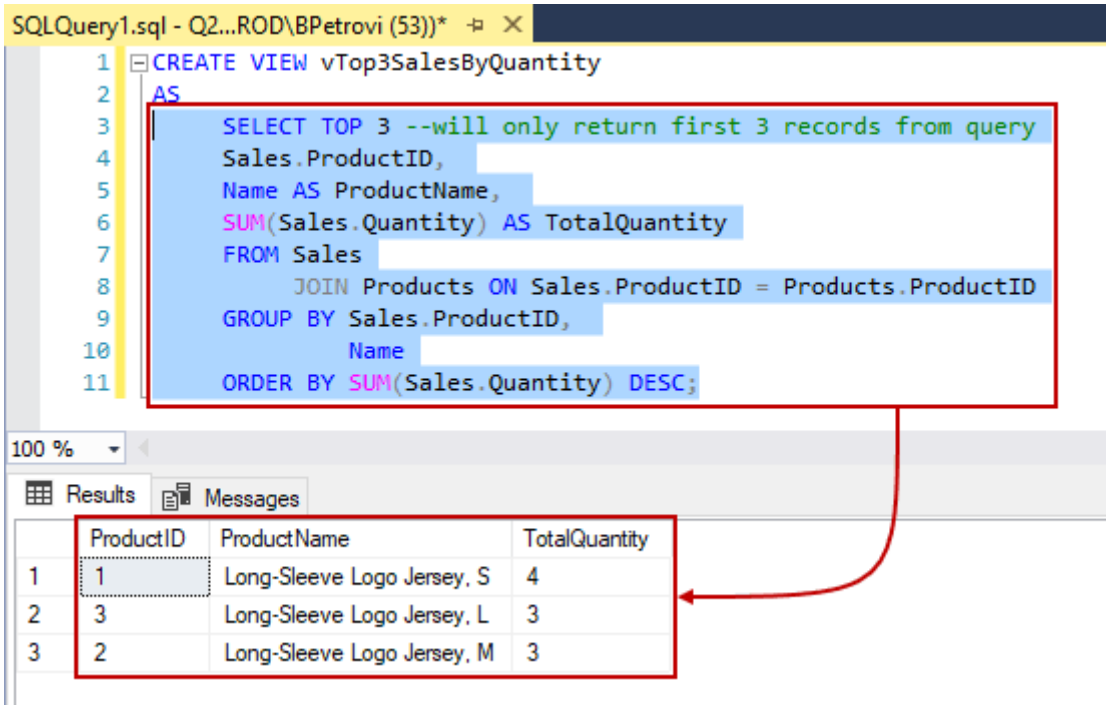
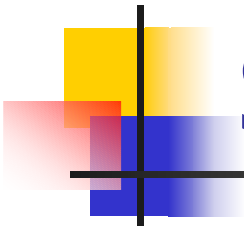
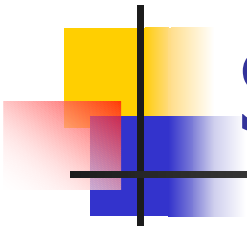
Il linguaggio SQL





SQL

- SQL è un acronimo per Structured Query Language. Fu sviluppato originalmente come linguaggio per il DBMS System R dall'IBM Research Laboratory (San Jose, California) nei tardi anni settanta. Al giorno d'oggi è ormai considerato uno standard per i database relazionali.
- SQL fornisce le primitive sia di un DDL che di un DML.
- L'ANSI (American National Standards Institute) e l'ISO (International Standardization Organization) hanno provveduto a definire una versione di riferimento di SQL fin dal 1986. Da allora vi sono le seguenti versioni:
  - SQL-89 (1989): versione ormai obsoleta con funzionalità minimali
  - SQL-2 (1992): è la versione di riferimento per i sistemi commerciali
  - SQL-3 (1999): include nuove caratteristiche frutto di ricerca recente
- SQL è considerata la lingua franca dei DBMS, i.e., è l'unico linguaggio che fornisce la garanzia di poter interagire con un database relazionale in modo indipendente dalle particolarità dell'interfaccia di quest'ultimo.



SQL

- SQL-2 è un linguaggio molto complesso e non esiste ancora un sistema commerciale che lo implementi perfettamente.
- Esistono quindi vari gradi di compatibilità:
  - Entry SQL (similare a SQL-89)
  - Intermediate SQL (sono incluse le caratteristiche che meglio si adattano alla richiesta del mercato)
  - Full SQL (sono implementate tutte le caratteristiche dello standard)
- D'altra parte molti sistemi commerciali presentano caratteristiche che non sono contemplate nello standard.
- In generale tutti i sistemi sono molto aderenti allo standard per quanto riguarda gli aspetti consolidati di SQL, mentre differiscono fra loro e rispetto a SQL-2 quando si considerano le caratteristiche più avanzate ed innovative.

Domini elementari in SQL

- Il dominio **`character`** permette di rappresentare singoli caratteri o stringhe:

  `character` [**`varying`**] [(lunghezza)] [**`character set`** nome]

  Esempi:

  1) stringhe composte da 20 caratteri: **`character (20)`**

  `2)` stringhe composte da al più 100 caratteri dell'alfabeto russo: **`character varying (100) character set Cyrillic`**

  Al posto di **`character`** e **`character varying`** si possono usare le forme compatte **`var`** e **`varchar`**.
- Il dominio **`bit`** permette di rappresentare attributi (*flag*) che specificano se un oggetto ha o meno una certa proprietà; infatti un attributo con dominio **`bit`** può assumere soltanto il valore 0 o il valore 1. E' anche possibile avere come domini stringhe bi bit tramite la sintassi seguente:

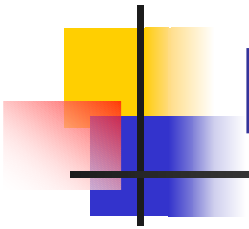
  bit [varying] [(lunghezza)]

  Esempi:

  1) stringhe di 20 bit: **`bit (20)`**

  2) stringhe di al più 35 bit: **`bit varying (35)`** o **`varbit(35)`**

Domini elementari in SQL



- Domini numerici esatti:
  - **`numeric`** [(Precisione [,Scala])]
  - **`decimal`** [(Precisione [,Scala])]
  - **`integer`**
  - **`smallint`**

  Esempi:

  1) Valori da –99 a +99: decimal (2)

  2) Valori da –999.99 a +999.99: numeric (5,2)
- Domini numerici approssimati (numeri a virgola mobile):
  - **`float`** [(Precisione)]
  - **`double precision`**
  - **`real`**

Domini elementari in SQL

- In SQL-2 sono stati introdotti dei domini specifici al fine di rappresentare informazioni temporali:
  - **`date`**
  - **`time`** [(Precisione)] [**`with`** time zone]
  - **`timestamp`** [(Precisione)] [**`with`** time zone]
- Ognuno dei domini precedenti è costituito da campi:
  - **`year`**, **`month`** e **`day`** per **`date`**
  - **`hour`**, **`minute`**, **`second`** per **`time`**
  - **`year`**, **`month`**, **`day`**, **`hour`**, **`minute`**, **`second`** per **`timestamp`**
- La precisione rappresenta il numero di cifre decimali destinate a rappresentare le frazioni di secondo.
- Nel caso venga specificata una time zone, si hanno a disposizione due ulteriori campi (**`timezone_hour`** e **`timezone_minute`**) che rappresentano la differenza fra l'ora locale e quella di Greenwich (Universal Coordinated Time).

Domini elementari in SQL

- Gli intervalli temporali sono un tipo di dominio che permette di rappresentare informazioni corrispondenti alla durata di un'azione, i.e., intervalli di tempo. La sintassi è la seguente:

  `interval` PrimaUnitàTemporale [**`to`** UltimaUnitàTemporale]
- La PrimaUnitàTemporale può essere accompagnata dalla precisione che indica il numero di cifre in base 10 che possono essere usate nella rappresentazione. Quando l'unità temporale più piccola è **`second`**, si può specificare il numero di cifre decimali che possono occorrere dopo la virgola.
- Esempi:

  1) intervalli fino a 99 anni e 11 mesi: **`interval year to month`**

  2) intervalli fino a 99999 anni e 11 mesi: **`interval year(5) to month`**

  3) intervalli fino a 9999 giorni, 23 ore, 59 minuti e 59,999999 secondi: **`interval day(4) to second(6)`**

Definizione di schemi

- In SQL uno schema è un insieme di domini, indici, tabelle, asserzioni, viste e privilegi.
- La sintassi per creare un nuovo schema è la seguente:

  `create schema` [NomeSchema]
  [[**`authorization`**] Autorizzazione]
  {DefElementoSchema}

  dove autorizzazione indica il nome dell'utente proprietario dello schema (se non specificato si assume che sia l'utente che ha lanciato il comando).

  NomeSchema è opzionale e, nel caso in cui non venga specificato, si assume il nome del proprietario.

  DefElementoSchema rappresenta gli elementi che compongono lo schema (non è necessario specificarli tutti al momento della creazione di quest'ultimo).

Definizione di tabelle

- La sintassi per la creazione di una tabella è la seguente:

  `create table` NomeTabella

  (NomeAttributo Dominio [ValoreDiDefault] [Vincoli]

  {, NomeAttributo Dominio [ValoreDiDefault] [Vincoli]}

  {AltriVincoli}

)
- Quindi una tabella è costituita da un insieme ordinato di attributi con i rispettivi domini (ed eventuali valori di default) e da un insieme (eventualmente vuoto) di vincoli.
- L'insieme denotato da AltriVincoli si riferisce a quei vincoli definiti su più attributi della tabella.
- Esempio:

```
create table Dipartimento
(Nome varchar (20) primary key,
 Indirizzo varchar(50),
 CodicePostale char(5)
)
```

Domini definiti dall'utente

- SQL permette all'utente di definire dei nuovi domini a partire da quelli elementari tramite la sintassi seguente:

  `create domain` NomeDominio **`as`** DominioEsistente

  [ValoreDiDefault]

  [Vincoli]
- Quindi un dominio definito dall'utente associa NomeDominio con il dominio DominioEsistente (può essere predefinito o definito in precedenza dall'utente), un eventuale valore di default ed eventuali vincoli.
- L'utilità di questi domini derivati è che permettono di associare un nome con un insieme di vincoli. In tal modo si può usarli ripetutamente nella definizione delle tabelle con il vantaggio che un eventuale cambiamento nella definizione originale si propaga automaticamente in tutti i punti in cui quest'ultima è stata utilizzata.

Valori di default

- I valori di default che compaiono nelle definizioni dei domini ed in quelle delle tabelle specificano i valori che devono assumere gli attributi corrispondenti quando, al momento dell'inserimento di una nuova riga (tupla), non vengono specificati dei valori per essi.
- Esempio:

  ```
  NumeroDipendenti smallint default 0
  ```

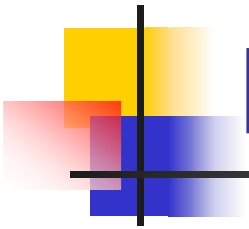
- Se non si specifica un valore di default, il sistema assume che esso sia NULL.
- Nel caso in cui siano stati specificati contemporaneamente un valore di default per il dominio ed uno per l'attributo corrispondente all'interno della definizione di una tabella, quest'ultimo ha la precedenza.

Vincoli intra-relazionali

- **Not null**: un attributo il cui dominio sia seguito dal vincolo **`not null`** non potrà mai assumere il valore NULL. Quindi in fase di inserimento è sempre necessario specificare un valore per l'attributo, a meno che non ve ne sia uno di default.
- **Unique**: si applica ad un attributo od un insieme di attributi, imponendo che l'attributo o l'insieme di attributi sia una chiave od una superchiave. Quindi non ci possono essere due righe distinte che assumano gli stessi valori (ad eccezione del valore NULL) in corrispondenza dell'attributo o dell'insieme di attributi specificati come **`unique`**.
- **Primary key**: può essere specificato solo una volta per tabella. L'attributo o gli attributi interessati diventano la chiave primaria e non possono assumere valori nulli e, ovviamente, devono essere una chiave.

Vincoli di integrità referenziale

- Un vincolo di integrità referenziale stabilisce una corrispondenza fra i valori degli attributi di una tabella (interna) ed i valori degli attributi di un'altra tabella (esterna).
- Il vincolo stabilisce che per ogni riga della tabella interna i valori degli attributi specificati nel vincolo (se non sono nulli) devono corrispondere a quelli di una riga della tabella esterna.
- L'unica restrizione imposta sugli attributi della tabella esterna è che questi siano soggetti al vincolo **`unique`**.
- Solitamente l'insieme di attributi della tabella esterna che entrano in gioco in un vincolo di integrità referenziale sono quelli che compongono la chiave primaria.



Esempio

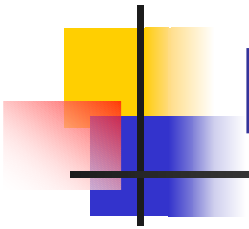
```
create table Impiegati
(Matricola integer primary key,
 Nome varchar (20),
 Cognome varchar (20),
 Dipartimento char (15),
 Stipendio numeric (9) default 0,
 Indirizzo varchar (100),
 foreign key(Dipartimento) references Dipartimenti(NomeDip),
 unique(Nome,Cognome)
)
```

Violazione di un vincolo di integrità referenziale

- Vi sono due modi di violare un vincolo di integrità referenziale:
  - Operare dei cambiamenti sulla tabella interna.
  - Operare dei cambiamenti sulla tabella esterna.
- Nel caso del primo tipo di violazione, l'operazione viene rifiutata dal sistema.
- Nel caso del secondo tipo di violazione, vi sono varie possibilità:
  - operazioni di aggiornamento:
    - **`cascade`**: il nuovo valore dell'attributo coinvolto nel vincolo della tabella esterna viene assegnato all'attributo corrispondente di tutte le righe vincolate della tabella interna.
    - **`set null`**: il valore NULL viene assegnato all'attributo corrispondente di tutte le righe vincolate della tabella interna al posto del valore modificato nella tabella esterna.
    - **`set default`**:il valore di default viene assegnato all'attributo corrispondente di tutte le righe vincolate della tabella interna al posto del valore modificato nella tabella esterna.
    - **`no action`**: l'operazione viene rifiutata dal sistema.

Violazione di un vincolo di integrità referenziale

- Operazione di cancellazione di un elemento della tabella esterna:
  - **`cascade`**: tutte le righe corrispondenti della tabella interna vengono eliminate.
  - **`set null`**: il valore NULL viene assegnato nella tabella interna all'attributo corrispondente al posto del valore cancellato nella tabella esterna.
  - **`set default`**: il valore di default viene assegnato nella tabella interna all'attributo corrispondente al posto del valore cancellato nella tabella esterna.
  - **`no action`**: l'operazione di cancellazione viene rifiutata dal sistema.
- L'asimmetria di comportamento in caso di violazione di un vincolo di integrità referenziale è dovuto al fatto che alla tabella esterna viene attribuita maggiore importanza, mentre la tabella interna deve semplicemente adattarsi alle variazioni.



Esempio

```
create table Impiegati
(Matricola integer primary key,
 Nome varchar (20),
 Cognome varchar (20),
 Dipartimento char (15),
 Stipendio numeric (9) default 0,
 Indirizzo varchar (100),
 foreign key (Dipartimento) references Dipartimenti(NomeDip)
                            on delete set null
                           on update cascade,
 unique(Nome,Cognome)
)
```

Modifica di schemi (alter)

- Il comando **alter** permette di modificare domini e tabelle preesistenti:

```
alter domain NomeDominio <set default ValoreDiDefault |
            drop default |
            add constraint Vincolo |
            drop constraint NomeVincolo>
alter table NomeTabella <
          alter column NomeAttributo
              <set default ValoreDiDefault | drop default> |
          add constraint Vincolo |
          drop constraint NomeVincolo |
          add column DefinizioneAttributo |
          drop column NomeAttributo>
```

- Quando si aggiunge un nuovo vincolo i dati esistenti devono soddisfarlo, altrimenti la modifica dello schema viene rifiutata.

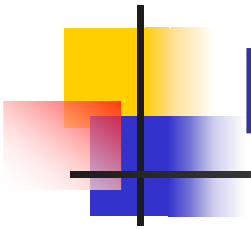
Modifica di schemi (drop)

- Il comando **drop** permette di eliminare dei componenti:

  drop <**schema** | **domain** | **table** | **view** | **assertion**>

  NomeComponente [**restrict** | **cascade**]

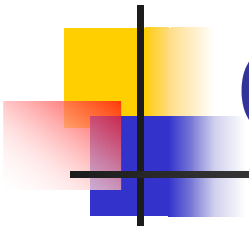
- L'opzione restrict è quella di default ed impedisce l'eliminazione del componente specificato se quest'ultimo non è vuoto oppure è in uso in un altro componente.
- L'opzione **cascade** al contrario, oltre ad eliminare il componente specificato, elimina anche tutti i componenti dipendenti.
- Quando si elimina un dominio definito dall'utente, gli attributi corrispondenti assumono come nuovo dominio quello di base usato nella definizione. Ad esempio se elimino il dominio **LongString** (creato con il comando **create domain LongString as char (100)**), tutti gli attributi aventi come dominio LongString assumono come nuovo dominio **char (100).**



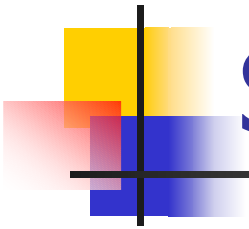
Metadati

- I DBMS relazionali memorizzano e gestiscono la descrizione delle tabelle presenti nei database sempre tramite strutture relazionali, i.e., tabelle. Questo meccanismo prende il nome di *riflessività*.
- Vi sono quindi due tipi di tabelle:
  - Le tabelle che contengono i dati veri e propri.
  - Le tabelle che contengono i metadati (dati che descrivono dati); queste ultime prendono il nome di catalogo della base di dati o dizionario.
- In teoria la riflessività permetterebbe di gestire i metadati con gli stessi comandi che permettono di gestire i dati. In pratica però si preferisce non seguire questo approccio per le seguenti ragioni:
  - Per chiarezza è opportuno distinguere i comandi che operano sugli schemi da quelli che operano sui dati.
  - Il dizionario cambia da sistema a sistema; quindi manipolazioni dirette sarebbero rivolte ad uno specifico sistema senza essere facilmente portabili su un altro DBMS.
  - Le manipolazioni dirette possono essere pericolose se non tengono conto di tutti i passaggi necessari per modificare uno schema; infatti potrebbero produrre degli schemi inconsistenti.

Query in SQL



- Le interrogazioni in SQL sono espresse in modo *dichiarativo*, i.e., basta esprimere le proprietà che devono possedere i dati per estrarli. Ci penserà il DBMS poi a stabilire la *procedura*, i.e., la sequenza di operazioni da eseguire per ottenere il risultato.
- Il DBMS si preoccupa di analizzare la query impartita dall'utente, traducendola nel suo linguaggio procedurale interno ed ottimizzandola.
- Quindi, a differenza di quanto avviene in algebra relazionale, l'utente non deve generalmente preoccuparsi di scrivere query ottimizzate, ma piuttosto leggibili e facili da modificare in caso di necessità.



SELECT

- Le interrogazioni in SQL vengono specificate per mezzo del comando select:

  `select` AttrEspr [[**`as`**] Alias] {, AttrEspr [[**`as`**] Alias]}

  `from` Tabella [[**`as`**] Alias] {, Tabella [[**`as`**] Alias]}

  [**`where`** Condizione]
- Il risultato consiste nella selezione fra le righe appartenenti al prodotto cartesiano (formato dalle tabelle specificate nella clausola **`from`**) di quelle che soddisfano la condizione specificata nella clausola **`where`**. Le colonne incluse nel risultato saranno quelle date dalla valutazione delle espressioni AttrEspr. Gli alias permettono di rinominare sia le colonne che le tabelle coinvolte.
- Il carattere speciale asterisco (**`*`**) può essere usato al posto delle espressioni AttrEspr per estrarre tutte le colonne delle tabelle coinvolte.

Esempio

Impiegati

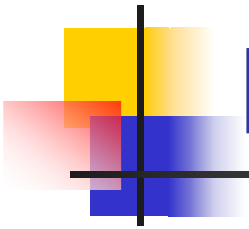
Numero	Nome	Età	Stipendio
101	Mary Smith	34	40000
103	Mary Bianchi	23	35000
104	Luigi Neri	38	61000
105	Nico Bini	44	38000
210	Marco Celli	49	60000
231	Siro Bisi	50	60000
252	Nico Bini	44	70000
301	Steve Smith	34	70000
375	Mary Smith	50	65000

Dipendenze

Capo	Impiegato
210	101
210	103
210	104
231	105
301	210
301	231
375	252

- La query "fornire i numeri di matricola, nomi ed età degli impiegati che guadagnano più di 40000 Euro all'anno" può essere formulata in SQL come segue:

```
select Numero, Nome, Età FROM Impiegati where Stipendio>40000
```



Esempio

- Fornire lo stipendio mensile dell'impiegato Marco Celli:

```
select Stipendio/12 as StipendioMensile
  from Impiegati where Nome='Marco Celli'
```

- Fornire i nomi e gli stipendi dei capi degli impiegati che guadagnano più di 40000 Euro all'anno:

```
select I1.Nome, I1.Stipendio
  from Impiegati as I1, Dipendenze as D,
       Impiegati as I2
 where I1.Numero=D.Capo and
       D.Impiegato=I2.Numero and
       I2.Stipendio>40000
```

- Nella query precedente vi sono tre alias: **`I1`**, **`D`** e **`I2`**. **`D`** è semplicemente uno pseudonimo di Dipendenze (tale tabella compare infatti una sola volta nella clausola **`from`**). **`I1`** e **`I2`** invece sono due variabili in quanto permettono di riferirsi alla stessa tabella (**`Impiegati`**) due volte nella stessa query.

Struttura della SELECT

- La struttura essenziale di una **`select`** è la seguente:

  `select` *ElementiSchema*

  `from` *ListaTabelle*

  `where` *Condizione*

- *ElementiSchema* specifica gli attributi che dovranno far parte della tabella risultato. L'uso dell'asterisco impone che tutti gli attributi di tutte le tabelle elencate nella clausola **`from`** compaiano nel risultato.
- La clausola **`from`** elenca tutte le tabelle coinvolte nella **`select`**.
- Le condizioni indicate nella clausola **`where`** vengono applicate al prodotto cartesiano delle tabelle specificate nella clausola **`from`**.

La clausola WHERE

- La condizione espressa nella clausola where è un'espressione logica costruita combinando delle semplici condizioni atomiche per mezzo degli operatori and, or e not.
- Le condizioni atomiche possono essere costruite applicando degli operatori di confronto ad espressioni formate a partire da nomi di attributi e costanti.
- Gli operatori di confronto disponibili sono: = (uguale), <> (diverso), < (minore), > (maggiore), <= (minore od uguale), >= (maggiore od uguale).
- Per i confronti fra stringhe SQL mette a disposizione l'operatore **`like`**. Grazie alla possibilità di usare i caratteri speciali _ (che rappresenta un carattere qualsiasi) e `%` (che rappresenta una qualunque stringa di caratteri, eventualmente vuota), è possibile utilizzare espressioni come **`'ab%r_'`** (i.e., tutte le stringhe che iniziano con i caratteri **`ab`** e che hanno una **`r`** prima dell'ultimo carattere).

  Esempio:
```
select Età, Stipendio from Impiegati
                    where Nome like '%Bi%'
```

Valori NULL

- A differenza dell'SQL-89 che era basato su una logica a due valori, SQL-2 adotta una logica a tre valori (true, false, unknown) in cui un predicato restituisce il risultato *unknown* se una delle espressioni a cui è applicato contiene il valore NULL.
- SQL mette a disposizione i predicati **`is null`** e **`is not null`** per gestire gli attributi che possono assumere valori nulli:

  NomeAttributo **`is`** [**`not`**] **`null`**

  Le condizioni della clausola **`where`** possono essere costruite anche tenendo conto di questi predicati.

  Esempio:

  ```
  select * from Impiegati where Età is not null
  ```

Duplicati

- Una differenza sostanziale rispetto all'algebra relazionale è che l'SQL permette di avere tabelle con righe che assumono gli stessi valori per tutti gli attributi.
- La ragione di questa scelta è che sarebbe troppo costoso in termini di utilizzo delle risorse del calcolatore implementare un algoritmo che rimuova automaticamente i duplicati.
- Esempio: la query

```
select I1.Nome, I1.Stipendio
 from Impiegati as I1, Dipendenze as D, Impiegati as I2
 where I1.Numero=D.Capo and D.Impiegato=I2.Numero and
       I2.Stipendio>40000;
```

produce il seguente risultato:

Nome	Stipendio
Marco Celli	61000
Steve Smith	60000
Steve Smith	60000
Mary Smith	70000

- Per eliminare i duplicati bisogna utilizzare la parola chiave **`distinct`** (dopo **`select`**).

Inner e outer join

- SQL-2 ha introdotto una sintassi alternativa per la rappresentazione dei join in modo da distinguere questi ultimi dalle altre condizioni presenti nella clausola **`where`**:

  `select` AttrEspr [[**`as`**] Alias] {, AttrEspr [[**`as`**] Alias]}
  `from` Tabella [[**`as`**] Alias] {[TipoJoin] **`join`** Tabella [[**`as`**] Alias]
  `on` CondizioneJoin}
  [**`where`** Condizione]

- TipoJoin può essere **`inner`**, **`right outer`**, **`left outer`** o **`full outer`** (la parola chiave **`outer`** è opzionale).
- In generale, con un **`inner join`** può succedere che alcune righe di una delle tabelle in gioco non vengano incluse nel risultato in quanto non esiste una riga corrispondente nell'altra tabella che soddisfi la condizione specificata. Per ovviare a ciò si può utilizzare un **`outer join`** che rimpiazza l'informazione mancante con dei valori NULL.

Inner e outer join

- Siano date le seguenti relazioni:

Impiegati

Impiegato	**Dipartimento**
Smith	Vendite
Black	Produzione
White	Produzione

Dipartimenti

Dipartimento	**Capo**
Produzione	Mori
Acquisti	Brown

- Se vogliamo i dipartimenti, i loro capi ed i relativi dipendenti (includendo anche i dipartimenti che al momento non ne hanno), useremo un right outer join come segue:

```
select Impiegato, Dipartimenti.Dipartimento, Capo
 from Impiegati right outer join Dipartimenti on
      Dipartimenti.Dipartimento=Impiegati.Dipartimento
```

Impiegato	Dipartimento	Capo
Black	Produzione	Mori
White	Produzione	Mori
NULL	Acquisti	Brown

- Alcuni sistemi per identificare l'outer join fanno seguire un simbolo * o + agli attributi coinvolti. Nel caso dell'esempio precedente si avrebbe quanto segue:

```
select Impiegato, Dipartimenti.Dipartimento, Capo
 from Impiegati, Dipartimenti
 where Dipartimenti.Dipartimento=*Impiegati.Dipartimento
```

Ordinamento delle righe

- Spesso nelle applicazioni è utile ordinare le righe della tabella generata da una query in base ad un certo criterio, soprattutto nel caso in cui il numero di righe totali sia molto elevato.
- SQL viene incontro a questa esigenza dell'utente mettendo a disposizione la clausola **`order by`**:

  order by AttributoDiOrdinamento [asc | desc]

  {, AttributoDiOrdinamento [asc | desc]}
- Le righe vengono ordinate in base al valore assunto in corrispondenza del primo attributo specificato e, a parità di valore, in base agli attributi successivi.

Ordinamento delle righe

- Ad esempio, per avere la lista di tutte le righe della tabella Impiegati ordinate in base all'età (ordinamento ascendente) ed allo stipendio (ordinamento discendente) si esegue la seguente query:

```
select * from Impiegati order by Età, Stipendio desc
```

Numero	Nome	Età	Stipendio
103	Mary Bianchi	23	35000
301	Steve Smith	34	70000
101	Mary Smith	34	40000
104	Luigi Neri	38	61000
252	Nico Bini	44	70000
105	Nico Bini	44	38000
210	Marco Celli	49	60000
375	Mary Smith	50	65000
231	Siro Bisi	50	60000

Operatori aggregati

- Gli operatori aggregati in SQL consentono di valutare proprietà che dipendono da un insieme di righe.
- In algebra relazionale al contrario tutte le condizioni sono valutate su ogni tupla **singolarmente**.
- Un tipico esempio di operatore aggregato è **`count`** che consente di calcolare il numero delle tuple che soddisfano una data condizione:

```
select count(*) as Cinquantenni from Impiegati
where Età=50
```

La query precedente fornisce il seguente risultato:

Cinquantenni
2

L'operatore **count**

- L'operatore aggregato count può essere utilizzato con la sintassi seguente:

  count (<* | [**distinct** | **all**] ListaAttributi>)
- Il carattere ***** restituisce il numero di righe.
- La parola chiave **distinct** specifica che deve essere restituito il numero di righe che hanno valori distinti relativamente agli attributi specificati in ListaAttributi.
- La parola chiave **all** (default) invece specifica che deve essere restituito il numero di righe che assumono valori diversi da NULL relativamente agli attributi specificati in ListaAttributi.
- Esempio:

```
select count(distinct Età) from Impiegati
```

Altri operatori aggregati

- <**sum** | **max** | **min** | **avg**> (**[distinct** | **all**] AttrEspr)
- **sum** e **avg** ammettono come espressioni o attributi soltanto quelli che rappresentano valori numerici o intervalli di tempo.
- **max** e **min** richiedono che sui valori dell'espressione o attributo forniti come parametri sia definito un ordinamento.
- Tutti questi operatori si applicano alle righe che soddisfano la condizione della clausola where e forniscono i seguenti risultati:
  - **sum**: restituisce la somma dei valori corrispondenti a AttrEspr;
  - **max** e **min**: restituiscono, rispettivamente il valore massimo e quello minimo di AttrEspr;
  - **avg**: restituisce la media dei valori di AttrEspr.

Esempi

- `select sum(Stipendio) as SpesaStipendi from Impiegati`

SpesaStipendi
499000

- `select avg(Età) as EtàMedia from Impiegati`

EtàMedia
40,6666666666667

- `select min(Età) as Giovane, max(Età) as Vecchio from Impiegati`

Giovane	Vecchio
23	50

Esempio di query non corretta

- La query

  ```
  select Nome, max(Stipendio) from Impiegati
  ```

  non è corretta.
- Intuitivamente, si vorrebbe che la query precedente fornisca lo stipendio più alto unitamente al nome dell'impiegato che lo percepisce.
- Tuttavia vi sono due problemi che impediscono di eseguire l'interrogazione:
  - Potrebbero esserci più di due righe con il valore massimo per la colonna Stipendio ed il sistema non avrebbe modo di scegliere.
  - Se anche non sussistesse il problema precedente, interrogazioni del genere potrebbero aver senso per min e max, ma non per i rimanenti operatori aggregati.
- Quindi non è possibile specificare attributi (od espressioni di attributi) con operatori aggregati come nel precedente esempio.

La clausola `group by`

- Spesso è necessario avere un controllo più fine di quello offerto dagli operatori aggregati. Infatti questi ultimi si applicano a tutte le righe prodotte dalla query, mentre sarebbe a volte utile che venissero applicati a dei sottoinsiemi di righe singolarmente.
- Per ottenere questo scopo, i.e., suddividere le righe risultato di una query in sottoinsiemi si usa la clausola **`group by`**:

```
select Dipartimento, sum(Stipendio)
from Impiegati group by Dipartimento
```

Impiegati

Nome	Cognome	Dipartimento	Ufficio	Stipendio	Città
Mary	Brown	Amministrazione	10	45000	Londra
Jacques	Dupont	Produzione	20	36000	Tolosa
David	Johnson	Amministrazione	20	40000	New York
Mario	Rossi	Distribuzione	16	45000	Roma
Charles	Brown	Planning	14	80000	Londra
Kurt	Maier	Planning	7	73000	Monaco
Stefan	Mortensen	Amministrazione	75	40000	Stoccolma
Alice	Duval	Produzione	20	46000	Tolosa

Risultato della query:

Dipartimento	Expr1001
Amministrazione	125000
Distribuzione	45000
Planning	153000
Produzione	82000

La clausola `group by`

- Usando la clausola group by è quindi possibile combinare operatori aggregati e attributi nella target list di una **`select`**.
- Tuttavia SQL impone dei vincoli sintattici agli attributi che possono essere utilizzati congiuntamente nella target list della **`select`**; infatti essi devono apparire nella clausola **`group by`**. Quindi la seguente query **non** è corretta (infatti ci sono più uffici per ogni dipartimento):

```
select Ufficio from Impiegati group by Dipartimento
```

- A volte tale vincolo appare troppo restrittivo:

Impiegati

Impiegato	Dipartimento
Black	Produzione
Smith	Vendite
White	Produzione

Dipartimenti

Dipartimento	Capo
Vendite	Brown
Produzione	Mori

La chiave primaria di Dipartimenti è Dipartimento. La query

```
select D.Dipartimento, count(*), D.Capo
from Impiegati I inner join Dipartimenti D on
(I.Dipartimento=D.Dipartimento) group by D.Dipartimento
```

è scorretta anche se, intuitivamente, è possibile associare un unico capo ad ogni dipartimento (visto che D.Dipartimento è chiave primaria di Dipartimenti).

La clausola `group by`

- In casi come quello precedente si dice che gli attributi che si vogliono ottenere sono funzionalmente dipendenti da quelli specificati nella clausola **`group by`**. In altre parole possono assumere un unico valore in corrispondenza dei singoli valori specificati per il raggruppamento.
- Modificare SQL per permettere questi casi comporterebbe una complicazione non banale del linguaggio. Quindi si è preferito costringere l'utente a specificare un insieme ridondante di attributi nella clausola **`group by`** per ottenere il risultato desiderato.
- Ad esempio, nel caso precedente la query corretta è:

```
select D.Dipartimento, count(*), D.Capo from
Impiegati I inner join Dipartimenti D on
(I.Dipartimento=D.Dipartimento) group by
D.Dipartimento, D.Capo
```

Dipartimento	Expr1001	Capo
Produzione	2	Mori
Vendite	1	Brown

La clausola **`having`**

- La clausola **`group by`** permette di suddividere le righe in sottoinsiemi per poi applicare ad ognuno di questi un operatore aggregato. In qualche caso può essere utile applicare gli operatori aggregati a sottoinsiemi che soddisfino una certa proprietà.
- Se la proprietà in questione è verificabile per ogni riga presa singolarmente, allora può essere inclusa nella clausola **`where`**.
- Quando al contrario la proprietà deve essere soddisfatta dal risultato fornito da un operatore aggregato, allora si rende necessaria la clausola **`having`**.
- Quindi nella clausola **`having`** vanno specificate le condizioni da applicare al termine dell'esecuzione di una query che faccia uso di **`group by`**.

Esempi

- **`select Dipartimento from Impiegati group by Dipartimento having sum(Stipendio)>100000`**

Impiegati

Nome	Cognome	Dipartimento	Ufficio	Stipendio
Mary	Brown	Amministrazione	10	45000
Jacques	Dupont	Produzione	20	36000
David	Johnson	Amministrazione	20	40000
Mario	Rossi	Distribuzione	16	45000
Charles	Brown	Planning	14	80000
Kurt	Maier	Planning	7	73000
Stefan	Mortensen	Amministrazione	75	40000
Alice	Duval	Produzione	20	46000

Risultato della query:

Dipartimento
Amministrazione
Planning

- **`select Dipartimento from Impiegati where Ufficio='20' group by Dipartimento having avg(Stipendio)>25000`**

Risultato della query:

Dipartimento
Amministrazione
Produzione

La clausola **`having`**

- Se è presente la clausola **`having`**, ma non quella **`group by`**, allora tutte le righe vengono considerate come un unico sottoinsieme. In questo caso se la condizione specificata non è soddisfatta, la tabella risultante sarà vuota.
- In generale la proprietà specificata nella clausola **`having`** può essere costruita seguendo le stesse regole sintattiche relative alla condizione della clausola **`where`**.
- Tuttavia, nonostante SQL permetta di utilizzare predicati sugli attributi che compaiono nella clausola **`group by`**, è preferibile specificare questi ultimi nella clausola **`where`**. In sostanza la regola da seguire è che nella clausola **`having`** vanno specificate soltanto proprietà costruite a partire da predicati su operatori aggregati.

Operatori insiemistici

- SQL mette a disposizione tre operatori che svolgono lo stesso ruolo di quelli insiemistici dell'algebra relazionale:
  - **`union`** (unione)
  - **`intersect`** (intersezione)
  - **`except`** o **`minus`** (differenza)
- Ciò che contraddistingue maggiormente gli operatori insiemistici dal resto dei costrutti di SQL è l'automatica eliminazione dei duplicati.
- Un'ulteriore differenza è che SQL non richiede che gli schemi delle relazioni (tabelle) siano uguali; infatti è sufficiente che i domini degli attributi siano compatibili.
- La sintassi è la seguente:

  select1 {<**`union`** | **`intersect`** | **`except`**> [**`all`**] *select2*}

  La parola chiave all (se specificata) consente di includere nel risultato anche gli eventuali duplicati.

Esempio

- ```
  select Nome from Impiegati
  union
  select Cognome from Impiegati
  ```

Nome	Cognome	Dipartimento	Ufficio	Stipendio
Mary	Brown	Amministrazione	10	45000
Jacques	Dupont	Produzione	20	36000
David	Johnson	Amministrazione	20	40000
Mario	Rossi	Distribuzione	16	45000
Charles	Brown	Planning	14	80000
Kurt	Maier	Planning	7	73000
Stefan	Mortensen	Amministrazione	75	40000
Alice	Duval	Produzione	20	46000

Impiegati

Nome
Alice
Brown
Charles
David
Dupont
Duval
Jacques
Johnson
Kurt
Maier
Mario
Mary
Mortensen
Rossi
Stefan
  ```

Esempio

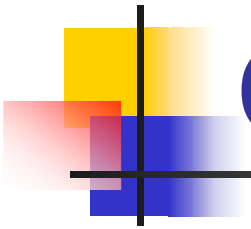
```
select Nome from Impiegati where
Dipartimento<>'Produzione'
union all
select Cognome from Impiegati where
Dipartimento<>'Produzione'
```

Nome	Cognome	Dipartimento	Ufficio	Stipendio
Mary	Brown	Amministrazione	10	45000
Jacques	Dupont	Produzione	20	36000
David	Johnson	Amministrazione	20	40000
Mario	Rossi	Distribuzione	16	45000
Charles	Brown	Planning	14	80000
Kurt	Maier	Planning	7	73000
Stefan	Mortensen	Amministrazione	75	40000
Alice	Duval	Produzione	20	46000

Impiegati

Nome
Mary
David
Mario
Charles
Kurt
Stefan
Brown
Johnson
Rossi
Brown
Maier
Mortensen

Query nidificate



- Generalmente la condizione della clausola **`where`** è costituita da predicati che stabiliscono un confronto fra due valori.
- SQL permette anche di eseguire dei confronti (all'interno della clausola **`where`**) tra un valore ed il risultato di una query (nidificata).
- Siccome nei casi di query nidificate il confronto avviene generalmente fra un nome di attributo (riferito ad una singola riga) ed un insieme di valori (il risultato della query nidificata), si rende necessario estendere gli usuali operatori di confronto con le parole chiave **`all`** o **`any`**.
- La parola chiave **`all`** specifica che il confronto con il valore dell'attributo è vero se è soddisfatto da tutti gli elementi restituiti dalla query nidificata.
- La parola chiave **`any`** specifica che il confronto con il valore dell'attributo è vero se è soddisfatto da almeno un elemento restituito dalla query nidificata.

Esempio

Impiegati

Nome	Cognome	Dipartimento	Ufficio	Stipendio
Mary	Brown	Amministrazione	10	45000
Jacques	Dupont	Produzione	20	36000
David	Johnson	Amministrazione	20	40000
Mario	Rossi	Distribuzione	16	45000
Charles	Brown	Planning	14	80000
Kurt	Maier	Planning	7	73000
Stefan	Mortensen	Amministrazione	75	40000
Alice	Duval	Produzione	20	46000
Alice	Smith	Distribuzione	4	32000

Dipartimenti

Nome	Indirizzo	Città
Amministrazione	Via De Gasperi, 89	Roma
Produzione	Via Roma, 56	Milano
Distribuzione	Viale dei Platani, 78	Firenze
Planning	Via Milano, 19	Torino

Risultato

Nome	Cognome	Dipartimento	Ufficio	Stipendio
Charles	Brown	Planning	14	80000
Kurt	Maier	Planning	7	73000

- Query che fornisce gli impiegati che lavorano in dipartimenti situati a Torino:

```
select * from Impiegati where Dipartimento = any (select
Nome from Dipartimenti where Città='Torino')
```

Esempio

Impiegati

Nome	Cognome	Dipartimento	Ufficio	Stipendio
Mary	Brown	Amministrazione	10	45000
Jacques	Dupont	Produzione	20	36000
David	Johnson	Amministrazione	20	40000
Mario	Rossi	Distribuzione	16	45000
Charles	Brown	Planning	14	80000
Kurt	Maier	Planning	7	73000
Stefan	Mortensen	Amministrazione	75	40000
Alice	Duval	Produzione	20	46000
Alice	Smith	Distribuzione	4	32000

Risultato

Nome	Cognome
Jacques	Dupont
Alice	Smith
Alice	Duval

- Query che fornisce nomi e cognomi di impiegati che hanno lo stesso nome di un impiegato del dipartimento Produzione:

```
select I1.Nome, I1.Cognome from Impiegati as I1, Impiegati
as I2 where I1.Nome=I2.Nome and
I2.Dipartimento='Produzione'
```

- Formulazione con query nidificata:

```
select Nome, Cognome from Impiegati where Nome= any
(select Nome from Impiegati where
Dipartimento='Produzione')
```

Esempio

- Query che fornisce i nomi dei dipartimenti in cui non lavorano impiegati con cognome Rossi:

```
select Nome from Dipartimenti where Nome <> all
(select Dipartimento from Impiegati where
Cognome='Rossi')
```

Risultato

Nome
Amministrazione
Produzione
Planning

- Query equivalente che fa uso dell'operatore insiemistico except:

```
select Nome from Dipartimenti except select
Dipartimento from Impiegati where
Cognome='Rossi'
```

Esempio

- L'effetto degli operatori aggregati max e min può essere simulato tramite operazioni nidificate.
- La query seguente fornisce il nome del dipartimento in cui lavora l'impiegato che guadagna di più:

```
select Dipartimento from Impiegati where
Stipendio = (select max(Stipendio) from
Impiegati)
```

Risultato

Dipartimento
Planning

- La query precedente può essere formulata in modo da fornire lo stesso risultato utilizzando soltanto una query nidificata:

```
select Dipartimento from Impiegati where
Stipendio >= all (select Stipendio from
Impiegati)
```

Appartenenza ad insiemi

- SQL fornisce due operatori (**`in`** e **`not in`**) in modo da permettere all'utente di scrivere query che consentano di esprimere condizioni di appartenenza ad insiemi.
- In realtà **`in`** e **`not in`** sono abbreviazioni sintattiche per, rispettivamente **`= any`** e **`<> all`**.
- Esempio: la query che fornisce i nomi dei dipartimenti in cui non lavorano impiegati con cognome Rossi:

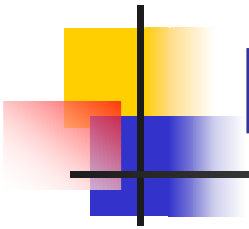
```
select Nome from Dipartimenti where Nome <> all
(select Dipartimento from Impiegati where
Cognome='Rossi')
```

può essere riscritta come segue:

```
select Nome from Dipartimenti where Nome not in
(select Dipartimento from Impiegati where
Cognome='Rossi')
```

Query nidificate complesse

- L'intuizione che sta dietro alle query nidificate, se si assume che queste ultime vengano eseguite prima di analizzare le righe delle query esterne, è che il risultato di una query nidificata venga memorizzato in una locazione temporanea, in modo da valutare la condizione della query esterna utilizzando il risultato temporaneo.
- Tutto ciò ha il vantaggio di produrre esecuzioni efficienti delle query nidificate in quanto la query più interna viene eseguita una sola volta ed il suo risultato viene utilizzato per valutare la condizione della query esterna su ognuna delle righe di quest'ultima.
- Tuttavia può capitare che la query nidificata faccia riferimento al contesto della query che le racchiude tramite una variabile definita in quest'ultima (*binding transfer*, i.e., trasferimento di legame).
- In questo caso, l'interpretazione corretta delle query presuppone che si esegua dapprima il prodotto cartesiano delle tabelle specificate nella query esterna e poi si valuti la condizione della clausola **`where`** su ognuna delle righe ottenute. Quindi la query nidificata deve essere eseguita nuovamente per ogni riga.



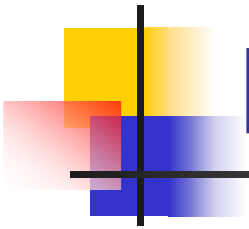
Esempio

- Data la tabella con schema Persone(Nome, Cognome, CodiceFiscale), trovare tutti gli omonimi, i.e., tutte le persone con lo stesso nome e lo stesso cognome, ma con codici fiscali differenti:

```
select * from Persone as P
where exists (select * from Persone as P1
              where P1.Nome=P.Nome and
                    P1.Cognome=P.Cognome and
                    P1.CodiceFiscale<>P.CodiceFiscale)
```

- L'operatore **`exists`** restituisce vero se la query a cui è applicato produce un risultato non vuoto, restituisce falso altrimenti.
- Data la tabella con schema Persone(Nome, Cognome, CodiceFiscale), trovare tutte le persone che non hanno omonimi:

```
select * from Persone as P
where not exists (select * from Persone as P1
              where P1.Nome=P.Nome and
                    P1.Cognome=P.Cognome and
                    P1.CodiceFiscale<>P.CodiceFiscale)
```



Modifica di dati in SQL

- SQL permette di eseguire, oltre alle interrogazioni, anche operazioni di modifica dei dati contenuti nel database.
- I comandi specifici per modificare i dati sono i seguenti:
  - **`insert`**: permette di aggiungere nuove righe ad una tabella.
  - **`delete`**: permette di cancellare delle righe da una tabella.
  - **`update`**: permette di modificare i dati contenuti nelle righe di una tabella.

Inserimenti

- La sintassi generale per l'inserimento di nuove righe in una tabella ha la seguente sintassi:

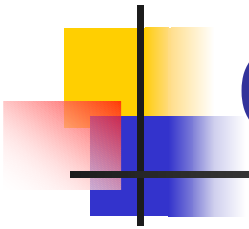
  `insert into` NomeTabella [**`(`**ListaAttributi**`)`**] <**`values (`**ListaValori**`)`** | selectSQL>
- La prima alternativa consente di inserire delle righe singolarmente, specificando di volta in volta i valori della nuova riga:

```
insert into Dipartimenti (Nome, Indirizzo, Città)
values ('Ricerca e sviluppo','Via Monte Napoleone,
67', 'Milano')
```

- La seconda alternativa consente di inserire un insieme di righe recuperate dal database tramite un'interrogazione:

```
insert into ImpiegatiProduzione (select Nome,
Cognome, Ufficio, Stipendio from Impiegati where
Dipartimento='Produzione')
```

- Nel caso in cui non vengano specificati tutti gli attributi in un'istruzione **`insert`**, a quelli mancanti viene assegnato il valore di default oppure il valore NULL, se non esiste un valore di default.



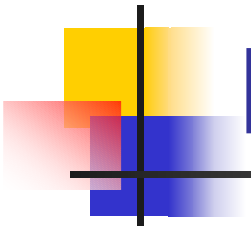
Cancellazioni

- La sintassi dell'istruzione di cancellazione è la seguente:

  `delete from` NomeTabella [**`where`** Condizione]
- Se la clausola **`where`** è assente, vengono cancellate tutte le righe della tabella specificata, altrimenti soltanto quelle che soddisfano la condizione.
- Nel caso in cui vi sia un vincolo referenziale con una politica di tipo cascade per le cancellazioni, eliminare delle righe nella tabella può provocare una reazione a catena in tutte le tabelle collegate.
- La seguente query cancella tutte le righe della tabella Dipartimenti:

  ```
  delete from Dipartimenti
  ```
- La seguente query cancella tutte le righe della tabella Dipartimenti che corrispondono a dipartimenti senza impiegati:

  ```
  delete from Dipartimenti where Nome not in (select Dipartimento from Impiegati)
  ```
- Si noti che l'istruzione delete cancella righe di dati, ma lascia inalterato lo schema del database.



Modifiche

- La sintassi dell'istruzione per le modifiche è la seguente:

  `update` NomeTabella

  `set` Attributo = <Espressione | selectSQL | **`null`** | **`default`**>

  {, Attributo = <Espressione | selectSQL | **`null`** | **`default`**>}

  [**`where`** Condizione]
- Se la clausola where non è presente, le modifiche vengono apportate a tutte le righe della tabella specificata, altrimenti soltanto a quelle che soddisfano Condizione.
- Il nuovo valore assegnato a Attributo può essere:
  - un'espressione,
  - il risultato di una query SQL,
  - il valore null,
  - il valore di default.

Esempi di modifiche

- Aumento di 5000 Euro dello stipendio dell'impiegato Mario Rossi:

```
update Impiegati set Stipendio=Stipendio+5000 where
Nome='Mario' and Cognome='Rossi'
```

- Aumento del 20% dello stipendio degli impiegati del dipartimento Produzione:

```
update Impiegati set Stipendio=Stipendio*1.2 where
Dipartimento='Produzione'
```

- Volendo aumentare del 10% lo stipendio degli impiegati che guadagnano 30000 Euro o meno e del 15% quello degli impiegati che guadagnano più di 30000 Euro, la seguente sequenza di query non produce il risultato desiderato:

```
update Impiegati set Stipendio=Stipendio*1.1 where
Stipendio<=30000

update Impiegati set Stipendio=Stipendio*1.15 where
Stipendio>30000
```

  Infatti, un impiegato con uno stipendio di 28000 Euro, dopo la prima query vedrebbe il proprio salario aumentato a 30800 Euro. Quindi soddisferebbe la clausola della seconda query e si vedrebbe aumentare ulteriormente lo stipendio del 15% (+4620 Euro) per un totale del 26,5%.