamento. Quest'ultime verrano ordinate in modo sequenziale rispetto l'ordine in cui sono state descritte.

L'ordine, a sua volta, può essere *ascendente* (`ASC`), *discendente* (`DESC`). Di default verrà considerato un ordinamento ascendente.

3. Operatori di aggregazione

Gli **operatori aggregati** costituiscono una delle più importanti estensioni di SQL. Ogni operatore di aggregazione viene applicato alla tabella contenente il risultato dell'interrogazione.

Essi sono **cinque**, ovvero:

- `count`: svolge il compito di contare il numero di righe all'interno della tabella.

  La sintassi completa per l'utilizzo dell'operatore è la seguente: `count * | [distinct, all] ListaDiAttributi`.

  In particolare, l'operatore può essere utilizzato:

  - Insieme al simbolo `*`, per contare *tutte le righe* all'interno della tabella;
  - Insieme alla keyword `distinct`, per contare tutte le righe distinte all'interno della `ListaDiAttributi`;
  - Insieme alla keyword `all` (quale è anche l'opzione di *default*), per cui vengono considerate solo le righe con valori *non nulli*. Restituisce 0 se opera in presenza di soli valori nulli.
- `sum`: effettua la somma dell'attributo che racchiude tra parentesi. Come è possibile intuire la sintassi per l'utilizzo dell'operatore è il seguente: `sum(Attributo)`. L'operatore `sum` ignora i *valori nulli*.
- `max`, `min`, `avg` : quali restituiscono rispettivamente massimo e minimo e medie dell'attributo tra parentesi.

  La sintassi per l'operatore è la seguente: `max(Attributo)` , `min(Attributo)`, `avg(Attributo)`. In particolare, l'operatore `avg`, ignora i *valori nulli*. In generale tutti i tre gli operatori, insieme anche all'operatore `sum`, restituiscono valore nullo se operano in presenza di soli valori nulli.

  [NB] Relativamente all'utilizzo di `distinct` o `all` , notiamo che in assenza di entrambi verrà considerato `all` di default.

3.1 Raggruppamento nelle Query

SQL mette a disposizione la clausola `group by` permettendo di specificare in che modo dividire la tabella in sottoinsiemi. Un esempio del suo utilizzo può essere il seguente:

```
SELECT DeptName, SUM(Salary) FROM Employee GROUP BY DeptName
```

Dall'esempio precedente si intuisce, quindi, che l'utilizzo di un *operatore aggregato* unito ad un raggruppamento mediante `group by`, fa sì che l'operatore aggregato venga applicato esclusivamente ai vari sottoinsiemi. Se un interrogazione fa uso di questa clausola (*group by*), allora l'argomento della `SELECT` sarà un sottoinsieme degli attributi che compongono la `group by` ed al più degli operatori di aggregazione.

3.2 Predicati sui gruppi

Abbiamo notato che se le condizioni che i sottoinsiemi devono soddisfare sono verificabili al livello delle singole righe, allora basta imporre quest'ultime all'interno della clausola WHERE. Lo stesso però non si può dire se le condizioni sono di tipo aggregato. In tal caso, infatti, sarà necessario imporre quest'ultime all'interno di una nuova clausola, ossia `HAVING`.

Più nello specifico, la clausola `HAVING`, impone le condizioni da applicare all'interno di un interrogazione che fa uso della clausola `GROUP BY`. Quindi, in altre parole, un sottoinsieme generato a partire da un raggruppamento farà parte del risultato se soddisfa il predicato definito all'interno di `HAVING`.

> *I predicati (o condizioni che dir si voglia) che prevedono un operatore aggregato, devono essere definiti all'interno della clausola HAVING.*

3.3 Interrogazioni di tipo insiemistico

Analogamente all'algebra relazionale, anche SQL mette a disposizione gli operatori insiemistici. Quest'ultimi sono: `UNION`, `INTERSECT` ed `EXCEPT`, indicando rispettivamente *unione*, *intersezione* e *differenza*.

Si noti che gli operatori insiemistici prevedono di default di eliminare qualsivoglia duplicato. Se ciò non è desiderabile, è utile accompagnare l'operatore insiemistico con la keyword `ALL`. Inoltre, differentemente dall'algebra relazionale, SQL non prevede che gli attributi su cui si effettua l'operarazione siano uguali, bensì che siano compatibili e di uguale numero, basandosi quindi sulla posizione degli attributi.

3.4 Interrogazioni nidificate

Il linguaggio SQL consente anche di scrivere **interrogazioni nidificate**. L'uso di quest'ultime può avvenire in ogni clausola, ma il suo utilizzo più comune avviene nella clausola `WHERE`. Notiamo come esista un **problema** di disomogeneità nel momento in cui avviene un confronto tra un attributo ed il risultato di una query nidificata. In tal caso, infatti, è bene utilizzare una tra le seguenti keyword, ossia `ANY` o `ALL`. Esse indicano rispettivamente che la condizione sia soddisfatta *per una* o *per tutte* le righe restituire dalla query.

Del tutto identici, relativamente alle keyword appena notate, sono `IN` e `NOT IN`.

```
-- Trovare tutti i nomi dei dipendenti che condividono il nome con un
dipendente in Produzione
SELECT NomeImpiegato
FROM Impiegato
WHERE NomeImpiegato = ANY (SELECT Nome
                           FROM Impiegato
                           WHERE Dipartimento = 'Produzione')
-- Trovare il nome del dipendente che guadagna lo stipendio massimo
SELECT NomeImpiegato
FROM Impiegato
WHERE NomeImpiegato = ANY (SELECT MAX(Stipendio)
                           FROM Impiegato
```

```sql
-- Utilizzo di ALL

-- Trovare i nomi dei dipartimenti in cui non lavora alcun dipendente
di nome Rossi
SELECT NomeDipartimento
FROM Dipartimento
WHERE NomeDipartimento <> ALL (SELECT Dipartimento
                               FROM Impiegato
                               WHERE Cognome = Rossi)
-- Trovare il nome del dipendente che guadagna lo stipendio massimo
SELECT Nome
FROM Impiegato
WHERE Stipendio >= ALL (SELECT Stipendio FROM Impiegato)
```

4. Algoritmi utili allo svolgimento delle Query in SQL

- Ricerca del **massimo**

```sql
SELECT M, N
FROM R1 JOIN R2 USING (ID)
WHERE Valore =
  (SELECT MAX (Valore) FROM R1)
```

- Ricerca del **minimo**

```sql
SELECT M, N
FROM R1 JOIN R2 USING (ID)
WHERE Valore =
  (SELECT MIN(Valore) FROM R1)
```

- Per ogni ID, trova il **totale di valori x** correlati al singolo ID

```sql
SELECT ID, COUNT(X) AS totaleValoriX
FROM R1
GROUP BY ID
```

- Trova i valori x associati a **tutti** i valori y

  [1]

```
SELECT DISTINCT X
FROM R1
WHERE NOT EXISTS
  -- Seleziono i valori Y non in relazione con X;
  (SELECT Y
   FROM R2
   WHERE Y NOT IN
     -- Cerco tutti i valori Y in relazione con X
    (SELECT Y
     FROM R2 JOIN R1 USING (X)))

-- Se la seconda SELECT non restituisce alcun valore, allora
ciò indica che quello specifico valore X è in relazione con
tutti i valori Y. Il NOT EXIST all'interno della prima SELECT,
quindi, risulta vero ed il valore X viene visualizzato.
```

  [2]

```
SELECT X
FROM R1
GROUP BY X
HAVING COUNT ( DISTINCT Y ) =
  (SELECT COUNT(*) FROM R2)
-- Seleziono un valore X quando che quest'ultimo è in
relazione con un numero di diversi Y pari al totale dei valori
Y.
```

- Ricerca dei valori x in relazione con **almeno** un valore y

```
SELECT X
FROM R1, R2
WHERE X = Y
```

- **Coppie** di ID sempre in relazione

```sql
CREATE VIEW R2 (ID2, Valore) AS
SELECT ID, Valore FROM R1

SELECT DISTINCT ID, ID2
FROM R1 JOIN R2 USING (Valore)
WHERE  ID < ID2
```

- Ricerca dei valori x in relazione con **al più** un valore y

```sql
SELECT X
FROM R1
GROUP BY X
HAVING COUNT(Y) <= 1
```

- Ricerca dei valori x maggiore di qualche valore y

```sql
SELECT *
FROM R1
WHERE ValoreX > ANY (SELECT ValoreY FROM R2)
```

- Ricerca degli ID con valore x inferiore alla media

```sql
SELECT ID
FROM R1
WHERE ValoreX <
  (SELECT AVG(ValoreX) FROM R1)
```

- Ricerca degli ID con valore maggiore in un giorno

```sql
SELECT ID
FROM R1 S
WHERE Valore =
  (SELECT MAX(Valore)
   FROM R2
   WHERE R1.Data = R2.Data)
```