Il Linguaggio SQL

SQL (Structured Query Language) nasce in ambiente IBM nel 1974, all'interno del progetto di System R.

Utilizzato come linguaggio di manipolazione, definizione e controllo in R*, Starburst, SQL/DS, DB2, ORACLE, INFORMIX, INGRES, SQL Server, ...

N.B. Ogni sistema fa uso di un proprio dialetto, nonostante sia definito uno standard ANSI (American National Standard Institute) e ISO (International Standards Organization)

Lo standard attuale è noto come SQL 92 ed è articolato in tre livelli:

- entry level
- intermediate
- full

SQL è utilizzabile sia in modo interattivo sia facendo uso di un linguaggio ospite ("embedded SQL")

Caratteristiche principali

linguaggio non procedurale

le istruzioni non contengono sequenze, cicli o test, ma permettono una descrizione diretta di *cosa* si vuole ottenere e non *come*

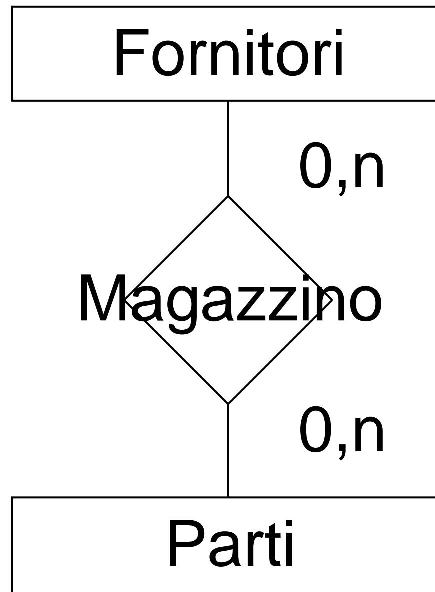
manipolazione "set-oriented" (insiemi di tuple)

gli insiemi di tuple interessati da una operazione sono definiti sulla base di *predicati*

completezza relazionale

> *ogni espressione dell'algebra relazionale può essere realizzata facendo uso di statement SQL*

Esempio guida - schema concettuale



Esempio guida

F

FNUM	**FNOME**	**VOLUME**	**CITTA**
F1	Sassi	20	Lecce
F2	Galli	10	Pisa
F3	Bisi	30	Pisa
F4	Celli	20	Lecce
F5	Adami	30	Ancona

P

PNUM	**PNOME**	**COLORE**	**PESO**	**CITTA**
P1	Dado	Rosso	12	Lecce
P2	Vite	Verde	17	Pisa
P3	Chiodo	Blu	17	Roma
P4	Chiodo	Rosso	14	Lecce
P5	Anello	Blu	12	Pisa
P6	Tappo	Rosso	19	Lecce

M

FNUM	**PNUM**	**QTA**
F1	P1	300
F1	P2	200
F1	P3	400
F1	P4	200
F1	P5	100
F1	P6	100
F2	P1	300
F2	P2	400
F3	P2	200
F4	P2	200
F4	P4	300
F4	P5	400

Oggetti e Operatori di base (cenni)

Dati Numerici:	INTEGER	(o INT)
	SMALLINT	
	DECIMAL(p,q)	{ p cifre, q decimali }
	FLOAT(p)	{ p cifre }
Dati Stringa:	CHAR[ACTER](n)	{ n < 255 }
	VARCHAR(n)	{ long strings }
Data/Tempo:	DATE	
	TIME	
	TIMESTAMP	{ = DATE + TIME }

Sono supportati gli operatori numerici di base, gli operatori logico-relazionali, e operatori per la manipolazione di stringhe.

Null Values (`NULL`)

SQL utilizza una logica a 3 valori (T = 'True', F= 'False', e ? = 'Unknown' o `NULL`), basata sulle seguenti tabelle di verità:

AND	T	?	F
T	T	?	F
?	?	?	F
F	F	F	F

OR	T	?	F
T	T	T	T
?	T	?	?
F	T	?	F

NOT	
T	F
?	?
F	T

Due `NULL` non sono considerati uguali se confrontati tra loro, ma lo sono ai fini dell'indiciamento e altre operazioni.

Valore di default (es.: stringa vuota, zero, ecc.): permette di accettare informazione incompleta anche nei casi in ciò non sarebbe lecito.

Nella definizione di un attributo sono pertanto possibili le opzioni `NOT NULL` e `NOT NULL WITH DEFAULT`.

Permette la creazione, modifica, e cancellazione sia di oggetti "logici" (le relazioni e le viste) sia di oggetti fisici (gli indici).

Creazione di una relazione (table)

```
CREATE   TABLE      F
( FNUM    CHAR(5)    NOT NULL,
  FNOME   CHAR(20)   NOT NULL WITH DEFAULT,
  VOLUME  SMALLINT   ,
  CITTA   CHAR(15)   NOT NULL WITH DEFAULT
   PRIMARY KEY (FNUM));
CREATE   TABLE      M
( FNUM    CHAR(5)    NOT NULL,
  PNUM    CHAR(5)    NOT NULL,
  QTA     SMALLINT
   PRIMARY KEY (FNUM,PNUM))
  FOREIGN KEY (FNUM) REFERENCES F
  FOREIGN KEY (PNUM) REFERENCES P   ;
```

Indici

CREATE INDEX FX on F (FNUM **DESC**) ; (*Il criterio di ordinamento di default è **ASC**.*)

```
DROP    INDEX    FX ;
```

È possibile creare *indici composti* su 2 o più attributi di una relazione:

```
CREATE   INDEX     C_FX on F (CITTA,VOLUME) ;
```

imporre l'unicità della chiave (non necessariamente primaria, ma priva di duplicati all'atto della creazione dell'indice):

```
CREATE   UNIQUE INDEX FX on F (FNUM);
CREATE   UNIQUE INDEX MX on M (FNUM,PNUM);
```

e, infine, forzare l'ordinamento dei dati attraverso la creazione di un indice clustered:

```
CREATE   CLUSTER INDEX CX on F (FNUM) ;
```

Data Manipulation Language

Il DML di SQL si basa su 4 statement di base:

SELECT: per il reperimento (di una o più tuple), ricava da relazioni di partenza una relazione, come gli operatori dell'algebra relazionale

INSERT: per l'inserimento di tuple in una relazione

UPDATE: per l'aggiornamento di tuple di una relazione

DELETE: per la cancellazione di tuple da una relazione

Tutti i quattro statement possono fare uso di una clausola

`WHERE` < condizione >

che è equivalente alla condizione dell'operatore di selezione in algebra relazionale e permette la specifica di condizioni logiche che i dati da reperire (modificare) devono soddisfare.

Lo statement SELECT

La forma generale dello statement è:

```
SELECT   [ALL | DISTINCT ]   <select_list> proiezione
FROM     <table_list>                      prodotto cartesiano
[WHERE   <condition>]                      selezione (include predicati
[GROUP BY <attribute_list>                     di join e altri)
 [HAVING <condition>]]
[ORDER BY <attribute_list>]
```

Solo le clausole SELECT e FROM sono obbligatorie

SELECT: quali informazioni devono essere fornite in uscita (attributi, medie di valori, ecc.)

N.B. La clausola SELECT realizza l'operatore di Proiezione dell'algebra relazionale, non quello di Selezione. Quest'ultimo è realizzato dalla clausola WHERE.

FROM: da quali relazioni si estraggono le informazioni

trovare "*Codici e volumi dei fornitori di Pisa*"

selezione con predicato su Città e proiezione su FNUM e VOLUME

```
SELECT   FNUM, VOLUME
FROM     F
WHERE    CITTA = "Pisa" ;
```

FNUM	VOLUME
F2	10
F3	30

È possibile (talvolta necessario in caso di ambiguità) fare uso di nomi di attributi **qualificati** con il nome della relazione

```
SELECT   F.FNUM AS "Fornitore", F.VOLUME AS "Volume Affari"
FROM     F
WHERE    F.CITTA = "Pisa" ;
```

e *rinominare* le colonne di uscita

Relazioni e tabelle

Le *tables* di SQL sono collezioni di tuple, con eventuali duplicati

Per produrre "relazioni" e non solamente "tables" è sufficiente utilizzare l'opzione `DISTINCT` (`UNIQUE` è un sinonimo).

```
SELECT   PNUM
FROM     M ;
```

```
SELECT DISTINCT   PNUM
FROM   M ;
```

PNUM
P1
P2
P3
P4
P5
P6
P1
P2
P2
P2
P4
P5

PNUM
P1
P2
P3
P4
P5
P6

"*Codici delle parti fornite dal fornitore F1 in quantità maggiore di 200.*"

```
SELECT   PNUM, QTA
FROM     M
WHERE    FNUM = "F1"
AND      QTA > 200 ;
```

PNUM	QTA
P1	300
P3	400

"*Tutti i dati dei fornitori di Pisa*"

```
SELECT   *
FROM     F
WHERE    CITTA = "Pisa" ;
```

"*Codice e volume dei fornitori di Pisa, in ordine discendente di volume*"

```
SELECT   FNUM, VOLUME
FROM     F
WHERE    CITTA = "Pisa"
ORDER BY VOLUME DESC ;
```

N.B. Gli attributi citati nella clausola ORDER BY devono comparire nella clausola SELECT (anche implicitamente se si usa *).

```
SELECT   FNUM, 10*QTA
FROM     M
WHERE    PNUM = "P1"
ORDER BY 2, 1 ;
```

Interrogazioni di range

"*Parti il cui peso è compreso tra 16 e 19*"

```
SELECT   PNUM
FROM     P
WHERE    PESO BETWEEN 16 AND 19 ;
```

equivale a:

```
SELECT   PNUM
FROM     P
WHERE    PESO >=16
AND      PESO <=19 ;
```

Interrogazioni di set

"*Parti il cui peso è 14, 18, o 19*"

```
SELECT   PNUM
FROM     P
WHERE    PESO IN (14, 18, 19) ;
```

equivale a:

```
SELECT   PNUM
FROM     P
WHERE    PESO = 14
OR       PESO = 18
OR       PESO = 19 ;
```

Esistono le forme NOT BETWEEN e NOT IN

Interrogazioni LIKE

Usando le wildcards % (stringhe arbitrarie) e _ (singolo carattere) è possibile ricercare stringhe arbitrarie in specifici attributi.

"*Tutti i fornitori il cui nome inizia con C*"

```
SELECT   *
FROM     F
WHERE    FNOME LIKE "C%" ;
```

Interrogazioni e NULL

Si supponga che il volume di F5 sia NULL, anzichè 30. Allora la query

```
SELECT   FNUM
FROM     F
WHERE    VOLUME > 25 ;
```

restituisce il solo F3. In generale, nessun operatore di confronto restituisce True in presenza di NULL.

Per selezionare tuple con NULL, si usa:

```
SELECT   FNUM
FROM     F
WHERE    VOLUME IS NULL ;
```

Interrogazioni di join

Quando in SELECT si specificano più relazioni, SQL realizza una operazione di *prodotto cartesiano* e, in combinazione con la clausola WHERE, un -join.

I predicati che confrontano attributi di diverse relazioni sono detti *predicati di join*.

"*Tutte le combinazioni di parti e di fornitori situati nella stessa città*"

```
SELECT   F.FNUM, F.CITTA, P.PNUM,
P.CITTA
FROM     F, P
WHERE    F.CITTA = P.CITTA ;
```

F.FNUM	F.CITTA	P.PNUM	P.CITTA
F1	Lecce	P1	Lecce
F1	Lecce	P4	Lecce
F1	Lecce	P6	Lecce
F2	Pisa	P2	Pisa
F2	Pisa	P5	Pisa
F3	Pisa	P2	Pisa
F3	Pisa	P5	Pisa
F4	Lecce	P1	Lecce
F4	Lecce	P4	Lecce
F4	Lecce	P6	Lecce

Vengono concatenate le tuple di F e P con lo stesso valore del campo CITTA. Il risultato è poi proiettato sui campi della select_list.

Join con "predicati locali"

"*Tutte le parti in città diverse da quella del fornitore F1.*"

```
SELECT   P.PNUM, P.CITTA, F.CITTA
FROM     F, P
WHERE    F.CITTA <> P.CITTA
AND      F.FNUM = "F1";
```

P.PNUM	P.CITTA	F.CITTA
P2	Pisa	Lecce
P3	Roma	Lecce
P5	Pisa	Lecce

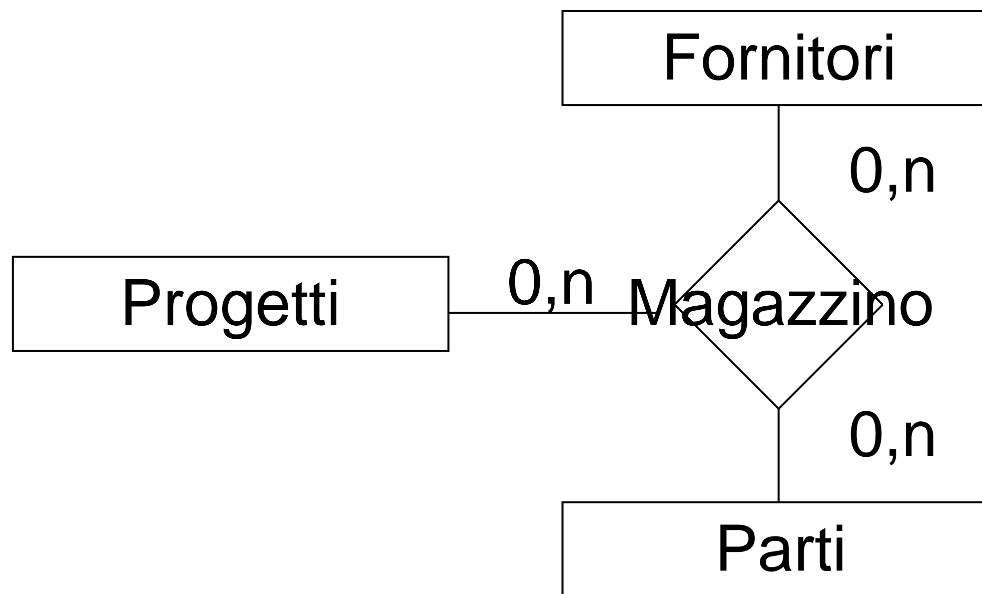
Self Join

Esprime una condizione su attributi della stessa relazione. Occorre distinguere i diversi riferimenti alle tuple facendo uso di "variabili di range" (alias)

```
SELECT   FIRST.FNUM, SECOND.FNUM
FROM     F FIRST, F SECOND
WHERE    FIRST.CITTA = SECOND.CITTA
AND      FIRST.FNUM < SECOND.FNUM ;
```

FIRST.FNUM	SECOND.FNUM
F1	F4
F2	F3

Altro esempio



DB schema = { F(FNUM, FNOME, VOLUME, CITTA),

P(PNUM,PNOME,COLORE,PESO,CITTA),

J(JNUM,JNAME,CITTA),

FPJ(FNUM, PNUM, JNUM, QTA) }

FPJ: La parte PNUM è stata fornita dal fornitore FNUM per il progetto JNUM in quantità QTA.

Q1: "*Fornitori del progetto J1, in ordine di codice*"

```
SELECT   DISTINCT FNUM
FROM     FPJ
WHERE    JNUM = "J1"
ORDER BY FNUM
```

Q2: "*Come Q1, ma ordinati per nome*"

```
SELECT   DISTINCT F.FNUM,F.FNOME
FROM     F, FPJ
WHERE    FPJ.JNUM = "J1"
AND      F.FNUM = FPJ.FNUM
ORDER BY F.FNOME
```

Q3: "*Tutte le coppie di città tali che un fornitore della prima ha rifornito un progetto della seconda*"

```
SELECT   DISTINCT F.CITTA, J.CITTA
FROM     F, J, FPJ
WHERE    F.FNUM = FPJ.FNUM
AND      J.JNUM = FPJ.JNUM
```

Q4: "*Parti fornite ad un progetto da un fornitore della stessa città del progetto*"

```
SELECT   DISTINCT FPJ.PNUM
FROM     F, J, FPJ
WHERE    F.FNUM = FPJ.FNUM
AND      J.JNUM = FPJ.JNUM
AND      F.CITTA = J.CITTA
```

N.B. È una cosiddetta "query ciclica" in quanto il grafo con "nodi" F, J, e FPJ e "archi" i predicati di join è ciclico.

Q5: "*Progetti per cui esiste almeno un fornitore di un'altra città*"

```
SELECT   DISTINCT J.JNUM
FROM     F, J, FPJ
WHERE    F.FNUM = FPJ.FNUM
AND      J.JNUM = FPJ.JNUM
AND      F.CITTA <> J.CITTA
```

Q6: "*Tutte le coppie di parti tali che esiste un fornitore che le abbia fornite entrambe*"

```
SELECT   FPJX.PNUM, FPJY.PNUM
FROM     FPJ FPJX, FPJ FPJY
WHERE    FPJX.FNUM = FPJY.FNUM
AND      FPJX.PNUM < FPJY.PNUM
```

Query innestate (subquery)

Una query si dice innestata se nella clausola WHERE è presente un altro statement SELECT.

La presenza di "nested queries" giustifica il termine Structured di SQL. È da notare che nei primi prototipi di SQL l'uso di nested queries era l'unico modo per poter eseguire dei join.

"*Nomi dei fornitori che hanno fornito la parte P1*"

```
SELECT   F.FNOME
FROM     F, M
WHERE    F.FNUM = M.FNUM
AND      M.PNUM = "P1"
```

diventa

```
SELECT   FNOME
FROM     F
WHERE    F.FNUM IN (  SELECT FNUM
                      FROM   M
         WHERE      PNUM = "P1"  )
```

La valutazione della subquery restituisce l'insieme di codici ("F1", "F2"). Il SELECT esterno è pertanto equivalente a

```
SELECT   FNOME
FROM     F
WHERE    F.FNUM IN ("F1", "F2")
```

e il risultato è { ("Sassi"), ("Galli") }.

"*Nomi dei fornitori che hanno fornito almeno una parte rossa*"

```
SELECT   F.FNOME
FROM     F, P, M
WHERE    F.FNUM = M.FNUM
AND      P.PNUM = M.PNUM
AND      P.COLORE = "Rosso"
```

diventa

```
SELECT   FNOME
FROM     F
WHERE    F.FNUM IN
  (      SELECT     FNUM
         FROM       M
         WHERE      PNUM IN
         (          SELECTPNUM
                    FROM      P
                    WHERE     COLORE = "Rosso" ))
```

La subquery piu interna ha risultato ("P1", "P4", "P6").

La subquery intermedia ha risultato ("F1", "F2", "F4") (in realtà occorrerebbe DISTINCT)

Anche se semanticamente una query innestata equivale ad una query senza nesting, non è detto che il DBMS (l'ottimizatore) le tratti allo stesso modo, per ciò che riguarda le modalità di esecuzione (la forma innestata può dar luogo a interpretazioni procedurali).

NB: Le due query sono *equivalenti* dal punto di vista dell'algebra relazionale; all'atto pratico, possono differire per la presenza di duplicati: in questo caso, la versione con subquery proietta la relaz. esterna su un campo chiave (alternativa), quindi privo di duplicati, mentre il join a tre produce uno stesso fornitore tante volte quante sono le parti rosse da lui fornite.

Subquery correlate

Nel SELECT innestato si può fare riferimento a variabili del SELECT esterno. In questo caso (almeno concettualmente) la subquery non può essere valutata in un singolo passo.

"*Nomi dei fornitori che hanno fornito la parte P1*"

```
SELECT   FNOME
FROM     F FX
WHERE    "P1" IN
  (      SELECT     PNUM
         FROM       M
         WHERE      M.FNUM = FX.FNUM )
```

FX è una variabile il cui range sono le tuple di F. Per ognuna di queste la subquery viene valutata, e restituisce tutte le parti relative al fornitore corrente. Se P1 è parte del risultato, allora FNOME del fornitore corrente fa parte del risultato finale.

"*Parti fornite da più di un fornitore*"

```
SELECT   DISTINCT PNUM
FROM     M FPX
WHERE    FPX.PNUM IN
  (      SELECT    FPY.PNUM
         FROM      M FPY
         WHERE     FPY.FNUM <> FPX.FNUM )
```

Per ogni tupla di FPX la subquery restituisce le parti fornite da fornitori diversi da quello corrente. Se FPX.PNUM è tra queste, allora fa parte del risultato.

Altri operatori

Gli operatori relazionali =, <, …., si possono usare se e solo se la subquery non ritorna più di un valore (se il risultato è vuoto, allora equivale a NULL)

"*Fornitori della stessa città di F1*"

```
SELECT   FNUM
FROM     F
WHERE    CITTA = (  SELECTCITTA
                    FROM  F
                    WHERE FNUM = "F1" );
```

"Parti di peso massimo per ogni colore"

```
SELECT   PNUM
FROM     P PX
WHERE    PX.PESO > (  SELECT PY.PESO
                            FROM   P PY
                            WHERE  PX.COLORE = PY.COLORE
                            AND PX.PNUM <> PY.PNUM );
```

Questa non è corretta, in quanto il risultato contiene in generale più di una tupla. È anche scorretta perchè esclude le parti che sono le sole ad avere un dato colore.

Il quantificatore esistenziale (EXISTS)

L'espressione EXISTS (SELECT ....) restituisce True se e solo se il risultato del SELECT non è vuoto.

Analogamente, NOT EXISTS (SELECT ....) restituisce True se e solo se il risultato del SELECT è vuoto.

"*Nomi dei fornitori che hanno (o non hanno) fornito la parte P1*"

```
SELECT   FNOME
FROM     F FX
WHERE    EXISTS      ( SELECT  *
                       FROM    M
                       WHERE   M.FNUM = FX.FNUM
                       AND M.PNUM = "P1" )
SELECT   FNOME
FROM     F FX
WHERE    NOT EXISTS  ( SELECT  *
                       FROM    M
                       WHERE   M.FNUM = FX.FNUM
                       AND M.PNUM = "P1" )
```

Si può fare uso del costrutto

<op_rel> ALL (SELECT …)

per verificare che una condizione sia vera per tutti i valori restituiti dalla subquery

"*Parti di peso massimo per ogni colore*"

```
SELECT   PNUM
FROM     P PX
WHERE    PX.PESO >= ALL
  (      SELECT    PY.PESO
         FROM      P PY
         WHERE     PX.COLORE = PY.COLORE )
```

Per una tupla di PX, la subquery restituisce i pesi di tutte le parti con il colore corrente. La parte viene selezionata se il suo peso è >= di quello di tutte le parti restituite.

Funzioni aggregate (column functions)

SQL mette a disposizione una serie di comode funzioni per elaborare i valori di un attributo (MAX, MIN, AVG, SUM) e per contare le tuple che soddisfano una condizione (COUNT).

Numero di fornitori presenti	`SELECT COUNT(*)` `FROM F ;`
Numero di fornitori che forniscono almeno una parte	`SELECT COUNT(DISTINCT FNUM)` `FROM M ;`
Numero di colori di parti presenti	`SELECT COUNT(DISTINCT COLORE)` `FROM P ;`

Peso medio di una parte fornita da F1	`SELECT AVG(PESO)`
	`FROM P, M`
	`WHERE P.PNUM = M.PNUM`
	`AND M.FNUM = "F1" ;`

Quantità totale fornita della parte P3	`SELECT SUM(QTA)`
	`FROM M`
	`WHERE PNUM = "P3" ;`

“Fornitori il cui volume è minore del volume massimo presente”

```
SELECT   FNUM
FROM     F
WHERE    VOLUME <
  (      SELECT    MAX(VOLUME)
         FROM      F )
```

N.B. ...WHERE VOLUME < MAX(VOLUME) non va bene, perchè MAX() si applica a tutte le tuple che soddisfano la clausola WHERE. Poichè questa include MAX(VOLUME).....

"*Parti di peso massimo per ogni colore*"

```
SELECT   PNUM
FROM     P PX
WHERE    PX.PESO =
  (      SELECT    MAX(PY.PESO)
         FROM      P PY
         WHERE     PX.COLORE = PY.COLORE )
```

La clausole GROUP BY e HAVING

Con la clausola GROUP BY <attribute_list> si possono raggruppare le tuple per valori omogenei degli attributi specificati.

Quantità totali fornite per ogni parte	`SELECT PNUM, SUM(QTA)`
	`FROM M`
	`GROUP BY PNUM ;`

Quantità totali fornite per ogni parte il cui codice è compreso tra P2 e P5	`SELECT PNUM, SUM(QTA)`
	`FROM M`
	`GROUP BY PNUM`
	`HAVING PNUM IN ("P2",”P3”,”P4”,"P5") ;`

La clausola HAVING è l'equivalente del WHERE applicata a gruppi di tuple.
"*Quantità totali fornite per ogni parte fornita da più di un fornitore*"
	`SELECT PNUM, SUM(QTA)`
	`FROM M`
	`GROUP BY PNUM`
	`HAVING COUNT(*) > 1 ;`

"*Quantità minime e massime fornite per ogni parte, escludendo le forniture di F1*"

```
SELECT   PNUM, MIN(QTA), MAX(QTA)
FROM     M
WHERE    FNUM <> "F1"
GROUP BY PNUM;
```

L'operatore UNION

È possibile fare l'unione dei risultati di due o più query se le relazioni prodotto sono dello stesso tipo o "compatibili" (dipende dalla versione di SQL). UNION non è strettamente necessario (esiste l'OR!).

"*Parti il cui peso è maggiore di 15 o che sono state fornite da F2*"

```
SELECT   PNUM
FROM     P
WHERE    PESO > 15
UNION
SELECT   PNUM
FROM     M
WHERE    FNUM = "F2"  ;
```

```
SELECT   DISTINCT P.PNUM
FROM     P, M
WHERE    P.PESO > 15
OR       (P.PNUM=M.PNUM
AND      M.FNUM="F2");
```

Attenzione alla priorità dei predicati: M.PNUM=P.PNUM AND (M.FNUM='F2' OR P.PESO>15) non prenderebbe le tuple di P che non sono state fornite da nessuno

Un esempio completo

"*Per tutte le parti la cui quantità fornita è maggiore di 300 (non considerando nel totale le forniture in quantità minore di 200), si forniscano codice e quantità massima fornita, ordinando il risultato su quest'ultima e, a parità, per valori discendenti del codice*"

```
SELECT    PNUM, MAX(QTA)        {5}
FROM      M                     {1}
WHERE     QTA >= 200            {2}
GROUP BY  PNUM                  {3}
HAVING    SUM(QTA) > 300        {4}
ORDER BY 2, PNUM DESC;          {6}
```

1) Si considera la relazione M.
2) Le tuple che non soddisfano QTA >= 200 vengono eliminate.
3) Si raggruppano le tuple restanti per valori di PNUM.
4) Si sommano le quantità in ogni gruppo e si selezionano i soli gruppi per cui SUM(QTA) > 300.
5) Si calcola MAX(QTA) per i restanti gruppi e si estraggono PNUM e MAX(QTA).
6) Si ordina il risultato su MAX(QTA) e quindi su valori discendenti di PNUM.

Lo statement INSERT

L'inserimento di tuple in una relazione può essere eseguito sia fornendo esplicitamente i valori (***single-record Insert***), sia utilizzando il risultato di una query (***multiple-record Insert***).

Insert singolo

```
INSERT
INTO     P( PNUM, CITTA, PESO )
VALUES   ("P7","Ancona",24 ) ;

INSERT
INTO     P
VALUES   ("P8","Molla","Rosa",14,"Napoli");
```

L'ordine deve corrispondere a quello dichiarato nella CREATE TABLE.

Insert multiplo

```
CREATE   TABLE     TEMP
( PNUM   CHAR(6),
  TOTQTY INTEGER ) ;

INSERT
INTO     TEMP ( PNUM, TOTQTY )
         SELECT    PNUM, SUM(QTA)
         FROM      M
         GROUP BY  PNUM ;
```

Lo statement UPDATE

Per l'UPDATE non esiste a priori una distinzione tra UPDATE singolo e multiplo. In generale, il numero di tuple da modificare dipende dai predicati presenti nella calusola WHERE.

Caso di UPDATE singolo

```
UPDATE   P
SET      COLORE = "Giallo",
         PESO = PESO + 5,
         CITTA = NULL
WHERE    PNUM = "P2"  ;
```

Caso di UPDATE multiplo

```
UPDATE   F
SET      VOLUME = 2*VOLUME
WHERE    CITTA = "Lecce"  ;
```

UPDATE con subquery

```
UPDATE   M
SET      QTA = 0
WHERE    "Lecce"    =
  (      SELECT     CITTA
         FROM       F
         WHERE      F.FNUM = M.FNUM )    ;
```

UPDATE di più relazioni

Per aggiornare le tuple di più relazioni, è necessario fare uso di più statement UPDATE. In generale:

Ogni statement UPDATE può modificare le tuple di una singola relazione (lo stesso vale per INSERT e DELETE).

```
UPDATE   F
SET      FNUM = "S9"
WHERE    FNUM = "F2"  ;

UPDATE   M
SET      FNUM = "S9"
WHERE    FNUM = "F2"  ;
```

Il problema di garantire l'integrità delle informazioni è demandato al DBMS, rendendo i due statement parte di una singola *transazione*.

Lo statement DELETE

Vale quanto detto a proposito dell'UPDATE, per ciò che riguarda il numero di tuple che vengono cancellate da una relazione.

Caso di DELETE singolo

```
DELETE
FROM     F
WHERE    FNUM = "F1"  ;
```

Caso di DELETE multiplo

```
DELETE
FROM     F
WHERE    CITTA = "Lecce"  ;
```

DELETE di tutte le tuple

```
DELETE
FROM     F          ;
```

DELETE con subquery

```
DELETE
FROM     M
WHERE    "Lecce"    =
  (      SELECT     CITTA
         FROM       F
         WHERE      F.FNUM = M.FNUM )    ;
```

Vincoli di integrità

In fase di creazion7e di una tabella è possibile definire

- chiave primaria, semplice o composta

```
CREATE TABLE F (FNUM    CHAR(4) NOT NULL,
        FNOME   CHAR(10) NOT NULL,
        VOLUME  SMALLINT,
        CITTA    CHAR(10)
        PRIMARY KEY (FNUM)
        );
```

Nota: la chiave `FNUM` *deve* essere dichiarata `NOT NULL`

• chiavi importate, con indicazione della tabella di riferimento

```
CREATE TABLE M(FNUM    CHAR(4) NOT NULL,
        PNUM    CHAR(4) NOT NULL,
        QTA INTEGER
        PRIMARY KEY (FNUM,PNUM)
        FOREIGN KEY (FNUM) REFERENCES F(FNUM)
         ON UPDATE CASCADE
         ON DELETE CASCADE
        FOREIGN KEY (PNUM) REFERENCES P(PNUM)
         ON UPDATE CASCADE
         ON DELETE CASCADE
        );
```

Nota: la clausola REFERENCES si riferisce implicitamente ad un attributo della tabella referenziata con lo stesso nome.

Azioni di mantenimento del vincolo di riferimento

Siano X la tabella che viene referenziata, con la chiave primaria A, e Y la tabella che effettua il riferimento con la chiave esterna A (quindi la clausola REFERENCES compare nella creazione di Y).
È necessario intraprendere una *azione di mantenimento* in caso di cancellazione di una riga di X che contiene un valore del campo A presente in almeno una riga di Y. Analogamente accade in caso di modifica di un valore del campo A in X.

`CASCADE` l'aggiornamento del valore nella tabella X si propaga alle righe riferite nella tabella Y; la cancellazione di una riga di X con campo A=a si propaga con la cancellazione di tutte le righe di Y con campo A=a

`RESTRICT` l'aggiornamento (o la cancellazione) nella tabella X di una riga con A=a viene rifiutato se esistono in Y righe con A=a; per effettuare la cancellazione occorrerà preventivamente (con altra istruzione SQL) eliminare le righe riferite in Y; per l'aggiornamento la procedura è più complessa, e può essere necessario prima eliminare le righe riferite Y, poi effettuare la modifica in X e re-inserire in Y le righe con il valore nuovo

`SET NULL` l'aggiornamento (o la cancellazione) nella tabella X di una riga con A=a provoca l'aggiornamento delle righe corrispondenti in Y, dove il campo A è posto a NULL

Azioni di mantenimento del vincolo di riferimento (ii)

La scelta di quale azione di mantenimento dei vincoli adottare è a carico del progettista e dipende dal significato delle informazioni memorizzate nelle tabelle.

In SQL standard, in assenza di specifica dell'azione di mantenimento si assume RESTRICT.

Molte implementazioni di SQL richiedono che, in presenza di controllo dell'integrità referenziale, vengano costruiti degli indici.

La definizione di una chiave primaria implica la creazione di un indice unico.

Esempio: all'esecuzione di

```
DELETE FROM F WHERE FNUM="F1";
```

se per la tabella M è stata specificata l'azione CASCADE, il sistema esegue, implicitamente, anche

```
DELETE FROM M WHERE FNUM="F1";
```

Creazione di “viste” logiche

Il risultato di una “select” è una tabella, che esiste soltanto per il tempo necessario alla visualizzazione.

Una “select” può quindi essere utilizzata per la definizione di una relazione. È possibile memorizzare tale definizione sotto forma di view (vista); in seguito, la vista può essere interrogata come le altre relazioni: il DBMS provvede a trasformare le select sulla vista in select equivalenti sulle tabelle reali.

“Vista dei totali forniti per ogni parte”

```
CREATE VIEW P_TOT  (PNUM, PNOME, TOT)
AS        SELECT P.PNUM, PNOME,  SUM(QTA)
          FROM P, M
          WHERE     P.PNUM = M.PNUM
          GROUP BY P.PNUM, PNOME;
```

Reperire i totali forniti per le parti P1, P3 e P5

```
SELECT * FROM P_TOT
WHERE PNUM IN ('P1', 'P3', 'P5');
```

La creazione di vista è permanente: la nuova tabella (virtuale) sarà disponibile fino ad esplicita cancellazione.

Esistono forti limitazioni all'inserimento e all'aggiornamento delle viste (meglio operare sulle tabelle base).

Le viste possono essere utilizzate anche per garantire diversi livelli di privatezza dei dati.

MULTIUTENZA - AUTORIZZAZIONI (privileges)

La presenza di una pluralità di utenti della base di dati pone il problema di *chi* può operare su *quali dati*. SQL adotta il concetto di *privilegio di accesso,* inteso come l'autorizzazione a eseguire una data azione su un oggetto della base di dati.
Ogni oggetto ha un *proprietario*, che può eseguire ogni operazione su di esso.
Se il proprietario vuole consentire ad altri di usare le tabelle e gli oggetti che ha creato deve concedere esplicitamente il privilegio di farlo.
Descrittori di privilegi:

- di uso
- di tabella
- di colonna

Ogni descrittore contiene:

- oggetto al quale si applica il privilegio: tabella o colonna)
- identificatore dell'utente che concede il privilegio
- identificatore dell'utente privilegiato
- identificatore dell'azione permessa
- INSERT, UPDATE, DELETE, SELECT, REFERENCES

AUTORIZZAZIONI (ii)

```
GRANT aut.tab ON tabella TO {PUBLIC | user}
                      [WITH GRANT OPTION]
GRANT ALL PRIVILEGES ON tabella TO user
REVOKE [...] FROM ...
```

La clausola WITH GRANT OPTION permette che chi riceve il privilegio lo trasmetta ad altri

Esempio: l'utente A può consultare le tabelle P e M, mentre l'utente B può consultare soltanto la tabella dei totali forniti per ogni parte

```
GRANT SELECT ON P TO A;
GRANT SELECT ON M TO A;
GRANT SELECT ON P_TOT TO B;
```

SQL da linguaggio “ospite”

È possibile eseguire enunciati SQL da un programma scritto in un linguaggio di programmazione quale COBOL, PL/1, RPG, C, ... ed avere una interazione tra variabili di programma e oggetti SQL

PROBLEMI:

- ✓ SQL opera in modo *“orientato agli insiemi”*, mentre i linguaggi di programmazione imperativi operano in modo *“orientato al record”*
- ✓ È necessario poter utilizzare variabili di programma per la composizione di statement SQL e poter copiare valori di campi in variabili di programma
- ✓ È necessario disporre di strumenti per comunicare al programma lo stato delle esecuzioni SQL

Per il primo punto si introduce la nozione di *cursore:*

- si associa un cursore ad una query di selezione
- si apre il cursore come se si trattasse di un file sequenziale
- si acquisisce un record alla volta dal cursore, con la possibilità di consultare i valori dei campi, aggiornare, cancellare
- si chiude il cursore

Per il secondo punto, ogni implementazione di SQL per uno specifico linguaggio ospite predispone adeguati strumenti sintattici.

Per il terzo punto, è spesso disponibile un parametro SQLCODE per restituire al programma ospite un codice che indica l'esito dell'istruzione SQL eseguita:

```
SQLCODE=0               ok,
SQLCODE<0               situazione di errore
```

Operazioni di cursore

Ogni cursore viene associato ad una specifica SELECT:

```
DECLARE <cursor_name> CURSOR
FOR <select_clause>
```

La query non viene eseguita all'atto della sua dichiarazione, ma solo quando il cursore viene aperto con

```
OPEN <cursor_name>
```

Quando è aperto il cursore identifica un raccolta *ordinata* di righe, e una specifica posizione all'interno dell'ordinamento: ***su*** una certa riga, ***prima*** di una certa riga, ***dopo*** l'ultima riga.

La OPEN porta il cursore sulla prima riga; con l'istruzione

```
FETCH <cursor_name>
INTO <target_list>
```

se ne ricopiano i valori nei parametri target (che devono corrispondere alle colonne indicate nella SELECT relativa) e si passa sulla riga seguente; se si era sull'ultima riga ci si posiziona dopo l'ultima riga.

La riga corrente può essere modificata con

```
UPDATE <table_name>
SET <assign_list>
WHERE CURRENT OF <cursor_name>
```

dove ogni assegnamento ha la forma:

```
<col_name> = [scalar_expression|NULL]
```

La riga corrente può essere cancellata con

```
DELETE FROM <table_name>
WHERE CURRENT OF <cursor_name>
```

Quando tutte le righe sono state esaminate il cursore può essere chiuso con

```
CLOSE <cursor_name>
```

Esempi:

1. Definizione e apertura cursore:

```
DECLARE X CURSOR FOR
SELECT * FROM M WHERE PNO=NOMEVAR

OPEN X
```

2. Update posizionata

```
UPDATE M
SET QTA=M.QTA+10
WHERE CURRENT OF X;
```

4. Cancellazione posizionata

```
DELETE FROM M WHERE CURRENT OF X
```

Composizione dei cataloghi

- La struttura di un DB è memorizzata in un insieme di *cataloghi.* I cataloghi sono, a loro volta, relazioni, interrogabili in SQL.
- La struttura dei cataloghi è standard, ma specifiche implementazioni possono non usarne tutte le parti.

```
SELECT TBNAME
FROM SYSTABLS
WHERE CREATOR='SYSTEM'

SYSTABLS    descrizioni tabelle
SYSCOLS     descrizione colonne
SYSIDXS     informazioni su indici
SYSVIEWS    descrizione viste
SYSSYNS     descrizione sinonimi
SYSCOLAU    autorizzazioni di colonna
SYSKEYS     descrizioni di chiavi
SYSVDEPS    descrizioni dipendenze
SYSAUTH     autorizzazioni di tabella
SYSTIMES    tempi di accesso
```

Descrive quali sono le tabelle di sistema

```
SELECT   COLNAME, COLTYPE,   COLLEN, COLSCALE
FROM     SYSCOLS
WHERE    TBNAME='SYSTABLS'
```

COLNAME	COLTYPE	COLLEN	COLSCALE
TBNAME	C	10	0
CREATOR	C	10	0
TBTYPE	C	1	0
COLCOUNT	N	3	0
CLUSTERRID	N	10	0
INDXCOUNT	N	3	0
CREATED	D	8	0
UPDATED	D	8	0
CARD	N	10	0
NPAGES	N	10	0

Descrive la struttura della tabella SYSCOLS

```
SELECT   COLNAME, COLTYPE, COLLEN, COLSCALE
FROM     SYSCOLS
WHERE    TBNAME='SYSCOLS';
```

COLNAME	COLTYPE	COLLEN	COLSCALE
COLNAME	C	10	0
TBNAME	C	10	0
TBCREATOR	C	10	0
COLNO	N	3	0
COLTYPE	C	1	0
COLLEN	N	3	0
COLSCALE	N	2	0
NULLS	C	1	0
COLCARD	N	10	0
UPDATES	C	1	0
HIGH2KEY	C	8	0
LOW2KEY	C	8	0

Gestione delle transazioni

In SQL una transazione è definita come una sequenza di operazioni che deve risultare indivisibile, ovvero può:

- a) andare completamente a buon fine, oppure
- b) fallire completamente, lasciando la base di dati intatta

Creazione di log-file che memorizza tutte le operazioni sul DB

Con l'uso del log-file possibile:
- determinare il fallimento di una transazione, ristabilendo la situazione iniziale
- ripristinare uno stato corretto del DB in seguito a guasti

Le modalità e i comandi di gestione delle transazioni e dei recuperi variano ampiamente da un sistema all'altro.

Problemi:

- dimensione dei file di log ed audit
- rallentamento del sistema per tenere traccia delle op.
- memorizzare su dispositivi fisici diversi il DB ed i file di log ed audit, per ridurre le probabilità di perdita di entrambi
- soltanto la presenza del log-file permette la gestione delle transazioni

TRANSAZIONI

```
BEGIN TRANSACTION
sequenza di operazioni DML
END TRANSACTION
```

oppure

```
ROLLBACK
```

MULTIUTENZA - LOCKING

Esplicito:

```
DATABASE nomedb EXCLUSIVE
LOCK TABLE nometabella
IN {SHARE | EXCLUSIVE} MODE
```

SHARE permette multiutenza in lettura

```
UNLOCK TABLE
```

Implicito:

Il sistema blocca automaticamente un gruppi di righe interessati da una operazione di INSERT, UPDATE, DELETE, e lo rilascia al termine dell'operazione.

All'interno di una transazione vengono bloccate <u>tutte</u> le righe interessate da INSERT, UPDATE, DELETE.

```
SET LOCK MODE TO [NOT] WAIT
```