www.art-net.info

Indice

Capitolo 3

SQL (ANSI-92)

(Gli esercizi sono stati implementati, quando non specificato diversamente, utilizzando SQL Plus 8 Oracle)

Introduzione

Cosa è SQL

SQL non identifica un prodotto commerciale, ma un linguaggio nello stesso modo in cui C e Basic indicano linguaggi generali e non compilatori particolari.
È un linguaggio, che serve per eseguire varie operazioni sia sui dati che sulle strutture che li contengono. La sigla, acronimo di *Structured Query Language*, è ormai diventata sinonimo di linguaggio standard per la gestione dei data base relazionali.

SQL è dunque un linguaggio per la gestione di data base relazionali, quindi assolve alle funzioni di Data Description Languge (linguaggio di descrizione dei dati e delle strutture che li conterranno), di Data Manager Language (linguaggio per la manipolazione dei dati) e di linguaggio di interrogazione.
Il termine SQL può generare confusione. La lettera S, iniziale di Structured (Strutturato), e la lettera L, iniziale di Language, sono abbastanza semplici, ma la lettera Q si presta a varie interpretazioni. Ovviamente Q sta per Query che se fosse interpretata alla lettera, limiterebbe il linguaggio SQL a uno strumento per interrogare il data base. In effetti SQL fa molto di più che porre delle domande.

Gli informatici chiamano questo linguaggio di alto livello o dichiarativo perché permette di svolgere operazioni dichiarando ***cosa*** si deve ottenere e non ***come*** si deve ottenere. Ricordiamo che i linguaggi di terza generazione o procedurali sono quelli dove bisogna specificare il *come si fa,* non è sufficiente dichiarare il *cosa si deve fare*. Non è così per questo linguaggio, che pur limitando le scelte del programmatore e l'efficienza del programma, libera lo sviluppatore dal gravoso compito di scrivere pagine e pagine di codice. Chi usa questo linguaggio però, non è solo chi programma, ma anche chi si avvicina all'informatica marginalmente e per riflesso. Queste persone sono gli impiegati, i professionisti, i commessi, i magazzinieri, ecc. insomma chiunque ha la necessita di manipolare o consultare basi di dati. Forse la causa del suo grande successo sta nella sua semplicità di utilizzo. Non bisogna, però farsi ingannare, perché se da un lato SQL è intuitivo e semplice, da un altro, per essere capito a fondo richiede di essere studiato con attenzione per capirne tutte le sfumature e le notevoli potenzialità.

Storia di SQL

Le origini di SQL risalgono all'inizio degli anni 70 in California, quando la società IBM sviluppa il System R, un applicativo per la gestione dei dati, il cui linguaggio veniva chiamato *Sequel*. Questo linguaggio rappresentava l'embrione di quello che sarebbe poi diventato l'attuale SQL. Infatti alla fine degli anni 70, sempre l'IBM, sviluppa un altro prodotto il DB2 (un sistema per la gestione di database relazionali o RDBMS, *Relational Data Base Management System*) che utilizza una primordiale versione di SQL. Da allora si sono succeduti un gran numero di prodotti che implementano questo linguaggio e ogni produttore, aggiungendo delle variazioni e estensioni proprie, ha contribuito alla creazione della miriade di dialetti che oggi vengono chiamati SQL.

Standardizzazione di SQL

I due enti che si occupano del processo di standardizzazione ANSI (*American National Standards Organization*) e ISO (*International Standards Organization*), stanno svolgendo, ormai da anni, azioni di promozione dello standard SQL.

Sebbene questi enti preparino le specifiche cui devono adeguarsi i progettisti dei vari DBMS tutti i prodotti che implementano SQL differiscono in maniera più o meno marcata dagli standard ufficiali, aggiungendo delle variazioni alla sintassi o ampliando alcune funzioni.

Nel 1986 è stato promulgato il primo standard; esso possedeva già gran parte delle primitive di formulazione di interrogazioni, ma offriva un limitato supporto per la definizione e manipolazione dei dati e delle strutture logiche che avrebbero dovuto contenerli.

Nel 1989 lo standard è stato ulteriormente esteso ma in modo limitato; l'aggiunta più significativa di questa versione è stata la definizione dell'integrità referenziale. Si fa riferimento a questa versione dello standard usando il nome *SQL-89*.

Nel 1992 è stata pubblicata un'altra versione, in gran parte compatibile con quella precedente ma arricchita da un gran numero di nuove funzionalità. Si fa riferimento a questa ulteriore versione usando i nomi *SQL-92* o *SQL-2*.

L'opera di standardizzazione sta continuando ancora, infatti si sta lavorando ad un ennesima versione chiamata SQL-3, allo stato attuale non ancora pubblicata.

Note sul corso

Per una migliore comprensione e digeribilità di SQL, nel corso appaiono numerosi esempi ed esercizi. Queste *query* sono state testate utilizzando, nella maggiore parte dei casi, Personal Oracle 8 ed inoltre è stata (quando non specificato diversamente) verificata la compatibilità delle sintassi con Access 8.0 (versione fornita con Office 97). Il costante riferimento ad Access si è reso necessario in quanto per la sua particolare diffusione renderà possibile, a chi leggerà queste dispense, di testare loro stessi le varie *query* e sintassi che incontrerà.

I capitoli dal 1° al 6° sono corredati di un efficace strumento di apprendimento e autovalutazione: gli esercizi svolti. Si è pensato di fornire questo strumento per un maggiore approfondimento e diversificazione nello studio di SQL.

Il primo capitolo ed in parte il secondo risulteranno essere particolarmente semplice se non banali per chi già conosce SQL; si noti però, che queste dispense sono state create pensando non ad un lettore molto esperto in informatica ma a qualcuno che pur avendo delle conoscenze informatiche di base si affacci per la prima volta alla studio di SQL. Seguendo la stessa filosofia, sono stati inseriti nel corso argomenti che non riguardano direttamente SQL, ma che sicuramente aiuteranno la sua comprensione. Si riporta la lista di questi argomenti.

L'ultimo capitolo (cap. 9) è più sintetico degli altri. Questo è dovuto al fatto che gli argomenti trattati riguardano più lo studio del DBMS che lo studio di SQL, dunque si consiglia per un maggiore approfondimento di consultare il manuale del DBMS che si sta usando.

Introduzione alle query (capitolo 1)

Prime elementari regole

La sintassi del linguaggio SQL è abbastanza flessibile, sebbene ci siano delle regole da rispettare come in qualsiasi linguaggio di programmazione.

```
SELECT COGNOME, NOME
FROM PRESIDENTE
WHERE COGOME = 'Pertini'
```

In questo esempio tutti i caratteri, a parte 'Pertini', sono scritti in maiuscolo, ma non deve essere necessariamente così. Avremmo potuto anche scrivere così:

```
select cognome, nome
from presidente
where cogome = 'Pertini'
```

si noti però che 'Pertini' è scritto sempre nello stesso modo, infatti i riferimenti ai contenuti di un data base devono essere scritti con lo stesso tipo di caratteri in cui sono stati registrati.

Le parole chiave che abbiamo usato nella query sono:

- SELECT
- FROM
- WHERE

'cognome' e 'nome' sono dei campi e 'presidente' è una tabella. Quindi il comando recita: *seleziona visualizzandoli, i campi cognome e nome della tabella presidente la dove cognome = 'Pertini'*.

Esempio:

ASSEGNI

Assegno	Beneficiari	Importo	Note
1	Computer Shop	500.000	Stampante
2	Assicurazioni ASSO	954.000	Assicurazioni automobili
3	SNAM	650.000	Riscaldamento casa
4	Supermarket GS	490.000	Alimentari

5	Scuola	490.000	Scuola di musica

con l'espressione:

```
select * from assegni;
```

si ottiene:

ASSEGNO	BENEFICIARI	IMPORTO	NOTE
1	Computer Shop	500.000	Stampante
2	Assicurazioni ASSO	954.000	Assicurazioni automobili
3	SNAM	650.000	Riscaldamento casa
4	Supermarket GS	490.000	Alimentari
5	Scuola	490.000	Scuola di musica

L'asterisco (*) di *select* * indica al database di fornire tutte le colonne associate alla tabella specificata dalla clausola FROM.

Come termina una istruzione SQL:
In alcune implementazioni si usa il punto e virgola (;) in altre il punto e virgola o lo slash (/).

Selezionare le colonne o cambiare l'ordine di apparizione

Esempi:

con l'espressione:

```
select importo, assegno from assegni;
```

dalla tabella precedente si ottiene:

IMPORTO	ASSEGNO
500.000	1
954.000	2
650.000	3
490.000	4

490.000 5

clausola DISTINCT (query senza duplicati)

Esaminando il contenuto del campo Importo della tabella ASSEGNI, si potrà notare che il valore 490.000 appare due volte. Possiamo visualizzare tale colonna facendo comparire i valori ripetuti una sola volta:

```
select distinct Importo from ASSEGNI;
```

```
……….
IMPORTO
……..
500.000
954.000
650.000
490.000
```

Altro esempio:

DOCENTI

Nome	Cognome	Materia
Lara	Bianco	Italiano
Lara	Bianco	Storia
Mario	Guidi	Diritto
Mario	Guidi	Economia
Anna	Rossi	Matematica

Con l'espressione:

```
SELECT DISTINCT NOME, COGNOME
FROM DOCENTI;
```

si ottiene:

NOME	COGNOME
Lara	Bianco
Mario	Guidi
Anna	Rossi

ESERCIZI (capitolo 1)

1) Le seguenti query non funzionano. Perché?

   a. Select * from persone

   b. Select *

   c. Select nome cognome FROM persone;

2) Le seguenti istruzioni forniscono lo stesso risultato?

   SELECT * FROM PERSONE;

   select * from persone;

3) Quale delle seguenti istruzioni SQL è corretta?

   a. select * from persone;

   b. select * from persone
      /

   c. select *
      from persone;

4) Utilizzando la tabella sottostante scrivere una query per selezionare soltanto il contenuto delle colonne ETA e NOME.

ANAGRAFICA

NOME	COGNOME	ETA
Giovanni	Bruni	62
Antonio	Rossi	43
Mario	Rossi	49
Piero	Bianchi	37

Esmeralda	Gnocca	31

5) Dalla tabella sottostante estrapolare, senza ripetizioni, le squadre di calcio presenti.

TIFOSERIA

NOME	COGNOME	SQUADRA_APPARTENENZA
ARTURO	ROSSI	LAZIO
GIOVANNI	ROSSI	LAZIO
MARIO	ROSSI	LAZIO
MARIA	MARCHETTI	NAPOLI
MARIA	MARCHETTI	ROMA

6) La seguente query è giusta? e se sbagliata, perché?

```
SELECT NOME,  DISTINCT SQUADRA_APPARTENENZA
FROM TIFOSERIA;
```

7) La seguente query è giusta? e se sbagliata, perché?

```
SELECT  DISTINCT COGNOME, NOME
FROM TIFOSERIA;
```

8) La seguente query è giusta? e se sbagliata, perché?

```
SELECT  DISTINCT *
FROM TIFOSERIA;
```

Espressioni, condizionali e operatori (capitolo 2)

Condizioni

Tutte le volte che si vuole trovare un particolare elemento o gruppo di elementi in un data base, occorre specificare una o più condizioni. Le condizioni sono introdotte dalla clausola WHERE.

Esempio:

STUDENTI

NOME	Cognome	ANNO	Classe	Sezione
Mario	Bianchi	1976	1	A
Anna	Bianco	1973	2	B
Marta	Carli	1974	3	A
Gianni	Rossi	1972	4	A
Giulio	Mancini	1972	5	A
Max	Zunini	1976	5	B

con l'espressione:

```
SELECT *
FROM STUDENTI
WHERE CLASSE = 5
AND SEZIONE = 'A';
```

si ottiene:

NOME	COGNOME	ANNO	CLASSE	SEZIONE
Giulio	Mancini	1972	5	A

Operatori aritmetici

Sono gli operatori aritmetici: + (somma), - (sottrazione), / (divisione), * (moltiplicazione).

L'operatore somma:

data la tabella

PREZZO

Elemento	PrezzoIngrosso
Pomodori	340
Patate	510
Banane	670
Rape	450
Arance	890
Mele	230

nella seguente espressione:

```
SELECT ELEMENTO, PREZZOINGROSSO, PREZZOINGROSSO + 150
FROM PREZZO;
```

l'operatore + aggiunge 150 lire a ogni prezzo e genera la seguente tabella:

ElEMENTO	PREZZOINGROSSO	PREZZOINGROSSO + 150
Pomodori	340	490
Patate	510	660
Banane	670	820
Rape	450	600
Arance	890	1040
Mele	230	380

Operatore sottrazione:

l'operatore meno svolge due funzioni. La prima è quella di cambiare il segno a un numero, la seconda è quella di sottrarre i valori di una colonna da quelli di un'altra colonna. Vediamo il primo caso:

MINMAX

Regione	TempMin	TempMax
Piemonte	-4	10
Toscana	4	13
Sicilia	10	19
Lombardia	-2	9
Friuli	-3	8

con l'espressione

```
SELECT REGIONE, -TEMPMIN, -TEMPMAX
FROM MINMAX;
```

si ottiene

REGIONE	TEMPMIN	TEMPMAX
Piemonte	4	-10
Toscana	-4	-13
Sicilia	-10	-19
Lombardia	2	- 9
Friuli	3	- 8

Vediamo il secondo caso:

con l'espressione

```
SELECT REGIONE,
(TEMPMAX – TEMPMIN) Differenza
FROM MINMAX;
```

REGIONE	DIFFERENZA
Piemonte	14
Toscana	9
Sicilia	9
Lombardia	11
Friuli	11

Operatore divisione (/):

Esempio: abbiamo la necessità di vendere a metà prezzo

PREZZO

Elemento	PrezzoIngrosso
Pomodori	340
Patate	510
Banane	670
Rape	450
Arance	890
Mele	230

con l'espressione

```
SELECT Elemento, PrezzoIngrosso, (PrezzoIngrosso / 2) PrezzoVendita
FROM PREZZO;
```

si ottiene

ELEMENTO	PREZZOINGROSSO	PREZZOVENDITA
Pomodori	340	170
Patate	510	255
Banane	670	335
Rape	450	225
Arance	890	445
Mele	230	115

Operatore moltiplicazione (*):

Esempio: abbiamo la necessità di moltiplicare per 0.9

PREZZO

Elemento	PrezzoIngrosso
Pomodori	340
Patate	510
Banane	670
Rape	450
Arance	890
Mele	230

con l'espressione

```
SELECT Elemento, PrezzoIngrosso, (PrezzoIngrosso * 0.9) NuovoPrezzo
FROM PREZZO;
```

si ottiene

ELEMENTO	PREZZOINGROSSO	MUOVOPREZZO
Pomodori	340	306
Patate	510	459
Banane	670	603
Rape	450	405
Arance	890	801
Mele	230	207

Operatori di confronto

L'operatore (=):

Esempio: dalla seguente tabella vogliamo estrapolare tutti i dati dell'amico Mario

AMICI

Nome	Cognome	Telefono
Giovanni	Bruni	0763 546432
Antonio	Rossi	06 756499
Mario	Rossi	02 435591
Piero	Bianchi	06 326799

```
SELECT *
FROM AMICI
WHERE NOME = 'Mario';
```

NOME	COGNOME	TELEFONO
Mario	Rossi	02 435591

Gli operatori: maggiore (>) , maggiore o uguale (>=), minore (<), minore o uguale (<=), diverso (<>):

Questi operatori si usano allo stesso modo di come si usa l'operatore di uguaglianza. Vediamo alcuni esempi:

ANAGRAFICA

Nome	Cognome	Età
Giovanni	Bruni	55
Antonio	Rossi	43
Mario	Rossi	49
Piero	Bianchi	37

Voglio sapere chi ha più di 44 anni

```
SELECT *
FROM ANAGRAFICA
WHERE Età > 44;
```

NOME	COGNOME	ETÀ
Giovanni	Bruni	55
Mario	Rossi	49

Voglio sapere chi ha un'età diversa da 55 anni

```
SELECT *
FROM ANAGRAFICA
WHERE Età <> 55;
```

NOME	COGNOME	ETÀ
Antonio	Rossi	43
Mario	Rossi	49
Piero	Bianchi	37

L'operatore IS:

Modifichiamo ora la tabella ANAGRAFICA inserendo un altro nominativo di cui però non sappiamo l'età. In questo caso nel campo ETA verrà inserito in maniera 'automatica' il valore **NULL** che identifica l'assenza di dati:

NOME	COGNOME	ETA
Giovanni	Bruni	55
Antonio	Rossi	43
Mario	Rossi	49
Piero	Bianchi	37
Esmeralda	Gnocca	

Ora vogliamo conoscere il nome e il cognome delle persone di cui non abbiamo il dato età:

```
SELECT *
FROM ANAGRAFICA
WHERE ETA IS NULL;
```

NOME	COGNOME	ETA
Esmeralda	Gnocca	

L'operatore IS funziona con la clausola NULL, ma possiamo sostituirlo anche con l'operatore (=): WHERE ETA = NULL;

Operatori di caratteri

Gli operatori di caratteri possono essere utilizzati per manipolare il modo in cui le stringhe devo essere ricercate.

Operatore LIKE:

È possibile estrarre da un database quei dati che assomigliano a un certo schema, senza essere perfettamente identici allo schema specificato. Vediamo qualche esempio:

ANATOMIA

Nome	Posizione
Fegato	Destra-Addome
Cuore	Petto
Faringe	Gola
Vertebre	Centro-Dorso
Incudine	Orecchio
Rene	Dorso

```
SELECT *
FROM ANATOMIA
WHERE Posizione LIKE '%Dorso%';
```

NOME	POSIZIONE
Vertebre	Centro-Dorso
Rene	Dorso

Si noti l'uso del segno '%' dopo LIKE. Esso rappresenta zero, uno o più caratteri. **Il corrispondente carattere implementato da Access è '*'**.

```
SELECT *
FROM ANATOMIA
WHERE Posizione LIKE 'Dorso%';
```

NOME	POSIZIONE
Rene	Dorso

```
SELECT *
FROM ANATOMIA
WHERE Nome LIKE 'F%';
```

Nome	Posizione
Fegato	Destra-Addome
Faringe	Gola

Vediamo altri esempi:

NOMINATIVI

Nome	Cognome	Provincia
Giovanni	Bruni	CO
Antonio	Rossi	MI
Mario	Rossi	CT
Piero	Bianchi	PV

```
SELECT *
FROM NOMINATIVI
WHERE Provincia LIKE 'C_';
```

NOME	COGNOME	PROVINCIA
Giovanni	Bruni	CO
Mario	Rossi	CT

Il carattere di sottolineatura (_) è un carattere che sostituisce un singolo carattere e non il carattere spazio. **Il suo corrispettivo implementato da Access è '?'. Access implementa anche il segno '#' che sostituisce qualsiasi singola cifra (0, 1, 2, 3, ecc,).**

Operatore di concatenazione (||)

Il simbolo || serve a concatenare due stringhe. **IL corrispettivo operatore che si usa con Access è '&'.** Vediamo alcuni esempi:

AMICI

COGNOME	NOME	TELEFONO	PR	CAP
ROSSI	ALESSANDRA	0761 556632	BG	01023
VERDI	ALESSANDRA	02 345622	MI	03456
MERRILI	TITTI	0732 433388	CO	01255
BANFI	BARBY	0722 114381	BR	03367
PERSIANI	MARIA	0581 931522	CA	09941
MAZZA	JO	0359 118267	PV	01944
BORDONI	CHIARA	0445 668193	CT	01042

```
SELECT NOME || COGNOME  NOMECOMPLETO
FROM AMICI;
```

NOMECOMPLETO
ALESSANDRAROSSI
ALESSANDRAVERDI
TITTIMERRILI
BARBYBANFI
MARIAPERSIANI
JOMAZZA
BORDONICHIARA

Access non implementa, oltre che l'operatore '||', anche questo modo di ridenominare le colonne estrapolate:

```
SELECT NOME || COGNOME  NOMECOMPLETO
```

Ecco un'altra applicazione dell'operatore di concatenazione:

```
SELECT COGNOME || ',   ' || NOME AMICO
FROM AMICI;
```

AMICO
ROSI, ALESSANDRA
VERDI, ALESSANDRA
MERRILI, TITTI
BANFI, BARBY
PERSIANI, MARIA
MAZZA, JO
BORDONI, CHIARA

Gli operatori logici

Per comprendere a pieno gli *operatori logici* bisogna introdurre i fondamenti dell'algebra di Boole

Algebra di BOOLE

L'elemento essenziale del pensiero umano è la logica che permette all'uomo di formulare ragionamenti e di elaborare informazioni che gli vengono fornite dall'esterno. L'uomo si avvale di una logica esprimibile con un linguaggio che gli è congeniale e che si sa essere il linguaggio binario.

Il tipo di logica dei calcolatori è un modello applicativo di un sistema logico costituito nel secolo scorso dal matematico inglese George Boole che da lui prende il nome di '*algebra booleana'*.
Il sistema logico di Boole trae la sua prima ispirazione dal tentativo di introdurre un '*calcolo logico'* o, più significativamente di '*matematizzare'* le leggi del pensiero logico.
George Boole pubblicò nel 1856 un trattato in cui espose il suo articolato *calcolo logico*.

L'algebra delle proposizioni è la logica di cui si avvalgono i calcolatori per interpretare ed eseguire le istruzioni dei programmi; è anche la logica usata nella progettazione e per il funzionamento dei circuiti elettronici.

Algebra delle proposizioni

Le frasi del linguaggio della logica si differenziano notevolmente da quelle del linguaggio comune perché per ogni frase logica ha senso chiedersi se ciò che enuncia è vero o falso.
Le frasi del linguaggio della logica prendono il nome di *proposizioni logiche* o, più semplicemente, *proposizioni* (o *enunciati*).
Per esempio, le frasi:

a: Roma è capitale d'Italia;

b: 10 è un numero dispari;

c: la rosa è un fiore;

sono proposizioni logiche. Infatti posiamo dire con certezza che gli enunciati *a* e c sono veri mentre l'enunciato *b* falso.

Consideriamo ora i seguenti enunciati:

d: che bello volare;

e: hai visto Maria?;

f : domani pioverà;

riferendoci a queste frasi non possiamo dire se sono vere o false. 'Che bello volare' può essere una proposizione vera per chi ama il volo ma non per chi ne ha paura. La verità o la falsità della frase dipendono solo dalle emozioni soggettive. Così non si può dire se le frasi *e* e *f* sono vere o false. Quindi le frasi *d, e, f* non sono proposizioni logiche.
Da questo possiamo dire:

In logica si chiama proposizione ogni frase per la quale ha senso dire che è "vera", o è "falsa".

La logica delle proposizioni è anche detta logica *bivalente* proprio perché ogni proposizione può avere uno solo dei due valori: *vero* o *falso*.

Vediamo ora le operazioni sulle *proposizioni*.

Congiunzione logica:
Date le due proposizioni:

a: Mantova è una città;
b: L'Italia è una nazione;

la proposizione:

r : Mantova è una città e l'Italia è una nazione

è una proposizione composta, ottenuta operando sulle proposizioni *a* e *b* per mezzo del connettivo e. Il valore di verità di *r* dipende dai valori delle due proposizioni. L'operazione binaria che da come risultato il valore di verità *r* si chiama *congiunzione logica.*

DEFINIZIONE: Si chiama congiunzione logica un'operazione che ad ogni coppia di proposizioni *a*, *b* associa la proposizione composta *r* che è vera se *a* e *b* sono entrambe vere e falsa in tutti gi altri casi.

Il connettivo *congiunzione* viene indicato con AND. La tavola della verità della *congiunzione logica* è la seguente e si ottiene considerando tutte le possibile coppie di valori di verità attribuibili ad *a* e *b*.

a	***b***	***r = a* AND *b***
FALSO	FALSO	**FALSO**
FALSO	VERO	**FALSO**
VERO	FALSO	**FALSO**
VERO	VERO	**VERO**

Disgiunzione logica:
Date le proposizioni:

a: il quadrato è un poligono
b: Dante è un poeta

la proposizione:

r : il quadro è un poligono o Dante è un poeta

è una proposizione composta, ottenuta operando sulle proposizioni *a* e *b* per mezzo del connettivo o. L'operazione binaria che da come risultato il valore di verità *r* si chiama *disgiunzione logica.*

DEFINIZIONE: Si chiama *disgiunzione logica* un'operazione che ad ogni coppia di proposizioni *a, b* associa la proposizione composta *r* che è vera se almeno una delle due proposizioni è vera, falsa se *a* e *b* sono entrambe false.

Il connettivo *disgiunzione logica* viene indicato con OR. La tavola della verità della *disgiunzione logica* è la seguente e si ottiene considerando tutte le possibile coppie di valori di verità attribuibili ad *a* e *b*.

a	***b***	***R = a* OR *b***
FALSO	FALSO	**FALSO**
FALSO	VERO	**VERO**
VERO	FALSO	**VERO**
VERO	VERO	**VERO**

Negazione logica:
Date le proposizioni:

a: 3 è un numero dispari
b: 3 non è un numero dispari

è evidente che la proposizione *b* è la negazione della proposizione *a*. Possiamo dire che *b* si ottiene da *a* mediante la sua negazione.

DEFINIZIONE: Si chiama negazione logica un'operazione che ad una proposizione *a,* associa la proposizione *b* la quale risulta vera se *a* è falsa e falsa se *a* è vera.

La *negazione logica* viene indicata con NOT. La tavola della verità della *negazione logica* è la seguente.

a	**NOT *a***
FALSO	**VERO**
VERO	**FALSO**

Negli esempi che utilizzeremo, per meglio comprendere gli *operatori logici,* non appariranno espressioni *booleane* complesse ma espressioni semplici, cioè composte da un solo operatore. Dobbiamo far presente però, che tali operatori sono utilizzabili come qualsiasi altro operatore matematico, dunque nulla ci impedisce di creare delle vere e proprie espressioni algebriche lunghe a piacere.

La breve panoramica sull'algebra booleana termina qui; vista la semplicità dell'argomento lasciamo al lettore il compito di utilizzare e 'trasferire' quanto a appreso, alle specifiche del linguaggio SQL.

Operatore AND:

Indica che entrambe le espressioni che si trovano ai suoi lati devono essere soddisfatte, vediamo un esempio:

FERIE

COGNOME	ID_DIPENDENTE	ANNI	FERIE_GODUTE
MARINI	101	2	4
ROSSI	104	5	23
ZAPPA	107	8	45
VERDI	233	4	80
BOLDI	210	15	100
GIALLI	211	10	78

vogliamo sapere quali impiegati hanno lavorato nell'azienda per 5 anni o più e hanno utilizzato più di 50 giorni di ferie.

```
SELECT COGNOME
FROM FERIE
WHERE ANNI >= 5
AND FERIE_GODUTE  > 50;
```

COGNOME
BOLDI
GIALLI

L'operatore OR:

È sufficiente che una sola espressione sia verificata per ottenere il valore TRUE, vediamo un esempio:
vogliamo sapere i cognomi dei dipendenti che hanno più di 5 anni di servizio o hanno goduto ferie per più di 80 giorni.

```
SELECT COGNOME
FROM FERIE
WHERE ANNI <= 5
OR FERIE_GODUTE > 80;
```

COGNOME
MARINI
ROSSI
VERDI
BOLDI

Operatore NOT:

Ha il compito di invertire il significato di una condizione, vediamo un esempio: vogliamo conoscere i cognomi che non iniziano per B.

```
SELECT COGNOME
FROM FERIE
WHERE COGNOME NOT  LIKE 'B%';
```

COGNOME
MARINI
ROSSI
VERDI
ZAPPA
GIALLI

Gli operatori di insieme

SQL mette a disposizione degli operatori insiemistici, da applicare nella scrittura delle nostre interrogazioni. Tali operatori operano sul risultato di più *select.* **Gli attributi interessati dagli *operatori di insieme* devono esser di tipo compatibile tra loro**.

Gli operatori disponibili sono gli operatori di UNION (unione), INTERSECT (intersezione) e MINUS (differenza), il significato è analogo ai corrispondenti operatori dell'algebra insiemistica che adesso vedremo brevemente:

Teoria sugli operatori insiemistici

Unione:

Si definisce ***unione*** fra A e B l'insieme formato dagli elementi che appartengono *almeno* a uno dei due insiemi A e B.
L'unione fra gli insiemi A e B si indica scrivendo: $A \cup B$.

$A = \{G, 9, R\}$ $A = \{9, B\}$

$$A \cup B = \{G, 9, R, B\}$$

Intersezione:
Si definisce ***intersezione*** fra *A* e *B* il sottoinsieme formato dagli elementi *comuni* agli insiemi *A* e *B*.
L'intersezione fra gli insiemi *A* e *B* si indica scrivendo: $A \cap B$.

$$A = \{G, 9, R\} \qquad B = \{9, B\}$$

$$A \cap B = \{9\}$$

Differenza fra due insiemi:
Si definisce ***differenza*** fra *A* e *B*, dati in questo ordine, il sottoinsieme formato dagli elementi di *A* che non appartengono a *B*.
La *differenza* fra gli insiemi *A* e *B* si indica scrivendo: $A \setminus B$.

$$A = \{G, 9, R\} \qquad B = \{9, B\}$$

$$A \setminus B = \{G, R\}$$

Chiariti questi concetti basilari passiamo ora a vedere i corrispondenti operatori del linguaggio SQL:

Operatore UNION e UNION ALL:

L'operatore UNION restituisce il risultato di più query escludendo le righe duplicate, vediamo un esempio:

CALCIO

NOME
MARINI
BRAVO
ROSSI
VERDI
MARRONI

CALCETTO

NOME
MARINI
BACCO
ROSSI
DINI
MARRONI

GIALLI
GIANNINI

FALCONE
GIANNINI

Vogliamo sapere quali persone giocano in una squadra o nell'altra:

```
SELECT NOME FROM CALCETTO
UNION
SELECT NOME FROM CALCIO;
```

```
NOME
----------
MARINI
BACCO
BRAVO
ROSSI
VERDI
DINI
MARRONI
FALCONE
GIALLI
GIANNINI
```

L'operatore UNION fornisce 10 nomi distinti estratti dalle due tabelle, se invece vogliamo vedere tutti i nomi che compaiono nelle due tabelle, ***duplicati inclusi***, aggiungiamo **ALL** alla precedente espressione:

```
SELECT NOME FROM CALCETTO
UNION ALL
SELECT NOME FROM CALCIO;
```

```
NOME
------------
MARINI
BACCO
ROSSI
DINI
MARRONI
FALCONE
GIANNINI
MARINI
BRAVO
ROSSI
VERDI
MARRONI
GIALLI
GIANNINI
```

Operatore INTERSECT:

Restituisce l'intersezione (valori comuni a tutti gli insiemi coinvolti) del risultato delle query. La seguente istruzione SELECT mostra l'elenco dei giocatori che appartengono sia alla squadra di calcio che di calcetto. (Le tabelle CALCIO e CALCETTO si trovano nella pagina precedente). **Questo operatore non è implementato da Access.**

```
SELECT NOME FROM CALCETTO
INTERSECT
SELECT NOME FROM CALCIO;
```

```
NOME
----------
MARINI
ROSSI
MARRONI
GIANNINI
```

Operatore MINUS (differenza):

Restituisce le righe della prima query che non sono presenti nella seconda. **Questo operatore non è implementato da Access.**

```
SELECT * FROM CALCIO
MINUS
SELECT * FROM CALCETTO;
```

NOME
BRAVO
VERDI
GIALLI

Altri operatori: IN e BETWEEN

Gli operatori IN e BETTWEEN forniscono una scorciatoia per quelle operazioni che possono essere svolte anche in altri modi. Ad esempio, per trovare tutti gli amici che vivono in provincia di Como, Pavia, e Bergamo,

AMICI

COGNOME	NOME	TELEFONO	PR	CAP
ROSSI	MARIA	0761 556632	BG	01023
VERDI	MARIA	02 345622	MI	03456
MARRONI	ANTONIO	0732 433388	CO	01255
BANFI	BARBY	0722 114381	BR	03367
PERSIANI	LUCA	0581 931522	CA	09941
MAZZA	ALBERTO	0359 118267	PV	01944
BATTISTI	CHIARA	0445 668193	CT	01042

possiamo usare le seguenti espressioni:

```
SELECT *
FROM AMICI
WHERE
PR = 'CO'
OR
PR = 'PV'
OR
PR = 'BG';
```

```
SELECT *
FROM AMICI
WHERE PR IN ('CO', 'PV', 'BG');
```

Il risultato che si ottiene per entrambe le espressioni è il seguente:

```
COGNOME    NOME       TELEFONO   PR       CAP
-------------------------------------------------------------------
ROSSI          MARIA       23423          BG       01023
MARRONI    ANTONIO    45567          CO       01255
MAZZA         ALBERTO   567878         PV       01944
```

Facciamo ora un altro esempio:

PREZZO

ELEMENTO	PREZZO_INGROSSO
POMODORI	340
PATATE	510
BANANE	670
RAPE	450
ARANCE	890

MELE	230

```
SELECT *
FROM PREZZO
WHERE PREZZO_INGROSSO >= 250
AND
PREZZO_INGROSSO <= 750;
```

```
SELECT *
FROM PREZZO
WHERE PREZZO_INGROSSO BETWEEN 250
AND 750;
```

Il risultato che si ottiene per entrambe le espressioni è il seguente:

ELEMENTO	PREZZO_INGROSSO
POMODORI	340
PATATE	510
BANANE	670
RAPE	450

ESERCIZI (capitolo 2)

Utilizzare la tabella AMICI, qui riportata, per rispondere ai quesiti dove non è specificata altra tabella.

AMICI

COGNOME	NOME	PR
ROSSI	MARIA	BG
VERDI	MARIA	MI
MARRONI	ANTONIO	CO
BANFI	BARBY	BR
PERSIANI	LUCA	CA
MAZZA	ALBERTO	PV
BATTISTI	CHIARA	CT

1) Scrivere una query per selezionare tutti i cognomi che iniziano con la lettera M.

2) Scrivere una query per selezionare gli amici che vivono in provincia di Bergamo (BG) e il cui nome è MARIA.

3) Quale è il risultato di questa query?

```
SELECT NOME, COGNOME
FROM AMICI
WHERE NOME = 'MARIA'
OR COGNOME = 'MAZZA';
```

4) Quale è il risultato di questa query?

```
SELECT NOME, COGNOME
FROM AMICI
WHERE NOME = 'MARIA'
AND COGNOME = 'MAZZA';
```

5) Quale scorciatoia potrebbe essere utilizzata in alternativa a *WHERE A >= 10 AND A <= 30* ?
6) Scrivere una interrogazione che dalla tabella sottostante estrapoli il nome delle donne sposate

PERSONE

ID_PERSONA	NOME	ID_CONIUGE	SESSO
1	ANTONIO	12	M
12	SILVIA	1	F
2	GIULIO	7	M
3	MARIA		F
6	ROBERTA	9	F
7	ANTONELLA	2	F
9	ARTURO	6	M

7) Scrivere una interrogazione che dalla tabella *PERSONE* estrapoli i nomi che iniziano per 'A' e finiscono per 'O'.

8) Scrivere una interrogazione che dalla tabella *PERSONE* estrapoli i nomi in cui la quarta lettera sia una 'O'.

9) Scrivere una interrogazione che ci visualizzi tutti i dati della tabella sottostante, più un colonna dal nome '*PrezzoVendita'* in cui dovranno comparire i rispettivi prezzi della colonna *PrezzoIngrosso* ma aumentati del 15%

PREZZI

Elemento	PrezzoIngrosso
Pomodori	340
Patate	510
Banane	670
Rape	450
Arance	890
Mele	230

10) Scrivere un interrogazione che inverta il segno dei valori presenti nella colonna PrezzoIngrosso della

tabella *PREZZI*.

Il database costituito dalle tabelle Cacciatori, Pescatori e Scalatori appartiene ad un circolo sportivo e contiene i nominativi degli iscritti a tale circolo. Gli iscritti sono raggruppati sulle tabelle in base allo sport che essi fanno. Chiaramente la stessa persona può fare più di uno sport e quindi comparire in più di una tabella. Per gli esercizi seguenti faremo riferimento a questo piccolo database.

CACCIATORI

Codice	Nome
35	ROSSI
27	NERI
12	BINACHI
2	BISCARDI

PESCATORI

Codice	Nome
4	ROSSI
77	MANCINI
49	CRUCIANI
11	MARCA

SCALATORI

Codice	Nome
27	NERI
11	MARCA
1	MICHELI
2	BISCARDI

11) Scrivere una *query* per visualizzare tutti i nominativi iscritti al circolo.

12) Scrivere una *query* per visualizzare i cacciatori che non siano anche scalatori.

13) Scrivere una *query* per visualizzare gli scalatori che non siano anche cacciatori.

14) Scrivere una *query* per visualizzare i pescatori che siano anche cacciatori.

15) Se dovessimo scrivere una *query* per visualizzare i cacciatori che siano anche pescatori potremmo utilizzare la soluzione dell'esercizio N° 14?

16) Scrivere una *query* per visualizzare tutti i nominativi iscritti al circolo il cui nome finisce con 'I' e ci sia almeno una 'A'.

17) Scrivere una *query* per visualizzare tutti i nominativi iscritti al circolo il cui nome finisce con 'I' o ci sia almeno una 'A'.

Funzioni (capitolo 3)

Le *funzioni*, nell'ambito dei linguaggi di terza generazioni (linguaggi procedurali), sono delle particolari procedure le quali passandogli dei valori (parametri) esse ci restituiscono (ritornano) un valore.
Anche se SQL non è un linguaggio procedurale, implementa le funzioni nella stessa maniera
ma con una differenza sostanziale:
Nei linguaggi procedurali noi stessi possiamo crearci delle funzioni, con SQL ciò non è possibile e quindi possiamo utilizzare solo quelle funzioni che ci mette a disposizione il DBMS che stiamo usando.

In questo capitolo vedremo molte funzioni, ma soltanto le prime 5 (COUTNT, SUM, AVG, MAX e MIN) sono definite nello standard SQL. Queste prime cinque funzioni sono le più importanti e dobbiamo impararle bene, esse sono sempre presenti nella maggior parte dei DBMS a differenza delle restanti, che a volte non appaiono affatto o sono implementate con una sintassi diversa.

Funzioni aggregate

Le funzioni che analizzeremo in questo paragrafo hanno la particolarità di restituire un solo valore. Inoltre, dato che operano su insiemi di righe, vengono anche chiamate **funzioni di gruppo**.
Gli esempi di questo paragrafo utilizzano la tabella IMPIEGATO:

IMPIEGATO

NOME	COGNOME	DIPARTIMENTO	UFFICIO	STIPENDIO
MARIO	ROSSI	AMMINISTRAZIONE	10	L. 4.500.000
CARLO	BIANCHI	PRODUZIONE	20	L. 360.000
GIUSEPPE	VERDI	AMMINISTRAZIONE	20	L. 4.000.000
FRANCO	NERI	DISTRIBUZIONE	16	L. 4.500.000
CARLO	ROSSI	DIREZIONE	14	L. 7.300.000
LORENZO	LANZI	DIREZIONE	7	L. 730.000
PAOLA	BORRONI	AMMINISTRAZIONE	75	L. 4.000.000
MARCO	FRANCO	PRODUZIONE	46	L. 4.000.000

COUNT

Restituisce il numero di righe che soddisfano la condizione specificata nella clausola WHERE.
Vediamo un esempio: voglio conoscere il numero di impiegati che appartengono al dipartimento produzione

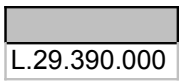
```
SELECT COUNT (*)
FROM IMPIEGATO
WHERE DIPARTIMENTO = 'PRODUZIONE';
```

2

SUM

Questa funzione somma tutti i valori di una colonna, vediamo un esempio: voglio ottenere la somma di tutti gli stipendi

```
SELECT SUM(STIPENDIO)
FROM IMPIEGATO;
```

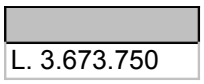


La funzione SUM opera soltanto con i numeri, se viene applicata a un campo non numerico, si ottiene un messaggio di errore.

AVG

Calcola la media aritmetica dei valori di una colonna. Vediamo un esempio: voglio conoscere lo stipendio medio della tabella IMPIEGATO (vedi pagina precedente).

```
SELECT AVG(STIPENDIO)
FROM IMPIEGATO;
```

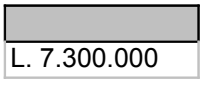


La funzione AVG opera soltanto con i numeri.

MAX

Questa funzione serve a trovare il valore massimo di una colonna. Per esempio vogliamo sapere a quanto ammonta lo stipendio maggiore.

```
SELECT MAX(STIPENDIO)
FROM IMPIEGATO;
```



La funzione MAX opera anche con i caratteri: la stringa ‘Maria’ è maggiore della stringa ‘Giovanna’.

MIN

Questa funzione opera in modo analogo a MAX, ad eccezione del fatto che restituisce il valore minimo di una colonna. Per trovare il minimo stipendio della tabella IMPIEGATO si usa la seguente espressione:

```
SELECT MIN(STIPENDIO)
FROM IMPIEGATO;
```

L. 360.000

La funzione MIN opera anche con i caratteri: la stringa ‘AAA’ è minore della stringa ‘BB’.

STDDEV
Deviazione standard

Questa funzione calcola la deviazione standard di una colonna di numeri. **Non esiste in Access** . Vediamo un esempio:

TEMPERATURE

CITTA	TEMPERATURA
ROMA	10
ANCONA	8
NAPOLI	15

```
SELECT STDDEV(TEMPERATURA)
FROM TEMPERATURE;
```

STDDEV(TEMPERATURA)
3,6055513

VARIANGE
Quadrato della deviazione standard

Questa funzione calcola il quadrato della deviazione standard. **Non esiste in Access**. Vediamo un esempio usando la tabella usata precedentemente:

```
SELECT VARIANCE(TEMPERATURA)
FROM TEMPERATURE;
```

```
VARIANCE(TEMPERATURA)
---------------------------------------
                  13
```

Assiomi delle funzioni aggregate:

- Restituiscono un solo valore
- La clausola *SELECT* può essere seguita solo e soltanto dalla funzione di aggregazione
- Vanno applicate a tipi di dato a loro compatibili

Funzioni temporali

Queste funzioni operano su *date* e *orari;* sono molto potenti e quando servono si rivelano essere molto utili. **Alcuni DBMS, come Access, non le implementano o usano sintassi diverse**.

ADD_MONTHS

Questa funzione aggiunge un numero di mesi a una data specificata. Vediamo un esempio usando la tabella sottostante.

PROGETTO

COMPITO	DATA_INIZIO	DATA_FINE
AVVIO PROGETTO	01-Apr-99	02-Apr-99
DEFINIZIONE SPECIFICHE	02-Apr-99	01-Mag-99
CONTROLLO TECNICO	01-Giu-99	30-Giu-99
PROGETTAZIONE	01-Lug-99	02-Set-99
COLLAUDO	03-Set-99	17-Dic-99

```
SELECT COMPITO, DATA_INIZIO,
ADD_MONTHS(DATA_FINE,2)
FROM PROGETTO;
```

COMPITO	DATA_INIZIO	ADD_MONTH
AVVIO PROGETTO	01-Apr-99	02-Giu-99
DEFINIZIONE SPECIFICHE	02-Apr-99	01-Lug-99
CONTROLLO TECNICO	01-Giu-99	31-Ago-99
PROGETTAZIONE	01-Lug-99	02-Nov-99
COLLAUDO	03-Set-99	17-Feb-00

LAST_DAY

Questa funzione fornisce l'ultimo giorno di un mese specificato (se il mese è di 30, 31, 29 o 28 giorni).

```
SELECT DATA_FINE, LAST_DAY(DATA_FINE)
FROM PROGETTO;
```

DATA_FINE	LAST_DAY(DATA_FINE)
02-Apr-99	30-Apr-99
01-Mag-99	31-Mag-99
30-Giu-99	30-Giu-99
02-Set-99	30-Set-99
17-Dic-99	31-Dic-99

MONTHS_BETWEEN

Questa funzione serve per sapere quanti mesi sono compresi tra il mese *x* e il mese *y*.

```
SELECT COMPITO, DATA_INIZIO, DATA_FINE,
MONTHS_BETWEEN(DATA_FINE, DATA_INIZIO) DURATA
FROM PROGETTO;
```

COMPITO	DATA_INIZ	DATA_FINE	DURATA
AVVIO PROGETTO	01-Apr-99	02-Apr-99	,03225806
DEFINIZIONE SPECIFICHE	02-Apr-99	01-Mag-99	,96774194
CONTROLLO TECNICO	01-Giu-99	30-Giu-99	,93548387
PROGETTAZIONE	01-Lug-99	02-Set-99	2,0322581
COLLAUDO	03-Set-99	17-Dic-99	3,4516129

NEW_TIME

Questa funzione consente di regolare l'ora e la data in funzione del fuso orario. Vediamo un esempio utilizzando la tabella Progetto:

```
SELECT DATA_FINE AST,
NEW_TIME(DATA_FINE, 'AST' ,  'PDT')
FROM PROGETTO;
```

AST	NEW_TIME(DATA, 'AST', 'PDT')
02-Apr-99	01-Apr-99
01-Mag-99	30-Apr-99
30-Giu-99	29-Giu-99
02-Set-99	01-Set-99
17-Dic-99	16-Dic-99

(*Vedi le sigle dei fusi orai nella pagina seguente*)

FUSI ORARI

SIGLA	FUSO ORARIO
AST o ADT	Atlantic Standard o Atlantic Daylight Time
BST o BDT	Bering Standard o Bering Daylight Time
CST o CDT	Central Standard o Central Daylight Time
EST o EDT	Eastern Standard o Eastern Daylight Time
GMT	Greenwich Mean Time
HST o HDT	Alaska-Hawaii Standard o Hawaii Daylight Time
MST o MDT	Mountain Standard o Mountain Daylight Time
NST	Newfoundland Standard Time
PST o PDT	Pacific Standard o Pacific Daylight Time
YST o YDT	Yukon Standard o Yukon Daylight Time

NEXT_DAY

Questa funzione imposta una nuova data, successiva a quella passatagli come primo parametro, in base al giorno della settimana passatogli come secondo parametro.

```
SELECT DATA_INIZIO,
NEXT_DAY(DATA_INIZIO, 'VENERDI')
FROM PROGETTO;
```

DATA_INIZ	NEXT_DAY(
01-Apr-99	02-Apr-99
02-Apr-99	09-Apr-99
01-Giu-99	04-Giu-99
01-Lug-99	02-Lug-99
03-Set-99	10-Set-99

SYSDATE

Questa funzione fornisce la data e l'ora del sistema. Vediamo degli esempi:

```
SELECT DISTINCT SYSDATE
FROM PROGETTO;
```

```
SYSDATE
---------
18-Mar-99
```

Per sapere a che punto del progetto si è arrivati oggi:

```
SELECT *
FROM PROGETTO
WHERE DATA_INIZIO > SYSDATE;
```

Funzioni aritmetiche

Si verifica spesso il caso in cui i dati che vengono estrapolati da un database richiedono delle operazioni matematiche. Molte implementazioni di SQL includono delle funzioni aritmetiche simili a queste. Gli esempi esposti si basano sulla tabella Numeri:

NUMERI

A	B
3,1415	4
-45	,707
5	9
-57,667	42
15	55
-7,2	5,3

ABS

Questa funzione calcola il valore assoluto del numero specificato. Vediamo un esempio:

```
SELECT ABS(A)VALORE_ASSOLUTO
FROM NUMERI;
```

```
VALORE_ASSOLUTO
---------------------------
         3,1415
             45
              5
         57,667
             15
            7,2
```

CEIL

Questa funzione fornisce il più piccolo numero intero che è maggiore o uguale al suo argomento.
Questa sintassi non è implementata da Access.

```
SELECT A, CEIL(A) MAX_INTERI
FROM NUMERI;
```

```
         A          MAX_INTERI
---------------------------------------
    3,1415               4
       -45             -45
         5               5
   -57,667             -57
        15              15
      -7,2              -7
```

FLOOR

Questa funzione fornisce il più grande numero intero che è minore o uguale al suo argomento.
Questa sintassi non è implementata da Access.

```
SELECT A, FLOOR(A) MINIMI_INTERI
FROM NUMERI;
```

```
         A          MINIMI_INTERI
-----------------------------------------------
    3,1415                 3
       -45               -45
         5                 5
   -57,667               -58
        15                15
      -7,2                -8
```

SIGN

La funzione SIGN restituisce –1 se il suo argomento è minore di zero e restituisce 1 se il suo argomento è maggiore o uguale a zero. **Questa sintassi non è implementata da Access.** Vediamo un esempio:

```
SELECT A, SIGN(A)
FROM NUMERI;
```

A	SIGN(A)
3,1415	1
-45	-1
5	1
-57,667	-1
15	1
-7,2	-1

È possibile anche utilizzare SIGN in una query SELECT . . . WHERE come questa:

```
SELECT A
FROM NUMERI
WHERE SIGN(A) = 1;
```

A
3,1415
5
15

Funzioni trigonometriche

Le funzioni trigonometriche **COS**, **SIN**, **TAN** sono molto utili in applicazioni in cui si richiede l'uso di tali calcoli. Tutte queste funzioni operano supponendo che l'angolo *n* sia espresso in radianti. **Queste funzioni, stranamente, sono implementate da Access.** Vediamo alcuni esempi usando la tabella ANGOLI:

ANGOLI

RADIANTI
3,14
6,28
1,57

COS

Calcola il coseno del parametro passatogli come angolo espresso in radianti:

```
SELECT RADIANTI, COS(RADIANTI)
FROM ANGOLI;
```

```
RADIANTI       COS(RADIANTI)
-----------------------------------------
   3,14             -,9999987
   6,28              ,99999493
   1,57              ,00079633
```

SEN

Calcola il seno del parametro passatogli come angolo espresso in radianti:

```
SELECT RADIANTI, SIN(RADIANTI)
FROM ANGOLI;
```

```
RADIANTI            SIN(RADIANTI)
-------------------------------------------------
   3,14                  ,00159265
   6,28                 -,0031853
   1,57                  ,99999968
```

TAN

Calcola la tangente del parametro passatogli come angolo espresso in radianti:

```
SELECT RADIANTI, TAN(RADIANTI)
FROM ANGOLI;
```

```
 RADIANTI           TAN(RADIANTI)
-------------------------------------------------
   3,14                -,0015927
   6,28                -,0031853
   1,57              1255,7656
```

Funzioni sulle potenze, logaritmi e radici

Per gli esempi verrà usata la tabella NUMERI sottostante:

NUMERI

A	B
3,1415	4
-45	,707
5	9
-57,667	42
15	55
-7,2	5,3

EXP

Questa funzione permette di elevare *e* a un esponente (*e* è una costante matematica che viene utilizzata in varie formule). Vediamo un esempio:

```
SELECT A, EXP(A)
FROM NUMERI;
```

A	EXP(A)
3,1415	23,138549
-45	2,863E-20
5	148,41316
-57,667	9,027E-26
15	3269017,4
-7,2	,00074659

LN

Questa funzione calcola il logaritmo naturale. **Questa funzione non è implementata da Access**
Vediamo un esempio in cui si vuole calcolare i logaritmi della colonna A della tabella NUMERI:

```
SELECT A, LN(A)
FROM NUMERI;
```

ERRORE:
ORA-01428: l'argomento '-45' è esterno all'intervallo

Il messaggio d'errore che viene visualizzato è dato dal fatto che non è possibile determinare un logaritmo di un valore negativo quando la base è positiva: non esiste nessun esponente che elevato ad *e* (valore positivo) ci da come risultato un valore negativo.
Il 'problema' può essere risolto inserendo all'interno della funzione LN la funzione ABS che ci restituisce i valori assoluti di quelli specificati:

```
SELECT A, LN(ABS(A))
FROM NUMERI;
```

A	LN(ABS(A))
3,1415	1,1447004
-45	3,8066625
5	1,6094379
-57,667	4,0546851
15	2,7080502
-7,2	1,974081

LOG

Questa funzione richiede due argomenti e calcola il logaritmo del secondo avendo come base il primo. Vediamo un esempio in cui si calcolano i logaritmi del numero 2 aventi come base i valori della colonna B della tabella NUMERI:

```
SELECT B, LOG(B, 2)
FROM NUMERI;
```

B	LOG(B,2)
4	,5
,707	-1,999129
9	,31546488
42	,18544902
55	,17296969
5,3	,41562892

Questa funzione non ci permette, però, di calcolare il logaritmo in cui la base è negativa, dunque il primo argomento che viene passato alla funzione dovrà essere sempre maggiore di zero.

POWER

Questa funzione non è implementata da Access. Questa funzione consente di elevare un numero alla potenza di un altro. Il primo argomento è elevato alla potenza del secondo. Vediamo un esempio:

```
SELECT A, B, POWER(A, B)
FROM NUMERI;
```

ERRORE:
ORA-01428: l'argomento '-45' è esterno all'intervallo

Sembrerebbe che non sia possibile (matematicamente) elevare un valore negativo ad un indice frazionario, ma non è così, il problema dunque sussiste forse solamente per SQL implementato da Oracle. Il problema può essere risolto usando opportune funzioni viste in precedenza o evitando di far calcolare la potenza di un numero negativo usando un indice frazionario. Facciamo un'altra prova:

```
SELECT A, B, POWER(B, A)
FROM NUMERI;
```

A	B	POWER(B,A)
3,1415	4	77,870231
-45	,707	5972090,5
5	9	59049
-57,667	42	2,467E-94
15	55	1,275E+26
-7,2	5,3	6,098E-06

SQRT

Questa funzione è implementata da Access con la sintassi 'SQR(*nome_campo*)'. La funzione SQRT restituisce la radice quadrata di un argomento. Poiché la radice quadrata di un numero negativo non esiste, non è possibile utilizzare questa funzione con i numeri negativi.
Vediamo una esempio:

```
SELECT B, SQRT(B)
FROM NUMERI;
```

B	SQRT(B)
4	2
,707	,84083292
9	3
42	6,4807407
55	7,4161985
5,3	2,3021729

Funzioni di caratteri

Queste funzioni ci permettono di manipolare i dati da visualizzare in tutti i modi e formati desiderati. Sono particolarmente utili quando abbiamo la necessità di rendere i dati più leggibili o quando vogliamo estrapolare delle informazioni sulle stringhe e i caratteri rappresentanti le informazioni.
Gli esempi presentati si basano sulla tabella CARATTERI sottostante:

CARATTERI

COGNOME	NOME	S	CODICE
ROSSI	GIGI	A	32
BIANCHI	MARIO	J	67
NERI	MARIA	C	65
BRUNI	ANGELO	M	87
SCURI	ANNA	A	77
VERDI	ANTONIO	G	52

CHR

Questa funzione fornisce il carattere corrispondente al codice ASCII passatogli. Vediamo un esempio:

```
SELECT CODICE, CHR(CODICE)
FROM CARATTERI;
```

CODICE	CH
32	
67	C
65	A
87	W
77	M
52	4

CONCAT

Questa sintassi non è accettata da Access. L'equivalente di questa funzione è stato utilizzato nel Capitolo 2, quando si è parlato di operatori. Il simbolo usato nel capitolo 2 è il seguente: || che unisce insieme due stringhe di caratteri, come la funzione CONCAT. Ecco un esempio:

```
SELECT CONCAT(NOME, COGNOME) "NOME E COGNOME"
FROM CARATTERI;
```

```
NOME E COGNOME
----------------------------------
GIGIROSSI
MARIOBIANCHI
MARIANERI
ANGELOBRUNI
ANNASCURI
ANTONIOVERDI
```

INITCAP

La funzione INITCAP trasforma in maiuscolo o lascia in maiuscolo il primo carattere di una parola e trasforma in minuscolo o lascia in minuscolo tutti gli altri caratteri. **Questa funzione non è implementata da Access**.
Vediamo un esempio:

```
SELECT NOME PRIMA, INITCAP(NOME) DOPO
FROM CARATTERI;
```

```
PRIMA      DOPO
--------------------------------
GIGI       Gigi
MARIO      Mario
MARIA      Maria
ANGELO     Angelo
ANNA       Anna
ANTONIO    Antonio
```

LOWER e UPPER

La funzione LOWER trasforma tutti i caratteri di una parola in maiuscolo; UPPER esegue l'operatore inversa. **Questa funzione non è implementata da Access**. Vediamo degli esempi:

```
UPDATE CARATTERI
SET NOME = 'Mario'
WHERE NOME = 'MARIO';
```

```
Aggiornata 1 riga.
```

Con questa espressione abbiamo modificato il formato con cui viene rappresentato uno dei nomi della tabella CARATTERI , al fine di accertarci che la funzione esegua il suo compito correttamente.

```
SELECT NOME, UPPER(NOME), LOWER(NOME)
FROM CARATTERI;
```

```
NOME          UPPER(NOME)      LOWER(NOME)
------------------------------------------------------------------------
GIGI           GIGI             gigi
Mario          MARIO            mario
MARIA          MARIA            maria
ANGELO         ANGELO           angelo
ANNA           ANNA             anna
ANTONIO        ANTONIO          antonio
```

LPAD e RPAD

La '**L**' e la '**R**' stanno per *left* e *right* mentre '**PAD**' significa in inglese cuscinetto.
Queste due funzioni richiedono da due a tre argomenti. Il primo argomento rappresenta le stringhe sulle quali operare. Il secondo argomento è il numero di caratteri da aggiungere alla stringa. Il terzo argomento (facoltativo) è il carattere da aggiungere, che può essere un singolo carattere o una stringa di caratteri; se non viene specificato, sarà automaticamente aggiunto uno spazio. **Questa funzione non è implementata da Access**. Vediamo alcuni esempi:

```
SELECT COGNOME, LPAD(COGNOME,20,'*')
FROM CARATTERI;
```

```
COGNOME      LPAD(COGNOME,20,'*')
------------------------------------------------------------
ROSSI        **************ROSSI
BIANCHI      ************BIANCHI
NERI         ***************NERI
BRUNI        **************BRUNI
SCURI        **************SCURI
VERDI        **************VERDI
```

```
SELECT COGNOME, LPAD(COGNOME,20,'tra')
FROM CARATTERI;
```

```
COGNOME    LPAD(COGNOME,20,'TRA
-------------------------------------------------------------
ROSSI          tratratratratraROSSI
BIANCHI          tratratratratBIANCHI
NERI           tratratratratratNERI
BRUNI          tratratratratraBRUNI
SCURI          tratratratratraSCURI
VERDI          tratratratratraVERDI
```

```
SELECT COGNOME, RPAD(COGNOME,20,'/')
FROM CARATTERI;
```

```
COGNOME    RPAD(COGNOME,20,'/')
-------------------------------------------------
ROSSI          ROSSI///////////////
BIANCHI          BIANCHI//////////////
NERI           NERI////////////////
BRUNI          BRUNI////////////////
SCURI          SCURI////////////////
VERDI          VERDI////////////////
```

LTRIM e RTRIM

La '**L**' e la '**R**' stanno per *left* e *right* mentre *To trim* in inglese significa anche tagliare.
Il primo argomento, come per le funzioni RPAD e LPAD, rappresenta le stringhe sulle quali operare. Il secondo argomento può essere un carattere o una stringa di caratteri. **Queste due funzioni non sono implementate da Access**.
Vediamo alcuni esempi:

```
SELECT NOME, RTRIM(NOME, 'O')
FROM CARATTERI;
```

```
NOME           RTRIM(NOME, 'O')
---------- -----------------------------------
GIGI           GIGI
MARIO          MARI
MARIA          MARIA
ANGELO         ANGEL
ANNA           ANNA
ANTONIO        ANTONI
```

```
SELECT NOME, RTRIM(NOME, 'N')
FROM CARATTERI;
```

```
NOME      RTRIM(NOME, 'N')
----------  ------------------------------
GIGI            GIGI
MARIO           MARIO
MARIA           MARIA
ANGELO          ANGELO
ANNA            ANNA
ANTONIO         ANTONIO
```

```
SELECT NOME, LTRIM(NOME, 'A')
FROM CARATTERI;
```

```
NOME       LTRIM(NOME, 'A')
---------- ------------------------------
GIGI           GIGI
MARIO          MARIO
MARIA          MARIA
ANGELO         NGELO
ANNA           NNA
ANTONIO        NTONIO
```

REPLACE

REPACE permette di sostituire una stringa Di caratteri con quella specifica. Richiede tre argomenti: il primo rappresenta le stringhe sulle quali effettuate le ricerche ; il secondo è la stringa da ricercare e sostituire; il terzo è facoltativo e specifica la stringa di sostituzione.
Se l'ultimo argomento non viene indicato, ogni ricorrenza della chiave di ricerca (stringa trovata) viene eliminata, senza essere sostituita da un'altra stringa. **Questa funzione non è implementata da Access**. Vediamo alcuni esempi:

```
SELECT NOME, REPLACE(NOME, 'R', '**')
FROM CARATTERI;
```

```
NOME      REPLACE(NOME,'R','**')
---------- ------------------------------------
GIGI            GIGI
MARIO           MA**IO
MARIA           MA**IA
ANGELO          ANGELO
ANNA            ANNA
ANTONIO         ANTONIO
```

```
SELECT NOME, REPLACE(NOME, 'A')
FROM CARATTERI;
```

```
NOME      REPLACE(NOME, 'A')
--------- ----------------------------------
GIGI          GIGI
MARIO         MRIO
MARIA         MRI
ANGELO        NGELO
ANNA          NN
ANTONIO       NTONIO
```

SUBSTR

Questa funzione consente si estrarre una serie di caratteri (sottostringa) da una stringa specificata. SUBSTR richiede tre argomenti: il primo è la stringa specificata da esaminare; il secondo è la posizione del primo carattere da estrarre; il terzo è il numero di caratteri da estrarre. Se il terzo parametro viene omesso la query visualizza tutti i caratteri rimanenti dopo la posizione specificata. **Questa funzione non è implementata da Access**. Vediamo alcuni esempi:

```
SELECT COGNOME, SUBSTR(COGNOME, 2, 3)
FROM CARATTERI;
```

```
COGNOME    SUBSTR(COGNOME, 2, 3)
---------- --------------------------------
ROSSI          OSS
BIANCHI        IAN
NERI           ERI
BRUNI          RUN
SCURI          CUR
VERDI          ERD
```

```
SELECT COGNOME, SUBSTR(COGNOME,2)
FROM CARATTERI;
```

```
COGNOME    SUBSTR(COCOGNOME, 2)
---------- --------------------------------
ROSSI          OSSI
BIANCHI        IANCHI
NERI           ERI
BRUNI          RUNI
SCURI          CURI
VERDI          ERDI
```

Se viene utilizzato un numero negativo come secondo argomento, la posizione iniziale viene determinata contando la stringa da destra verso sinistra iniziando dal suo ultimo carattere. Vediamo alcuni esempi:

```
SELECT COGNOME, SUBSTR(COGNOME,-4, 2)
FROM CARATTERI;
```

```
COGNOME    SUBSTR(COGNOME, -4, 2)
---------- ------------------------------------
ROSSI       OS
BIANCHI     NC
NERI        NE
BRUNI       RU
SCURI       CU
VERDI       ER
```

```
SELECT COGNOME, SUBSTR(COGNOME,-5, 2)
FROM CARATTERI;
```

```
COGNOME    SU
---------- ----------
ROSSI       RO
BIANCHI     AN
NERI
BRUNI       BR
SCURI       SC
VERDI       VE
```

Vediamo un esempio in cui vogliamo visualizzare solo le iniziali dei nominativi presenti in CARATTERI:

```
SELECT SUBSTR(NOME, 1, 1) ||' - '|| SUBSTR(COGNOME, 1, 1)
FROM CARATTERI;
```

```
SUBST
---------
G - R
M - B
M - N
A - B
A - S
A - V
```

INSTR

La funzione INSTR permette di sapere in quale punto di una stringa si trova un particolare schema di caratteri. Il primo argomento della funzione è la stringa da esaminare. Il secondo argomento è lo schema da ricercare. Il terzo e il quarto argomento sono numeri che indicano dove iniziare le ricerche e quale tipo di corrispondenza fornire. Vediamo alcuni esempi:

```
SELECT COGNOME, INSTR(COGNOME, 'I', 2, 1)
FROM CARATTERI;
```

```
COGNOME    INSTR(COGNOME, 'I', 2, 1)
-----------------------------------------------------
ROSSI                    5
BIANCHI                  2
NERI                     4
BRUNI                    5
SCURI                    5
VERDI                    5
```

```
SELECT COGNOME, INSTR(COGNOME, 'I', 2, 2)
FROM CARATTERI;
```

```
COGNOME    INSTR(COGNOME,'I',2,2)
-----------------------------------------------------
ROSSI                    0
BIANCHI                  7
NERI                     0
BRUNI                    0
SCURI                    0
VERDI                    0
```

```
SELECT COGNOME, INSTR(COGNOME, 'R', 2, 2)
FROM CARATTERI;
```

```
COGNOME    INSTR(COGNOME, 'R', 2, 2)
-----------------------------------------------
ROSSI                    0
BIANCHI                  0
NERI                     0
BRUNI                    0
SCURI                    0
VERDI                    0
```

```
SELECT COGNOME, INSTR(COGNOME, 'N', 2, 1)
FROM CARATTERI;
```

```
COGNOME    INSTR(COGNOME, 'N', 2, 1)
-----------------------------------------------------
ROSSI                    0
BIANCHI                  4
NERI                     0
BRUNI                    4
SCURI                    0
VERDI                    0
```

```
SELECT NOME, INSTR(NOME, 'N', 2, 2)
FROM CARATTERI;
```

```
NOME      INSTR(NOME, 'N', 2, 2)
-----------------------------------------
GIGI                     0
MARIO                    0
MARIA                    0
ANGELO                   0
ANNA                     3
ANTONIO                  5
```

Il valore di default per il terzo e il quarto argomento è 1.

Access implementa questa funzione usando un'altra sintassi, vediamo come:

InStr([inizio,]stringa1, stringa2[, confronto])

La sintassi della funzione InStr è composta dai seguent argomenti:

inizio Facoltativa. Espressione numerica che definisce la posizione di inizio per ciascuna ricerca. Se omessa, la ricerca inizia dalla posizione del primo carattere.
stringa1 Obbligatoria. Espressione stringa oggetto della ricerca.
stringa2 Obbligatoria. Espressione stringa cercata.
confronto Facoltativa. Specifica il tipo di confronto di stringa. Per ulteriori informazioni consultare la guida in linea.

LENGTH

La sintassi accettata da Access per tale funzione è: ***Len*(*nome_campo*)**

La funzione LENGTH restituisce la lunghezza del suo argomento, come in questo esempio:

```
SELECT NOME, LENGTH(NOME)
FROM CARATTERI;
```

NOME	LENGTH(NOME)
GIGI	4
MARIO	5
MARIA	5
ANGELO	6
ANNA	4
ANTONIO	7

Funzione USER

Questa funzione restituisce il nome dell'utente corrente alla tabella specificata. **Questa funzione non è implementata da Access**. Vediamo un esempio:

```
SELECT USER
FROM CARATTERI;
```

USER
DEMO
DEMO
DEMO
DEMO
DEMO
DEMO

ESERCIZI (capitolo 3)

1) Le funzioni di gruppo sono anche chiamate in un altro modo, quale?

2) La seguente query è giusta? e se sbagliata, perché?

```
SELECT  SUM(NOME)
FROM  PERSONE;
```

3) Esiste una funzione che trasforma in maiuscolo il primo carattere di una stringa e in minuscolo tutti gli altri, quale è questa funzione?

4) La seguente query è sbagliata? e se giusta, perché?

```
SELECT COUNT(NOME)
FROM PERSONE;
```

5) Applicando la seguente *query* alla tabella sottostante

```
SELECT COUNT(NOME)
FROM PERSONE;
```

PERSONE

ID_PERSONA	NOME
1	ANTONIO
12	SILVIA
2	GIULIO
3	
6	ROBERTA
7	ROBERTA
9	MARIA

otteniamo uno di questi valori, quale?

a. 7
b. 8
c. 6
d. 5

6) Applicando la seguente *query* alla tabella sottostante

```
SELECT COUNT(*)
FROM PERSONE;
```

PERSONE

ID_PERSONA	NOME
1	ANTONIO
12	SILVIA
2	GIULIO
3	
6	ROBERTA
7	ROBERTA
9	MARIA

otteniamo uno di questi valori, quale?

a. 7
b. 8
c. 6
d. 5

7) Possiamo unire in un'unica colonna due colonne distinte come possono essere ad esempio COGNOME e NOME presenti nella medesima tabella? e se si quali sono gli operatori o le funzioni in grado di farlo?

8) La seguente query è giusta? e se sbagliata, perché?

```
SELECT SUBSTR NOME,1,5
FROM nome-tabela;
```

9) La seguente query è sbagliata? e se giusta, perché?

```
SELECT DISTINCT COUNT(NOME)
FROM PRESONE;
```

10) Applicando la seguente *query* alla tabella sottostante

```
SELECT COUNT(DISTINCT NOME)
FROM PERSONE;
```

PERSONE

ID_PERSONA	NOME
1	ANTONIO
12	SILVIA
2	GIULIO
3	
6	ROBERTA
7	ROBERTA
9	MARIA

otteniamo uno di questi valori, quale?

a. 7
b. 8
c. 6
d. 5

11) *NOMINATIVI*

COGNOME	NOME	S	CO
ROSSI	GIGI	A	32
BIANCHI	MARIO	J	67
NERI	MARIA	C	65
BRUNI	ANGELO	M	87
SCURI	ANNA	A	77
VERDI	ANTONIO	G	52

Da questa tabella scrivere una query per ottenere il seguente risultato:

INIZIALI	CODICE
G.R.	32

Questo capitolo è dedicato alle clausole utilizzate con l'istruzione SELECT , in particolare saranno trattate le seguenti clausole :

- WHERE
- ORDER BY
- GROUP BY
- HAVING

Negli esempi di questo capitolo, quando non verrà specificato diversamente, si utilizzerà la seguente tabella:

ASSEGNI

ASSEGNO	BENEFICIARIO	IMPORTO	NOTE
1	COMPUTER SHOP	50 000	DISCHETTI E CD-ROM
2	LIBRERIE CULTURA	245 000	LIBRI, CANCELLERIA
3	COMPUTER SHOP	200 000	TELEFONO CELLULARE
4	BIOGAS SRL	88 000	GAS
5	SUPERMARCHET GS	150 000	ALIMENTARI
16	ASSICURAZIONI ASSO	425 000	ASSICURAZIONE CASA
17	GAS S.P.A.	25 000	GAS
21	COMPUTER SHOP	34 000	CONTROLLER
20	ABITI BELLA	110 000	PANTALONI
9	ABITI BELLA	224 000	COMPLETO DONNA
8	COMPUTER SHOP	134 000	JOYSTICK

WHERE

La clausola WHERE serve per implementare delle condizioni verificabili a livello delle singole righe.
Questa clausola è abbastanza semplice da usare ed è già stata utilizzata precedentemente in questo corso, vediamo un esempio:

```
SELECT * FROM ASSEGNI
```

```
WHERE IMPORTO < 150000;
```

ASSEGNO	BENEFICIARIO	IMPORTO	NOTE
1	COMPUTER SHOP	50 000	DISCHETTI E CD-ROM
4	BIOGAS SRL	88 000	GAS
17	GAS S.P.A.	25 000	GAS
21	COMPUTER SHOP	34 000	CONTROLLER
20	ABITI BELLA	110 000	PANTALONI
8	COMPUTER SHOP	134 000	JOYSTICK

Come possiamo vedere dall'esempio la condizione 'IMPORTO < 150000', implementata tramite la clausola WHERE, è stata posta a tutte le righe della tabella e solo per quelle righe dove tale condizione è risultata soddisfatta che sono stati estrapolati e visualizzati i dati secondo gli argomenti dell'istruzione SELECT.

ORDER BY

A volte potrebbe essere necessario presentare i risultati di una query in un certo ordine, la clausola ORDER BY assolve a questo scopo. Vediamo alcuni esempi:

```
SELECT *
FROM ASSEGNI
ORDER BY BENEFICIARIO;
```

ASSEGNO	BENEFICIARIO	IMPORTO	NOTE
20	ABITI BELLA	110 000	PANTALONI
9	ABITI BELLA	224 000	COMPLETO DONNA
16	ASSICURAZIONI ASSO	425 000	ASSICURAZIONE CASA
4	BIOGAS SRL	88 000	GAS
1	COMPUTER SHOP	50 000	DISCHETTI E CD-ROM
3	COMPUTER SHOP	200 000	TELEFONO CELLULARE
8	COMPUTER SHOP	134 000	JOYSTICK
21	COMPUTER SHOP	34 000	CONTROLLER
17	GAS S.P.A.	25 000	GAS
2	LIBRERIE CULTURA	245 000	LIBRI, CANCELLERIA
5	SUPERMARCHET GS	150 000	ALIMENTARI

E’ possibile ordinare i record in senso inverso, con la lettera o il numero più alti in prima posizione? Si che è possibile, tramite la parola chiave DESC. Vediamo un esempio:

```
SELECT *
FROM ASSEGNI
ORDER BY BENEFICIARIO DESC;
```

ASSEGNO	BENEFICIARIO	IMPORTO	NOTE
5	SUPERMARCHET GS	150 000	ALIMENTARI
2	LIBRERIE CULTURA	245 000	LIBRI, CANCELLERIA
17	GAS S.P.A.	25 000	GAS
1	COMPUTER SHOP	50 000	DISCHETTI E CD-ROM
21	COMPUTER SHOP	34 000	CONTROLLER
3	COMPUTER SHOP	200 000	TELEFONO CELLULARE
8	COMPUTER SHOP	134 000	JOYSTICK
4	BIOGAS SRL	88 000	GAS
16	ASSICURAZIONI ASSO	425 000	ASSICURAZIONE CASA
20	ABITI BELLA	110 000	PANTALONI
9	ABITI BELLA	224 000	COMPLETO DONNA

Esiste anche la parola chiave facoltativa ASC per l’ordinamento ascendente. Comunque questa parola chiave è raramente utilizzata in quanto superflua. Infatti ORDER BY, se non viene specificato diversamente, ordina per l’appunto in modo ascendente.

La clausola ORDER BY può essere applicata a più campi. Vediamo alcuni esempi:

```
SELECT  BENEFICIARIO, NOTE
FROM ASSEGNI
ORDER BY BENEFICIARIO, NOTE;
```

BENEFICIARIO	NOTE
ABITI BELLA	COMPLETO DONNA
ABITI BELLA	PANTALONI
ASSICURAZIONI ASSO	ASSICURAZIONE CASA
BIOGAS SRL	GAS
COMPUTER SHOP	CONTROLLER
COMPUTER SHOP	DISCHETTI E CD-ROM
COMPUTER SHOP	JOYSTICK
COMPUTER SHOP	TELEFONO CELLULARE
GAS S.P.A.	GAS
LIBRERIE CULTURA	LIBRI, CANCELLERIA
SUPERMARCHET GS	ALIMENTARI

```
SELECT  BENEFICIARIO, NOTE
FROM ASSEGNI
ORDER BY BENEFICIARIO, NOTE DESC;
```

BENEFICIARIO	NOTE
ABITI BELLA	PANTALONI
ABITI BELLA	COMPLETO DONNA
ASSICURAZIONI ASSO	ASSICURAZIONE CASA
BIOGAS SRL	GAS
COMPUTER SHOP	TELEFONO CELLULARE
COMPUTER SHOP	JOYSTICK
COMPUTER SHOP	DISCHETTI E CD-ROM
COMPUTER SHOP	CONTROLLER
GAS S.P.A.	GAS
LIBRERIE CULTURA	LIBRI, CANCELLERIA
SUPERMARCHET GS	ALIMENTARI

Possiamo far riferimento ai campi da ordinare dopo ORDER BY indicando invece del loro nome il valore dell'ordine di apparizione all'interno della tabella. Vediamo un esempio:

```
SELECT *
FROM ASSEGNI
ORDER BY 3;
```

ASSEGNO	BENEFICIARIO	IMPORTO	NOTE
17	GAS S.P.A.	25 000	GAS
21	COMPUTER SHOP	34 000	CONTROLLER
1	COMPUTER SHOP	50 000	DISCHETTI E CD-ROM
4	BIOGAS SRL	88 000	GAS
20	ABITI BELLA	110 000	PANTALONI
8	COMPUTER SHOP	134 000	JOYSTICK
5	SUPERMARCHET GS	150 000	ALIMENTARI
3	COMPUTER SHOP	200 000	TELEFONO CELLULARE
9	ABITI BELLA	224 000	COMPLETO DONNA
2	LIBRERIE CULTURA	245 000	LIBRI, CANCELLERIA
16	ASSICURAZIONI ASSO	425 000	ASSICURAZIONE CASA

I dati sono stati visualizzati ordinandoli per il campo IMPORTO che è appunto il terzo campo che appare nella tabella ASSEGNI.

GROUP BY

Questa clausola ci permette di formare dei sottoinsiemi per quelle colonne specificate. Vediamo cosa significa quanto affermato.

```
SELECT BENEFICIARIO
FROM ASSEGNI
GROUP BY BENEFICIARIO;
```

```
BENEFICIARIO
------------------------
ABITI BELLA
ASSICURAZIONI ASSO
BIOGAS SRL
COMPUTER SHOP
GAS S.P.A.
LIBRERIE CULTURA
SUPERMARCHET GS
```

Il risultato della query è una lista di beneficiari, che appaiono però una sola volta, anche se nella tabella di origine la maggior parte di essi compare più volte.

Questa clausola è usata molto spesso per applicare le funzioni di gruppo non a tutte le righe indistintamente, ma a sottoinsiemi di esse. Vediamo un esempio:

Vogliamo sapere quanto è stato elargito, in totale, per ogni beneficiario:

```
SELECT BENEFICIARIO, SUM(IMPORTO)
FROM ASSEGNI
GROUP BY BENEFICIARIO;
```

```
BENEFICIARIO                SUM(IMPORTO)
-------------------------------------------------------------
ABITI BELLA                  334 000
ASSICURAZIONI ASSO           425 000
BIOGAS SRL                    88 000
COMPUTER SHOP                418 000
GAS S.P.A.                    25 000
LIBRERIE CULTURA             245 000
SUPERMARCHET GS              150 000
```

In questa query viene applicata la funzione di gruppo SUM per ogni sottoinsieme di BENEFICIARIO.

Da questo punto in poi dovremo considerare che alla tabella ASSEGNI sono stati aggiunti i seguenti record:

22	ABITI BELLA	79 000	PANTALONI
23	BIOGAS SRL	399 000	GAS
24	LIBRERIE CULTURA	224 000	LIBRI, CANCELLERIA
25	COMPUTER SHOP	88 000	CONTROLLER

Quindi la tabella così modificata apparirà nel seguente modo:

ASSEGNI

ASSEGNO	BENEFICIARIO	IMPORTO	NOTE
1	COMPUTER SHOP	50 000	DISCHETTI E CD-ROM
2	LIBRERIE CULTURA	245 000	LIBRI, CANCELLERIA
3	COMPUTER SHOP	200 000	TELEFONO CELLULARE
4	BIOGAS SRL	88 000	GAS
5	SUPERMARCHET GS	150 000	ALIMENTARI
16	ASSICURAZIONI ASSO	425 000	ASSICURAZIONE CASA
17	GAS S.P.A.	25 000	GAS
21	COMPUTER SHOP	34 000	CONTROLLER
20	ABITI BELLA	110 000	PANTALONI
9	ABITI BELLA	224 000	COMPLETO DONNA
8	COMPUTER SHOP	134 000	JOYSTICK
22	ABITI BELLA	79 000	PANTALONI
23	BIOGAS SRL	399 000	GAS
24	LIBRERIE CULTURA	224 000	LIBRI, CANCELLERIA
25	COMPUTER SHOP	88 000	CONTROLLER

È possibile applicare la clausola GROUP BY anche a più di un campo per volta. Vediamo come funziona:

```
SELECT BENEFICIARIO, NOTE
FROM ASSEGNI
GROUP BY BENEFICIARIO, NOTE;
```

BENEFICIARIO	NOTE
ABITI BELLA	COMPLETO DONNA
ABITI BELLA	PANTALONI
ASSICURAZIONI ASSO	ASSICURAZIONE CASA
BIOGAS SRL	GAS
COMPUTER SHOP	CONTROLLER
COMPUTER SHOP	DISCHETTI E CD-ROM
COMPUTER SHOP	JOYSTICK
COMPUTER SHOP	TELEFONO CELLULARE
GAS S.P.A.	GAS
LIBRERIE CULTURA	LIBRI, CANCELLERIA
SUPERMARCHET GS	ALIMENTARI

In questa query le righe selezionate sono 11 contro le 15 della tabella originale, cosa è successo?
È avvenuto che la dove il beneficiario presentava le stesse note, veniva visualizzato una volta sola.
Si veda quante volte appare, nella tabella ASSEGNI, in NOTE 'libri cancelleria', 'gas'(per beneficiario Biogas SRL), 'controller' e 'pantaloni'.

Vediamo altri esempi:

Vogliamo sapere oltre a quanto è stato elargito per ogni beneficiario, quante volte il singolo beneficiario compare nella tabella:

```
SELECT BENEFICIARIO, SUM(IMPORTO), COUNT(BENEFICIARIO)
FROM ASSEGNI
GROUP BY BENEFICIARIO;
```

BENEFICIARIO	SUM(IMPORTO)	COUNT(BENEFICIARIO)
ABITI BELLA	413 000	3
ASSICURAZIONI ASSO	425 000	1
BIOGAS SRL	487 000	2
COMPUTER SHOP	506 000	5
GAS S.P.A.	25 000	1
LIBRERIE CULTURA	469 000	2
SUPERMARCHET GS	150 000	1

Abiti Bella compare nella tabella tre volte, Assicurazioni ASSO una volta, ecc.

Voglio sapere il totale dell'*importo* per ogni *nota* che facendo parte dello stesso *beneficiario* compaia una o più volte. Voglio sapere quante volte quella stessa nota appare per lo stesso beneficiario. Voglio inoltre visualizzare le note:

```
SELECT BENEFICIARIO, NOTE, SUM(IMPORTO), COUNT(BENEFICIARIO)
FROM ASSEGNI
GROUP BY BENEFICIARIO, NOTE;
```

BENEFICIARIO	NOTE	SUM(IMPORTO)	COUNT(BENEFICIARIO)
ABITI BELLA	COMPLETO DONNA	224 000	1
ABITI BELLA	PANTALONI	189 000	2
ASSICURAZIONI ASSO	ASSICURAZIONE CASA	425 000	1
BIOGAS SRL	GAS	487 000	2
COMPUTER SHOP	CONTROLLER	122 000	2
COMPUTER SHOP	DISCHETTI E CD-ROM	50 000	1
COMPUTER SHOP	JOYSTICK	134 000	1
COMPUTER SHOP	TELEFONO CELLULARE	200 000	1
GAS S.P.A.	GAS	25 000	1
LIBRERIE CULTURA	LIBRI, CANCELLERIA	469 000	2
SUPERMARCHET GS	ALIMENTARI	150 000	1

Nella prima riga della tabella estrapolata con la query precedente, vediamo che COUNT(BENEFICIARIO) vale 1; ciò significa che la nota 'Completo donna' per quel beneficiario è presente nella tabella di origine una sola volta. Mentre invece, la nota 'Pantaloni', sempre per il medesimo beneficiario vale 2, questo significa che quella nota per quel beneficiario è presente nella tabella ben due volte.

Adesso vogliamo estrapolare i stessi dati della query precedente, ma ordinandoli per le *note*.

```
SELECT BENEFICIARIO, NOTE, SUM(IMPORTO), COUNT(BENEFICIARIO)
FROM ASSEGNI
GROUP BY BENEFICIARIO, NOTE
ORDER BY NOTE;
```

BENEFICIARIO	NOTE	SUM(IMPORTO)	COUNT(BENEFICIARIO)
SUPERMARCHET GS	ALIMENTARI	150 000	1
ASSICURAZIONI ASSO	ASSICURAZIONE CASA	425 000	1
ABITI BELLA	COMPLETO DONNA	224 000	1
COMPUTER SHOP	CONTROLLER	122 000	2
COMPUTER SHOP	DISCHETTI E CD-ROM	50 000	1
BIOGAS SRL	GAS	487 000	2
GAS S.P.A.	GAS	25 000	1
COMPUTER SHOP	JOYSTICK	134 000	1
LIBRERIE CULTURA	LIBRI, CANCELLERIA	469 000	2
ABITI BELLA	PANTALONI	189 000	2
COMPUTER SHOP	TELEFONO CELLULARE	200 000	1

Per gli esempi futuri faremo riferimento alla tabella DIPENDENTI sottostante:

DIPENDENTI

NOME	DIVISIONE	STIPENDIO	GIORNI_MUTUA	FERIE_GODUTE
ROSSI	VENDITE	2 000 000	33	5
BIANCHI	VENDITE	2 100 000	1	0
BRUNI	RICERCA	3 300 000	0	9
VERDI	ACQUISTI	1 800 000	32	20
GIALLI	RICERCA	4 800 000	0	0
NERI	RICERCA	3 400 000	2	1
MANCINI	AMMINISTRAZIONE	2 400 000	9	24
MARCHETTI	VENDITE	2 000 000	99	12

HAVING

Abbiamo visto come tramite la clausola GROUP BY le righe possano venire raggruppate in sottoinsiemi. Una particolare interrogazione può avere la necessità di estrapolare solo quei sottoinsiemi di righe che soddisfano certe condizioni, in questo caso però non è possibile usare la clausola WHERE in quanto tale clausola verifica la condizione che la segue, su tutte le righe e non in maniera singola sui valori estrapolati per ogni sottoinsieme di righe.
Vediamo un esempio:

Vogliamo conoscere le medie dei stipendi per ogni divisione che superano i 2.200.000 di lire

```
SELECT DIVISIONE, AVG(STIPENDIO)
FROM DIPENDENTI
GROUP BY DIVISIONE
WHERE AVG(STIPENDIO)  > 2200000;
```

```
ERRORE alla riga 4:
ORA-00933: comando SQL terminato erroneamente
```

Nella query scritta sopra possiamo vedere come la clausola WHERE sia stata posta per ultima, infatti bisogna verificare la condizione solo dopo che sono stati formati i sottoinsiemi dalla clausola GROUP BY.
Il risultato però, è comunque un avviso di errore, proprio perché non è possibile utilizzare WHERE per verificare condizioni sui risultati di funzioni di gruppo. Vediamo quindi come possiamo risolvere il problema:

```
SELECT DIVISIONE, AVG(STIPENDIO)
FROM DIPENDENTI
HAVING AVG(STIPENDIO) > 2200000;
```

```
DIVISIONE                 AVG(STIPENDIO)
-------------------------------------------------------
AMMINISTRATORE       2 400 000
RICERCA                   3 833 333,3
```

Come abbiamo potuto vedere dall'esempio la clausola HAVING sostituisce la clausola WHERE la dove nella condizione appaiono funzioni di gruppo o quando la condizioni deve essere verificata su sottoinsiemi di righe.

Vediamo altri esempi:

```
SELECT DIVISIONE, AVG(STIPENDIO)
FROM DIPENDENTI
GROUP BY DIVISIONE
HAVING DIVISIONE = 'VENDITE';
```

```
DIVISIONE      AVG(STIPENDIO)
-------------------------------------------
VENDITE           2033333,3
```

In questo caso è stata estrapolata la media degli stipendi della sola divisione vendite. Proviamo a ottenere lo stesso risultato usando a posto della clausola HAVING la clausola WHERE:

```
SELECT DIVISIONE, AVG(STIPENDIO)
FROM DIPENDENTI
GROUP BY DIVISIONE
WHERE DIVISIONE = 'VENDITE';
```

```
ERRORE alla riga 4:
ORA-00933: comando SQL terminato erroneamente
```

Il risultato che si ottiene è un avviso di errore, in quanto la clausola WHERE è stata messa dopo la clausola GROUP BY e quindi si è tentato di usarla non per singole righe, ma per sottoinsiemi di righe. Vediamo come si può aggirare l'ostacolo senza, comunque, usare la clausola GROUP BY:

```
SELECT DIVISIONE, AVG(STIPENDIO)
FROM DIPENDENTI
WHERE DIVISIONE = 'VENDITE'
GROUP BY DIVISIONE;
```

```
DIVISIONE      AVG(STIPENDIO)
-------------------------------------------
VENDITE           2033333,3
```

In questo caso la clausola WHERE è stata usata non sugli insiemi delle righe, in quanto posta prima della clausola GROUP BY.

Voglio conoscere la media dei giorni di ferie godute per dipartimento, ma solo di quei dipendenti che percepiscono stipendi superiori ai 2.050.000 di lire:

```
SELECT DIVISIONE, AVG(FERIE_GODUTE)
FROM DIPENDENTI
WHERE STIPENDIO > 2050000
GROUP BY DIVISIONE;
```

```
DIVISIONE                AVG (FERIE_GODUTE)
-------------------------------------------------------------
AMMINISTRATORE           24
RICERCA                   3,3333333
VENDITE                   0
```

In questo caso abbiamo usato la clausola WHERE in quanto la condizione va verificata per tutte le righe e non per singoli valori estrapolati da singoli sottoinsiemi di righe.

Adesso vogliamo escludere dal risultato della query precedente quelle divisioni la cui media delle ferie godute, calcolata solo per quei dipendenti il cui stipendio supera i 2.050.000 di lire, è uguale a zero:

```
SELECT DIVISIONE, AVG(FERIE_GODUTE)
FROM DIPENDENTI
WHERE STIPENDIO > 2050000
GROUP BY DIVISIONE
HAVING AVG(FERIE_GODUTE)  <> 0;
```

```
DIVISIONE            AVG(FERIE_GODUTE)
-------------------------------------------------------------
AMMINISTRATORE       24
RICERCA               3,3333333
```

Riepilogo

Dagli esempi precedenti, dalle cose che sono state dette e da eventuali esperimenti che potremmo fare, possiamo enunciare degli assiomi che riguardano l'uso delle clausole viste in questo capitolo:

1. WHERE non può essere usato per verificare condizioni su risultati di funzioni di gruppo.
2. WHERE non può essere usato per verificare condizioni su sottoinsiemi delle varie righe.

1. GROUP BY tutte le colonne che vengono selezionate (colonne che seguono la clausola SELECT) devono essere elencate nella clausola GROUP BY.

1. HAVING può essere seguita da una o più funzioni di gruppo e verificare condizioni su i valori ritornati. da tali funzioni.
2. HAVING può verificare condizioni sui valori dei sottoinsiemi creati dalla clausola GROUP BY.
3. HAVING può verificare condizioni combinate sui valori dei sottoinsiemi creati dalla clausola GROUP BY e condizioni sui valori ritornati da funzioni di gruppo.
4. HAVING i campi che vi appaiono devono essere specificati nella clausola GROUP BY
5. HAVING è necessario, per il suo utilizzo, la presenza della clausola GROUP BY.
6. HAVING non è necessario che venga posta dopo la clausola GROUP BY.

1. ORDER BY è necessario che venga posta dopo la clausola GROUP BY e dopo la clausola HAVING.

Riassumendo vediamo come può essere la forma sintetica di una query che fa uso delle clausole viste in questo capitolo:

select *lista attributi o espressioni*
from *lista tabelle*
[**where** *condizioni semplici*]
[**group by** *lista attributi di raggruppamento*]
[**having** *condizioni aggregate*]
[**order by** *lista attributi di ordinamento*]

ESERCIZI (capitolo 4)

1) È corretta questa query? se giusta o sbagliata, spiegare il perché.

```
SELECT COGNOME, AVG(STIPENDIO), REPARTO
FROM DIPENDENTI
WHERE REPARTO = 'VENDITE'
ORDER BY COGNOME
GROUP BY REPARTO;
```

2) Si può applicare la clausola ORDER BY a una colonna che non appare fra quelle citate nell'istruzione SELECT ?

3) Quando usiamo la clausola HAVING, dobbiamo necessariamente utilizzare anche la clausola GROUP BY ?

4) Si può applicare l'istruzione SELECT a una colonna che non appare fra quelle citate nella clausola GROUP BY ?

5) Scrivere una query che ci permetta di estrapolare da una tabella contenente dati su libri, quei *generi* in cui non sono presenti libri con prezzo inferiore o uguale a £10.000.
Quanto detto, significa in parole più semplici, che dobbiamo visualizzare quei *generi* dove non ci sono libri il cui costo sia uguale o inferiore a £ 10.000. Se ad esempio il libro Le Crociate che appartiene al genere di Storia costa £ 9.500, non dovrà apparire nella *select* il genere *Storia.*

LIBRI (TITOLO, AUTORE, GENERE, PREZZO, EDITORE)

6) Se applicassimo la seguente *select*

```
SELECT DIVISIONE
FROM DIPENDENTI
GROUP BY DIVISIIONE
HAVING MAX(STIPENDIO) < 4 800 000 ;
```

alla tabella *DIPENDENTI,* indicare quali sarebbero le divisioni estrapolate.

DIPENDENTI

NOME	DIVISIONE	STIPENDIO	GIORNI_MUTUA	FERIE_GODUTE
ROSSI	VENDITE	2 000 000	33	5
BIANCHI	VENDITE	2 100 000	1	0
BRUNI	RICERCA	3 300 000	0	9
VERDI	ACQUISTI	1 800 000	32	20
GIALLI	RICERCA	4 800 000	0	0
NERI	RICERCA	3 400 000	2	1
MANCINI	AMMINISTRAZIONE	2 400 000	9	24
MARCHETTI	VENDITE	2 000 000	99	12

7) Scrivere una *select* che ci permetta di estrapolare, dalla tabella *DIPENDENTI,* le *divisioni* in cui non compaiono lavoratori che non hanno goduto di giorni di ferie. Ovvero, bisogna visualizzare solo quelle divisioni dove non c'è neanche un dipendente con zero giorni di ferie.

8) Scrivere una *query* che, dalla tabella *ASSEGNI,* ci permette di estrapolare i beneficiari che gli è stato versato perlomeno un assegno di cifra superiore alle 400 000.

ASSEGNI

ASSEGNO	BENEFICIARIO	IMPORTO	NOTE
1	COMPUTER SHOP	50 000	DISCHETTI E CD-ROM
2	LIBRERIE CULTURA	245 000	LIBRI, CANCELLERIA
3	COMPUTER SHOP	200 000	TELEFONO CELLULARE
4	BIOGAS SRL	88 000	GAS
5	SUPERMARCHET GS	150 000	ALIMENTARI
16	ASSICURAZIONI ASSO	425 000	ASSICURAZIONE CASA
17	GAS S.P.A.	25 000	GAS
21	COMPUTER SHOP	34 000	CONTROLLER
20	ABITI BELLA	110 000	PANTALONI
9	ABITI BELLA	224 000	COMPLETO DONNA
8	COMPUTER SHOP	134 000	JOYSTICK

9) Scrivere una *query* che, dalla tabella *ASSEGNI,* ci permette di estrapolare i beneficiari la cui media degli importi degli assegni versati sia superiore alle 300 000. La lista dei beneficiari deve apparire ordinata in modo discendente.

Combinazione di tabelle (capitolo 5)

Questo capitolo tratta un importante tipo di operazione tra le tabelle: il Join.
Il vocabolo join significa unione e nel caso di SQL sta ad indicare unione tra tabelle. Esistono vari tipi di join, ma tutti derivano o possono essere ricondotti a vari operatori dell'algebra insiemistica. L'importanza principale del join risiede nella possibilità che ci offre per correlare e visualizzare dati appartenenti a tabelle diverse o alla medesima tabella, logicamente correlati tra di loro. I semplici dati, da noi uniti, possono assumere la forma di complesse informazioni così come noi li vogliamo.

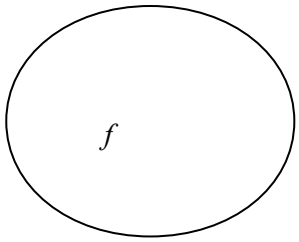
CROSS JOIN

Per comprendere a pieno l'operazione CROSS JOIN (unione incrociata) bisogna aver ben chiaro il concetto di prodotto cartesiano:

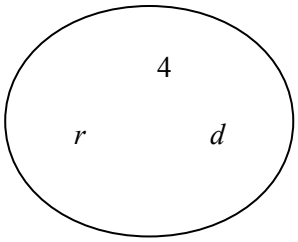
…… **Prodotto cartesiano**……

Dati due insiemi D1 e D2 si chiama prodotto cartesiano di D1 e D2, l'insieme delle coppie ordinate ($v1$, $v2$), tali che $v1$ è un elemento di D1 e $v2$ un elemento di D2.

Vediamo cosa significa quanto affermato con un esempio:



insieme A



insieme B

$$A \text{ x } B = \{(2, r), (2, s), (2, d), (2, 4), (f, r), (f, s), (f, d), (f, 4), (r, r), (r, s), (r, d), (r, 4)\}$$

Come possiamo vedere il prodotto cartesiano fra i due insiemi è dato da tutti gli elementi di A combinati con ogni elemento di B. Nella rappresentazione delle varie coppie dobbiamo rispettare l'ordine di apparizione degli elementi, in quanto l'appartenenza dell'elemento all'insieme è individuabile proprio dalla suo ordine di apparizione. Nell'esempio abbiamo usato solo due insiemi ma il prodotto cartesiano è applicabile anche a più di due insiemi.

……

Ora considerando che le tabelle non sono altro che insiemi i cui elementi sono le righe ecco che possiamo individuare l'operazione di CROSS JOIN in quella di prodotto cartesiano appartenente alle teorie degli insiemi. Dunque il prodotto cartesiano tra due o più tabelle si traduce in una istruzione chiamata CROSS JOIN. Il CROSS JOIN si ottiene in maniera molto semplice elencando dopo la FROM le tabelle che devono essere coinvolte. Vediamo un esempio di CROSS JOIN:

Per lo scopo usiamo due tabelle: TAB1 e TAB2

TAB1

```
COLONTAB1
-----------------
RIG1 TAB1
RIG2 TAB1
RIG3 TAB1
RIG4 TAB1
RIG5 TAB1
```

TAB2

```
COLONTAB2
-----------------
RIG1 TAB2
RIG2 TAB2
RIG3 TAB2
```

```
SELECT *
FROM TAB1, TAB2;
```

```
COLONTAB1       COLONTAB2
---------------- ----------------
RIG1 TAB1       RIG1 TAB2
RIG2 TAB1       RIG1 TAB2
RIG3 TAB1       RIG1 TAB2
RIG4 TAB1       RIG1 TAB2
RIG5 TAB1       RIG1 TAB2

RIG1 TAB1       RIG2 TAB2
RIG2 TAB1       RIG2 TAB2
RIG3 TAB1       RIG2 TAB2
RIG4 TAB1       RIG2 TAB2
RIG5 TAB1       RIG2 TAB2

RIG1 TAB1       RIG3 TAB2
RIG2 TAB1       RIG3 TAB2
RIG3 TAB1       RIG3 TAB2
RIG4 TAB1       RIG3 TAB2
RIG5 TAB1       RIG3 TAB2
```

Questo è il risultato che si ottiene dal CROSS JOIN delle tabelle TAB1 e TAB2, come si può vedere non è altro che un prodotto cartesiano. Chiaramente avremmo potuto usare anche più di due tabelle.

Il CROSS JOIN non è particolarmente utile e viene usato raramente, ma se in una CROSS JOIN si utilizza la clausola WHERE potremmo ottenere join molto più interessanti.

NATURAL JOIN

Il NATURAL JOIN è un tipo di operazione che ci permette di correlare due o più tabelle sulla base di valori uguali in attributi contenenti lo stesso tipo di dati.
Vediamo un esempio:

Per lo scopo usiamo due tabelle: PERSONE e AUTO. La tabella AUTO fa riferimento alla persona proprietaria dell'auto attraverso il campo PROPRIETARIO in cui sono riportati i numeri di patente. Lo stesso tipo di dato è presente nella tabella PERSONE nel campo PATENTE.

PERSONE

NOME	PATENTE
ANTONIO	123
GIOVANNI	156
ARTURO	172

AUTO

TARGA	PROPRIETARIO
VT AC73949	156
ROMA J1003	172
MI GH3434	300
NA G666223	301

Vogliamo ottenere un join delle righe delle due tabelle la dove i valori dei campi PROPRIETARIO e PATENTE sono uguali .

```
SELECT *
FROM PERSONE, AUTO
WHERE  PATENTE =  PROPRIETARIO;
```

NOME	PATENTE	TARGA	PROPPRIETARIO
GIOVANNI	156	VT AC73949	156
ARTURO	172	ROMA J1003	172

Nel caso le due tabelle originarie avessero avuto i campi interessati al join (PATENTE e PROPRIETARIO) con lo stesso nome in entrambe, avremmo dovuto specificare dopo la WHERE prima del nome del campo il nome della tabella a cui facevamo riferimento. Facciamo un esempio considerando le tabelle PERSONE e AUTO così modificate

PERSONE

NOME	NUM_PATENTE
ANTONIO	123
GIOVANNI	156
ARTURO	172

AUTO

TARGA	NUM_PATENTE
VT AC73949	156
ROMA J1003	172
MI GH3434	300
NA G666223	301

In questo caso siamo obbligati a specificare l'appartenenza dei campi alle tabelle:

```
SELECT *
FROM PERSONE, AUTO
WHERE  PERSONE.NUM_PATENTE  =  AUTO.NUM_PATENTE;
```

NOME	NUM_PATENTE	TARGA	NUM_PATENTE
GIOVANNI	156	VT AC73949	156
ARTURO	172	ROMA J1003	172

INNER JOIN

È un tipo di join in cui le righe delle tabelle vengono combinate solo se i campi collegati con join soddisfano una determinata condizione.
Vediamo un esempio:

Vogliamo ottenere un join delle righe delle due tabelle PERSONE e AUTO, la dove i valori dei campi PROPRIETARIO e PATENTE sono uguali e dove il valore del campo NOME è uguale ad 'ARTURO'.

```
SELECT *
FROM PERSONE, AUTO
WHERE  PATENTE =  PROPRIETARIO
AND NOME = 'ARTURO';
```

NOME	PATENTE	TARGA	PROPRIETARIO
ARTURO	172	ROMA J1003	172

Esistono anche, delle parole chiavi specifiche per eseguire l'operazione di INNER JOIN.

```
SELECT *
FROM PERSONE INNER JOIN AUTO
ON
(PERSONE.PATENTE = AUTO.PROPRIETARIO AND NOME = 'ARTURO');
```

NOME	PATENTE	TARGA	PROPRIETARI
ARTURO	172	ROMA J1003	172

Il risultato che otteniamo è lo stesso, **ma la sintassi usata non è accettata da SQL Plus 8.0 Oracle** (prodotto usato per testare la maggior parte degli esempi di questo corso), infatti questa query e quelle successive in cui appaiono parole chiave specifiche, sono state testate utilizzato Microsoft Access.
Usando le parole chiave specifiche dobbiamo indicare, per alcuni DBMS come nel caso di Access, a quale tabella appartengono i campi. C'è inoltre da far notare che quello che segue la clausola ON va messo tra parentesi se è presente più di una condizione.

Vediamo altri esempi:
Usando le parole chiave specifiche, vogliamo ottenere lo stesso JOIN che abbiamo usato come esempio nel paragrafo del NATURA JOIN (pagina precedente):

```
SELECT *
FROM PERSONE INNER JOIN AUTO
ON PERSONE.PATENTE = AUTO.PROPRIETARIO;
```

NOME	PATENTE	TARGA	PROPRIETARI
GIOVANNI	156	VT AC73949	156
ARTURO	172	ROMA J1003	172

Possiamo renderci conto che questo tipo di join è simile al *natural join*; *i*nfatti **il *natural join* è un particolare caso di *inner join*.**

OUTER JOIN

Con l'OUTER JOIN è possibile estrapolare anche quei dati, appartenenti ad una delle tabelle, che non verrebbero estrapolati nei tipi di join visti fino a questo momento. Infatti OUTER significa esterno; dati esterni al normale tipo di join.
Dobbiamo specificare quale è la tabella di cui vogliamo estrapolare i dati anche se non soddisfano la condizione di join, questo lo facciamo indicando con LEFT o RIGHT se la tabella in questione è quella che appare a destra o a sinistra del comando JOIN.

```
SELECT . . .
FROM tabella1 [LEFT | RIGHT] JOIN tabella2
ON tabella1.campox condizione tabella2.campoy
```

Vediamo alcuni esempi:

Vogliamo visualizzare nel nostro JOIN oltre a tutte le persone che possiedono un auto e l'auto appartenuta, anche quelle che non possiedono nessuna auto:

```
SELECT *
FROM PERSONE LEFT JOIN AUTO
ON PERSONE.PATENTE = AUTO.PROPRIETARIO;
```

NOME	PATENTE	TARGA	PROPRIETARI
ANTONIO	123		
GIOVANNI	156	VT AC739	156
ARTURO	172	ROMA J1003	172

SELF JOIN

Il SELF JOIN ci consente di unire una tabella con se stessa. La sintassi è simile a quella della query vista nel paragrafo che trattava il CROS JOIN. Vediamo un esempio usando la tabella TAB2:

TAB2

COLONTAB2

RIG1 TAB2
RIG2 TAB2
RIG3 TAB2

```
SELECT  R1.COLONTAB2,  R2.COLONTAB2
FROM  TAB2  R1,   TAB2  R2;
```

```
R1.COLONTAB2    R2.COLONTAB2
-----------------------------------------------
RIG1 TAB2          RIG1 TAB2
RIG2 TAB2          RIG1 TAB2
RIG3 TAB2          RIG1 TAB2

RIG1 TAB2          RIG2 TAB2
RIG2 TAB2          RIG2 TAB2
RIG3 TAB2          RIG2 TAB2

RIG1 TAB2          RIG3 TAB2
RIG2 TAB2          RIG3 TAB2
RIG3 TAB2          RIG3 TAB2
```

Come possiamo vedere dalla query otteniamo un prodotto cartesiano. Dopo la parola chiave SELECT siamo costretti a simulare l'esistenza di due tabelle mente invece ne abbiamo una soltanto. Dopo la parola chiave FROM faremo riferimento al nome delle colonne e alla tabella a cui appartengono:

SELECT *nomeTabellaInesitente1.nomeColonna*, *nomeTabellaInesitente2.nomeColonna*
FROM *nomeColonna nomeTabellaInesitente1*, *nomeColonna nomeTabellaInesitente2*
[WHERE *condizioni*];

Questo tipo di *select* non è particolarmente utile a meno che non si utilizzi la clausola *where* per unire dati che soddisfano una particolare condizione. Vediamo un esempio:

Vogliamo estrapolare dalla tabella sottostante tutte le coppie sposate:

PERSONE

ID_PERSONA	NOME	CONIUGE	SESSO
1	ANTONIO	12	M
12	SILVIA	1	F
2	GIULIO	7	M
3	MARIA		F
6	ROBERTA	9	F
7	ANTONELLA	2	F
9	ARTURO	6	M

```
SELECT   T1.NOME,  T2.NOME
FROM  PERSONE T1,  PERSONE T2
WHERE  T1.ID_PERSONA  =  T2.CONIUGE;
```

T1.NOME	T2.NOME
ANTONELLA	GIULIO
ANTONIO	SILVIA
ARTURO	ROBERTA
GIULIO	ANTONELLA
ROBERTA	ARTURO
SILVIA	ANTONIO

La *select* funziona ma però ci accorgiamo che le coppie vengono ripetute e questo non è esattamente quello che volevamo. Come possiamo risolvere il problema? Lascio ai lettore il compito di trovare la soluzione.

JOIN tra tabelle usando operatori di confronto che non siano il segno di uguale (=)

Possiamo usare dopo la clausola *where* anche tipi di operatori che non siano l'operatore di uguale (=). In casi del genere il join che si ottiene è abbastanza inusuale, ma può accadere che si renda necessario eseguire query di questo tipo.
La sintassi, dunque sarà uguale a quella di tutti i join visti in precedenza con la sola differenza che la dove appare il segno di uguale (=) possiamo usare, al suo posto, qualsiasi altro operatore di confronto.

JOIN su più di due tabelle

Come il prodotto cartesiano può essere eseguito su più di due insiemi, anche i vari tipi di *join* possono essere applicati a più di due tabelle. Fa eccezione il *self join*, ma che comunque può simulare l'esistenza anche di più di due tabelle; vediamo un esempio senza però visualizzare il risultato della query:

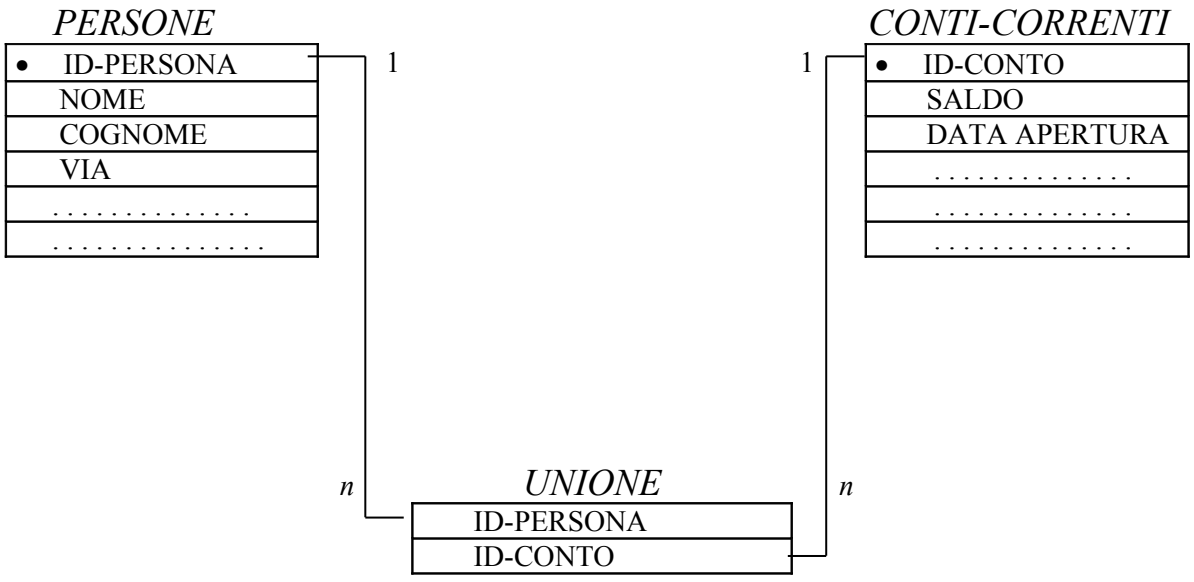
```
SELECT  T1.NOME, T2.NOME, T3.NOME
FROM PERSONE T1,  PERSONE T2, PERSONE T3
WHERE  . . .
```

Un altro caso particolare è quello dell'*outer join* che per essere applicato a più di due tabelle ha bisogno di una sintassi particolare; vediamo uno schema generale e rimandiamo l'approfondimento di tale tipo di sintassi al capitolo seguente.

```
SELECT . . .
FROM tabella1 [LEFT | RIGHT] JOIN ( specifiche di join
tra altre due o più tabelle)
ON . . .
```

I restanti tipi di *join* non presentano particolari sintassi o eccezioni ad essere applicati su più di due tabelle.

Vediamo ora un caso di *join* che si presenta frequentemente tra tre tabelle: chi conosce bene il modello relazionale sa che non è possibile 'correlare' due tabelle usando una relazione di tipo *molti a molti*. In questo caso si utilizza una terza tabella che contiene come chiavi esterne la chiave primaria della prima tabella e la chiave primaria della seconda tabella. Nell'esempio seguente abbiamo la tabella *PERSONE* correlata con la tabella *CONTI-CORRENTI*; il tipo di relazione è di tipo *molti a molti*. Infatti una *persona* può avere più conti correnti e un *conto corrente* può appartenere a più persone.



Se vogliamo visualizzare i dati così correlati usiamo la seguente query:

```
SELECT . . .
FROM PERSONE, UNIONE, CONTI-CORRENTI
WHERE PERSONE.ID-PERSONA = UNIONE.ID-PERSONA
AND UNIONE.ID-CONTI = CONTI-CORRENTI.ID-CONTI;
```

Alla *select* precedente nulla ci impedisce di aggiungere altre condizioni, magari per poter visualizzare soltanto i conti appartenenti al Sig. Rossi Antonio. Lascio al lettore il compito di aggiungere all'espressione l'ulteriore condizione.

ESERCIZI (capitolo 5)

Nella tabella *ISCRITTI* sono presenti gli associati ad un club, nella tabella *FAMILIARI* sono registrati gli appartenenti alla famiglia di ogni associato. Per alcuni esercizi faremo riferimento a queste due tabelle.

ISCRITTI

NOME	MATRICOLA
GIOVANNI	1
ANTONIO	21
RICCARDO	9

FAMILIARI

NOME	PARENTELA	MATRICOLA_DEL_PARENTE_ISCRITTO	ETA
GIULIA	FIGLIA	21	5
MARIA	MOGLIE	21	35
RUGERO	FIGLIO	1	21

1) Dalle tabelle *ISCRITTI* e *FAMILIARI* estrapolare un'unica tabella in cui compaiono per ogni iscritto i suoi familiari e il tipo di parentela. Gli iscritti senza familiari non dovranno apparire.

2) Dalle tabelle *ISCRITTI* e *FAMILIARI* estrapolare un'unica tabella in cui compaiono per ogni iscritto i suoi familiari e il tipo di parentela. Gli iscritti senza familiari dovranno apparire.

3) Dalle tabelle *ISCRITTI* e *FAMILIARI* estrapolare un'unica tabella in cui compaiono per ogni iscritto i suoi familiari e il tipo di parentela, solo se tali familiari hanno un'età inferiore ai 20 anni. Gli iscritti con famigliari che non soddisfano la condizione non dovranno essere visualizzati.

4) La tabella *MARCHE* contiene dati riguardanti le industrie costruttrici di automobili; la tabella *MODELLI* contiene dati riguardanti i vari modelli di auto esistenti. I nomi dei campi in neretto rappresentano le chiavi primarie.

MARCHE

COD_CASA
NOME_CASA
STATO
CITTA

MODELLI

NOME_AUTO
COD_CASA
TIPO
CILINDRATA

Scrivere una *select* che restituisca come risultato il codice della casa, il nome della casa, la città e il nome delle auto per le automobili di tipo s*port* che hanno una cilindrata compresa tra 1000 cc e 2000 cc, estremi inclusi.

5) Scrivere una *select* che restituisca come risultato, dalle tabelle dell'esercizio N° 4, il codice della casa e il nome della casa che produce più di due modelli di automobili di tipo *sport*. Utilizzare la 'tecnica' dell'unione fra tabelle.

6) Date due tabelle (TABELLA1 e TABELLA2) che contengono una colonna chiamata *NUMERO*, come fareste a trovare quei numeri che appartengono a entrambe le tabelle? Scrivere una query.

7) Date le tabelle *STUDENTI*(*matricola, nome_esame, voto*), *MATRICOLA*(*matricola, cognome, nome*) scrivere una *select* che abbia come risultato il cognome e la matricola degli studenti che hanno sostenuto l'esame di informatica riportando una votazione compresa tra 23 e 28, oppure hanno sostenuto l'esame di di informatica.

Subquery (capitolo 6)

Se siamo in grado di padroneggiare con la maggior parte delle 'sintassi' viste nei capitoli precedenti, siamo a buon punto e possiamo realizzare interrogazioni molto complesse. Sebbene ciò però, è facile incappare in casi in cui non è possibile estrapolare i dati in maniera immediata e semplice. Per far fronte a situazione di questo tipo SQL ci mette a disposizione un altro potente strumento sintattico: *la subquery.*
Essa è una query che sta all'interno di un'altra interrogazione. La query interna passa i risultati alla query esterna che li verifica nella condizione che segue la clausola *WHERE;* vediamo i vari tipi di *subquery.*

Subquery che ci restituiscono un solo valore

Per gli esempi di questo paragrafo si farà riferimento alla tabella *DIPENDENTI*:

DIPENDENTI

NOME	DIVISIONE	STIPENDIO	GIORNI_MUTUA	FERIE_GODUTE
ROSSI	VENDITE	2 000 000	33	5
BIANCHI	VENDITE	2 100 000	1	0
BRUNI	RICERCA	3 300 000	0	9
VERDI	ACQUISTI	1 800 000	32	20
GIALLI	RICERCA	4 800 000	0	0
NERI	RICERCA	3 400 000	2	1
MANCINI	AMMINISTRAZIONE	2 400 000	9	24
MARCHETTI	VENDITE	2 000 000	99	12

Vogliamo conoscere il nome dei dipendenti le cui ferie godute superino la media delle ferie godute da tutti. A qualcuno potrebbe venire in mente di scrivere una query di questo tipo:

```
SELECT NOME
FROM DIPENDENTI
WHERE FERIE_GODUTE > AVG(FERIE_GODUTE);
```

ma il risultato che si otterrebbe è un messaggio di errore; infatti non è possibile far seguire la calusola *WHERE* da funzioni di gruppo. Facciamo finta di fare un altro esperimento:

```
SELECT NOME
FROM DIPENDENTI
HAVING FERIE_GODUTE > AVG(FERIE_GODUTE);
```

anche in questo caso quello che si otterrebbe è un messaggio di errore: non è possibile usare la clausola *HAVING* in espressioni dove non compare la clausola *GROUP BY* e non è possibile, nella nostra interrogazione, eseguire raggruppamenti, dunque dobbiamo per forza usare dopo la clausola *WHERE* una *subquery.*

```
SELECT NOME
FROM DIPENDENTI
WHERE FERIE_GODUTE > (SELECT AVG(FERIE_GODUTE)
                      FROM DIPENDENTI);
```

```
NOME
----------
BRUNI
VERDI
MANCINI
MARCHETTI
```

Sapendo a priori che il valore medio delle ferie godute da ogni dipendente è pari a 8,875 giorni possiamo verificare che l'espressione scritta è corretta e estrapola esattamente i dati che ci interessavano.
È evidente che il risultato della *subquery* è un unico valore; infatti non è possibile, con questo tipo di sintassi, estrapolare dalla *subquery* più di un valore e non è possibile usare le clausole *GROUP BY* e *HAVING*.
Ricapitolando **elenchiamo delle regole valide per l'utilizzo di questo tipo di *subquery* :**

- La *subquery* deve restituire un unico valore
- Nella *subquery* non possono apparire le clausole *GROUP BY* e *HAVING*
- La *subquery* deve comparire alla destra dell'operatore di confronto
- Non si possono confrontare due *subquery* (conseguenza della regola precedente).

Subquery con IN

Questo operatore ci consente di estrapolare dalla *subquery* non un solo valore, ma più valori da cui verrà verificata la corrispondenza. Cerchiamo di capire con un esempio questa sintassi:

ISCRITTI

NOME	MATRICOLA
GIOVANNI	1
ANTONIO	21
RICCARDO	9

FAMILIARI

NOME	PARENTELA	MATRICOLA_DEL_PARENTE_ISCRITTO	ETA
GIULIA	FIGLIA	21	5
MARIA	MOGLIE	21	35
RUGERO	FIGLIO	1	21

Nella tabella *ISCRITTI* sono presenti gli associati ad un club di cacciatori, nella tabella *FAMILIARI* sono registrati gli appartenenti alla famiglia di ogni associato. Vogliamo visualizzare gli associati che hanno almeno un famigliare:

```
SELECT *
FROM ISCRITTI
WHERE MATRICOLA
IN
(SELECT MATRICOLA_DEL_PARENTE_ISCRITTO FROM FAMILIARI);
```

NOME	MATRICOLA
ANTONIO	21
GIOVANNI	1

Come possiamo vedere sono stati estrapolati solo i nominativi Antonio e Giovanni, gli unici che hanno familiari, dunque la *subquery* (quella tra parentesi) estrapola una serie di matricole le quali la dove c'è corrispondenza con le matricole della prima *select* la condizione è verificata. Adesso vogliamo visualizzare gli associati che hanno uno o più figlie.

```
SELECT *
FROM ISCRITTI
WHERE MATRICOLA
 IN
    (SELECT MATRICOLA_DEL_PARENTE_ISCRITTO
     FROM FAMILIARI
     WHERE PARENTELA = 'FIGLIA');
```

NOME	MATRICOLA
ANTONIO	21

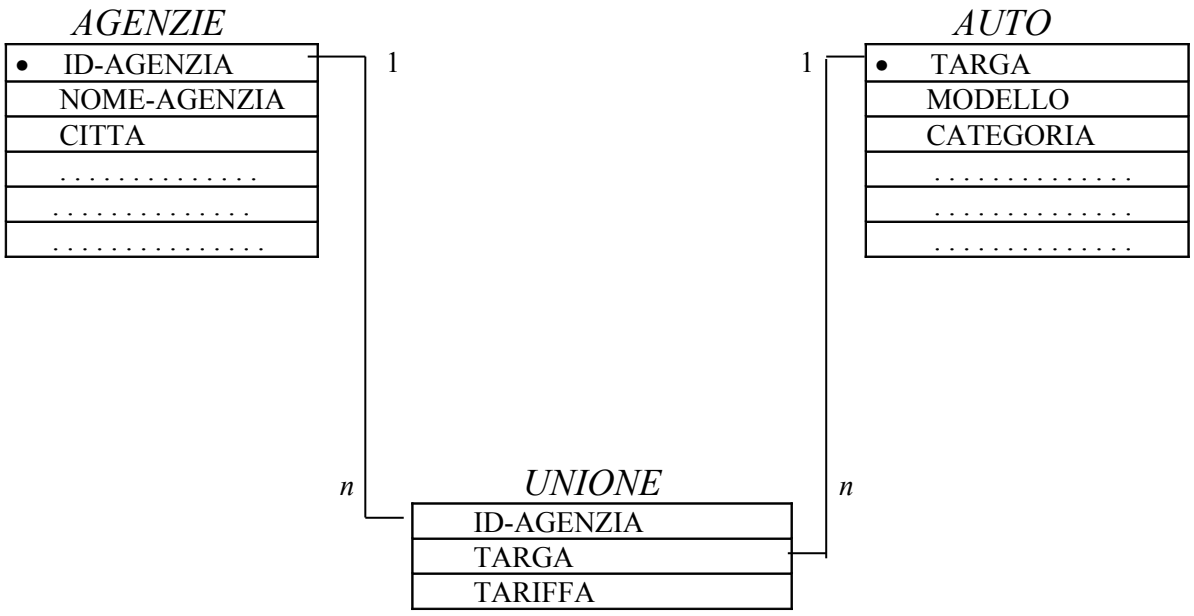
Il 'meccanismo' che abbiamo usato è simile a quello della query precedente, con la differenza che abbiamo aggiunto una ulteriore condizione nella *subquery*. Possiamo aggiungere tante ulteriori condizioni quante ne servono; addirittura considerando una *subquery* come una *query* qualsiasi, nessuno ci impedisce di confrontare nella condizione i valori estrapolati da *'subsubquery'*. Vediamo nel prossimo paragrafo di comprendere meglio quanto detto.

Subquery annidate

Con il termine *annidate* si identificano quelle *query* che si trovano all'interno di *subquery:*

SELECT *nome campi*
FROM *nome tabella*
WHERE (*SUBQUERY condizione* (*SUBQUERY condizione* (*SUBQUERY*)));

nello schema soprastante abbiamo inserito tre *subquery* nella prima *select* una dentro l'altra come se fossero scatole cinesi, ma avremo potuto inserirne anche più. La potenza di una simile struttura è notevole anche se la sua complessità richiede nella sua applicazione una particolare attenzione e tecnica di 'costruzione'. Cercheremo con un esempio di comprendere questo tipo di struttura sintattica. La struttura sottostante rappresenta il database che conserva le informazioni inerenti all'attività che andremo ad analizzare:



Si ha una azienda che noleggia auto. L'azienda opera su tutto il territorio nazionale tramite agenzie presenti nelle maggiori città. Le automobili non possono essere associabili alle singole agenzie e il loro costo di noleggio dipende oltre che dalla categoria dell'auto (utilitaria, sport, gran turismo, ecc.) anche dall'agenzia da dove è stata noleggiata: una automobile noleggiata all'agenzia di Milano costa di più della stessa automobile noleggiata tramite l'agenzia di Catania.
Nello schema si vedono tre tabelle; la tabella *Agenzie* è associata alla tabella *Auto* tramite la tabella *Unione*. Questo tipo di struttura è necessaria per far fronte al tipo di relazione, *molti a molti*, che si ha tra una agenzia e un'auto. Infatti la stessa auto viene associata a tutte le agenzie e ad una agenzia associamo tutte le auto; in questo modo per mezzo dell'attributo *tariffa* sappiamo il costo di ogni auto noleggiata per ogni agenzia.
Supponiamo ora che vogliamo conoscere il nome delle agenzie che applicano una tariffa inferiore alle 100.000 di lire per automobili di categoria sport.
Scomponendo il problema in più moduli che poi risolveremo con delle singole *select,* arriveremo alla soluzione in modo semplice:

Iniziamo dalla selezione delle automobili di tipo sport:

```
SELECT TARGA
FROM AUTO
WHERE CATEGORIA = 'sport' ;
```

Questa query la chiameremo *Q*1.

Continuiamo selezionando le tariffe che soddisfano le condizioni del quesito:

```
SELECT ID_AGENZIA
FROM UNIONE
WHERE TARGA IN (Q1)
AND TARIFFA < 100.000;
```

Questa query la chiamiamo *Q*2.

A questo punto abbiamo a disposizione i codici delle agenzie che soddisfano il nostro quesito, ma non abbiamo ancora a disposizione i nomi di tali agenzie. Inoltre i codici delle agenzie sono inutilmente ripetuti più volte, dunque la successiva query che soddisfa completamente il quesito è:

```
SELECT NOME_AGENZIA
FROM AGENZIE
WHERE ID_AGENZIA IN (Q2);
```

Vediamo ora la query scritta in modo completo:

```
SELECT NOME_AGENZIA
FROM AGENZIE
WHERE ID_AGENZIA IN
        (SELECT ID_AGENZIA
         FROM UNIONE
         WHERE TARGA IN
                (SELECT TARGA
                 FROM AUTO
                 WHERE CATEGORIA = 'SPORT';)
         AND TARIFFA < 100000;);
```

Considerando il modo in cui abbiamo costruito questa interrogazione, possiamo renderci conto che la maniera migliore per effettuare la lettura e comprensione di una query di questo tipo, è iniziare ad analizzare le subquery più interne e man mano passare a quelle più esterne. Questa considerazione è importante ed è valida anche per la scrittura della query. In quest'ultimo caso però, decidere quale sarà la subquery più interna è difficile, comunque sia dobbiamo affidarci non tanto a delle eventuali regole assiomatiche ma alla nostra logica che se utilizzata con rigore non può tradirci.

EXISTS

Questa è una parola chiave che riceve una *subquery* come argomento e restituisce il valore *vero* se la *subquery* restituisce almeno un dato qualsiasi e *falso* se si verifica il contrario. Chiariamo con un esempio quanto detto:

DIPENDENTI

NOME	DIVISIONE	STIPENDIO	GIORNI_MUTUA	FERIE_GODUTE
ROSSI	VENDITE	2 000 000	33	5
BIANCHI	VENDITE	2 100 000	1	0
BRUNI	RICERCA	3 300 000	0	9
VERDI	ACQUISTI	1 800 000	32	20
GIALLI	RICERCA	4 800 000	0	0
NERI	RICERCA	3 400 000	2	1
MANCINI	AMMINISTRAZIONE	2 400 000	9	24
MARCHETTI	VENDITE	2 000 000	99	12

Vogliamo estrapolare tutti i dati di NOME e DIVISIONE da questa tabella solo se e soltanto se è presente il nominativo Neri:

```
SELECT *
FROM DIPENDENTI
WHERE EXISTS
                (SELECT *
                 FROM DIPENDENTI
                  WHERE NOME = 'NERI');
```

NOME	DIVISIONE
ROSSI	VENDITE
BIANCHI	VENDITE
BRUNI	RICERCA
VERDI	ACQUISTI
GIALLI	RICERCA
NERI	RICERCA
MANCINI	AMMINISTRAZIONE
MARCHETTI	VENDITE

Da questa interrogazione i dati vengono estrapolati perché essendo presente la stringa ‘NERI’ la parola chiave EXISTS restituisce il valore *true*. Nel caso la stringa ‘NERI’ non fosse stato presente la parola chiave EXISTS avrebbe restituito *false* e in questo caso l’interrogazione non avrebbe visualizzato nessun valore.

SOME, ANY, ALL

Questi tre operatori svolgono funzioni simili alle parole chiave IN e EXSIST. Si consiglia di consultare altro testo per approfondire l’argomento.

ESERCIZI (capitolo 6)

1) La seguente query è giusta? e se errata perché?

```
SELECT COGNOME, NOME, MATRICOLA, ETA
FROM  DIPENDENTI
WHERE  (SELECT AVG(ETA) FROM DIPENDENTI) <  ETA;
```

2) La seguente query è giusta? e se errata perché?

```
SELECT MATRICOLA
FROM  DIPENDENTI
WHERE ETA >  (SELECT ETA FROM DIPENDENTI);
```

3) Dalla tabella *PERSONE* scrivere una interrogazione che estrapoli tutti i dati delle persone che possiedono almeno un'auto. Usare la tecnica delle subquery.

PERSONE

NOME	PATENTE
ANTONIO	123
GIOVANNI	156
ARTURO	172

AUTO

TARGA	PROPRIETARIO
VT AC3949	156
ROMA J1003	172
MI GH3434	300
NA G666223	301

4) Nella tabella *ISCRITTI* sono presenti gli associati ad un club, nella tabella *FAMILIARI* sono registrati gli appartenenti alla famiglia di ogni associato;

ISCRITTI

NOME	MATRICOLA
GIOVANNI	1
ANTONIO	21
RICCARDO	9

FAMILIARI

NOME	PARENTELA	MATRICOLA_DEL_PARENTE_ISCRITTO	ETA
GIULIA	FIGLIA	21	5
MARIA	MOGLIE	21	35
RUGERO	FIGLIO	1	21

scrivere una *select* che dalle tabelle *ISCRITTI* e *FAMILIARI,* ci visualizzi gli iscritti che non hanno nessun familiare.

5) La tabella *MARCHE* contiene dati riguardanti le industrie costruttrici di automobili; la tabella *MODELLI*
contiene dati riguardanti i vari modelli di auto esistenti. I nomi dei campi in neretto rappresentano le chiavi primarie.

MARCHE

COD_CASA
NOME_CASA
STATO
CITTA

MODELLI

NOME_AUTO
COD_CASA
TIPO
CILINDRATA

Scrivere una *select* che restituisca come risultato il codice della casa e il nome della casa che produce più di due modelli di automobili di tipo *sport*. Utilizzare la 'tecnica' delle *subquery*.

6) Data una tabella *STUDENTI*(*matricola, esame, voto*), scrivere una qurery che abbia come risultato la *matricola* degli studenti che hanno effettuato più esami dello studente matricola 23.

Manipolare i dati (capitolo 7)

Fino a questo punto del corso abbiamo trattato quella parte di SQL che assolve alle funzioni di QL (Query Language) ovvero interrogazioni dei dati senza nessuna possibilità di manipolarli come ad esempio cambiarne il valore. In questo capitolo vedremo invece quella parte di SQL che assolve alle funzioni di DML (Data Manipulation Language). Questa parte di SQL ci consente di *inserire* dati nelle tabelle, di *modificarli* e di *cancellarli*; le corrispondenti istruzioni che assolvono a tale scopo sono: INSERT, UPDATE, DELETE. Possiamo operare con queste tre istruzioni sui dati in modo selettivo o in modo globale. Nel modo selettivo useremo delle *select* e/o delle condizioni per far riferimento a particolari valori o a particolari posizioni nella tabella; nel modo globale non faremo uso di *select* o di condizioni. Vediamo in dettaglio quanto esposto.

INSERT

Supponiamo di disporre delle seguenti tabelle:

PERSONE

Nome	Cognome	Indirizzo	Citta
Giovanni	Verdi	Via Bella	Roma
Antonio	Rossi	Via Casalacio	Roma

GIOCATORI

Cognome	Nome	Squadra
Mancini	Arturo	S. Lorenzo

e di voler inserire i nominativi presenti in PERSONE nella tabella GIOCATORI:

```
INSERT INTO GIOCATORI
            (COGNOME, NOME)
SELECT COGNOME, NOME
FROM PERSONE;
```

il risultato che si ottiene è il seguente:

GIOCATORI

Cognome	Nome	Squadra
Mancini	Arturo	S. Lorenzo
Verdi	Giovanni	
Rossi	Antonio	

Vediamo un altro esempio:

Supponiamo di disporre una tabella DIPENDENTI e di aver appena creato la tabella DIRIGENTI. Abbiamo la necessità di inserire nella tabella DIRIGENTI tutti i dipendenti che siano dirigenti:

```
INSERT INTO DIRIGENTI

SELECT NOME, COGNOME
FROM DIPENDENTI
WHERE QULIFICA = 'DIRIGENTE';
```

Questa *insert* inserisce dentro la tabella DIRIGENTI quello che ha selezionato la restante istruzione formata da una *select*. Vediamo graficamente cosa è accaduto:

Nel database prima della insert:

DIPENDENTI

COGNOME	NOME	MANSIONE
ROSSI	MARIO	RAGIONIERE
NERI	GIULIO	DIRIGENTE
BIANCHI	MARIA	IMPIEGATO
VERDI	LUIGI	DIRETTORE

DIRIGENTI

NOME	COGNOME

Nel database dopo la insert:

DIPENDENTI

COGNOME	NOME	QUALIFICA
ROSSI	MARIO	RAGIONIERE
NERI	GIULIO	DIRIGENTE
BIANCHI	MARIA	IMPIEGATO
VERDI	LUIGI	DIRETTORE

DIRIGENTI

NOME	COGNOME
NERI	GIULIO

La *insert* dell'esempio precedente era di tipo selettivo, vediamo ora una *insert* di tipo globale: Supponiamo di avere la necessità di inserire nella tabella DIRIGENTI un nuovo dirigente il cui nominativo non è presente nella tabella DIPENDENTI in quanto il dirigente in questione non è un dipendente ma un libero professionista 'assoldato' dall'azienda.

```
INSERT INTO DIRIGENTI
VALUES ('GIOVANNI', 'GIOVANNETTI');
```

dopo questa *insert* la tabella DIRIGENTI verrà così trasformata:

DIRIGENTI

NOME	COGNOME
GIULIO	NERI

GIOVANNI	GIOVANNETTI

Supponiamo ora che dobbiamo inserire nella tabella DIPENDENTI un nuovo lavoratore ma non sappiamo quale è la sua qualifica:

```
INSERT INTO DIPENDENTI
(NOME, COGNOME)
VALUES ('AMERIGO', 'VESPUCCI');
```

dopo questa *insert* la tabella DIPENDENTI verrà così trasformata:

DIPENDENTI

COGNOME	NOME	QUALIFICA
ROSSI	MARIO	RAGIONIERE
NERI	GIULIO	DIRIGENTE
BIANCHI	MARIA	IMPIEGATO
VERDI	LUIGI	DIRETTORE
VESPUCCI	AMERIGO	

Una importante differenza tra i due tipi di *insert* è che tramite l'*insert* con *value* si inserisce un record per volta, mentre tramite *insert* con la *select* è possibile inserire molti record alla volta.

È evidente che i valori che inseriamo in una tabella tramite l'istruzione *INSERT* devono essere dello stesso tipo del campo che li riceve; simile discorso vale per la dimensione o la lunghezza del dato da inserire che non deve superare la dimensione o la lunghezza del campo che lo riceve. Una stringa lunga 23 caratteri non può essere contenuta in un campo di tipo stringa lungo 20 caratteri.
Abbiamo usato le parole '*dimensione*' e '*lunghezza*': '*lunghezza*' viene usato per campi di tipo stringa, '*dimensione*' viene usato per campo di tipo numerico.

Campi contatori

I campi contatori sono dei particolari campi i quali contengono valori numerici incrementali. Generalmente questi campi con i suoi valori vengono utilizzati come chiave primaria; infatti è possibile far si che ad ogni nuovo inserimento di record, viene immesso un valore incrementale che è univoco.

In Access per ottenere tutto ciò basta dichiarare un campo di tipo *contatore* e questo automaticamente, ad ogni inserimento di record, si caricherà di un valore incrementato di una unità. Quindi per il primo record avremo il valore 1, per il secondo il valore 2 e così via.

In Oracle il discorso è più complesso, infatti bisogna creare una sequenza e poi permettere che questa generi un valore incrementale da immettere nel campo. Vediamo un esempio di come si crea una sequenza e di come si utilizza:

```
CREATE SEQUENCE sq_atleti     /* Viene creata una sequenza di nome sq_atleti */
        INCREMENT BY 1          /* L'incremento della sequenza è unitario */
        START WITH 1            /*Il primo valore della sequenza sarà 1 */
        NOMAXVALUE  /* Non deve esistere un valore massimo oltre il quale non si possono generare valori  */
        NOCYCLE  /*Non permette che i valori vengano generati  da capo in un nuovo ciclo */
        NOCACHE  /* Non permette di preallocare i valori della sequenza in memoria */
```

```
                ORDER;    /* Garantisce che i numeri della sequenza vengano assegnati alle istanze che li richiedono
nell'ordine con cui vengono ricevute le richieste */
  INSERT INTO atleti
  VALUES('Giovanni',
              'Rossi',
               sq_atleti.NEXTVAL);   /* Viene generato e inserito nel corrispondente campo il valore incrementele */
```

La sintassi di questo esempio è stata testata solo su Oracle.

UPDATE

Questa istruzione serve per modificare i dati contenuti in una tabella. Consideriamo di avere una tabella PRODOTTI con i campi *prodotto* e *prezzo*; ora vogliamo modificare i corrispettivi *prezzi* dei prodotti aumentandoli del 10%:

Tabella prima dell'update:

PRODOTTI

PRODOTTO	PREZZO
MARMELLATA	L. 3.000
CIOCCOLATA	L. 4.500
BISCOTTI	L. 2.500

```
UPDATE PRODOTTI
SET PREZZO = PREZZO * 1.10;
```

Tabella dopo l'*update:*

PRODOTTI

PRODOTTO	PREZZO
MARMELLATA	L. 3.300
CIOCCOLATA	L. 4.950
BISCOTTI	L. 2.750

Questo particolare tipo di *update* viene applicato a tutte le righe per la colonna specificata; esiste anche un altro tipo di *update* le cui righe la dove deve avvenire la modifica vengono selezionate usando una *select*:

Alla tabella PRODOTTI modificata da l'ultimo *update* vogliamo modificare il prezzo della cioccolata aumentandolo ulteriormente di un altro 3%:

```
UPDATE PRODOTTI
SET PREZZO = PREZZO * 1.03
WHERE PRODOTTO = 'CIOCCOLATA';
```

Tabella dopo l'*update:*

PRODOTTI

PRODOTTO	PREZZO
MARMELLATA	L. 3.300
CIOCCOLATA	L. 5.099
BISCOTTI	L. 2.750

È anche possibile modificare più campi alla volta. Vediamo un esempio: Nella tabella PRODOTTI dobbiamo modificare il prezzo della cioccolata aumentandolo ulteriormente di un 15% e dobbiamo cambiare la denominazione da 'cioccolata' in 'nutella'

```
UPDATE PRODOTTI
SET PREZZO = PREZZO * 1.15,
    PRODOTTO = 'NUTELLA'
WHERE PRODOTTO = 'CIOCCOLATA';
```

Tabella dopo l'*update:*

PRODOTTI

PRODOTTO	PREZZO
MARMELLATA	L. 3.300
NUTELLA	L. 5.864
BISCOTTI	L. 2.750

DELETE

Oltre a inserire dati in una tabella o modificarli, occorre anche poterli cancellare. Questa istruzione assolve allo scopo. Va fatto notare che DELETE non cancella un singolo campo per volta ma una riga o più

righe per volta; inoltre questa istruzione cancella i record e non l'intera tabella. Vediamo ora alcuni esempi per meglio comprendere come funziona DELETE.
Si da il caso che dalla tabella PRODOTTI si voglia cancellare la riga della marmellata:

```
DELETE FROM PRODOTTI
WHERE PRODOTTO = 'MARMELLATA';
```

Tabella dopo la delete:

PRODOTTI

PRODOTTO	PREZZO
NUTELLA	L. 5.864
BISCOTTI	L. 2.750

Si da il caso che, ancora non contenti di aver cancellato la sola riga della marmellata, ora vogliamo cancellare tutte le dighe della tabella:

```
DELETE FROM PRODOTTI;
```

Tabella dopo delete senza condizione:

PRODOTTI

PRODOTTO	PREZZO

Come si può vedere dalla sintassi dei due esempi sull'istruzione DELETE, non è possibile cancellare l'intera tabella o singoli campi all'interno delle righe selezionate. Qualcuno a questo punto si potrebbe domandare come far scomparire l'intera tabella o come cancellare un singolo campo? Per cancellare un campo all'interno di una riga possiamo usare l'istruzione UPDATE, mentre invece come far scomparire un'intera tabella lo vedremo al capitolo seguente.

Ricapitolando possiamo affermare che DELETE ci permette di:

- cancellare una sola riga

- cancellare più righe
- cancellare tutte le righe

ROLLBACK, COMMIT

Per le sintassi di questi comandi si è fatto riferimento al tool SQL Plus 8.0 di Oracle. **Tali comandi non sono implementati da Access.**

Quando si inseriscono o si cancellano o si modificano i dati in un database è possibile intervenire al fine di annullare l'operazione o confermarla definitivamente. Ciò è particolarmente utile quando ci accorgiamo di aver eseguito uno dei tre comandi, visti in questo capitolo, sui dati errati, o quando vogliamo confermare definitivamente il comando mandato in esecuzione.

Per far maggiore chiarezza a quanto affermato va detto che i DBMS i quali implementano i comandi ROLLBACK e COMMIT, non rendono effettivi istantaneamente i comandi DELETE, UPDATE e INSERT, ma tengono memoria temporaneamente delle modifiche effettuate in un'altra area. Questo fa si che un utente, che non sia quello che ha eseguito uno dei comandi DELETE, UPDATE e INSERT, non veda le modifiche appartate; mentre l'altro, quello che ha eseguito uno dei tre comandi, veda le tabelle in oggetto come se fossero state modificate definitivamente. Dunque il comando di COMMIT rende definitive le modifiche apportate e il comando ROLLBACK elimina ogni modifica da queste ultime. Cerchiamo di meglio comprendere di quanto detto:

COMMIT esplicito

Rappresenta il comando vero e proprio che l'utente digita per confermare e rendere definitive le modifiche apportate. La sintassi è la seguente e va digitata dopo aver mandato in esecuzione uno o più dei tre comandi visti in questo capitolo.

```
COMMIT;
```

COMMIT implicito

Ci sono delle azioni che mandano in esecuzione il COMMIT in maniera automatica. Una di queste azioni è ad esempio l'uscita dal tool che ci consente di mandare in esecuzione le sintassi SQL o eseguire dei seguenti comandi: CREATE TABLE, CREATE VIEW, DROP TABLE, DROP VIEW. Ci sono altri comandi o azioni che provocano il COMMIT; per ulteriori informazioni si consiglia di consultare il manuale del DBMS che si sta utilizzando.

ROLLBACK automatico

Abbiamo finito di effettuare una serie di aggiornamenti al database, quando ad un certo punto per un fatto accidentale il computer si spegne o si blocca. Ebbene se non è stato effettuato un COMMIT esplicito i nostri aggiornamenti non sortiranno alcun effetto sul database. È questo un caso di ROLLBACK automatico.

ROLLBACK esplicito

È il caso in cui volendo non rendere effettive le modifiche apportate eseguiamo il comando ROLLBACK. La sintassi è la seguente:

```
ROLLBACK;
```

Riepilogo

In questo capitolo abbiamo visto le tre istruzioni più importanti per poter manipolare i dati. Gli esempi esposti erano particolarmente semplici; c'è però da far notare che con le istruzioni *INSERT, UPDATE* e *DELETE* possiamo utilizzare *select* o condizioni complesse e articolate quanto vogliamo.

Creare e mantenere tabelle (capitolo 8)

Il capitolo precedente era dedicato a quella parte di SQL che assolve alle funzioni di DML (Data Manipulation Language). In questo capitolo tratteremo invece, quella parte di SQL che implementa le funzioni di DDL (Data Description Language) ovvero quelle funzioni che ci permettono di progettare, di manipolare e di distruggere quelle strutture che contengono i dati. L'acronimo Description infatti, sta a indicare '*descrizione*' o meglio '*definizione*' delle strutture che conterranno i dati.

Vedremo in dettaglio le istruzioni CREATE TABLE, ALTER TABLE, DROP TABLE, per creare, modificare ed eliminare tabelle. Esistono anche i comandi CREATE DATABASE e DROP DATABASE per creare e distruggere database; questi comandi però non verranno mostrati in questo corso in quanto la loro implementazione varia notevolmente da un DBMS ad un altro. Inoltre tutti i DBMS, oggi in commercio, hanno dei tools di tipo grafico che permettono di creare e di distruggere un database in modo facile e senza scrivere codice SQL.

CREATE TABLE

Questa istruzione, come facilmente si immagina, serve a creare una tabella; il suo schema sintattico è il seguente:

```
CREATE  TABLE  nomeTabella
          (
            campo1 tipoDati  [valore di default] [vincoli],
            campo2 tipoDati  [valore di default] [vincoli],
            campo3 tipoDati  [valore di default] [vincoli],
                  . . . . . . . . . . . . . . . . . . . . . . .
           );
```

Vediamo un esempio:

```
CREATE TABLE ANAGRAFICA
(
      NOME VACHAR2(25),
      COGNOME VARCHAR2(25),
      DATA_NASCITA DATE
 );
```

ANAGRAFICA

NOME	COGNOME	DATA_NASCITA

Questa soprastante è la tabella *ANAGRAFICA* che è stata creata; essa ha tre campi: due di tipo testo che possono contenere fino a 25 caratteri alfanumerici ognuna e un campo di tipo data che può contenere date o orari.
Particolare attenzione deve essere rivolta alle parole chiave utilizzate per la dichiarazione dei tipi dei campi, infatti le parole chiave preposte a questo scopo potrebbero cambiare da DBMS a DBMS. Per gli esempi di questo capitolo, come del resto anche per gli altri, si è fatto riferimento a SQL Plus 8 Oracle; la sintassi e le parole chiave della *create* precedente sono solo in parte accettate da Access. Provate a lanciare questa *create* dal modulo Query di Access, avendo però sostituito prima i tipi dati *VARCHAR2* in *CHAR* e vi accorgerete che la tabella verrà creata, va però fatto notare che non sarà sempre così.

In Access esistono i corrispondenti tipi di dato che implementa Oracle ma le parole chiave utilizzate per dichiararli cambiano. Vediamo tramite lo schema seguente le parole chiave che utilizza Oracle per implementare la dichiarazione dei tipi di dato.

Parole chiave Oracle per dichiarare i tipi di dato

CHAR	Dati alfanumerici di lunghezza variabile compresa tra 1 e 2000 caratteri. Se la stringa immessa è più piccola della dimensione del campo, allora verranno aggiunti tanti spazi a destra della stringa; infatti viene usata quando la lunghezza della stringa da immettere è sempre la stessa.
DATE	Dati di tipo temporale: secoli, anni, mesi, giorni, ore, minuti, secondi.
LONG	Stringhe alfanumeriche come CHAR. La differenza sta nella lunghezza, un LONG può avere una lunghezza variabile fino a 2 gigabyte.
LONG RAW	Dati binari (grafici, suoni, video, ecc.) fino a 2 gigabyte.
NUMBER	Dati numerici positive o negativi.
RAW	Come LONG RAW ma fino a 255 byte.
ROWID	Stringa esadecimale che rappresenta l'indirizzo univoco di una riga di una tabella.
VARCHAR2	Simile a CHAR, ma con una lunghezza variabile compresa tra 1 e 4000 caratteri. Viene usata quando la lunghezza della stringa immessa è variabile.

NOT NULL

In un campo dichiarato con il vincolo NOT NUILL non è possibile non inserire un valore. Se non inseriamo un valore valido in un campo di questo tipo i valori inseriti nella riga non vengono accettati. Facciamo un esempio:

Vogliamo creare una tabella di nome di *TIFOSI* con i campi: *IDENTIFICATIVO, NOME, SQUDRA.* Il campo *IDENTIFICATIVO* deve contenere obbligatoriamente un valore.

```
CREATE TABLE TIFOSI
(
      IDENTIFICATIVO CHAR(9) NOT NULL,
      NOME VARCHAR2(25),
      SQUADRA VARCHAR2(25)
```

```
);
```

TIFOSI

IDENTIFICATIVO	NOME	SQUADRA

Questa soprastante è la tabella generata dalla *create* dell'esempio. Se vogliamo che i dati inseriti nella riga vengono accettati dobbiamo per forza aver inserito anche un valore valido nel campo *IDENTIFICATIVO*, infatti questa è la funzione del vincolo *NOT NULL*. Anche questa *create*, dopo aver sostituito i tipi dati *VARCHAR2* in *CHAR*, se lanciata dal modulo Query di Access viene accettata e genera la tabella corrispondente.

Qualcuno potrebbe pensare che più valori NULL in una tabella e magari nella stessa colonna possiedono lo stesso valore, ma non è così: due valori NULL nella stessa tabella non hanno lo stesso valore. Il valore NULL non è lo zero o il carattere space, ma è un '*nulla*' virtuale ogni volta diverso e mai uguale ad ogni altro.

PRIMARY KEY

Questo vincolo ci permette di dichiarare un campo come chiave primaria. Ricordiamo che la chiave primaria rappresenta quel campo il cui valore rappresenta in maniera univoca la riga; quindi non è possibile trovare all'interno della tabella due valori uguali della chiave primaria. Facciamo un esempio:

Abbiamo bisogno di creare una tabella di nome *CALCIATORI* con i campi: *ID_IDENTIFICATIVO, NOME, COGNOME*. Il campo *ID_IDENTIFICATIVO* deve essere dichiarato chiave primaria:

```
CREATE TABLE CALCIATORI
  (
        ID_IDENTIFICATIVO CHAR(3) PRIMARY KEY,
        NOME VARCHAR2(20),
        COGNOME VARCHAR2(20)
  );
```

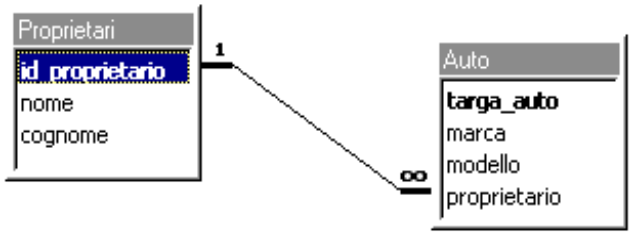
CALCIATORI

ID_IDENTIFICATIVO	NOME	COGNOME

Questa *create* ha costruito una tabella in cui il campo *ID_IDENTIFICATIVO* è chiave primaria; questo significa che non è possibile avere all'interno della tabella due valori del campo *ID_IDENTIFICATIVO* uguali. **Access non ci accetta la clausola *PRIMARY KEY* se la usiamo insieme al comando CREATE TABLE.**

FOREIGN KEY

La chiave esterna di una tabella rappresenta la chiave primaria della tabella master a cui è associata.



In questa immagine acquisita da Access possiamo vedere la tabella master *Proprietari* con la chiave primaria *id_proprietario* e la tabella slave *Auto* con la chiave esterna *proprietario.* Tramite la chiave esterna *proprietario*, riusciamo a sapere a chi appartiene quella auto.

Per sapere come si implementa un chiave esterna su di una tabella consultare il paragrafo ALTER TABLE.

UNIQUE

Questo vincolo non ci permette di inserire nella stessa colonna più di una volta lo stesso valore. Vediamo, con un esempio, di capire quale è la sua utilità:

Immaginiamo di aver bisogno di una tabella *PERSONE* la cui chiave primaria (ID_PERSONA) sia costituita da un valore numerico che si incrementi in maniera automatica ad ogni inserimento di una nuova persona. Vediamo come potrebbe essere e cosa potrebbe contenere questa tabella:

PERSONE

ID_PERSONA	NOME
0	GIOVANNI
1	MARIA
2	MARIA

Nella tabella appaiono due MARIA. Considerando che nella realtà rappresentata dalla tabella diamo per scontato che non ci sono due persone con lo stesso nome è evidente che la doppia MARIA costituisce un errore dovuto al doppio inserimento. Dunque la chiave primaria *ID_PERSONE* non assolve al suo scopo di rappresentare in maniera univoca l'entità persona. Dovremo dunque rinunciare a questo tipo di chiave costituita da un contatore che si incrementa automaticamente ad ogni inserimento? (consultare il sottoparagrafo *Campi contatori* del paragrafo *INSERT* del capitolo 7, per saperne di più sui contatori) Sicuramente no! Infatti il problema si risolve facilmente inserendo il vincolo di non poter inserire valori uguali nella colonna NOME:

```
CREATE TABLE PERSONE
  (
    ID_PERSONA NUMBER PRIMARY KEY,
    NOME VARCHAR2(25) UNIQUE
  );
```

Nella tabella generata da questa *create* non sarà più possibile, grazie al vincolo UNIQUE, inserire inutilmente due volte la stessa persona. Qualcuno però potrebbe obbiettare che la realtà che stiamo rappresentando con la tabella potrebbe cambiare a tal punto da far si che ci siano persone con lo stesso nome. A questo punto dovremmo rinunciare a questo tipo di chiave? Niente affatto! Dovremo però utilizzare una tabella così creata:

```
CREATE TABLE PERSONE
  (
    ID_PERSONA NUMBER PRIMARY KEY,
    NOME VARCHAR2(25),
    COGNOME VARCHAR2(25),
    UNIQUE (NOME, COGNOME)
  );
```

In questo caso potremmo essere sicuri, come vuole la teoria dei database relazionali, che tutte le persone 'presenti' nella tabella appaiono una sola volta. Quanto affermato non è esattamente vero: ricordiamo che il valore NULL è un '*nulla*' virtuale ogni volta diverso. Cosa accadrebbe nella tabella così creata se inserissimo due volte i dati di Mario Rossi, ma senza inserire il nome? Accadrebbe che il sistema non riconosce i due Rossi come la stessa persona, in quanto il valore del campo NOME risulterebbe diverso. Quindi per ovviare anche a questa eventualità dobbiamo utilizzare una tabella creata in questo modo:

```
CREATE TABLE PERSONE
  (
    ID_PERSONA NUMBER PRIMARY KEY,
    NOME VARCHAR2(25) NOT NULL,
    COGNOME VARCHAR2(25) NOT NULL,
    UNIQUE (NOME, COGNOME)
  );
```

Il vincolo UNIQUE è utilizzato non solo per far fronte a ipotetiche situazioni tanto complesse, ma anche semplicemente a non permettere di non inserire un dato fondamentale per l'entità che si sta rappresentando.

La parola chiave UNIQUE non è implementata da Access.

DEFAULT

Questa parola chiave, che non rappresenta un vincolo, ci permette di assegnare un valore ad un attributo quando viene inserita una riga senza che sia specificato un valore per l'attributo stesso. Cercheremo di capire con un esempio quanto affermato:

Vogliamo che nel campo *NUMERO_FIGLI* della tabella *DIPENDENTI* che creeremo venga inserito automaticamente il valore 0 se non specificato diversamente. Vediamo lo schema sintattico di una possibile *create* per assolvere allo scopo:

```
CREATE  TABLE  DIPENDENTI
     (
        . . . . . .
        . . . . . .
        NOME VARCHAR2(25),
        COGNOME VARCHAR2(25),
        NUMERO_FIGLI  NUMBER  DEFAULT 0,
        . . . . . .
     );
```

Nella ipotetica tabella *DIPENDENTI* quando si conferma l'inserimento della riga senza aver inserito alcun valore nel campo *NUMERO_FIGLI* a tale campo verrà inserito automaticamente il valore impostato come *default.*

CHECK

Questa parola chiave ci permette di verificare che il valore che si sta inserendo nel campo rientri in un certo range o abbia un determinato formato. In caso contrario il valore in questione non verrà accettato. Vengono riportate come esempio alcune *CREATE TABLE* **testate su Oracle**:

```
CREATE TABLE componenti_gruppi (
        rif_gruppo              NUMBER,
        componente              CHAR(8),
        capogruppo              NUMBER CHECK (capogruppo IN (0,1)),    /* Permette l'inserimento dei
soli valori 0  o 1 */
        dal                     DATE NOT NULL
);
```

```
CREATE TABLE indirizzi_elettronici (
        rif_indirizzo           NUMBER,
```

```
        tipo                    CHAR(1) CHECK (tipo in ('E','W')), /* Permette l'inserimento dei soli
caratteri 'E' o 'W' */
        address                 VARCHAR2(80)
CREATE TABLE classi_di_rischio0 (
        id_classe               NUMBER(3) CHECK (id_classe >= 900),  /* Permette l'inserimento solo se
è verificata la condizione  */
        descrizione             VARCHAR2(100),
        stato                   CHAR(1) CHECK (stato IN ('A','D'))
);



CREATE TABLE tipi_di_rischio0 (
        id_classe               NUMBER(3) NOT NULL,
        id_tipo_rischio         NUMBER(3) CHECK (id_tipo_rischio >= 1 AND id_tipo_rischio < 900),
        descrizione             VARCHAR2(100),
        premio_minimo           NUMBER,
        durata_minima           NUMBER,
        proroga_minima          NUMBER,
        tasso_grandi_clienti    NUMBER,
        tasso_clienti_medi      NUMBER,
        tasso_clienti_base      NUMBER,
        coefficienti_autonomia  NUMBER,
        numero_appendice        NUMBER,
        stato_rischio           CHAR(2) CHECK (stato_rischio IN ('RO','RN','RS')),
        stato                   CHAR(1) CHECK (stato IN ('A','D')),
        dal                     DATE NOT NULL
);
```

La parola chiave CHECK non è implementata da Access.

Creare una tabella da una già esistente

Se per ipotesi dobbiamo creare una tabella che dovrà contenere i dati duplicati di un'altra tabella già esistente si può usare l'istruzione *CREATE TABLE* con una particolare sintassi; vediamo lo schema:

```
CREATE TABLE  nomeTabella  (campo1, campo2, . . .)
AS (SELECT Campo1, Campo2, . . .
     FROM vecchiaTabella
     WHERE  . . . );
```

questa *create* così strutturata consente di creare una nuova tabella caricando i dati dalla vecchia nei corrispondenti campi della nuova. La corrispondenza dei campi avviene per posizione, cioè il primo campo che viene indicato dopo 'CREATE TABLE (. . . ' corrisponde con il primo campo che viene indicato dopo 'AS (SELECT . . . ' , stesso discorso vale per il secondo campo, per il terzo e così via. A conferma di quanto detto va fatto notare che i nomi dei campi della nuova tabella e quelli della vecchia non devono necessariamente corrispondere, ovvero essere uguali. Saranno uguali invece, i tipi di dato dei campi tra

nuova e vecchia tabella, infatti in questa particolare *create* non appare nessuna dichiarazione al tipo di dati dei campi. La clausola *WHERE* infine ci permette di selezionare e quindi di caricare nella nuova tabella, solo le righe che soddisfano una particolare condizione. **Questa sintassi non è implementata da access.**

ALTER TABLE

Molte volte nasce la necessità di modificare una tabella già esistente modificando il tipo e il nome di una colonna o aggiungendone una nuova. Questo accade principalmente per due motivi: o in fase di progettazione non si è tenuto conto di tutti i campi necessari o la realtà che si sta rappresentando è cambiata a tal punto da rendere necessaria una o più modifiche su quella particolare tabella.

Si utilizza ALTER TABLE anche in fase di implementazione o progettazione del database, per semplificare il lavoro. Ad esempio, prima si creano tutte le tabelle per mezzo di uno script, poi per mezzo di un altro script vi si inseriscono tutte le chiavi primarie e poi le chiavi esterne. In questo modo si avrà una lista separata di tutte le chiavi primarie (script che inserisce le chiavi primarie), una lista di tutte le chiavi esterne (script che aggiunge le chiavi esterne) e così via. Vediamo alcuni esempi:

Aggiungere un campo:

```
ALTER TABLE  DIRIGENTI
ADD NOTE  VARCHAR2 (2000);
```

Questa *alter table* aggiunge un capo di nome *NOTE* e di tipo *VACHAR2* con lunghezza massima di 2000 caratteri alla tabella *DIRIGENTI*. **Access non implementa il tipo dati *VARCHAR2***.

Modificare il tipo ad un campo:

Supponiamo ora che alla tabella *DIRIGENTI* vogliamo modificare il tipo del campo *TITOLO_STUDIO* che è un *CHAR(30)* in *CHAR(60)*:

```
ALTER TABLE  DIRIGENTI
MODIFY TITOLO_STUDIO CHAR(60);  /* MODIFY non è accettato da Access */
```

Modificare l'opzione NOT NULL:

L'opzione MODIFY può anche essere utilizzata per cambiare l'opzione di un campo da *NOT NULL* a *NULL*; vediamo un esempio:

Supponiamo che vogliamo cambiare l'opzione del campo *ID_DIRIGENTE* da *NOT NULL* a *NULL*

```
ALTER TABLE DIRIGENTI
MODIFY ID_DIRIGENTE  CHAR(3) NULL;    /* MODIFY non è accettato da Access */
```

è possibile anche effettuare l'operazione inversa, ovvero modificare l'opzione *NULL* di un campo in *NOT NULL* a patto però che in quella colonna non appaiono valori nulli.

Aggiunge un CHECK:

Viene mostrata la tabella *CREDITI* attraverso la sua *CREATE* ; come si può vedere non esiste nessun *CHECK*:

```
CREATE TABLE crediti (
        id_credito      CHAR(8),
        rif_cliente     CHAR(8),
        tipo_rischio    NUMBER(3),
        ammontare       NUMBER NOT NULL,
        residuo         NUMBER NOT NULL
);
```

Per aggiungere un *CHECK* si utilizza la seguente sintassi:

```
ALTER TABLE crediti
        ADD CONSTRAINT ck_crediti
        CHECK (residuo <= ammontare);          /* Questa sintassi è stata testata solo su Oracle */
```

una volta lanciata questa *ALTER TABLE* non sarà più possibile inserire un valore numerico nel campo *residuo* maggiore del valore presente nel campo *ammontare* o inserire un valore numerico nel campo *ammontare* minore del valore presente nel campo *residuo*.

Modificare un CHECK:

Per meglio comprendere come modificare un *CHECK* verrà mostrato un esempio. Abbiamo la necessità di creare una tabella *STUDENTI* strutturata nel seguente modo:

```
CREATE TABLE studenti (
        nome varchar(20),
        cognome varchar(20),
        livello_corso CHAR(2) CHECK (livello_corso IN ('1°', '2°', '3°'))
);
            /* Questa sintassi è testata solo su Oracle */
```

Come si può vedere nella tabella già esiste un controllo *CHECK* . Ci accorgiamo però, solo dopo aver creato la tabella, che il suo *CHECK* deve essere modificato. Per modificare un *CHECK* si utilizza la stessa sintassi utilizzata per aggiungere un *CHECK*:

```
ALTER TABLE STUDENTI
        ADD CONSTRAINT ck_livello_corso
        CHECK (livello_corso IN ('1°', '2°'));

                /* Questa sintassi è stata testata solo su Oracle */
```

Ancora una volta ci siamo accorti di aver sbagliato il nostro *CHECK* e dunque dovremo rimodificarlo. Per modificare un *CHECK* già modificato dovremo prima distruggere lo specifico *CHECK;* vediamo come:

```
ALTER TABLE STUDENTI
        DROP CONSTRAINT ck_livello_corso;

                /* Questa sintassi è stata provata solo su Oracle */
```

A questo punto non essendoci più alcun *CHECK* potremmo aggiungerne un altro.

Inserire chiavi primarie

A volte nasce la necessita di dichiarare un particolare campo di una tabella, chiave primaria. Vediamo quali sono le sintassi per ottenere ciò:

```
ALTER TABLE nome_tabella
        ADD CONSTRAINT nome_chiave PRIMARY KEY (nome_campo);
```

Può presentarsi il caso che la chiave primaria debba esse di tipo composto:

```
ALTER TABLE nome_tabella
        ADD CONSTRAINT nome_chiave PRIMARY KEY
                        (nome_campo_a, nome_campo_b, nome_campo_c );
```

Aggiungere chiavi esterne

La necessità di aggiungere chiavi esterne è abbastanza frequente. Vediamo quali sono le sintassi:

```
ALTER TABLE nome_tabella
        ADD CONSTRAINT nome_chiave FOREIGN KEY (nome_campo)
        REFERENCES nome_tabella_master;
```

Alcuni DBMS come Access non supportano l'uso della clausola *MODIFY* ; altri DBMS implementano, per l'istruzione ALTER TABLE, sintassi diverse o ulteriori estensioni; quindi si consiglia di consultare la documentazione del sistema che si sta utilizzando per conoscere la sintassi esatta.

DROP TABLE

Questa istruzione serve per eliminare completamente una tabella dal database. Se a questa tabella ci sono associati degli *indici* o delle *view* essi verranno eliminati insieme alla tabella.
Ci rendiamo conto che questa istruzione è particolarmente pericolosa. È consigliabile usarla solo per tabelle temporanee, ovvero per quelle tabelle non importanti per il database e create per un uso temporaneo e marginale. Vediamo alcuni esempi:

Vogliamo eliminare la tabella *DIRIGENTI* dal nostro database.

```
DROP TABLE DIRIGENTI;
```

Il comando non chiede conferma dell'utente ed elimina definitivamente la tabella. Quindi bisogna far attenzione a eliminare la tabella giusta; infatti è possibile trovare in database molto grandi tabelle con lo stesso nome ma comunque appartenenti ad utenti diversi. In questo caso è bene indicare prima del nome della tabella il nome dell'utente; facciamo finta che il nome dell'utente della tabella *DIRIGENTI* sia Arturo:

```
DROP TABLE Arturo.DIRIGENTI;
```

È importante utilizzare sempre questa sintassi per non correre il grave rischio di cancellare la tabella sbagliata.

VIEW e INDICI (capitolo 9)

Le *view* e gli *indici* sono oggetti abbastanza diversi tra di loro; le *view* esistono in RAM, gli *indici* vengono memorizzati su disco. Malgrado questa sostanziale differenza, i due oggetti, hanno in comune il fatto di essere associati a una o più tabelle e di mostrarci i dati in un ordine o formato diverso da quello originale. Cerchiamo di capire meglio di cosa si tratta.

VIEW

Fino a questo punto del corso abbiamo utilizzato oggetti (database, tabelle e campi) che avevano la caratteristica di esistere fisicamente su disco. In questo paragrafo invece andremo a considerare qualcosa che non esiste fisicamente su disco. Questo particolare oggetto è ogni volta ricostruito in RAM, ma è come se esistesse fisicamente su disco. Infatti può essere trattato, a parte qualche piccola limitazione, come se fosse una tabelle vere e propria.

Le *VIEW* vengono create mediante l'istruzione ***CREATE VIEW*** associandogli un nome ed una lista di attributi. I valori presenti nelle *VIEW* possono essere modificati o estrapolati da i comuni comandi applicabili alle tabelle. Vediamo quali sono questi comandi:

- SELECT
- INSERT
- INPUT
- UPDATE
- DELETE
- DROP

Le *VIEW* possono anche essere usate per creare *query* complesse; una volta creata la *VIEW* su di essa è possibile eseguire *query*. Le modifiche dei dati su una *VIEW* si ripercuotono su tutte le tabelle da cui sono state create e viceversa.
Per creare le *VIEW* useremo la seguente sintassi:

```
CREATE VIEW  nomeView [(colonna1, colonna2, . . . )] AS
SELECT  nomi_delle_colonne
FROM . . .
. . . ;
```

questo comando indica a SQL di creare una *VIEW,* di nome *nomeView,* strutturata con le colonne specificate. La *select* che segue serve per estrapolare i dati da una o più tabelle e inserirli nella *VIEW*. Vediamo un altro esempio:

```
CREATE VIEW  pagamenti (nome, cognome, indirizzo, città,  totale) AS
```

```
SELECT clienti.nome, clienti.cognome, clienti.indirizzo, clienti.città,  fatture.totale
FROM clienti, fatture
WHERE clienti.id_cliente =  fatture.riferimentoCliente
AND fattuare.saldata = 'no';
```

lanciando questa *CREATE* si genera una *VIEW* con i campi *nome, cognome, indirizzo, città, totale* in cui avremo tutti i dati dei clienti che hanno pagamenti di fatture in sospeso. I dati della *select* vengono ricavati dal join di due tabelle: *clienti* e *fatture*.

Una *VIEW* di questo tipo è particolarmente utile, basta pensare che si aggiorna in tempo reale man mano che vengono inseriti o modificati i dati nelle tabelle *clienti* e *fatture.*

Modificare i dati di una *VIEW*

Abbiamo già detto che le *VIEW* possono essere considerate come qualsiasi tabelle, quindi possiamo cancellare, modificare, ecc. usando i comandi e le sintassi già viste precedentemente per le semplici tabelle. Quando però, la *VIEW è* il risultato di join tra più tabelle le cose potrebbero complicarsi un po. Si consiglia di essere cauti, magari facendo varie prove, o di consultare il manuale del *DBMS* che si sta utilizzando.

Perché si utilizzano le *VIEW*

Vediamo alcune tipiche applicazioni delle *VIEW*

- Proteggere i dati:

  Uno dei grossi problemi dei database è la sicurezza e la riservatezza dei dati. Le viste ci aiutano ad ottenere questo scopo: si immagini di avere una certa tabella a cui non vogliamo che un certo tipo di utenza possa accedere a tutti i campi indistintamente.
  Per questo tipo di utenza creeremo una vista in cui appariranno soltanto alcuni dei campi, della tabella in questione.

- Convertire le unità di misura:

  Ad esempio se il campo *IMPORTO* contenente valori espressi in sterline e vogliamo visualizzarli in lire l'uso di una *VIEW* risulta essere particolarmente utile. Vediamo un esempio:

  ```
  CREATE VIEW fattureRegnoUnito  (azienda, lire)  AS
  SELECT nomeDitta, importo * 3000
  FROM fattureRegnoUnito;
  ```

- Semplificare la costruzione di *query* complesse:

  In alcuni casi quando si devono estrapolare dati da più tabelle, può essere conveniente per una maggiore semplicità, creare una *VIEW* e a questa applicare una *selct.* Questo tipo di uso delle *VIEW* può sembrare poco ortodosso, ma è sicuramente un modo veloce e relativamente semplice per venire a capo a problemi di interrogazione apparentemente insolubili con le

normali sintassi che seguono la parola chiave *SELEC*

Access non implementa le *VIEW*.

INDICI

Provate ad immaginare un biblioteca dove esistono lunghe cassettiere contenenti migliaia di schede ordinate afabeticamente per titolo, con i dati di ogni singolo libro. Supponete che il bibliotecario in un attimo di follia le lanci tutte in aria e poi le riponga a caso nelle cassettiere. Ora per ritrovare la scheda di un qualsiasi libro dovremo sfogliare le schede una per una fino a trovare il titolo che ci interessava. Mediamente, se le schede sono 1.000.000 dovremmo sfogliare e leggere l'intestazione (titolo del libro) di 500.000 schede prima di trovare quella giusta. Ciò che è stato descritto è quello che accade quando un qualsiasi DBMS cerca un record in una tabella senza usare gli indici; infatti i record all'interno di una tabella rispettano l'ordine di inserimento e non quello alfabetico. Immaginate ora che il bibliotecario non voglia più riordinare le schede ma comunque decida di creare un indice ordinato alfabeticamente in cui ad ogni titolo corrisponde la posizione esatta della scheda. Il bibliotecario costatando che il metodo adottato è funzionale decide di creare un altro indice ordinato alfabeticamente per gli autori, in cui per ogni autore si ha la posizione delle schede contenenti i dati dei libri scritti da quell'autore. Dunque, con il metodo degli indici, il bibliotecario effettuerà le ricerche in maniera particolarmente veloce e mirata. Questo è quello che fanno i DBMS, quando utilizzano gli indici.

In altre parole gli indici sono delle tabelle speciali associate alle tabelle dati, che vengono poi utilizzate durante le operazioni che agiscono su queste ultime.

Contrariamente a molti linguaggi gestionali mirati al trattamento dei file, SQL permette di creare più indici su una stessa tabella. Tuttavia quando si crea un indice, SQL memorizza, oltre ai dati della tabella, anche quelli dell'indice. Quindi ogni variazione alla tabella comporta una variazione agli opportuni puntatori alle righe della tabella e non è detto che ciò sia sempre conveniente. Ad esempio se una tabella cambia spesso dati, allora la presenza di molti indici rallenta il lavoro di aggiornamento. **Riportiamo una lista che ci aiuta a valutare quando è opportuno usare gli indici:**

- Gli indici occupano spazio su disco.
- Possiamo ottimizzare le *query,* tramite l'uso di indici, se queste forniscono modeste quantità di dati (non più del 23%). In caso contrario, allora gli indici non migliorano la velocità di lettura delle query.
- Gli indici di piccole tabelle non migliorano le prestazioni.
- I migliori risultati si ottengono quando le colonne su cui sono stati costruiti gli indici contengono grandi quantità di dati o tanti valori NULL.
- Gli indici rallentano le operazioni di modifica dei dati. Di questo bisogna tenerne conto quando si effettuano molti aggiornamenti. Infatti prima di un massiccio aggiornamento del database sarebbe meglio distruggere tutti gli indici e poi ricrearli.
- Se la condizione delle *query* riguarda un solo campo allora è opportuno usare un indice composto da quella sola colonna. Se la condizione delle *query* riguardano la combinazione di più campi allora è opportuno creare un indice contenente quei campi.

Vediamo la sintassi per creare un indice:

CREATE INDEX *nomeIndice*
ON *nomeTabella* (*nomeColonna1,* [*nomeColonna2*], . . .);

inutile dirlo, la sintassi del comando *CREATE INDEX* varia da DBMS a DBMS. Questa specifica sintassi è stata testata su Personal Oracle 8 e su Access.

SOLUZIONI ESERCIZI CAPITOLO 1

1) a. Non funziona perché manca il carattere di fine istruzione.
   b. È incompleta; manca la clausola FROM seguita dal nome di una tabella.
   c. Manca la virgola tra *nome* e *cognome.*

2) Si.

3) Sono tutte corrette.

4)
```
SELECT  ETA, NOME
FROM ANAGRAFICA;
```

5)
```
SELECT DISTINCT SQUADRA_APPARTENENZA
FROM TIFOSERIA;
```

6) La *select* è sbagliata in quanto non è possibile far precedere nomi di campi alla clausola DISTINCT.

7) La *select* è giusta.

8) La *select* è giusta.

SOLUZIONI ESERCIZI CAPITOLO 2

1)
```
SELECT COGNOME
FROM AMICI
WHERE COGNOME LIKE 'M%' ;
```

Alcuni DBMS, come Access, implementano '*' invece '%'

2)
```
SELECT COGNOME, NOME
FROM AMICI
WHERE NOME = 'MARIA'
AND PR = 'BG';
```

3)

NOME	COGNOME
MARIA	ROSSI
MARIA	VERDI
ALBERTO	MAZZA

4) La query non estrapola nessun nominativo, in quanto nessuna persona che si chiama Maria fa di cognome Mazza;

5)
```
BETWEEN 10 AND 30
```

6)
```
SELECT NOME
FROM PERSONE
WHERE CONIUGE IS NOT NULL
AND SESSO = 'F';
```

7)
```
SELECT NOME
FROM PERSONE
WHERE NOME LIKE 'A%O';
```

Bisogna ricordare che il corrispondente carattere ‘%’ implementato da Access è ‘*’.

8)
```
SELECT NOME
FROM PERSONE
WHERE NOME LIKE  ‘_ _ _O %’;
```

Bisogna ricordare che il corrispondente carattere ‘_’ implementato da Access è ‘?’.

9)
```
SELECT  *,  (PrezzoIngrosso * 1.5) PrezzoVendita
FROM PREZZI;
```

Bisogna ricordare che l’assegnazione del nome alla colonna che conterrà i nuovi prezzi, non è possibile utilizzando Access 8.0.

10) Le espressioni che possiamo utilizzare sono ameno due:

```
SELECT  – PrezzoIngrosso
FROM PREZZI;
```

```
SELECT (PrezzoIngrosso  *  –1)
FROM PREZZI;
```

11)
```
SELECT *
FROM CACCIATORI
UNION
SELECT *
FROM PESCATORI
UNION
SELECT *
FROM SCALATORI;
```

12)
```
SELECT *
FROM CACCIATORI
MINUS
SELECT *
FROM SCALATORI;
```

Bisogna ricordare che l’operatore MINUS non è implementato da Access 8.0

13)
```
SELECT *
FROM SCALATORI
MINUS
SELECT *
FROM CACCIATORI;
```

14)
```
SELECT *
FROM PESCATORI
INTERSECT
SELECT *
FROM CACCIATORI;
```

Bisogna ricordare che l'operatore INTERSECT non è implementato da Access 8.0

15) Si.

16)
```
SELECT *
FROM CACCIATORI
WHERE NOME LIKE ' %A%I'
UNION
SELECT *
FROM PESCATORI
WHERE NOME LIKE ' %A%I'
UNION
SELECT *
FROM SCALATORI
WHERE NOME LIKE ' %A%I';
```

Bisogna ricordare che il corrispondente carattere ' %' implementato da Access è '*'.

17)
```
SELECT *
FROM CACCIATORI
WHERE NOME LIKE ' %A %' OR NOME LIKE ' %I'
UNION
SELECT *
FROM PESCATORI
WHERE NOME LIKE ' %A%' OR NOME LIKE ' %I'
UNION
SELECT *
FROM SCALATORI
WHERE NOME LIKE ' %A%' OR NOME LIKE ' %I';
```

SOLUZIONI ESERCIZI CAPITOLO 3

1) Funzioni aggregate.

2) La *query* è sbagliata perché non è possibile applicare la funzione SUM( ) su una colonna che contiene dati di tipo *caratteri* o *stringa.*

3) INITCAP. È necessario ricordare però, che questa funzione non è implementata da Access 8.0

4) È giusta in quanto conta il numero di nomi selezionati.

5) c

6) a

7) La funzione in grado di farlo è CONCAT e l'operatore è '||'. È necessario ricordare però, che sia la funzione CONCAT che l'operatore '||' non sono implementati da Access 8.0.

8) È sbagliata, infatti mancano le parentesi che devono racchiudere i parametri passati alla funzione.

9) È sbagliata.

10) La risposta giusta è d. È necessario però, far notare che questa sintassi non viene accettata da Access 8.0

11)
```
SELECT SUBSTR(NOME, 1, 1) ||'.'||
SUBSTR(COGNOME, 1, 1) ||'.' INIZIALI, CODICE
FROM CARATTERI
WHERE CODICE = 32;
```

È necessario ricordare che questa funzione non esiste in Access 8.0.

SOLUZIONI ESERCIZI CAPITOLO 4

1) La sintassi è errata:
   - Tutte le colonne selezionate (quelle che seguono la clausola SELECT) devono essere elencate in GROUP BY.
   - La clausola GROUP BY deve apparire prima della clausola ORDER BY.

2) Si se non appare la clausola GROUP BY; no se appare la clausola GROUP BY.

3) Si.

4) No.

5)
```
SELECT GENERE
FROM LIBLRI
GROUP BY GENERE
HAVING MIN(PREZZO) > 10000;
```

6) Tutte meno la divisione Ricerca.

7)
```
SELELET DIVISIONE
FROM DIPENDENTI
GROUP BY DIVISIONE
HAVING MIN(FERIE_GODUTE) > 0;
```

8)
```
SELECT BENEFICIARI
FROM ASSEGNI
GROUP BY BENEFICIARIO
HAVING MAX(IMPORTO) > 400 000;
```

9)
```
SELECT BENEFICIARIO
FROM ASSEGNI
GROUP BY BENEFICIARIO
HAVING AVG(IMPORTO) > 300 000
ORDER BY BENEFICIARIO DESC;
```

SOLUZIONI ESERCIZI CAPITOLO 5

1) Possiamo ottenere quanto richiesto utilizzando una delle seguenti *select.*
La seconda *select* utilizza usa una sintassi non implementabile tramite SQL Plus 8.0; notiamo che i nomi dei campi specificati in ON sono preceduti dal nome della tabella a cui appartengono.

```
SELECT ISCRITTI.NOME, FAMILIARI.NOME, PARENTELA
FROM ISCRITTI, FAMILIARI
WHERE MATRICOLA = MATRICOLA_DEL_PARENTE_ISCRITTO;

SELECT ISCRITTI.NOME, FAMILIARI.NOME, PARENTELA
FROM ISCRITTI INNER JOIN  FAMILIARI
ON ISCRITTI.MATRICOLA = FAMILIARI.MATRICOLA_DEL_PARENTE_ISCRITTO;
```

2)
```
SELECT ISCRITTI.NOME, FAMILIARI.NOME, PARENTELA
FROM ISCRITTI LEFT JOIN FAMILIARI
ON ISCRITTI.MATRICOLA = FAMILIARI.MATRICOLA_DEL_PARENTE_ISCRITTO;
```

3) Possiamo utilizzare una delle seguenti *select.*

```
SELECT ISCRITTI.NOME, FAMILIARI.NOME, PARENTELA
FROM ISCRITTI INNER JOIN FAMILIARI
ON
(ISCRITTI.MATRICOLA = FAMILIARI.MATRICOLA_DEL_PARENTE_ISCRITTO AND ETA < 20);

SELECT ISCRITTI.NOME, FAMILIARI.NOME, PARENTELA
FROM ISCRITTI, FAMILIARI
WHERE ISCRITTI.MATRICOLA = FAMILIARI.MATRICOLA_DEL_PARENTE_ISCRITTO
AND ETA < 20;
```

4) Una possibile soluzione del problema è la seguente *select*:

```
SELECT MARCHE.COD_CASA, NOME_CASA, CITTA, NOME_AUTO
FROM MARCHE, MODELLI
WHERE MARCHE.COD_CASA = MODELLI.COD_CASA
AND CILINDRATA BETWEEN 1000 AND 2000
AND TIPO = 'SPORT';
```

Possiamo ottenere lo stesso risultato utilizzando le parole chiave *INNER JONI* e/o adoperare gli operatori di confronto classici invece di *BETWEEN* e *END*.

5) Possiamo utilizzare una delle seguenti *select.*

```
SELECT  MARCHE.COD_CASA,  NOME_CASA
FROM MARCHE,  MODELLI
WHERE  MARCHE.COD_CASA = MODELLI.COD_CASA
AND TIPO = 'SPORT'
GROUP BY  MARCHE.COD_CASA,  NOME_CASA
HAVING COUNT(*) > 2;
```

```
SELECT MARCHE.COD_CASA,  NOME_CASA
FROM MARCHE  INNER JOIN  MODELLI
ON
     (MARCHE.COD_CASA = MODELLI.COD_CASA
          AND TIPO = 'SPORT')
GROUP BY  MARCHE.COD_CASA,  NOME_CASA
HAVING COUNT(*) > 2;
```

6) Possiamo utilizzare una delle seguenti soluzioni:

```
SELECT NUMERO
FROM TABELLA1
INTERSECT
SELECT NUMERO
FROM TABELLA2;
```

La parola chiave INTERSECT non è implementata da Access.

```
SELECT TABELLA1.NUMERO
FROM TABELLA1, TABELLA2
WHERE  TABELLA1.NUMERO = TABELLA2.NUMERO;
```

```
SELECT TABELLA1.NUMERO
FROM TABELLA1 INNER JOIN TABELLA2
ON  TABELLA1.NUMERO = TABELLA2.NUMERO;
```

Quest'ultima sintassi non è accettata da SQL Plus 8.0

SOLUZIONI ESERCIZI CAPITOLO 6

1) la query è sbagliata perché la subquery appare a sinistra.

2) la query è sbagliata perché la subquery restituisce una serie di valori e non un solo valore.

3)
```
SELECT *
FROM PERSONE
WHERE PATENTE IN
                    (SELECT  PROPRIETARIO
                     FROM AUTO;);
```

4)
```
SELECT NOME
FROM ISCRITTI
WHERE MATRICOLA NOT IN
                        (SELECT MATRICOLA_DEL_PARENTE_ISCRITTO
                         FROM FAMILIARI);
```

5)
```
SELECT COD_CASA, NOME_CASA
FROM MARCHE
WHERE COD_CASA IN
                       (SELECT COD_CASA
                        FROM MODELLI
                        WHERE TIPO = 'SPORT'
                        GROUP BY COD_CASA
                        HAVING COUNT(*) > 2 ;);
```

6)
```
SELECT MATRICOLA
FROM ESAMI
GROUP BY MATRICOLA
HAVING COUNT(*) >
                    (SELECT COUNT(*)
                     FROM ESAMI
                     WHERE MATRICOLA = “23”;);
```

Note legali:

1. Queste dispense sono liberamente utilizzabili e pubblicabili sul web, a patto che:
    a) Non vengano rimosse queste note legali.
    b) Non venga rimosso il link al sito web della Art Net.

2. Non è permessa la pubblicazione su carta (riviste, libri, stampa, ecc.), salvo esplicita autorizzazione.

3. La Art Net (studio di consulenza informatica che si occupa di: sviluppo software conto terzi o per clienti finali, di outsourcing e body rental, di docenze e corsi d'informatica, di creazione siti web, di sviluppo applicativi gestionali tradizionali o web based) detiene tutti i diritti di copyright.

4. Nel caso di uso di questo materiale, non conforme a queste note, la Art Net perseguirà civilmente e penalmente i trasgressori.

5. Sebbene questi appunti siano stati redatti con coscienza, cura e buona fede, la Art Net non può assumersi alcuna responsabilità circa l'attendibilità del loro contenuto.

Art Net di Arrigoni Roberto via Giovanni XXIII, 4 Civita Castellana (Viterbo) P.I. 01598360566
Tel 0761 51.51.11