4 - SQL

SQL è il linguaggio di riferimento per le basi di dati relazionali.
Nel tempo la Structured Query Language ha subito diverse iterazioni e standardizzazioni, fino ad arrivare ad SQL-3 ma l'utilizzo di riferimento in questo caso sarà SQL-2

Definizione dei dati in SQL (Creazione di una base di dati)

Prima di cominciare il capitolo, qualche accenno sulla sintassi che verrà utilizzata:

- Le **parentesi angolari** $\langle\rangle$ permettono di isolare un termine della sintassi
- Le **parentesi quadre** indicano che il termine all'interno è opzionale, ossia può non comparire oppure comparire una sola volta
- Le **parentesi graffe** indicano che il termine racchiuso può non comparire o essere ripetuto un numero arbitrario di volte
- Le **barre verticali** indicano che deve essere scelto uno tra i termini separati dalle barre (un elenco di termini in alternativa può essere racchiuso tra parentesi angolari)

Le parentesi tonde dovranno sempre essere intese come termini del linguaggio SQL, non come simboli della definizione della grammatica

I domini elementari

SQL mette a disposizione alcune famiglie di domini elementari, da cui si possono poi definire i domini da associare agli attributi dello schema

Caratteri

Il dominio $\mathrm{CHARACTER}$ permette di rappresentare singoli caratteri oppure stringhe.
La lunghezza può essere fissa o variabile, in caso di quelle variabile si esplicita la lunghezza massima, oltre a poter prevedere una specifica della famiglia dei caratteri di default (latino, cirillico etc.).
La sintassi prevista è:

$$\mathrm{CHARACTER}\ [\mathrm{varying}][(Lunghezza)]$$
$$[\mathrm{CHARACTER\ SET}]NomeFamigliaCaratteri$$

Tipi numerici esatti

Questa famiglia contiene i domini che permettono di rappresentare valori esatti, interi o con una parte decimale di lunghezza prefissata.
La sintassi prevista è:

$$\text{NUMERIC}[(Precisione[, Scala])]$$
$$\text{DECIMAL}[(Precisione[, Scala])]$$
$$\text{INTEGER}$$
$$\text{SMALLINT}$$

I numeri NUMERIC e DECIMAL rappresentano tutti i numeri in base decimale, mentre il parametro $Precisione$ specifica il numero di cifre significative (con un dominio $\text{DECIMAL}(4)$ si possono rappresentare per esempio valori da -9999 a +9999).
Il parametro Scala si specifica la scala di rappresentazione (ossia quante cifre compaiono dopo la virgola nella scala di rappresentazione, tutto ciò sempre specificando la precisione), se non specificata è sempre 0.
La differenza tra DECIMAL e NUMERIC consiste nella precisione, per il primo ci deve essere un requisito minimo, per il secondo invece si rappresenta un valore esatto.
Nel caso in cui non interessi la parte frazionaria e quindi la precisione della rappresentazione decimale, allora si possono usare i domini INTEGER e SMALLINT, che si basano sulla rappresentazione interna binaria del calcolatore (la precisione è lasciata all'implementazione)
Qualora comunque la precisione non sia specificata, il sistema usa un valore caratteristico della implementazione.

Tipi numerici approssimativi

Per la rappresentazione di valori reali approssimativi SQL fornisce i seguenti tipi:

$$\text{FLOAT}[(Precisione)]$$
$$\text{REAL}$$
$$\text{DOUBLE PRECISION}$$

Tutti questi domini ovviamente permettono di descrivere numeri approssimati mediante rappresentazione con virgola mobile, in cui ciascun numero corrisponde ad una coppia di valori: mantissa e esponente.
Per ottenere un valore approssimativo del numero reale si moltiplica la mantissa per la potenza di 10 con il grado dell'esponente (esempio: $1,7 \times 10^{15}$ rappresenta $0.17E16$, oppure -4×10^{-7} rappresenta $-0.4E-6$).
Al dominio FLOAT può essere associata una precisione che rappresenta il numero delle cifre dedicate alla mantissa, la precisione invece dell'esponente è dipendente dall'implementazione.
La precisione del dominio REAL è fissa, quella del dominio DOUBLE PRECISION è di dimensione doppia rispetto al dominio precedente

Istanti temporali

Per la rappresentazione di istanti di tempo in SQL usiamo questa famiglia di domini con i seguenti tipi:

$$\text{DATE}$$
$$\text{TIME}[(Precisione)][with\ time\ zone]$$
$$\text{TIMESTAMP}[(Precisione)][with\ time\ zone]$$

Ciascuno di questi domini è strutturato e decomponibile con un insieme di campi:

- Il dominio DATE ammette i campi year, month, day
- Il dominio TIME ammette i campi hour,minute,$\mathrm{seconds}$
- Il dominio $\mathrm{TIMESTAMP}$ ammette tutti i campi precedentemente descritti
  Se l'opzione $with\ time\ zone$ è specificata allora risulta possibile accedere a due campi, $\mathrm{timezone\_hour}$ e $\mathrm{timezone\_minute}$, che rappresentano la differenza tra il fuso orario locale e quello standard (standard UTC)

Intervalli temporali

Questa famiglia di domini permette di rappresentare intervalli di tempo (come la durata di un evento), la sintassi è:

$$\mathrm{INTERVAL}\ PrimaUnitàDiTempo[\mathrm{to}\ UltimaUnitàdiTempo]$$

$\mathrm{PrimaUnitàDiTempo}$ e$\mathrm{UltimaUnitàDiTempo}$ definiscono le unità di misure, dalla più precisa alla meno precisa, in questo modo si possono definire domini come $\mathrm{interval\ year\ to\ month}$ per indicare per indicare che la durata di intervallo deve essere misurata in numero di anni e di mesi.
Notiamo che comunque ci sono due insieme distintivi nelle unità di misura: $\mathrm{year\ to\ month}$ e $\mathrm{day\ to\ seconds}$, questo perché non si possono paragonare per esempio giorni e mesi in modo esatto (un mese potrebbe avere dai 28 ai 31 giorni), rendendo difficile eventuali operazioni aritmetiche.

Definizioni di schema

SQL consente la definizione di uno schema di base di dati come collezione di oggetti (tabelle, domini, viste etc.).
Uno schema viene definito dalla seguente sintassi:

$$\mathrm{CREATE\ SCHEMA}\ [NomeSchema]\ [[authorization]\mathrm{Autorizzazione}]\ \{\mathrm{DefinizioneElementoSchema}\}$$

$\mathrm{Autorizzazione}$ rappresenta il nome dell'utente proprietario dello schema, se omesso si assume che chi abbia lanciato il comando sia il proprietario, $NomeSchema$ viene poi rinominato con lo stesso nome del proprietario.
Dopo il comando $\mathrm{CREATE\ SCHEMA}$ compaiono le definizioni dei suoi componenti, ma non è necessario che questa definizione avvenga contemporaneamente alla creazione dello schema.

Definizioni di tabelle

Una tabella SQL è costituita da una collezione ordinata di attributi e da un insieme (eventualmente vuoto) di vincoli.
Lo schema della tabella $\mathrm{DIPARTIMENTO}$ viene definita per esempio tramite la seguente istruzione SQL:

```
CREATE TABLE Dipartimento
(
    Nome varchar(20) PRIMARY KEY,
    Indirizzo varchar(50),
    Città varchar(20)
)
```

Definendolo quindi con uno schema più generale:

$$\begin{aligned}&\text{CREATE TABLE } NomeTabella\\&(Nomeattributo\ Dominio[ValoreDiDefault][Vincoli])\\&\quad\{,NomeAttributoDominio[ValoreDiDefault][Vincoli]\}\\&AltriVincoli\end{aligned}$$

Ogni tabella viene quindi definita associandole un nome ed elencando gli attributi che compongono lo schema.
Per ogni attributo abbiamo un nome, dominio ed eventuali insiemi di vincoli che devono essere rispettati dai valori dell'attributo.
Dopo la definizione degli attributi si possono definire i vincoli che coinvolgono più attributi della tabella.

Definizione di domini

Nella definizione delle tabelle si può fare riferimento ai domini predefiniti del linguaggio o a domini definiti dall'utente a partire da quelli predefiniti, infatti proprio da questi è possibile definirli in questa maniera:

$$\begin{aligned}&\text{CREATE DOMAIN } NomeDominio\ as\ TipoDiDato\\&[ValoreDiDefault]\\&[Vincolo]\end{aligned}$$

Un dominio quindi è caratterizzato dal proprio nome, un dominio elementare (predefinito o definito dall'utente in precedenza), da un eventuale valore di default e da un insieme di vincoli (eventualmente vuoti) per gli eventuali valori del dominio.
Definendo un dominio si può rendere più facile la modifica della definizione, infatti se si vuole modificare la definizione di attributi in uno stesso dominio sarà necessario soltanto modificare quest'ultimo e si applicherà a tutte le tabelle del dominio.

Specifica valori di default

Il termine $ValoreDiDefault$ nei domini e nelle tabelle permette di specificare un valore predefinito quando viene inserito un attributo in una riga della tabella senza specificare un valore.
In modo predefinito il valore di default risulta sempre nullo.
La sintassi specifica è la seguente:

$\text{DEFAULT}\langle GenericoValore|user|NULL\rangle$

- $GenericoValore$ rappresenta un valore compatibile con il dominio
- $user$ impone come valore di default l'identificativo dell/ utente che esegue il comando
- $NULL$ corrisponde al valore di default base

Quando un attributo o un dominio è definito a a partire da un altro a cui è stato già specificato un valore di default automaticamente quello ha la maggiore priorità, diventando valore effettivo.

Vincoli intrarelazionali

Sia nella definizione di domini che di tabelle è possibile definire dei vincoli, ovvero delle proprietà che devono essere verificate da ogni istanza della base di dati.
Ricordiamo che i vincoli intrarelazionali coinvolgono una sola relazione su un unico attributo.

Not Null

Il valore nullo come sappiamo è un particolare valore che indica l'assenza di informazioni, ma SQL non permette la distinzione dei diversi casi, per questo bisogna avere delle soluzioni ad-hoc, come l'introduzione di altri attributi o l'uso di particolare codifica.
Il vincolo NOT NULL indica che il valore nullo non è ammesso come valore dell'attributo e deve essere necessariamente specificato in fase di inserimento (ma anche successivamente), ma nel caso sia presente un valore di default non è necessario l'inserimento forzato.
Un esempio può essere:

```
Cognome varchar(20) NOT NULL
```

Unique

Il vincolo UNIQUE si applica ad un attributo o a un insieme di attributi di una tabella e impone che i valori (o le n-uple dei valori sull'insieme degli attributi) siano una superchiave, ossia per tutte le righe differenti della tabella non ci siano gli stessi valori.
Un'eccezione viene fatta per il valore nullo, in quanto si assume che siano tutti diversi tra loro.
La definizione del vincolo può avvenire in due modi:

1. Quando si vuole specificare questo vincolo su un unico attributo, in quel caso viene dichiarato nella specifica di quell'attributo:

```
Matricola character(6) UNIQUE
```

2. Quando avviene su un insieme di attributi in una tabella, usando la sintassi seguente:

$$UNIQUE(Attributo, \{, Attributo\})$$

Un esempio di sintassi è il seguente:

```
Nome varchar(20) NOT NULL,
Cognome varchar(20) NOT NULL,
UNIQUE (Cognome,Nome)
```

Si noti che si potrebbe pensare che la definizione:

```
Nome varchar(20) NOT NULL UNIQUE
Cognome varchar(20) NOT NULL UNIQUE
```

sia uguale in modo logico, ma in non lo è, nel primo caso si presuppone che non ci siano righe uguali con nome e cognome uguale, nel secondo caso invece si presuppone che non esistano o lo stesso nome o lo stesso cognome ripetuto più di una volta

Primary Key

SQL permette di specificare il vincolo $\mathrm{PRIMARY\ KEY}$ soltanto una volta per tabella e per singolo attributo o più attributi che costituiscono l'identificatore.
Gli attributi che fanno parte della chiave primaria non possono essere nulli, quindi si implica che ci sia una definizione $\mathrm{NOT\ NULL}$ omessa.
Un esempio di dichiarazione può essere:

```
Nome varchar(20),
Cognome varchar(20),
PRIMARY KEY (Cognome, Nome)
```

Vincoli interrelazionali

I vincoli interrelazionali più diffusi e significativi sono i **vincoli di integrità referenziale**, in SQL per loro definizione viene usato il vincolo di $\mathrm{FOREIGN\ KEY}$, chiamato anche **chiave esterna**;
Questa chiave esterna crea un legame tra i valori di un attributo della tabella su cui è definito (chiamata **interna**) e i valori di attributo di un altra tabella (chiamata **esterna**).
Il vincolo impone che per ogni riga della tabella interna il valore dell'attributi specificato (se diverso da nullo) sia presente nelle righe della tabella esterna tra i valori del corrispondente attributo;
Questo vincolo ha come unico requisito che la sintassi dell'attributo a cui si fa riferimento alla tabella esterna sia soggetto ad UNIQUE, ossia che questo sia un identificatore, infatti

tipicamente la chiave esterna fa riferimento la chiave primaria della tabella.
Nel vincolo possono essere coinvolti più attributi, in tal caso l'unica differenza è che bisognerà confrontare n-uple di valori piuttosto di singoli valori.

Possiamo definirlo in due modi:

1. Nel caso sia un unico attributo coinvolto si può usare il costrutto sintattico REFERENCES, con il quale si specificano la tabella esterna e l'attributo della tabella esterna:

```
CREATE TABLE Impiegato
(
Matricola character(6) PRIMARY KEY
Nome varchar(20) NOT NULL,
Cognome varchar(20) NOT NULL,
Dipart varchar(15)
       REFERENCES Dipartimento(NomeDip)
Ufficio numeric(9) DEFAULT 0,
UNIQUE(Cognome,Nome)
)
```

2. Nel caso ci sia un insieme di attributi si utilizza FOREIGN KEY, posto al termine della definizione degli attributi

```
FOREIGN KEY (Nome,Cognome)
            REFERENCES Anagrafica(Nome,Cognome)
```

La corrispondenza tra gli attributi locali e quelli esterni avviene in base all'ordine, infatti il primo attributo corrispondente a FOREIGN KEY corrisponde al primo argomento di REFERENCES e via via gli altri attributi.

Per tutti i vincoli visti finora quando il sistema rileva una violazione il comando di aggiornamento viene rifiutato segnalando l'errore all'utente, invece per quelli referenziali SQL permette di scegliere altre reazioni da adottare quando viene rilevata una violazione, per esempio una violazione può essere la modifica del contenuto della tabella interna in due modi:

- Inserire una nuova riga
- Modificare il valore dell'attributo referente
  In questo caso non viene offerto particolare supporto e l'operazione viene semplicemente rifiutata.

Per le modifiche sulla tabella esterna le alternative esistono, dato dal particolare significato della tabella esterna che sul piano applicativo rappresenta la tabella principale (**master**) alle cui variazioni la tabella interna (**slave**) deve adeguarsi.
Per poter agire con le operazioni di modifica possiamo usare uno dei seguenti modi:

- CASCADE: il nuovo valore dell'attributo della tabella esterna viene riportato su tutte le corrispondenti righe della tabella interna
- SET NULL: all'attributo referente viene assegnato il valore nullo al posto del valore modificato nella tabella esterna
- SET DEFAULT: all'attributo referente viene assegnato il valore di default al posto del valore modificato nella tabella esterna
- NO ACTION: l'azione di modifica non viene consentita e il sistema quindi non ha bisogno di riparare la violazione

Per le violazioni da cancellazione di un elemento della tabella esterna si ha a disposizione lo stesso insieme di reazioni:

- CASCADE: tutte le righe della tabella interna corrispondenti alla riga cancellata vengono cancellate
- SET NULL: all'attributo referente viene assegnato il valore nullo al posto del valore cancellato nella tabella esterna
- SET DEFAULT: all'attributo referente viene assegnato il valore di default al posto del valore modificato nella tabella esterna
- NO ACTION: l'azione di cancellazione non viene consentita

In generale per ogni evento è possibile associare una politica diversa in base a come la si vuole gestire.
Nel caso della politica CASCADE si assume che le righe della tabella interna siano strettamente legate alle corrispondenti righe della tabella esterna, per cui se si apporta una modifica alla tabella esterna si devono modificare in modo conseguente tutte le righe della tabella interna, per quanto riguarda le altre politiche ci sono dipendenze meno strette tra prima e seconda tabella.
Le violazioni possono generare una reazione a catena qualora la tabella interna compaia a sua volta come tabella esterna in un altro vincolo di integrità.

La politica di reazione viene specificata immediatamente dopo il vincolo di integrità secondo la seguente sintassi:

$$\text{ON } \langle \text{DELETE} \mid \text{UPDATE} \rangle \\ \quad \langle \text{CASCADE} \mid \text{SET NULL} \mid \text{SET DEFAULT} \mid \text{NO ACTION} \rangle$$

Modifica degli schemi

SQL fornisce primitive per la manipolazione degli schemi della base di dati che permettono di modificare le definizioni di tabelle precedentemente introdotte.

I comandi che vengono utilizzati a questo fine sono ALTER e DROP

Alter

Il comando ALTER permette di modificare domini e schemi di tabelle, le forme che troviamo sono:
Per i domini:

$$\begin{aligned}\mathrm{ALTER\ DOMAIN}\ NomeDominio\langle&\mathrm{SET\ DEFAULT}\ ValoreDiDefault\|\\&\mathrm{DROP\ DEFAULT}\ |\\&\mathrm{ADD\ CONSTRAINT}\ DefVincolo\ |\\&\mathrm{DROP\ CONSTRAINT}\ NomeVincolo\rangle\end{aligned}$$

Per le tabelle:

$$\begin{aligned}&\mathrm{ALTER\ TABLE}\ NomeTabella\langle\\&\mathrm{ALTER\ COLUMN}\ NomeAttributo\langle\mathrm{SET\ DEFAULT}\ NuovoDefault\ |\\&\qquad\qquad\qquad\qquad\mathrm{DROP\ DEFAULT}\rangle\ |\\&\mathrm{ADD\ CONSTRAINT}\ DefVincolo\ |\\&\mathrm{DROP\ CONSTRAINT}\ NomeVincolo\ |\\&\mathrm{ADD\ COLUMN}\ DefAttributo\ |\\&\mathrm{DROP\ COLUMN}\ NomeAttributo\rangle\end{aligned}$$

Tramite $\mathrm{ALTER\ DOMAIN}$ e $\mathrm{ALTER\ TABLE}$ è possibile aggiungere e rimuovere vincoli e modificare i valori di default associati ai domini e agli attributi, inoltre è possibile aggiungere ed eliminare attributi e vincoli sullo schema di una tabella.
Quando si definisce un nuovo vincolo questo deve essere soddisfatto dai dati già presenti, altrimenti l'inserimento viene rifiutato

Drop

Mentre il comando ALTER effettua delle modifiche sui domini o sullo schema delle tabelle il comando DROP permette di rimuovere dei componenti come schemi, domini, tabelle, viste o asserzioni.
Il comando usa la seguente sintassi:

$$\begin{aligned}&\mathrm{DROP}\ \langle\mathrm{SCHEMA}\ |\ \mathrm{DOMAIN}\ |\ \mathrm{TABLE}\ |\ \mathrm{VIEW}\ |\ \mathrm{ASSERTION}\rangle\ NomeElemento\\&\quad[\ \mathrm{RESTRICT}\ |\ \mathrm{CASCADE}\ ]\end{aligned}$$

L'operazione $\mathrm{RESTRICT}$ specifica che il comando non deve essere eseguito in presenza di oggetti **non vuoti**, nei diversi casi:

- Uno **schema** non è rimosso se contiene tabelle o altri oggetti
- Un **dominio** non è rimosso se appare in qualche definizione di tabella
- Una **tabella** non è rimossa se possiede delle righe o se è presenta qualche definizione di tabella o vista
- Una **vista** non è rimossa se è utilizzata nella definizione di altra tabelle o viste.
  L'opzione $\mathrm{RESTRICT}$ è un opzione di default.

Nel caso si specifichi l'opzione CASCADE tutti gli oggetti specificati devono essere rimossi.
Nei diversi casi:

- Quando si rimuove uno **schema** non vuoto anche tutti gli oggetti che fanno parte dello schema vengono eliminati
- Rimuovendo un **dominio** che compare nelle definizioni di qualche attributo l'opzione fa si che il nome del dominio venga rimosso, ma gli attributi che sono stati definiti utilizzando quel dominio rimangano associati al dominio elementare
- Quando si rimuove una **tabella** tutte le righe vengono perse, se la tabella poi compariva in qualche definizione di tabella o vista viene rimossa anche questa
- Eliminando una **vista** che compare nella altre definizioni di altre tabelle o viste viene anche queste tabelle e viste vengono rimosse
  Quindi in generale l'opzione CASCADE attiva una reazione a catena per cui tutti gli elementi che dipendono da un elemento vengono rimossi fin quando non si giunge ad una situazione dove non esistono dipendenze non risolte

Interrogazioni in SQL

Useremo per tutto questo capitolo le seguenti tabelle per gli esempi:

IMPIEGATO

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Bianchi	Produzione	20	36	Torino
Giovanni	Verdi	Amministrazione	20	40	Roma
Franco	Neri	Distribuzione	16	45	Napoli
Carlo	Rossi	Direzione	14	80	Milano
Lorenzo	Gialli	Direzione	7	73	Genova
Paola	Rosati	Amministrazione	75	40	Venezia
Marco	Franco	Produzione	20	46	Roma

DIPARTIMENTO

Nome	Indirizzo	Città
Amministrazione	Via Tito Livio, 27	Milano
Produzione	P.le Lavater, 3	Torino
Distribuzione	Via Segre, 9	Roma
Ricerca	Via Venosa, 6	Milano

La parte di SQL dedicata alla formulazione di interrogazioni fa parte del DML;
D'altronde la separazione tra DML e DDL non è rigida e parte dei servizi di definizione di interrogazioni vengono riutilizzati nella specifica di alcuni aspetti avanzati dello schema

Dichiaratività di SQL

SQL esprime le interrogazioni in modo **dichiarativo**, ossia si specifica l'obbiettivo dell'interrogazione e non il modo in cui ottenerlo, seguendo quindi i principi del calcolo relazionale (contrapponendosi a quelli procedurali come l'algebra relazionale).
Un interrogazione SQL per essere eseguita viene passata all'ottimizzatore di interrogazioni (**query optimizer**), un componente del DBMS che analizza interrogazione e formula a partire da quest'ultima un'interrogazione equivalente in calcolo relazionale.
In generale esistono diversi modi per effettuare la stessa interrogazione, il programmatore però deve effettuare la scelta non basandosi sull'efficienza ma sulla leggibilità e modificabilità.

Interrogazioni semplici in SQL

Le operazioni di interrogazione base in SQL vengono specificate per mezzo della struttura essenziale:

```
SELECT [DISTINCT] ListaAttributi <- Target list
FROM ListaTabelle <- clausola FROM
[WHERE Condizione] <- clausola WHERE
```

Una descrizione più precisa della sintassi è tale:

$$\begin{aligned}&\text{SELECT } AttrEspr[[\text{as}]Alias]\{,AttrEspr[[\text{as}]Alias]\}\\&\text{FROM } Tabella[[\text{as}]Alias]\{,Tabella[[\text{as}]Alias]\}\\&[\text{WHERE } Condizione]\end{aligned}$$

Questa interrogazione quindi seleziona, tra le righe che appartengono al prodotto cartesiano delle tabelle elencate nella clausola FROM, quelle che soddisfano le condizioni in WHERE, ottenendo come risultato una tabella con una riga per ogni riga prodotta dalla clausola FROM e filtrata dalla clausola WHERE, le cui colonne si ottengono dalla valutazione delle espressioni

Le operazioni più semplici si ottengono usando anche le clausole:

- **Singola tabella**:

```
SELECT *
FROM Agenti
```

- **Selezione**:

```
SELECT *
FROM Ordini
WHERE Ammontare > 10000
```

- **Proiezione**:

```
SELECT CognomeENome, Città
FROM Clienti
```

- **Prodotto cartesiano**:

```
SELECT *
FROM Clienti, Ordini
```

Clausola SELECT

La clausola SELECT specifica gli elementi dello schema della tabella risultato.
Come argomento della clausola SELECT può comparire il carattere speciale asterisco ($*$) che rappresenta la selezione di tutti gli attributi delle tabelle elencate nella clausola FROM
Esempio:

```
SELECT *
FROM Impiegato
WHERE Cognome="Rossi"
```

Possono anche comparire generiche espressioni sul valore degli attributi di ciascuna riga selezionata
Esempio:

```
SELECT Stipendio/12 as Stipendio mensile
FROM Impiegato
WHERE Cognome="Bianchi"
```

Risultato:

StipendioMensile
3,00

Clausola FROM

Quando si desidera formulare un'interrogazione che coinvolge righe appartenenti a più di una tabella si pone come argomento della clausola FROM l'insieme di tabelle che si vuole accedere.
Sul prodotto cartesiano delle tabelle elencate verranno applicate le condizioni contenute nella clausola WHERE, quindi un join può essere specificato in modo esplicito indicando le condizioni che esprimono il legame tra le diverse tabelle

Esempio:

```
SELECT Impiegato.Nome, Impiegato.Cognome, Dipartimento.Città
FROM Impiegato, Dipartimento
WHERE Impiegato.Dipart = Dipartimento.Nome
```

È necessario utilizzare l'operatore punto "." per specificare il nome della tabella quando le tabelle presenti possiedono più attributi con lo stesso nome, qualora poi non ci siano ambiguità è un ridondante se utilizzato.

Clausola WHERE

La clausola WHERE ammette come argomento un'espressione booleana costruita combinando predicati semplici con gli operatori AND,OR,NOT;
Ciascun predicato semplice usa gli operatori classici come $<, >, <>, =, \leq, \geq$ per confrontare da un lato un'espressione costruita a partire dai valori degli attributi per la riga, e dall'altro lato un valore costante o un'altra espressione.

Esempio:

```
SELECT Nome, Cognome
FROM Impiegato
WHERE Ufficio=20 AND Dipart='Amministrazione'
```

Oltre ai normali predicati di confronto relazionali, SQL mette a disposizione un operatore LIKE per il confronto delle stringhe, che permette di effettuare confronti con stringhe in cui compaiono caratteri speciali come _ e %, il primo rappresenta nel confronto un carattere arbitrario, il secondo una stringa di un numero arbitrario (anche eventualmente nullo).
Un confronto come LIKE '$\mathrm{ab\%ba\_}$' sarà perciò soddisfatto da una qualsiasi stringa di caratteri che inizia con ab e che ha la coppia di caratteri ba prima dell'ultima posizione (come $\mathrm{abcdedcbac}$)

Gestione dei valori nulli

Un valore nullo in un attributo può significare che un certo attributo non è applicabile, o che il valore è applicabile ma non conosciuto, oppure che non si conosca quale delle due situazioni sia applicabile.
Per selezionare i termini con i valori nulli SQL fornisce il predicato $\mathrm{IS\ NULL}$, la cui sintassi è:

$$\mathit{Attributo}\ \mathrm{IS}\ [\mathrm{NOT}]\ \mathrm{NULL}$$

Il predicato risulta vero solo se l'attributo ha valore nullo, mentre $\mathrm{NOT\ NULL}$ è la sua negazione.
Ricordiamo che da SQL-2 viene utilizzata la logica a tre valori che prevede il valore $\mathrm{unknown}$

Interpretazione formale delle interrogazioni in SQL

È possibile costruire una corrispondenza tra interrogazioni SQL ed equivalenti interrogazioni espresse in algebra relazionale (ricordiamo presente nel paragrafo Dichiaratività di SQL).
Data un interrogazione SQL nella sua formula più semplice:

$$\begin{aligned}&\text{SELECT } T_1.\,Attributo_{11}, \ldots, T_h Attributo_{hm}\\&\text{FROM } Tabella_1\ T_1, \ldots, Tabella_n\ T_n\\&\text{WHERE } Condizione\end{aligned}$$

si può costruire un'interrogazione equivalente in algebra relazionale usando la seguente traduzione (in cui per semplicità omettiamo le ridenominazioni che ci permettono di considerare tutti i join come prodotti cartesiani):

$$\Pi_{T_1.Attributo_{11},\ldots,T_hAttributo_{hm}}(\sigma_{Condizione}(TABELLA_1 \bowtie \cdots \bowtie TABELLA_n))$$

Per interrogazioni più complicate la forma di conversione non è più applicabile.
Una condizione essenziale per l'esecuzione di queste traduzioni è però che l'interrogazioni di partenza non usi funzionalità di SQL non presenti nell'algebra e calcolo relazionale come la valutazione degli Operatori Aggregati.
I risultati delle interrogazioni SQL differiscono anche dalle espressioni dell'algebra e del calcolo relazionale nella gestione dei duplicati

Duplicati

Una differenza significativa tra SQL e algebra relazionale è data dalla gestione di duplicati: Mentre in algebra relazionale una tabella viene vista come una relazione dal punto di vista matematico e quindi come un insieme di elementi (tuple) diversi tra loro, in SQL si possono avere in una tabella più righe uguali (dette **duplicati**).
Per emulare il comportamento dell'algebra relazionale sarebbe necessario effettuare l'eliminazione dei duplicati tutte le volte che si eseguono operazioni di proiezione, ma la rimozione è costosa e spesso non necessaria, per questo in SQL si è stabilito di permettere la presenza di duplicati all'interno delle tabelle, lasciando che sia l'interrogazione ad escluderli esplicitamente.
L'eliminazione dei duplicati viene esplicitata con la parola chiave DISTINCT posta dopo SELECT

Join interni ed esterni

Una sintassi alternativa per la specifica dei join permette di distinguere, tra le condizioni che compaiono nell'interrogazione, quelle che rappresentano condizioni di join e quelle che rappresentano condizioni di selezioni sulle righe.
La sintassi proposta è la seguente:

$$\begin{aligned}&\text{SELECT } AttrEspr[[\text{as}]Alias]\{, AttrEspr[[\text{as}]Alias]\}\\&\text{FROM } Tabella[[\text{as}]Alias]\\&\{[TipoJoin]\ \text{JOIN } Tabella[[\text{as}]Alias]\text{on}CondizioneDiJoin\}\\&[\text{WHERE } AltraCondizione]\end{aligned}$$

Mediante questa sintassi la condizione di join non compare come argomento della clausola WHERE, ma viene spostata nella clausola FROM, associata alle tabelle che vengono coinvolte nel join.
Il parametro $TipoJoin$ specifica quale tipo di join usare, tra cui:

- INNER, di default
- $\mathrm{RIGHT\ OUTER}$
- $\mathrm{LEFT\ OUTER}$
- $\mathrm{FULL\ OUTER}$
  L'inner join rappresenta il tradizionale theta-join dell'algebra relazionale

Con il join interno le righe che vengono coinvolte nel join sono in generale un sottoinsieme delle righe di ciascuna tabella, può infatti capitare che alcune righe non vengano considerate in quanto non vengano rispettate le condizioni;
Questo comportamento non è desiderabile solitamente, si preferisce infatti utilizzare il join esterno per mantenere le righe che fanno parte di una o entrambe le tabelle coinvolte.
Esistono poi tre variante dei join esterni:

- LEFT, fornisce come risultato il join esteso con le righe della tabella che compare a sinistra per le quali non esiste una corrispondente riga nella tabella di destra
- RIGHT, che si comporta nella maniera opposta del LEFT
- FULL, che restituisce il join interno esteso con le righe escluse di entrambe le tabelle

Normalmente il join naturale non è consigliato, dato che può introdurre dei rischi nelle applicazioni in quanto il suo comportamento può maturare profondamente al variare dello schema delle tabelle

Uso di variabili

Nelle interrogazioni SQL è possibile associare un nome alternativo, detto **alias**, alle tabelle che compaiono come argomento della clausola FROM.
Il nome viene usato per fare riferimento alla tabella nel contesto dell'interrogazione.
Questa funzionalità può essere sfruttata per fare riferimento a una tabella in modo compatto, ricorrendo a brevi alias ed evitando così di scrivere per esteso il nome della tabella tutte le volte che ne viene richiesto l'uso, ma ci sono però altre ragioni per usare gli alias;
Come prima cosa, utilizzando gli alias è possibile fare accesso più volte alla stessa tabella, come avviene nel calcolo relazionale quando si usano più variabili associate alla stessa tabella e in modo simile all'uso dell'operatore di ridenominazione ρ dell'algebra relazionale.
Tutte le volte che si introduce un alias per una tabella si dichiara in effetti una variabile che rappresenta le righe della tabella di cui è alias.
Quando la tabella compare una sola volta in un'interrogazione, non c'è differenza tra l'interpretare l'alias come uno pseudonimo o come una nuova variabile, quando compare invece più volte è necessario considerare l'alias come una nuova variabile.

Esempio:

```sql
SELECT I1.Cognome, I1.Nome
FROM Impiegato I1, Impiegato I2
WHERE I1.Cognome = I2.Cognome and
      I1.Nome <> I2.Nome and
      I2.Dipart = 'Produzione'
```

Ordinamento

SQL permette di specificare un ordinamento delle righe del risultato di un'interrogazione tramite la clausola $\mathrm{ORDER\ BY}$, con la quale si chiude l'interrogazione.
La sintassi è la seguente:

$$\mathrm{ORDER\ By}\, AttrDiOrdinamento\ [\mathrm{ASC}|\mathrm{DESC}]\\ \{, AttrDiOrdinamento\ [\mathrm{ASC}|\mathrm{DESC}]\}$$

Le righe vengono ordinate in base al primo attributo nell'elenco, per le righe che hanno lo stesso valore del primo attributo si considerano i valori degli attributi successivi in sequenza. L'ordine su ciascun attributo può essere ascendente o discendente in base al qualificatore utilizzato, se omesso viene usato un ordinamento ascendente

Esempio:

```sql
SELECT *
FROM Automobile
ORDER BY Marca DESC, Modello
```

Operatori Aggregati

Nell'algebra relazionale ogni condizione viene valutata su una signola tupla alla volta in modo indipendente, SQL invece permette di valutare delle proprietà che dipendono da un insieme di tuple.
SQL mette a disposizione 5 operatori:

$$\mathrm{COUNT,SUM,MAX,MIN,AVG}$$

Gli operatori aggregati vengono gestiti come un'estensione delle normali interrogazioni, considerando prima le parti FROM e WHERE, per poi applicare l'operatore alla tabella contenente il risultato dell'interrogazione.

L'operatore COUNT usa la seguente sintassi:

$$\mathrm{COUNT}(\langle *|[\mathrm{DISTINCT}||\mathrm{ALL}]\ ListaAttributi)$$

La prima opzione (*) restituisce il numero di righe, $\mathrm{DISTINCT}$ restituisce il numero di diversi valori degli attributi in $ListaAttributi$ (elimina i duplicati), ALL restituisce il numero di righe che possiedono valori diversi dal valore nullo in $ListaAttributi$, nel caso di omissione di

qualunque opzione quest'ultima è quella di default.
Esempio:

```
SELECT COUNT(DISTINCT Stipendio)
FROM Impiegato
```

Gli altri 4 operatori operatori aggregati invece ammettono come argomento un attributo o un'espressione, eventualmente preceduta dalle parole chiave $\mathrm{DISTINCT}$ e ALL.
Le funzioni aggregate SUM e AVG ammettono come argomento solo espressioni che rappresentano valori numerici o valori di tempo.
Le funzioni MAX e MIN richiedono solo sull'espressione sia definito un ordinamento per cui si possano applicare anche su stringhe di caratteri o su istanti di tempo
La sintassi è la seguente:

$$\langle \mathrm{SUM} \mid \mathrm{MAX} \mid \mathrm{MIN} \mid \mathrm{AVG}\ ([\mathrm{DISTINCT}\mid\ \mathrm{ALL}] \mathit{AttrEspr})$$

Gli operatori si applicano sulle righe che soddisfano la condizione presente nella clausola WHERE.

Interrogazioni con raggruppamento

SQL mette a disposizione la clausola $\mathrm{GROUP\ BY}$ che permette di specificare come dividere le tabelle in sottoinsiemi.
La clausola ammette come argomento un insieme di attributi e l'interrogazione raggrupperà le righe che possiedono gli stessi valori per questo insieme di attributi.
Esempio:

```
SELECT Dipart, SUM(Stipendio)
FROM (Impiegato)
GROUP BY Dipart
```

L'interrogazione viene eseguita prima come se la clausola $\mathrm{GROUP\ BY}$ non esistesse, selezionando gli attributi che compaiono nella clausola o all'interno dell'espressione argomento dell'operatore aggregato;
La tabella ottenuta poi viene analizzata, dividendo le righe in insiemi caratterizzati dallo stesso valore degli attributi che compaiono come argomento della clausola.
Dopo che le righe son state raggruppate in sottoinsiemi, l'operatore aggregato viene applicato separatamente su ogni sottoinsieme;
Il risultato dell'interrogazione è costituito da una tabella con righe che contengono l'esito della valutazione dell'operatore aggregato affiancato al valore dell'attributo che è stato usato per l'aggregazione

La sintassi SQL prevede che la clausola $\mathrm{GROUP\ BY}$ in un interrogazione possa comparire come argomento di la clausola SELECT soltanto se viene utilizzato lo stesso sottoinsieme

degli attributi

Predicati sui gruppi

Si potrebbe voler prendere in considerazione solo i sottoinsiemi che soddisfano determinate condizioni, nel caso questo sia verificabile per singole righe allora basta porre gli opportuni predicati come argomento della clausola WHERE, altrimenti sarà necessario usare il costrutto HAVING:
La clausola HAVING descrive le condizioni che si devono applicare al termine dell'esecuzione di un'interrogazione che fa uso della clausola GROUP BY, e ogni sottoinsieme di righe costruito dalla GROUP BY fa parte del risultato dell'interrogazione solo se il predicato argomento di HAVING risulta soddisfatto
Esempio:

```
SELECT Dipart, SUM(Stipendio) AS SommaStipendi
FROM Impiegato
GROUP BY Dipart
HAVING SUM(Stipendio)>100
```

La sintassi permette anche la definizione che presentano la clausola HAVING senza una corrispondente clausola GROUP BY, in questo caso l'intero insieme di righe è trattato come unico raggruppamento, ma si ha un campo limitato di applicabilità, poiché se la condizione non vien soddisfatta il risultato sarà vuoto.
Questa clausola permette l'utilizzo di espressioni booleane su predicati semplici (cioè i confronti tra il risultato della valutazione di un operatore aggregato e una generica espressione).
Sintatticamente nella clausola è ammessa anche la presenza diretta degli attributi argomento di GROUP BY, ma è preferibile raccogliere tutte le condizioni su questi attributi nell'ambito della clausola WHERE.

Per sapere quali predicati di un interrogazione che fa uso di raggruppamento vanno dati come argomento a WHERE rispetto ad HAVING basta rispettare il seguente criterio:
Solo i predicati in cui compaiono operatori aggregati devono essere argomento della clausola HAVING

Interrogazioni di tipo insiemistico

SQL mette a disposizione degli operatori insiemistici, simili a quelli dell'algebra relazionale, tra cui:

- UNION, unione
- INTERSECT, intersezione
- EXCEPT, chiamato anche MINUS
  La sintassi è la seguente:

$$SelectSQL\{\langle \text{UNION} \mid \text{INTERSECT} \mid \text{EXCEPT} \rangle [\text{ALL}] SelectSQL\}$$

Gli operatori insiemistici, al contrario del resto del linguaggio, assumo come default l'azione di eliminazione dei duplicati, ci sono due ragioni per questa differenza:

1. L'eliminazione dei duplicati rispetta molto meglio il tipico significato di questi operatori
2. L'esecuzione di queste operazioni (in particolare differenza e intersezione) richiede di effettuare un'analisi delle righe che rende molto limitato il costo aggiuntivo dell'eliminazione dei duplicati

Qualora si voglia adottare una diversa interpretazione degli operatori sarà sufficiente usare la parola chiave ALL.

Interrogazioni nidificate

SQL consente di scrivere interrogazioni che presentano al loro interno altre interrogazioni, chiamate **nidificate**.
La nidificazione può avvenire nelle 3 clausole di interrogazione e consiste nel confrontare un valore con una collezione di valori (ossia il risultato di un interrogazione).
Per risolvere il problema della disomogeneità dei termini di confronto, SQL ha esteso i normali operatori di confronto con due altre opzioni: ALL e ANY;
Il confronto ANY è vero se il primo operando sta nella relazione specificata con almeno un elemento del secondo operando, ALL ugualmente ma tutti gli elementi devono stare nella relazione.
Esempio di utilizzo:

```
SELECT *
FROM Impiegato
WHERE Dipart = ANY (SELECT Nome
                    FROM Dipartimento
                    WHERE CIttà='Firenze')
```

Talvolta l'interrogazione nidificata fa riferimento al contesto dell'interrogazione che la racchiude, tipicamente ciò accade tramite una variabile definita nell'ambito di una della query più esterna e usata nell'ambito della query più interna (si parla di un **passaggio di binding** da un contesto all'altro), così la nuova interpretazione è: per ogni riga della query esterna, valutiamo per prima cosa la query nidificata, quindi calcoliamo il predicato a livello di riga sulla query esterna.
Per quanto riguarda lo **scope** delle variabili SQL, vale la restrizione che una variabile è usabile solo nell'ambito della query in cui è stata definita o nell'ambito della query nidificata all'interno di essa.

Modifica dei dati in SQL

La parte di Data Manipulation Language comprende i comandi per interrogare e modificare il contenuto della basi di dati, andremo a vedere esattamente questi

Inserimento

Il comando di inserimento di righe nella base di dati presenta due sintassi, la prima permette di inserire le singole righe all'interno delle tabelle:

$$\begin{aligned}&\text{INSERT INTO } NomeTabella\ [ListaAttributi]\\&\quad\langle \text{VALUES}(Listavalori)|\rangle\\&\quad SelectSQL\end{aligned}$$

L'argomento della clausola VALUES rappresenta esplicitamente i valori degli attributi della singola riga.
Questa prima forma è usata principalmente all'interno dei programmi per riempire una tabella con i dati forniti direttamente dagli utenti.

La seconda forma permette di inserire insieme di righe, estratti dal contenuto della base di dati:

$$\begin{aligned}&\text{INSERT INTO } NomeTabella\\&\quad\langle \text{SELECT VALUES}(Listavalori)|\rangle\\&\quad\text{FROM Tabella}\\&\quad\text{WHERE Condizione}\end{aligned}$$

Se in un inserimento non vengono specificati i valori di tutti gli attributi della tabella, agli attributi mancanti viene assegnato quello di default, in assenza di questo avviene uno nullo; Se esiste il vincolo NOT NULL quest viene violato e non avviene l'inserimento.

Cancellazione

Il comando DELETE elimina le righe delle tabelle della base di dati,seguendo la seguente sintassi:

$$\text{DELETE FROM } NomeTabella\ [\text{WHERE } Condizione]$$

Quando il comando WHERE non ha argomenti il comando elimina tutte le righe della tabella.
Nel caso esista un vincolo di integrità referenziale con politica CASCADE, allora la cancellazione apporta anche la cancellazione di righe appartenenti ad altra tabelle

$$\begin{aligned}&\text{UPDATE } NomeTabella\\&\quad\text{set } Attributo = \langle Espressione \mid SelectSQL|\text{NULL}|\text{DEFAULT}\rangle\\&\qquad\{, Attributo = \langle Espressione \mid SelectSQL|\text{NULL}|\text{DEFAULT}\rangle\}\\&\qquad[\text{WHERE } Condizione]\end{aligned}$$

Se il comando non presenta la clausola WHERE si presuppone che la condizione sia soddisfatta e si modificano tutte le righe

Modifica

Il comando UPDATE permette di aggiornare uno o più attributi delle righe di una tabella che soddisfano eventualmente una condizione:

$$\begin{aligned}&\mathrm{UPDATE}\ NomeTabella\\&\quad\mathrm{SET}\ Attributo = \langle Espressione|SelectSQL|\mathrm{NULL}|\mathrm{DEFAULT}\rangle\\&\quad\{, Attributo = \langle Espressione|SelectSQL|\mathrm{NULL}|\mathrm{DEFAULT}\rangle\}\\&[\mathrm{WHERE}\ Condizione]\end{aligned}$$

Se il comando non presenta una clausola WHERE si suppone sempre che la condizione sia soddisfatta e si esegue la modifica su tutte le righe

Caratteristiche evolute di SQL

Vincoli di integrità generici

SQL permette di specificare vincoli più complessi grazie alla clausola CHECK con la seguente sintassi:

$$\mathrm{CHECK}(Condizione)$$

Le condizioni ammissibili sono le stesse della clausola WHERE e la condizione deve essere sempre verificata affinchè la base di dati sia corretta.

Una possibile dimostrazione dell'utilità del costrutto è la descrizione dei vincoli predefiniti, che possono essere descritti in maniera più veloce e compatta:

```
create table Impiegato
(Matricola character(6)
           check (Matricola is not null and
                  1 = (select count(*)
                       from Impiegato I
                       where Matricola=I.Matricola)),
 Cognome   character(20) check (Cognome is not null),
 Nome      character(20) check (Nome is not null and
                  2 > (select count(*)
                       from Impiegato I
                       where Nome = I.Nome
                       and Cognome = I.Cognome)),
 Dipart    character(15) check (Dipart in
                       (select NomeDip
                        from Dipartimento))
)
```

Usando questa clausola però si perde la possibilità di associare ai vincoli una politica di reazione alle violazioni

Asserzioni

Grazie alla clausola CHECK è possibile definire anche un ulteriore componente dello schema di una base di dati, le asserzioni;

Le asserzioni rappresentano dei vincoli che non sono associati a un attributo o a una tabella in particolare, bensì appartengono direttamente allo schema.
Mediante le asserzioni è possibile esprimere tutti i vincoli che abbiamo specificato fin qui nella definizione delle tabelle, permettono inoltre di esprimere vincoli che non sarebbero altrimenti definibili come vincoli che coinvolgono più tabelle o che richiedono che una tabella abbia una cardinalità minima.
La sintassi prevista è tale:

$$\text{CREATE ASSERTION } NomeAsserzione \text{ CHECK } (Condizione)$$

Le asserzioni possono essere verificate immediatamente dopo ogni modifica dello stato del DB **(controllo immediato)** o solo al termine di un insieme di operazioni/transazioni **(controllo differito).**
I comandi $\text{SET CONSTRAINTS IMMEDIATE}$ e $\text{SET CONSTRAINTS DEFERRED}$ consentono di specificare o passare da una modalità all'altra.

Viste

Oltre alla costruzione di tabelle di base, SQL permette di definire tabelle virtuali chiamate **viste** (già viste precedentemente in 3 - Linguaggi di Interrogazioni per Basi di Dati Relazionali), queste ultime sono il risultato di un'espressione SQL a partire da altre tabelle, sia base che virtuali.
La vista non è memorizzata, quando usata in un'interrogazione è espansa con la sua definizione.
La sintassi SQL per definire una vista logica è la seguente:

$$\text{CREATE VIEW } NomeVista[(ListaAttributi)]\text{AS } SelectSQL$$
$$[\text{WITH}[\text{LOCAL}|\text{CASCADE}]\text{ CHECK OPTION}]$$

L'interrogazione deve restituire un insieme di attributi compatibile con gli attributi nello schema della vista;
L'ordine nella target list della clausola SELECT deve corrispondere all'ordine degli attributi nello schema.

Le operazioni di modifica su vista sono soggette a restrizioni, perché spesso non sono riconducibili a modifiche sulle tabelle base usate per definire la vista.
Le modifiche sono ammesse solo quando la vista è definita con un'espressione che soddisfa le seguenti condizioni:

- La clausola SELECT non ha l'opzione DISTINCT e i valori degli attributi della vista non sono calcolati
- La clausola FROM riguarda una sola tabella base o virtuale, a sua volta modificabile, ovvero sono escluse tabelle virtuali ottenute per giunzione
- La clausola WHERE non contiene $SottoSelect$
- Non sono presenti gli operatori GROUP BY e HAVING

- Le colonne definite nelle tabelle di base con il vincolo NOT NULL devono far parte della tabella virtuale

Nell'ambito delle viste modificabili, CHECK OPTION specifica che possono essere ammessi aggiornamenti solo sulle righe della vista, e dopo gli aggiornamenti le righe devono continuare ad appartenere alla vista, ad esempio vorremmo costruire una vista contenente tutti gli impiegati con stipendio superiore a 5000 euro:

```
CREATE VIEW ImpiegatiAmmin(Matricola, Nome, Cognome, Stipendio) AS
SELECT Matricola, Nome, Cognome, Stipendio
FROM Impiegato
WHERE Dipart = 'Amministrazione' AND Stipendio > 5000
```

Creiamo una vista per gli impiegati amministratori "poveri" a partire dalla vista precedente:

```
CREATE VIEW ImpAmminPoveri AS
SELECT*
FROM ImpiegatiAmmin
WHERE Stipendio < 9000 WHIT CHECK OPTION
```

La vista è stata definita con CHECK OPTION (opzione di default CASCADE) in cui ogni aggiornamento sulla vista, per poter essere propagato, non deve eliminare righe dalle vista.

Le viste hanno diverse utilità come:

- Semplificare interrogazioni complesse, o addirittura non esprimibili in SQL su tabelle base
  Esempio
  Volendo trovare il numero medio degli agenti per zona, non possiamo scrivere qualcosa del genere:

```
SELECT AVG(COUNT(*))
FROM Agenti GROUP BY Zona
```

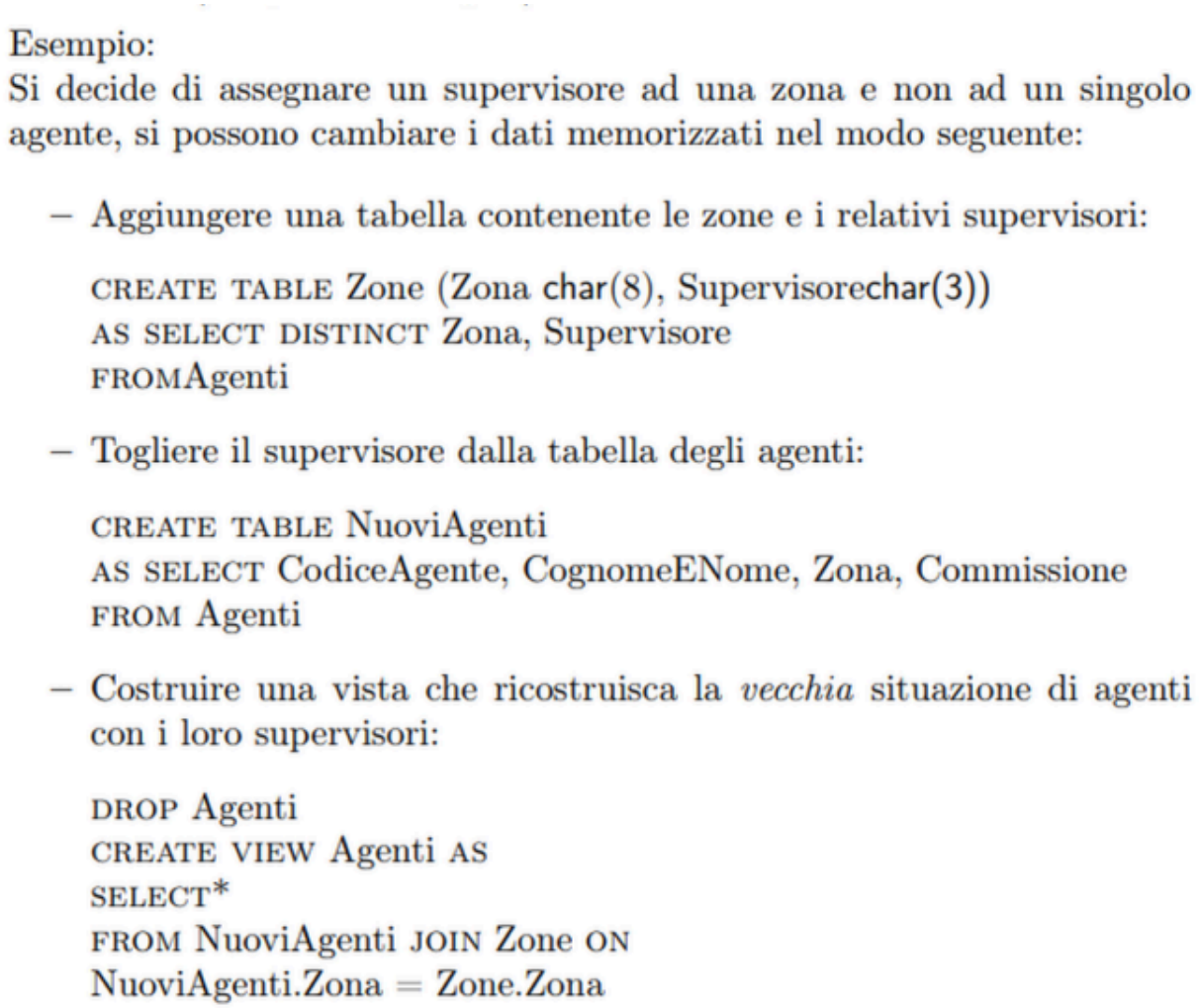
  Si può formulare una interrogazione alternativa con l'aiuto di una tabella virtuale:

```
CREATE VIEW AgentiXZona(Zona, NAgenti) AS
SELECT Zona, COUNT(*)
FROM Agenti
GROUP BY Zona;

SELECT AVG(NAgenti) fromAgentiXZona;

DROP AgentiXZona
```

- Per nascondere alle applicazioni alcune modifiche dell'organizzazione logica dei dati (indipendenza logica)



Esempio:
Si decide di assegnare un supervisore ad una zona e non ad un singolo agente, si possono cambiare i dati memorizzati nel modo seguente:

– Aggiungere una tabella contenente le zone e i relativi supervisori:

```
CREATE TABLE Zone (Zona char(8), Supervisorechar(3))
AS SELECT DISTINCT Zona, Supervisore
FROMAgenti
```

– Togliere il supervisore dalla tabella degli agenti:

```
CREATE TABLE NuoviAgenti
AS SELECT CodiceAgente, CognomeENome, Zona, Commissione
FROM Agenti
```

– Costruire una vista che ricostruisca la *vecchia* situazione di agenti con i loro supervisori:

```
DROP Agenti
CREATE VIEW Agenti AS
SELECT*
FROM NuoviAgenti JOIN Zone ON
NuoviAgenti.Zona = Zone.Zona
```

- Per dare visioni diverse degli stessi dati (viste utente)
  Esempio:
  Aggregare gli ordini per agente, per applicazioni di tipo statistico si può fornire la vista OrdiniPerAgente definita prima:

```
CREATE VIEW OrdiniPerAgewnte(CodiceAgente, TotaleOrdini) AS
SELECT CodiceAgente, SUM(Ammontare) FROM Ordini
GROUP BY CodiceAgente
```

Procedure

Lo standard SQL-2 prevede la definizione di **Procedure**, ovvero dei brevi sottoprogrammi memorizzati nel database come parte dello schema (motivo per cui vengono dette anche **stored procedures**). Esse permettono di assegnare un nome a un'istruzione SQL ed eventuali parametri. Una volta definita, la procedura è utilizzabile come un qualunque comando SQL. La presenza di procedure memorizzate nella base di dati consente di azzerare il tempo che sarebbe stato necessario per scrivere ogni volta le stesse istruzioni in SQL e, di conseguenza, riducono il rischio di commettere errori involontari scrivendo istruzioni errate.

Consideriamo una procedura SQL che assegna lo sconto per ogni cliente in base al codice cliente:

```
PROCEDURE AssegnaSconto (:Cod char(2), :Sc integer))
UPDATE Clienti
SET Sconto = :Sc
WHERE CodiceCliente = :Cod;
```

Quest'ultima può essere invocata all'interno del linguaggio ospite utilizzando il comando seguente:

```
$ AssegnaSconto (:CodCli, :ScRid)
```

È bene sapere che lo standard SQL-2 non tratta la scrittura di procedure complesse, ma solo quelle composte da un singolo comando SQL. Questo è invece permesso in SQL-3, dove viene fornita una ricca sintassi per la definizione di procedure, integrando anche le strutture di controllo

Trigger

I trigger, detti anche regole attive, sono dei costrutti per rendere la base di dati in grado di reagire a eventi definiti dall'amministratore tramite l'esecuzione di opportune azioni.
Seguono il paradigma Evento-Condizione-Azione (ECA), infatti ogni trigger si attiva al verificarsi di una condizione prestabilita. Un trigger attivato, avvia l'esecuzione di una specifica sequenza di istruzioni. Le basi di dati che contengono trigger sono dette basi di dati attive.
Nello specifico il paradigma seguito da un trigger funziona come segue:

- **Evento**: Tipicamente una modifica dello stato del database consiste in una operazione di INSERT, DELETE, UPDATE. Se l'evento accade, il trigger si attiva.
- **Condizione**: Predicato booleano espresso in SQL che identifica se l'azione del trigger deve essere eseguita. Quando il trigger si attiva, viene valutata la condizione, quindi il trigger viene considerato.
- **Azione**: Consiste in una sequenza di update SQL o una procedura. Quando la condizione è verificata, allora l'azione viene eseguita, quindi il trigger viene eseguito.

Ogni trigger è caratterizzato da:

- Nome
- Target, ovvero la tabella controllata
- Modalità
  - Nella modalità BEFORE il trigger è considerato e possibilmente eseguito prima dell'evento, ad esempio quando si vuole verificare una modifica prima di eseguirla.

    - Nella modalità AFTER il trigger è considerato ed eseguito dopo l'evento, più comune.
- Evento: INSERT, DELETE, o UPDATE
- Granularità
    - Granularità statement-level: il trigger è considerato e possibilmente eseguito su un insieme di tuple, agendo in modalità set-oriented.
    - Granularità row-level: il trigger è considerato e possibilmente eseguito una volta per ogni tupla modificata
- Tabelle di transizione
    - Se la modalità è row-level, ci sono due variabili di transizione (old e new) che rappresentano il valore precedente o successivo alla modifica di una tupla.
    - Se la modalità è statement-level, ci sono due tabelle di transizione ($\mathrm{old\ table}$ e $\mathrm{new\ table}$) che contengono i valori precedenti e successivi delle tuple modificate dallo statement.
- Azione
- Timestamp di creazione

Un trigger ha questo tipo di sintassi:

$$
\begin{aligned}
&\mathrm{CREATE\ TRIGGER}\ NomeTrigger\\
&Modo\ Evento\ \mathrm{ON}\ TabellaTarget\\
&[\ \mathrm{REFERENCING}\ Referenza\ ]\\
&[\ \mathrm{FOR\ EACH}\ Livello\ ]\\
&[\ \mathrm{WHEN}\ (\ PredicatoSQL\ )\ ]\\
&StatementProceduraleSQL
\end{aligned}
$$

Un esempio può essere questo:

$$
\begin{aligned}
&\mathrm{CREATE\ TRIGGER}\ ImpiegatiSenzaDip\\
&\mathrm{AFTER\ INSERT\ INTO}\ Impiegati\\
&\mathrm{FOR\ EACH\ ROW}\\
&\mathrm{WHEN}\ (\ New.Dipart\ \mathrm{IS\ NULL}\ )\\
&\mathrm{UPDATE}\ Impiegati\\
&\quad \mathrm{SET}\ Dipart = \text{'Nuovi Arrivati'}\\
&\quad \mathrm{WHERE}\ Matr = New.Matr
\end{aligned}
$$

È bene ricordare che l'azione eseguita al verificarsi della condizione di un trigger, può a sua volta attivare un ulteriore trigger, rendendoli quindi **a cascata**, ma se non vi si è attenti si potrebbe generare una catena di attivazioni potenzialmente infinita, mandando in stallo l'intera applicazione.
I trigger sono uno strumento molto potente che permette di gestire vincoli di integrità, calcolare dati derivati, gestire eccezioni e codificare regole aziendali. Un ulteriore vantaggio derivante dall'utilizzo dei trigger consiste nel riuscire a codificare la logica del sistema in maniera centralizzata e condivisa da tutte le applicazioni, con conseguenti vantaggi in fase di lettura e manutenzione del codice, infatti in caso di modifiche al comportamento del sistema è sufficiente intervenire nell'ambito della definizione dei trigger e non in più parti del

codice.
Lo svantaggio è che i trigger sono standardizzati solo per SQL-3, per cui potrebbero presentarsi casi (se pur sempre più rari) di non portabilità del codice.

Note

Si consiglia la visione del PDF 4.1 - Intro to MySQL.pdf per poter fare le eventuali esercitazioni