Laboratorio di Basi di dati

Linguaggio SQL

Enrico Mensa

Indice degli argomenti - PARTE I

0) Database di esempio

Durante gli esempi adopereremo questo database:

S

S#	SNAME	STATUS	CITY
S1	Smith	40	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	40	London
S5	Adams	30	Athens

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Scew	Red	17	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

Dove S sono i fornitori, P sono le parti e SP sono quante parti vendono i fornitori.

SP

S#	P#	QTY
S1	P1	200
S1	P1	700
S2	P3	400
S2	P3	200
S2	P3	200
S2	P3	500
S2	P3	600
S2	P3	400
S2	P3	800
S2	P5	100
S3	P3	200
S3	P4	500
S4	P6	300
S4	P6	300
S5	P2	200
S5	P2	100
S5	P5	500
S5	P5	100
S5	P6	200
S5	P1	100
S5	P3	200
S5	P4	800
S5	P5	400
S5	P6	500

Simboli:
[] -> Opzionale
{} -> Ripetibile da 0 a N volte
| -> OR
<> -> terminale

1) Operazioni su tabelle

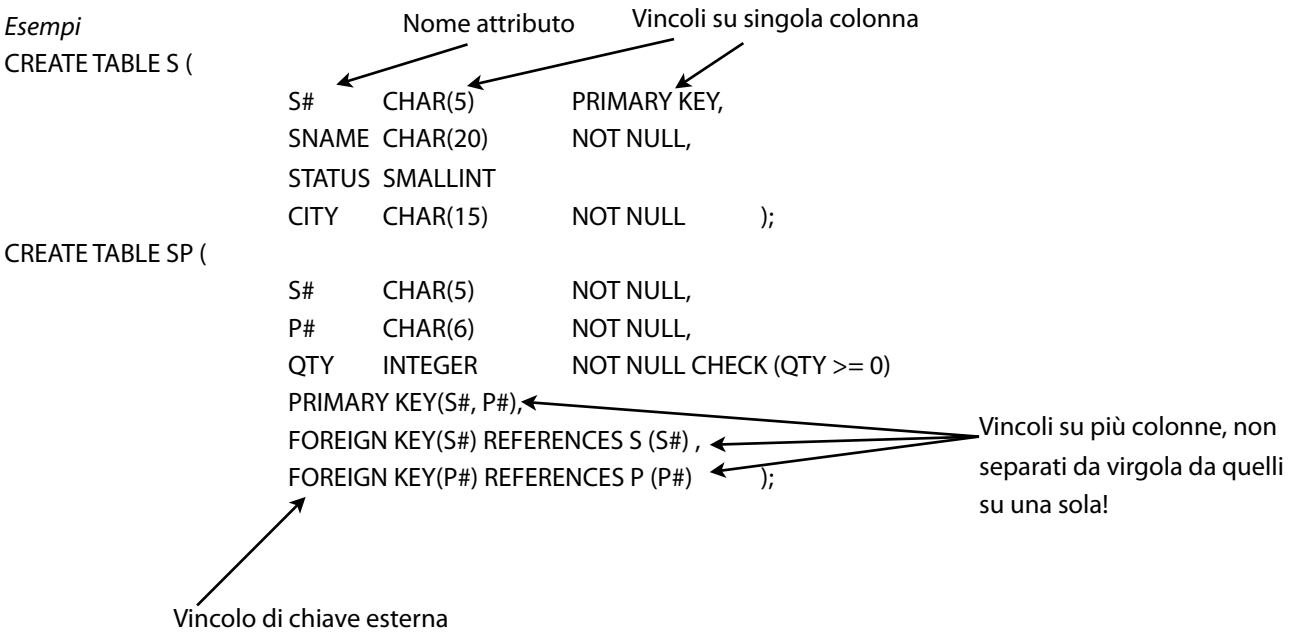
1.1) CREATE TABLE

CREATE TABLE <*Nome tabella*> (**definizione_colonna** {, **definizione_colonna**} {**vincoli_su_più_colonne**})

- definizione_colonna := <*Nome colonna*> tipo-dato [DEFAULT <*Espressione*>] {**vincoli_su_colonna**}
- vincoli_su_colonna :=
[CONSTRAINT <*Nome vincolo*>] { [NOT] NULL | UNIQUE | PRIMARY KEY | **def_key_esterna** | **altri_vincoli** }
- def_key_esterna := REFERENCES <*Nome tabella*> (<*Nome colonna*>)
- vincoli_su_più_colonne :=
[CONSTRAINT <*Nome vincolo*>] { PRIMARY KEY(elenco-col) | FOREIGN KEY (elenco-col) |
REFERENCES <*Nome tabella*> (elenco-col)| **altri_vincoli** }
- altri_vincoli := CHECK(asserzione)

Quindi, possiamo definire una colonna specificandone in ordine il nome, il tipo, eventualmente il valore di default, e poi i suoi vincoli.
I vincoli su una colonna sono definibili tramite la parola chiave **CONSTRAINT** ed il suo nome, e poi se quell'attributo è NULL, piuttosto che NOT NULL, se ogni valore deve essere distinto (UNIQUE) oppure se è chiave primaria (PRIMARY KEY). Si possono poi definire eventuali chiavi esterne ad una colonna di una tabella e infine una eventuale condizione booleana (espressa dal CHECK).
Dopo i vincoli sulle singole colonne si possono definire quelli su più colonne. (attenzione l'ultimo dei vincoli da una colonna non è separato da virgola!)

Esempi



1.2) ALTER TABLE

ALTER TABLE <*Nome tabella*> (ADD **definizione_colonna** | DROP nome-colonna)

- definizione_colonna := <*Nome colonna*> tipo-dato [DEFAULT <*Espressione*>] {**vincoli_su_colonna**}
- vincoli_su_colonna :=
[CONSTRAINT <*Nome vincolo*>] { [NOT] NULL | UNIQUE | PRIMARY KEY | **def_key_esterna** | **altri_vincoli** }
- def_key_esterna := REFERENCES <*Nome tabella*> (<*Nome colonna*>)

Esempio

ALTER TABLE S ADD Discount SMALLINT;	Aggiunge un attributo DISCOUNT di tipo SMALLINT a S.
ALTER TABLE S DROP Discount;	Toglie l'attributo DISCOUNT da S.

1.3) DROP TABLE

DROP TABLE <*Nome tabella*>
La cancellazione di una tabella comporta anche l'eliminazione dell'indice.

Esempio
DROP TABLE S;

2) Operazioni su indici

Un'indice è una struttura dati definita a livello logico-fisico che rende più performante la ricerca su un certo attributo.

2.1) CREATE INDEX

CREATE [UNIQUE] INDEX <*Nome indice*> ON <*Nome tabella*> (nome-colonna {, nome-colonna});

Il nome dell'indice è utile solamente per le operazioni di drop. La clausola UNIQUE verifica che l'attributo su cui si applica l'indice sia UNIQUE. Se non lo è, l'operazione fallisce.

Esempi
CREATE UNIQUE INDEX XS ON S (S#);
CREATE UNIQUE INDEX XSP ON SP(S#, P#);

2.2) DROP INDEX

DROP INDEX <*Nome indice*>

3) La SELECT

L'operazione di SELECT è composta di tre parti.

SELECT <*attributo*$_1$, *attributo*$_2$, ... *attributo*$_n$> | *
FROM <*Nome Tabella*>
WHERE <*Condizione*>;

Abbiamo in sostanza una selezione ed una proiezione, dove la proiezione si fa sugli n attributi espressi nella prima riga (con * si indicano tutti gli attributi) della tabella *Nome Tabella* rispetto alla condizione espressa dal WHERE.
Si noti che non vengono cancellati i duplicati. Nella clausola WHERE è possibile inserire più condizioni usando gli operatori **AND** e **OR**.
È possibile anche fare una SELECT con gli attributi completamente specificati ad esempio:

```
SELECT S.SNAME, S.STATUS
FROM S;
```

SNAME	STATUS
Smith	40
Jones	10
Blake	30
Clark	40
Adams	30

3.1) SELECT con DISTINCT

SELECT DISTINCT <*attributo*$_1$, *attributo*$_2$, ..., *attributo*$_n$> | *
FROM <*Nome Tabella*>
WHERE <*Condizione*>;
Come sopra ma elimina i duplicati.

3.2) SELECT con espressioni

```
SELECT P#, 'Peso in grammi:', WEIGHT*478
FROM P;
```

Come la SELECT normale ma aggiunge una colonna tra P# e WEIGHT nel risultato che ripete il testo 'Peso in grammi:'.

3.3) SELECT con ordinamento

È possibile effettuare la SELECT con ordinamento aggiungendo una riga:
ORDER BY <*Attributo*> **(ASC|DESC)** dove con ASC si effettua un ordine ascendente e con DESC un ordine decrescente.

È inoltre possibile esplicitare l'attributo su cui basarsi per l'ordinamento dando il numero d'ordine dell'argomento stesso, ovvero:

```
SELECT P#, 'Peso in grammi:', WEIGHT*478
FROM P
ORDER BY 3,P# ASC;
```

Ordinerà basandosi prima sul terzo argomento della SELECT (WEIGHT) e poi in caso di pesi uguali si baserà sulla P# (infatti ci sono due argomenti nell'ORDER BY).

4) Il JOIN

Il join permette la connessione fra più tabelle fornendo quindi una ricerca più estesa.

4.1) Equi-join semplice (INNER)

L'equi-join semplice è esprimibile come SELECT con più tabelle nella clausola FROM.

Trovare tutte le informazioni di fornitori e parti in modo che entrambi appartengano alla stessa città:

```
SELECT P.*, S.*
FROM P, S
WHERE S.CITY = P.CITY;
```

Questa operazione è sintatticamente scrivibile come (INNER è facoltativo, chiarifica la lettura):

```
SELECT P.*, S.*
FROM P [INNER] JOIN P ON S.CITY = P.CITY;
```

4.2) OUTER JOIN

In modo analogo si scrive l'OUTER JOIN nelle sue tre versioni (LEFT, RIGHT, FULL) (OUTER facoltativo, chiarifica la lettura):

```
SELECT P.*, S.*
FROM P LEFT [OUTER] JOIN P ON <Condizione>;
SELECT P.*, S.*
FROM P RIGHT [OUTER] JOIN P ON <Condizione>;
SELECT P.*, S.*
FROM P FULL [OUTER] JOIN P ON <Condizione>;
```

4.3) NATURAL JOIN

Il natural-join non esiste come costrutto quindi dobbiamo specificare noi nella SELECT i campi (escludendo quindi la metà di quelli coinvolti nella condizione)

```
SELECT S#, SNAME, STATUS, S.CITY, P#, PNAME, COLOR, WEIGHT
FROM S,P
WHERE S.CITY = P.CITY;
```

In questo caso si è escluso P.CITY che è uguale a S.CITY per via del join.

4.3) TETHA JOIN

Il Θ - join si effettua allo stesso modo degli altri join, ma con una condizione a piacere (Θ, appunto).

Ricava tutte le combinazioni di fornitori e prodotti tali che la città del fornitore sia diversa dalla città del prodotto:

```
SELECT P.*, S.*
FROM P, S
WHERE S.CITY <> P.CITY;
```

4.4) JOIN di una tabella con se stessa

È possibile effettuare il join di una tabella con se stessa tramite la scrittura:

SELECT nome1.Attributo, nome2.Attributo
FROM S <*nome1*> , S <*nome2*>
WHERE nome1.chiave < nome2.chiave

Nella clausola FROM si effettua una ridenominazione, in tutta la query la singola tabella S verrà sdoppiata con i nomi *nome1* e *nome2* i quali potranno essere usati come al solito. Interessante la clausola WHERE dove abbiamo imposto la maggioranza di una delle due tabelle per la chiave: questo accade per evitare banali accoppiamenti di una tupla con se stessa (si mette il < o > invece del <> per evitare sia che nome1.PK venga accoppiata con se stessa, ma anche che nome2.PK venga accoppiata dall'altra parte).

Trovare le coppie di codici di prodotti tali che entrambi i prodotti presenti nella coppia siano forniti dallo stesso fornitore:

```
SELECT FIRST.S#, SECOND.S#
FROM S FIRST, S SECOND
WHERE FIRST.CITY = SECOND.CITY AND FIRST.S# < SECOND.S#;
```

5) Valori NULL

5.1) IS NULL / IS NOT NULL

<*colonna*> **IS** [**NOT**] **NULL**

Non è possibile trattare i valori NULL "così come sono", ma è necessario invece l'utilizzo delle clausole **IS NULL** e **IS NOT NULL**.

Esempio: Ricavare i codici fornitori con stato NULL

```
SELECT S# FROM S
WHERE STATUS IS NULL;
```

6) Le funzioni aggregate

Le funzioni aggregate permettono il calcolo di particolari funzioni altrimenti non implementabili. L'argomento delle funzioni può essere preceduto da DISTINCT. Vengono anche utilizzate con la GROUP BY (capitolo 8).
Non è possibile annidare più funzioni aggregate.

6.1) COUNT(attributo)

Conta gli elementi non nulli in corrispondenza della colonna *attributo*. Per calcolare anche i nulli, bisogna adoperare COUNT(*).

6.2) SUM(attributo)

Somma i valori lungo la colonna *attributo*. Agisce solo su valori numerici.

6.3) AVG(attributo)

Calcola la media dei valori lungo la colonna *attributo*. Agisce solo su valori numerici.

6.4) MAX(attributo) / MIN(attributo)

Restituisce il valore massimo/minimo contenuto nella colonna *attributo*.

Esempio: Calcola la quantità di forniture del prodotto P2:

```
SELECT COUNT(*) FROM SP
WHERE P# = 'P2';
```

7) La ricerca testuale

7.1) LIKE

<nome colonna> ***LIKE*** *<stringa>*

È possibile usare il costrutto **LIKE** nel modo sopracitato per selezionare quelle tuple che contengono la stringa espressa come secondo parametro. Vi sono alcuni caratteri speciali:
- Il carattere _ sta per qualsiasi carattere (ma uno solo!)
- Il carattere % sta per qualsiasi sequenza di caratteri, anche nulla

Tramite questi due caratteri speciali possiamo creare una stringa adatta alla query che vogliamo andare a fare.
Esiste anche la versione **NOT LIKE**.

Esempio: S# di almeno tre caratteri il cui primo 'S'

```
SELECT S# FROM S
WHERE S# LIKE 'S_ _%';
```

Se volessimo utilizzare il carattere % nella ricerca, dovremmo introdurre la clausola **ESCAPE**. Con ESCAPE *<char>* possiamo definire un carattere di escape che se anteposto ad un carattere speciale lo rende non speciale, cioè deve essere letto normalmente.

Esempio: ADDRESS contiene la sottostringa '%Torino'

```
SELECT ADDRESS FROM S
WHERE ADDRESS LIKE '%$%Berkley%' ESCAPE '$';
```

Il carattere '%' dopo il carattere di escape $ verrà letta come '%' e non come carattere speciale.

8) La GROUP BY

8.1) GROUP BY

SELECT .... FROM ...
GROUP BY *<attributo>{,<attributo>}*

L'operazione di GROUP BY è usata in coincidenza con le funzioni aggregate per raggruppare il risultato basandosi su alcuni attributi. Si noti che non è possibile esprimere nella SELECT attributi che non siano coinvolti nella GROUP BY. Questo poiché la GROUP BY in sostanza modifica la "granularità" della relazione, facendo perdere tutti gli attributi non coinvolti.

Esempio: Calcolare la quantità totale fornita *per ogni* prodotto

```
SELECT P#, SUM(QTY)
FROM SP
GROUP BY P#;
```

Ovvero, raccogliamo per P# le varie SUM(QTY) (*quelle con P# uguali vengono sommate*), dopodiché stampiamo sia P# che SUM(QTY).

S#	P#	QTY
S1	P1	200
S1	P1	700
S2	P3	400
S2	P3	200
S2	P3	200
S2	P3	500
S2	P3	600
S2	P3	400

S2	P3	800
S2	P5	100
S3	P3	200
S3	P4	500
S4	P6	300
S4	P6	300
S5	P2	200
S5	P2	100

S5	P5	500
S5	P5	100
S5	P6	200
S5	P1	100
S5	P3	200
S5	P4	800
S5	P5	400
S5	P6	500

Risultato ---->

P#	QTY
P1	900
P2	300
P3	3.300
P4	1.800
P5	1.100
P6	700

8.2) HAVING

SELECT .... FROM ...
GROUP BY ...
HAVING *<condizione>*

Intuitivamente la HAVING è per il GROUP BY ciò che la WHERE è per la SELECT, ovvero è possibile esprimere una ulteriore condizione per filtrare il risultato.

Esempio: Ricavare i codici delle parti fornite da più di un fornitore

```
SELECT P# FROM SP
GROUP BY P#
HAVING COUNT(*) > 1
```

9) Interrogazioni stratificate

Le interrogazioni stratificate sono più interrogazioni che, in un sistema "a cipolla", usano come operando il risultato di un'altra operazione.

9.1) IN / NOT IN

SELECT .... FROM ...
WHERE <*attributo*> **IN**
(SELECT <*attributo*>... FROM ... WHERE ...)

Tramite IN si può selezionare dal risultato di una certa interrogazione qualcos'altro. È traducibile in un join.

Esempio: Trovare il nome dei fornitori che forniscono il prodotto P2

```
SELECT SNAME FROM S
WHERE S# IN
        (SELECT S# FROM SP WHERE P# = 'P2');
```

Ovvero, una volta ottenuti tutti i codici fornitori (S#) che forniscono P2 dalla seconda SELECT, la diamo in pasto alla prima che ci restituisce il SNAME. Sarebbe equivalente a:
SELECT S.NAME **FROM** S,SP **WHERE** SP.S# = S.S# **AND** SP.P# = 'P2'

Esempio2: Trovare il nome dei fornitori che forniscono almeno un prodotto rosso

```
SELECT SNAME FROM S
WHERE S# IN
        (SELECT S# FROM SP WHERE P# IN (
                SELECT P# FROM P WHERE COLOR = 'Red'));
```

L'utilità dell'IN si vede nella sua versione negativa **NOT IN** che invece non è traducibile in un join, infatti non esiste un join che restituisca i "non accoppiamenti".

Esempio: Trovare il nome dei fornitori che *non* forniscono il prodotto P2

```
SELECT SNAME FROM S
WHERE S# NOT IN
        (SELECT S# FROM SP WHERE P# = 'P2');
```

9.2) SELECT annidate a risultato unico (op)

SELECT .... FROM ...
WHERE <*attributo*> **op**
(SELECT <*attributo*>... FROM ... WHERE ...)

Se si sa a priori che il risultato della SELECT più profonda sarà unico allora si può adoperare un operatore {<, =, >, >=, >=} che confronti il risultato della SELECT più interna con le condizioni all'esterno.

Esempio: Trovare il codice dei fornitori che operano nella stessa città di S1

```
SELECT S# FROM S
WHERE CITY =
        (SELECT CITY FROM S WHERE S# = 'S1');
```

Esempio2: Ricavare i codici dei fornitori il cui stato è minore del valore massimo attualmente presente nella tabella

```
SELECT S# FROM S
WHERE STATUS <
        (SELECT MAX(STATUS) FROM S);
```

Se si vuole invece considerare un insieme di risultati è necessario l'operatore op ANY.

9.3) op ANY / op ALL

SELECT .... FROM ...
WHERE <*attributo*> **op ANY**
(SELECT <*attributo*>... FROM ... WHERE ...)

Si vuole selezionare una tupla ricavata dalla seconda SELECT solo se almeno una tupla della seconda SELECT rispetta la condizione espressa dall'operatore.

Esempio: Ricavare i codici dei fornitori il cui stato è minore del valore massimo attualmente presente nella tabella

```
SELECT S# FROM S
WHERE STATUS < ANY
        (SELECT STATUS FROM S);
```

Il risultato darebbe S1, S3, S5 poiché S2 ed S4 hanno valore massimo, quindi non hanno neanche una tupla dove lo status sia minore del massimo.

La versione **op ALL** vuole che la condizione sia rispettata da tutte le tuple.

10) Interrogazioni incrociate

Le interrogazioni incrociate utilizzano nelle SELECT più interne valori ottenuti dalle SELECT più esterne.

10.1) EXISTS / NOT EXISTS

SELECT ... FROM ...
WHERE **EXISTS** (SELECT ... FROM ... WHERE...)

Il risultato dell'EXISTS (SELECT ... FROM ...) è vero solo se il risultato della SELECT che la segue è diverso da vuoto. Se il risultato dell'EXISTS è vero, allora a quell'insieme viene applicata normalmente la SELECT esterna.

Esempio: Ricavare i nomi dei fornitori che forniscono il prodotto 'P2'

```
SELECT SNAME FROM S
WHERE EXISTS
        (SELECT * FROM SP
        WHERE SP.S# = S.S# AND SP.P# = 'P2');
```

Si noti che il riferimento a S.S# nella SELECT interna è alla relazione S, che invece è trattata dalla SELECT esterna. Concettualmente si ha "selezionare i nomi dei fornitori che forniscono il prodotto p2, purché esistano". Si può parafrasare la richiesta in: "Selezionare i nomi dei fornitori tali che esista una fornitura relativa al prodotto P2".

La versione **NOT EXISTS** è del tutto equivalente ma, ovviamente, con il NOT:

Esempio: Ricavare i nomi dei fornitori che *non* forniscono il prodotto 'P2'

```
SELECT SNAME FROM S
WHERE NOT EXISTS
        (SELECT * FROM SP
        WHERE SP.S# = S.S# AND SP.P# = 'P2');
```

Il calcolo dell'EXISTS può essere pensato come un for, se è vera la condizione, allora il record viene "ritornato" alla SELECT sopra.

10.2) EXISTS e op ALL

Da quanto detto si può intuire l'utilizzo combinato di EXISTS ed op ALL per alcune interrogazioni.

Esempio: Ricavare i nomi dei fornitori che forniscono *tutti* il prodotti

```
SELECT SNAME FROM S
WHERE NOT EXISTS
        (SELECT * FROM P
        WHERE NOT EXISTS
                (SELECT * FROM SP
                WHERE SP.S# = S.S#
                AND SP.P# = P.P#));
```

Preso un fornitore, non deve esistere un prodotto che non abbia una associazione con quel fornitore (ovvero che quel fornitore non fornisca).

Vale la pena di vederne alcuni passi:

S#	SNAME	STATUS	CITY
S1	Smith	40	London
...	...	...	...

P#	PNAME
P1	Nut
...	....

S#	P#	QTY
S1	P1	200
...	...	...

Prendo S1, prendo P1, verifico che ci sia almeno una associazione.

S#	SNAME	STATUS	CITY
S1	Smith	40	London
...	...	...	...

P#	PNAME
P2	Bolt
...	....

Prendo S1, prendo P2, verifico che ci sia almeno una associazione.
Ma non c'è!
Il risultato della NOT EXISTS più interna è vero (perché effettivamente non esiste associazione) e questo fa sì che sia vero che "P2 è un prodotto che non ha associazioni" e che sia ritornato.
Ma la NOT EXISTS superiore allora ha risultato falso poiché c'è ritorno di prima (non è vero che non esiste un prodotto siffatto). Quindi, S1 non viene selezionato.

Esempio2: Trovare tutti i codici dei fornitori che forniscono almeno tutti i prodotti forniti da S2

```
SELECT S# FROM S
WHERE NOT EXISTS
        (SELECT * FROM TEMP
        WHERE NOT EXISTS
                (SELECT * FROM SP
                WHERE SP.S# = S.S#
                AND SP.P# = P.P#));
```

Abbiamo concettualmente suddiviso in passi:

- Selezioniamo tutto ciò che è in fornisce S2
- Salviamo tale risultato nella relazione TEMP
- Selezioniamo tutti i fornitori che producono quei prodotti

11) Operatori insiemistici

Date due relazioni è possibile effettuare alcune operazioni insiemistiche. È necessario che le due relazioni siano definite sullo stesso insieme di attributi

11.1) UNION

A **UNION** B

L'insieme delle tuple x tali che x appartenga ad A oppure a B oppure ad entrambi. Le repliche sono eliminate.

Esempio: Ricavare i codici dei prodotti che o pesano più di 16 o sono forniti dal fornitore S2 o entrambe le cose.

```
SELECT P# FROM P
WHERE WEIGHT > 16
UNION
SELECT P# FROM SP
WHERE S# > 'S2';
```

11.2) INTERSECT

A **INTERSECT** B

L'insieme delle tuple x tali che x appartenga ad A ed appartenga anche a B.

11.3) EXCEPT

A **EXCEPT** B

L'insieme delle tuple di A che non sono presenti in B. In oracle, EXCEPT si chiama **MINUS**.

12) Operatore di aggiornamento

12.1) UPDATE

UPDATE <*Nome Tabella*> **SET** <*colonna*> = <*espressione*> {,<*colonna*> = <*espressione*>} [**WHERE** <*predicato*>]

Tutti i record della tabella <*Nome Tabella*> che rispettano l'eventuale <*predicato*> (se non specificato, tutti i valori) vengono modificati secondo le varie opzioni nel **SET**.

Esempio: Raddoppiare gli stati di tutti i fornitori di Londra

```
UPDATE S
        SET STATUS = STATUS * 2
        WHERE CITY = 'London';
```

Esempio2: Aggiornare a 0 la quantità fornita per tutti fornitori di Londra

```
UPDATE S
        SET QTY = 0
        WHERE 'London' = (SELECT CITY FROM S
                          WHERE S.S# = SP.S#);
```

L'aggiornamento simultaneo di più tabelle è impossibile. Quindi, volendo modificare campi coinvolti da vincolo di integrità referenziale, bisogna eseguire due UPDATE separati stando attenti che nessuno "intervenga" nel mezzo.

13) Operatore di cancellazione

13.1) DELETE FROM

DELETE FROM <*Nome Tabella*> [**WHERE** <*predicato*>]

Cancella dalla tabella <*Nome Tabella*> tutti i record che soddisfano il predicato (tutti se non è espresso il predicato).

Esempio: Cancellare il fornitore S1

```
DELETE FROM S
        WHERE S# = 'S1';
```

Attenzione, si perde l'integrità!

Esempio2: Cancellare le forniture dei fornitori di Londra

```
DELETE FROM SP
        WHERE 'London' = (SELECT CITY FROM S
                          WHERE S.S# = SP.S#);
```

14) Operatore di inserimento

14.1) INSERT INTO

INSERT INTO <*Nome Tabella*> [**(** <*colonna*>, {, <*colonna*>} **)**] **VALUES (***costante {, costante}***)**
oppure
INSERT INTO <*Nome Tabella*> [**(** <*colonna*>, {, <*colonna*>} **)**] SELECT ... FROM ... WHERE

I valori costanti inseriti dopo VALUES saranno assegnati rispettivamente agli attributi specificati nelle colonne elencate prima. Tutti gli attributi non specificati assumeranno valore NULL.

Esempio: Inserisci un nuovo prodotto proveniente da Atene che pesa 24 grammi

```
INSERT INTO P (P#, CITY, WEIGHT)
        VALUES ('P7', 'Athens', 24);
```

L'omissione della lista dei campi equivale a specificare tutti i campi della tabella (nell'ordine di creazione delle colonne)

Esempio2: Inserisci un nuovo prodotto proveniente da Atene che pesa 24 grammi che si chiama Sproket di colore rosa

```
INSERT INTO P
        VALUES ('P7', 'Sproket', 'Pink', 24, 'Athens');
```

È oltremodo possibile **inserire più record** tramite l'ausilio di una tabella TEMP.

Esempio3: Per ogni prodotto trovare il codice e la corrispondente quantità totale di forniture, salvando poi il risultato nel DB.

```
CREATE TABLE TEMP ( P# CHAR(6), TOTQTY INTEGER);
INSERT INTO TEMP (P#, TOTQTY)
        SELECT P#, SUM(QTY) FROM SP GROUP BY P#;
```

Dopo aver usato la TEMP a piacimento è possibile fare DROP TABLE TEMP per eliminarla.

Indice degli argomenti - PARTE II

1) Le viste (view)

CREATE VIEW <*nuovoNomeVista*> [(*nome-colonna* {, *nome-colonna*})]
AS <*subquery*> [**WITH** [**LOCAL** | **CASCATED**] **CHECK OPTION**]

1.1) Cos'è una vista

Una vista è una tabella virtuale, non esiste realmente anche se l'utente la percepisce come tale. Non ha quindi spazio sul supporto magnetico a differenza delle tabelle reali. La definizione di una view è memorizzata nel **catalogo** (si veda dopo).

Esempio: Creare una vista contente il codice, la città e lo stato di tutti i fornitori con stato maggiore di 15.

```
CREATE VIEW GOOD_SUPPLIERS AS
        SELECT S#, STATUS, CITY FROM S
        WHERE STATUS > 15
```

Dopodiché gli utenti potranno usare la tabella GOOD_SUPPLIERS esattamente come una relazione normale. La SELECT che si trova su AS non viene eseguita ma salvata sul catalogo.

1.2) Renaming degli attributi

Se i nomi delle colonne della vista non sono specificati sono utilizzati quelli nella SELECT.
I nomi delle colonne devono essere specificati se:
- Rappresentano il risultato di una funzione interna (SUM, ecc...)
- Rappresentano il risultato di un'espressione o sono costanti
- C'è omonimia fra le colonne (esempio sotto)

Esempio: Definire una vista contenente tutte le coppie di città x,y tali che un fornitore localizzato nella città x fornisca un prodotto immagazzinato nella città y.

```
CREATE VIEW CITY_PAIRS (SCITY, PCITY) AS
        SELECT DISTINCT S.CITY, P.CITY FROM S
        WHERE S.S# = SP.S# AND SP.P# = P.P#
```

1.3) Cancellazione di viste

DROP VIEW <*nome vista*>

Se si cancella una tabella di base, si cancellano automaticamente tutte le viste che vi si riferiscono.

1.4) I vantaggi delle viste

- Livello aggiuntivo di sicurezza (si possono proibire gli accessi a certi parte di tabella)
- Nascono la complessità dei dati
- Semplificano le query per gli utenti nabbi
- Presentano i dati in una maniera differente (usando il renaming ad esempio)
- Isolano le applicazioni (livello esterno) dalla definizione di tabelle di base (livello logico)
- Sono utili per "salvarsi" query complesse

1.5) Il problema dell'aggiornamento dei dati

In Oracle sono aggiornabili le viste per cui ad un'operazione di aggiornamento di una riga della vista corrisponde un'operazione di aggiornamento di una sola tabella di base.

Una vista non è intrinsecamente aggiornabile se rientra in uno o più di questi cinque casi:

1. Definita su attributi di tabelle che **non hanno la propria chiave primaria** mantenuta nella vista
2. Contenente **GROUP BY** e / o **ORDER BY**
3. Contenente **funzioni aggregate**
4. Contenente **DISTINCT**
5. Contenente **operatori insiemistici**

Una nota particolare all'inserimento che è effettuabile solo se tutti i le colonne con vincolo NOT NULL della tabella di base sono comprese nella vista.

Vediamo alcuni esempi con relativi tentativi di modifica.

```
CREATE VIEW S#_CITY
        AS SELECT S#, CITY
                FROM S;
```

La vista è aggiornabile!
(ogni modifica è possibile)

- Aggiornamento possibile poiché possiede S# (la chiave)
- Cancellazione di (S1, London) da S#_CITY possibile, viene cancellato (*S1, Smith, 40, London*)
- Modifica di (S1, London) in (S2, Rome) da S#_CITY possibile, viene modificato (*S1, Smith, 40, London)* in *(S1, Smith, 40, Rome)* all'interno di S
- Inserimento di (S6, Rome) in S#_CITY possibile, viene inserito (*S6, NULL, NULL, Rome*) all'interno di S

```
CREATE VIEW STATUS_CITY
        AS SELECT STATUS, CITY
                FROM S;
```

La vista non è aggiornabile!

- Aggiornamento non possibile poiché non possiede S# (la chiave)
- Cancellazione di (20, London) da STATUS_CITY non possibile, quale cancello?
- Modifica di (20, London) in (20, Rome) da STATUS_CITY non possibile, quale modifico
- Inserimento di (50, Athens) in STATUS_CITY non possibile, non posso inserire (*NULL, NULL, 40, Rome*) all'interno di S (poiché l'attributo S# essendo chiave primaria ha il vincolo di NOT NULL)

```
CREATE VIEW BIG_SUPPLIERS(S#, SNAME, P#, QTY)
        AS SELECT S#, STATUS, P#, QTY
                FROM S, SP
                WHERE  QTY > 100;
```

La vista non è aggiornabile, se io cancellassi (S1, Smith, P1, 200) dovrebbe essere cancellata solo la tupla corrispondente a S1 P1 in SP oppure anche quella del fornitore S1 in S?

1.6) CHECK OPTION

Se la CHECK OPTION è presente si possono aggiornare solo righe appartenenti alla vista e dopo l'aggiornamento le righe devono ancora appartenere alla vista.
Se una vista è definita basandosi su altre viste, allora con:
- LOCAL si verifica la correttezza dell'aggiornamento solo sulla visita aggiornata
- CASCADED (default) si verifica la correttezza su tutte le viste coinvolte

LOCAL e CASCADED non sono presenti in Oracle.

```
CREATE VIEW HEAVY_REDPARTS(P#, PNAME, P#, WT)
        AS SELECT P#, PNAME, WT
                FROM RED_PARTS
                WHERE  WT >= 10
        WITH CHECK OPTION;
```

Si supponga che RED_PARTS sia una vista che contiene tutte le parti rosse RED_PARTS(P#, PNAME, WEIGHT)

In tal caso non si potranno aggiornare le tuple con peso minore di 10.

In Oracle è inoltre possibile la clausola **WITH READ ONLY** che impedisce di apporre modifiche dalla vista alla tabella di base.

2) Il catalogo di sistema

Il catalogo di sistema contiene i descrittori di alcuni elementi rilevanti per il sistema stesso.
- Tabelle
- Viste
- Indici
- Utenti
- Privilegi d'accesso
- Piani delle applicazioni

Esso è strutturato in tabelle come i dati, l'ottimizzatore usa il catalogo per decidere le strategie di accesso.

1.1) Parti fondamentali del catalogo

Oracle ha centinaia di tabelle, tali tabelle mantengono anche informazioni sul livello fisico degli oggetti e sulla statistica dei dati in essi contenuta. L'utente è in grado di ispezionare le tabelle ma non di modificarle. Vi sono tre tipi di oggetti:

- Creati dall'utente (prefisso USER_)
- D'uso dell'admin del DB (prefisso DBA_)
- Usati da tutti (prefisso ALL_)

Le tabelle fondamentali di Oracle sono (è facile comprendere il significato degli attributi):

- ALL_TABLES(TABLE_NAME, OWNER, NUM_ROWS, ...) *//una riga per ogni tabella del DB*
- ALL_TAB_COL(COLUMN_NAME, TABLE_NAME, DATA_TYPE, ...) *//una riga per ogni colonna di ogni tabella del DB*
- ALL_INDEXES(INDEX_NAME, TABLE_NAME, OWNER, ...) *//una riga per ogni indice*
- USER_TABLES(TABLE_NAME, TABLESPACE_NAME, NUM_ROWS, BLOCKS, EMPTY_BLOCKS, AVG_SPACE)
- USER_TAB_COLUMNS(COLUMN_NAME, TABLE_NAME, DATA_TYPE, NULLABLE, ...)
- USER_CONSTRAINTS(OWENER, CONSTRAINT_NAME, CONSTRAINT_TYPE, TABLE_NAME, VALIDATED, ...)
- USER_CONS_COLUMS(OWNER, CONSTRAINT_NAME, TABLE_NAME, COLUMN_NAME, ...) *//descrive le colonne di cui l'utente è proprietario e che sono specificate nella definizione di vincoli USER_CONSTRAINTS*
- ALL_VIEWS(OWNER, VIEW_NAME, TEXT_LENGTH, TEXT) *//una riga per ogni vista, text contiene il testo della query*

1.2) Interrogazione del catalogo

L'utente può, volendo, interrogare il catalogo con istruzioni SQL.

Esempio: Trovare quali tabelle contengono una colonna S#

```
SELECT TABLE_NAME
FROM ALL_TAB_COL
WHERE COLUM_NAME='S#'
```

1.3) La gestione delle dipendenze nelle viste

Avendo la tabella di base T con sopra definita una vista V. Se T viene eliminata la vista V non può essere utilizzata quindi viene marcata come invalida (pur rimanendo nel catalogo). Tale processo è ricorsivo.
Se la tabella T dovesse essere ricostruita, quando la vista V verrà utilizzata allora la sua definizione verrà parsificata e, se coerente con la nuova T, verrà rimarcata come valida.

3) La sicurezza

Si vogliono proteggere i dati da letture non autorizzate e alterazione/distruzione dei dati.
Si hanno due concetti principali:
✦ **Sicurezza**: verifica che gli utenti siano *autorizzati* a eseguire le azioni che vogliono eseguire
✦ **Integrità**: verifica che le operazioni eseguite siano corrette (paragrafo successivo)

In entrambi i casi sono necessari dei **vincoli**, i quali vengono salvati nel catalogo. Il DBMS offre gli strumenti per controllare il diritto degli utenti ad accedere alle risorse. Il sistema:
- Memorizza sotto forma di autorizzazioni le decisioni prese (SQL GRANT e REVOKE sono i comandi SQL per fornire/revocare autorizzazioni)
- Verifica a tempo a compile time e run time l'abilitazione dell'utente ad accedere ai dati
- Deve essere in grado di identificare gli utenti (tramite una password)

I dati possono indicati nei comandi di protezione specificando l'intera tabella piuttosto che definendo la loro posizione utilizzando le coordinate riga/colonna.
In ogni caso non deve essere possibile accedere al DBMS per altre vie, talvolta è utile utilizzare una *audit trail* ovvero un tracciamento degli accessi. Inoltre, la crittografia dei dati ne aumenta la sicurezza.

Vi sono due meccanismi di protezione, le viste e le autorizzazioni.

3.1) Primo meccanismo di protezione: le viste

Il primo dei due meccanismi di protezione sono le viste. Tramite le viste è possibile nascondere insiemi di dati agli utenti che non siano abilitati. Inoltre, creando viste contenenti funzioni aggregate, è possibile fornire unicamente dati statistici riguardanti una certa relazione.

3.2) Secondo meccanismo di protezione: le autorizzazioni

Le autorizzazioni sono singolarmente definite su ogni utente. Possono essere definiti **privilegi** su:
- Tabelle e viste (operazioni sui dati, ecc)
    - sulle tabelle: SELECT, UPDATE, DELETE, INSERT
    - sulle viste: REFERENCES
- Database (creazione di tabelle, eliminazione di tabelle, ecc)
- Sistema (creazione di database, ecc)

L'istruzione di GRANT
GRANT <*privilegio*> [, <*privilegio*>]
ON <*oggetto*> [, <*oggetto*>]
TO <*utente*> [, <*utente*>]
[**WITH GRANT OPTION**]

Tramite l'istruzione di GRANT possiamo fornire un'autorizzazione. Se la clausola **WITH GRANT OPTION** è presente allora quell'utente potrà a sua volta dare ad altri i diritti che gli sono stati assegnati.

Esempio: fornire un permesso di lettura di tutta la tabella e aggiornamento della colonna stipendio della tabella impiegati all'utente pippo

GRANT SELECT, UPDATE(stipendio)
ON impiegati **TO** pippo;

Volendo è possibile usare la clausola **ALL PRIVILEGES** per fornire tutti i privilegi invece di specificarli uno ad uno.
Volendo assegnare un permesso a tutti gli utenti del database è possibile usare **PUBLIC** al posto del nome utente.

Esempio: garantire tutti i privilegi a tutti gli utenti sulla tabella impiegati

```
GRANT ALL PRIVILEGES
ON impiegati
TO PUBLIC;
```

L'istruzione di REVOKE

REVOKE <*privilegio*> [, <*privilegio*>]
ON <*oggetto*> [, <*oggetto*>]
FROM <*utente*> [, <*utente*>]
[**RESTRICT** | **CASCADE**];

Permette di revocare permessi precedentemente forniti con una GRANT.
La clausola **RESTRICT** (default) fa sì che il comando non venga eseguito se comporta la revoca di privilegi in cascata ad altri utenti.
La clausola **CASCADE** fa sì che il privilegio venga revocato in cascata a tutti gli utenti che l'avevano ricevuto da colui al quale è stato revocato.

Esempio: revocare a pippo il privilegio di cancellare dati dalla tabella impiegati

```
REVOKE DELETE
ON impiegati
TO pippo;
```

4) L'integrità

Nei sistemi attuali le verifiche di integrità sono per lo più eseguite da procedure codificate dagli utenti in un linguaggio di programmazione. I vincoli sono espressi in maniera dichiarativa affidando al sistema la verifica della loro consistenza.

Esempio: lo status deve essere sempre positivo per tutti i fornitori **FORALL** SX (SX.STATUS > 0)

4.1) Vincolo di integrità (generico)

Tale vincolo mantiene il database "corretto". Se utente dovesse provare ad eseguire un'operazione che viola tale vincolo può:

- impedire l'operazione
- eseguire un'azione compensativa tale da raggiungere un nuovo stato corretto

Vediamo quali vincoli sono implementati in SQL-92

4.2) 1° tipologia di vincolo di integrità in SQL-92: Vincolo di tabella

Permettono di specificare restrizioni sui dati permessi in ciascuna colonna di una tabella. Abbiamo:

1. <*nome colonna*> not null (non sono ammessi valori nulli nella colonna <*nome colonna*>)
2. <*lista colonne*> unique (valori duplicati non ammessi per la combinazione degli attributi <*lista colonne*>)
3. <*lista colonne*> primary key (<*lista colonne*> è la primary key della tabella)
4. check <*condizione*> (il vincolo è soddisfatto se la condizione è vera per ogni tupla della tabella. Volendo può fare riferimento anche a colonne che non sono della tabella)

I vincoli di tabella sono verificati dopo ogni istruzione SQL che inserisca in una tabella o modifichi colonne soggette ai vincoli. Se il vincolo è violato, l'istruzione SQL subisce rollback.

4.3) 2° tipologia di vincolo di integrità in SQL-92: Politiche di integrità

Tali politiche vengono specificate all'atto della CREATE TABLE insieme al vincolo di integrità referenziale.
La loro utilità è mantenere l'integrità in seguito a modifiche della tabella *slave* (referenziante) a fronte di modifiche operate alla tabella *master* (referenziata), specificando operazioni customizzate.
Anche questi vincoli vengono verificati dopo ogni comando SQL che potrebbe causarne la violazione.
In Oracle esistono azioni custumizzabili in seguito alla DELETE sulla relazione *master* ma non in seguito all'UPDATE.
Per quanto concerne la INSERT (sempre sulla relazione *master*) non sono specificate operazioni, poiché il vincolo non viene comunque violato.

Quindi, vediamo quali sono queste operazioni customizzabili sulla delete.

ON DELETE [**CASCADE** | **SET NULL** | **NO ACTION**]:

* Con la clausola **CASCADE**:
  A fronte della cancellazione di una riga nella tabella master si cancellano in *cascata* anche tutte le righe appartenenti alla tabella slave.
* Con la clausola **SET NULL**:
  A fronte della cancellazione di una riga nella tabella master si mette a NULL la chiave esterna di tutte le righe appartenenti alla tabella slave.
* Con la clausola **NO ACTION**:
  A fronte della cancellazione di una riga nella tabella master la cancellazione stessa non viene permessa.

La clausola NO ACTION esiste anche per l'operazione di aggiornamento, quindi si può scrivere **UPDATE NO ACTION**.

4.4) 3° tipologia di vincolo di integrità in SQL-92: Asserzioni

CREATE ASSERTION <*nome asserzione*>
CHECK (<*condizione*>)
[[**NOT**] DEFERRABLE [{initially deferred | initially immediate}]]

La condizione è un predicato SQL arbitrario.
Se l'asserzione è vera il vincolo è soddisfatto.
La valutazione può essere
- *immediata*: vincolo valutato dopo ogni istruzione SQL che la può violare
- *differita*: vincolo valutato al commit della transazione
-

Esempio: creare un vincolo che specifichi che ogni prodotto può essere fornito da al più 10 fornitori differenti

```
CREATE ASSERTION tooManyS
CHECK (NOT EXISTS
        (SELECT * FROM SP
        GROUP BY P#
        HAVING COUNT(DISTINCT S#) < 10))
DEFERRABLE
INITALLY DEFERRED;
```

5) Il Data Base Administrator

È un utente particolare che possiede tutti i privilegi.
Il DBA ha i seguenti compiti:
- responsabilità sul funzionamento del sistema
- gestione della struttura della base di dati (tabelle, ecc)
- gestione degli accessi e delle autorizzazioni
- controllo delle prestazioni del sistema

6) I triggers

È possibile fare sì che il DBMS sia in grado di reagire automaticamente a certi eventi che accadono al database. Tale reazione è in realtà un programma.

6.1) Gli eventi di Oracle

- Istruzioni di DML (INSERT, UPDATE, DELETE)
- Istruzioni di DDL (CREATE, DROP, ALTER)
- Eventi generici come il verificarsi di errori di sistema, login/out di cerci utenti e applicazioni, startup/shutdown del DB

6.2) Paradigma E-C-A

Un trigger obbedisce al cosiddetto paradigma Evento-Condizione-Azione (se accade E ed è vero C, allora esegui A). In Oracle l'azione è specificata tramite un blocco di istruzioni PL/SQL (Procedural Language) linguaggio specifico di Oracle per implementare procedure residenti sul server in grado di manipolare i dati tramite SQL.

6.3) Definizione di un trigger

CREATE [**OR REPLACE**] **TRIGGER** <*nome trigger*>
{**BEFORE** | **AFTER**}
{**INSERT** | **DELETE** | **UPDATE OF** <*colonne*> } **ON** <*nome tabella*>
[**FOR EACH ROW** [**WHEN** (<*trigger condition*>)]]
<*trigger body*>

Esaminiamolo:

- **BEFORE** | **AFTER**: specifica se l'azione deve essere avvenire prima o dopo l'evento
- **INSERT** | **DELETE** | **UPDATE OF** <*colonne*>: è l'evento
- **FOR EACH ROW**: specifica se l'azione verrà eseguita una volta per evento o tante volte, una per ogni riga coinvolta dall'evento. Se non specificata il trigger è di tipo *statement* ossia la condizione viene controllata una volta per tutte.
- **WHEN** <*trigger condition*>: è la condizione
- <*trigger body*>: è l'azione

Esempio: il trigger seguente controlla che la somma totale delle forniture di ciascun fornitore non sia superiore a 1000. Se sì, viene eseguito un blocco PL/SQL che visualizza un messaggio

```
CREATE TRIGGER insert_update_sp_on_tot_qty
BEFORE INSERT OR UPDATE OF qty ON SP
FOR EACH ROW
WHEN (1000 < SELECT SUM(qty)
        FROM SP WHERE S# = new.S#) //new.S# è la riga inserita o modificata
---- inizia qui l'azione ----
DECLARE totqty number(5); //variabile locale che implementa l'azione
BEGIN
        SELECT SUM(qty) INTO totqty //inserisce in totqty la quantità totale delle forniture di quel fornitore
        FROM SP WHERE S# = :new.S#;
        DBMS_OUTPUT.put_line('Ci sono attualmente forniture del fornitore '||:new.S#||' per un totale di '||totqty);
END;
---- termina qui l'azione ----
```

L'azione non fa altro che reperire la quantità che ha superato il limite e stampare un messaggio informativo.

Come si è visto nell'esempio ci sono alcune variabili predefinite per identificare le righe scatenanti l'evento.
Per i trigger di tipo FOR EACH ROW:

- Se scatenati da UPDATE
  - :old e :new assumono rispettivamente i valori della riga *prima* e *dopo* l'accadere l'evento
- Se scatenati da INSERT
  - :new assume i valori della riga *dopo* l'evento (:old non è definito)
- Se scatenati da DELETE
  - :old asusme i valori della riga *prima* dell'evento (:new non è definito)

6.3) Riassunto delle tipologie dei triggers (specifica dell'INSTEAD OF)

Abbiamo quindi:
- FOR EACH ROW triggers, che verificano la condizione per ogni riga coinvolta nell'evento
- Statement triggers, che verificano la condizione una tantum

Ci sono poi gli INSTEAD OF triggers, di tipo particolare, disponibile in Oracle.
Questo trigger viene attivati all'esecuzione di un'azione di aggiornamento su una vista. È utile nel caso in cui la vista non sia intrinsecamente aggiornabile e si voglia definire la sequenza di operazioni da effettuarsi sulle tabelle di base in risposta ad un'operazione di aggiornamento sulla vista.

CREATE [**OR REPLACE**] **TRIGGER** <*nome trigger*>
{ **INSTEAD OF** }
{**INSERT** | **DELETE** | **UPDATE** } **ON** <*nome vista*>
[**FOR EACH ROW** [**WHEN** (<*trigger condition*>)]]
<*trigger body*>
Attenzione: il trigger body verrà eseguito **al posto** dell'operazione specificata sulla vista.

Esempio: Dopo aver creato la vista (in cui si hanno i fornitori che forniscono più di 100 di un relativo prodotto), l'operazione si delete venga fatta su tutte le forniture di quel fornitore e del fornitore su entrambe le tabelle di base (SP ed S)

```
CREATE VIEW BIG_SUPPLIERS2 (S#, SNAME, P#, QTY) AS
SELECT S#, SNAME, P#, QTY
FROM S, SP
WHERE S.S# = SP.S# AND QTY > 100;
```

Il trigger:

```
CREATE TRIGGER delete_from_big_suppliers
INSTEAD OF DELETE ON BIG_SUPPLIERS
FOR EACH ROW
---- inizia qui l'azione ----
BEGIN
        DELETE FROM SP WHERE S# = :old.S#; //Cancello le forniture
        DELETE FROM S WHERE S# = :old.S#; //Quindi il fornitore
END;
---- termina qui l'azione ----
```

Indice degli argomenti - PARTE III

1) A cosa serve un cursore

Un cursore è uno strumento di programmazione che può essere inserito in un programma (scritto in linguaggi di terza generazione).
Tramite un cursore si possono ottenere i dati del database per poi trattarli sul programma stesso. Un cursore permette quindi di scandire le tuple restituite da una interrogazione sulla base di dati.

Chiaramente il cursore porta con se una struttura dati comoda per poter trattare i dati stessi, oltre che alcune variabili generiche di controllo (ad esempio se l'interrogazione ha dato risultato vuoto, ecc.)

2) Uso del cursore

2.1) Definizione del cursore

DECLARE <*nomeCursore*> [**SCROLL**] **CURSOR FOR** <*SELECT ...*>
[**FOR** <**READ ONLY** |
UPDATE [**OF** <*attributo*> {, <*attributo*> }] >]

Con Scroll si da il permesso di leggere dal risultato della SELECT in qualsiasi posizione/direzione
Con READ ONLY è possibile solo leggere le tuple, con UPDATE invece è anche possibile modificarle (sempre da programma).

2.2) Apertura del cursore ed esecuzione dell'interrogazione

OPEN <*nomeCursore*>

Tramite questo comando il risultato della query associata al cursore viene allocato in memoria ed è disponibile all'uso.

2.3) Accesso alle singole tuple

FETCH [<*posizione*> **FROM**] <*nomeCursore*> **INTO** <*lista_variabili_programma*>

Permette di accedere al risultato della interrogazione, una tupla alla volta, e mette a disposizione il valore dei suoi attributi inserendole nella lista del programma.
Tramite <*posizione*> possiamo indicare a quale tupla accedere, con **next** si può accedere alla successiva rispetto alla corrente (quindi c'è un puntatore che tiene conto di qual'è l'attuale). Invece di next, se SCROLL è attivo, possiamo avere: **prior**, **first**, **last**, **absolute** *numeroIntero*, **relative** *numeroIntero*.

2.4) Update con cursori

UPDATE <*nome tabella*> **SET** <*attributo*> = <*espressione*> {,<*attributo*> = <*espressione*>}
WHERE CURRENT OF <*nomeCursore*>

Ovviamente questo è possibile se il cursore è stato definito come FOR UPDATE. Tale aggiornamento modifica la tupla attualmente puntata da *nomeCursore*.

2.5) Chiusura di un cursore

CLOSE <*nomeCursore*>

Rilascia la memoria occupata dal risultato della interrogazione e chiude il cursore. Il numero di cursori aperti simultaneamente da un'applicazioni ha un limite, quindi è importante chiudere i cursori.

2.6) Uso delle variabili del linguaggio ospite

Declare *valoreStatus* **INTEGER**; *//variabile programma ospite*
Declare CursoreFornitore **cursor for**
SELECT SNAME, CITY
FROM S
WHERE STATUS > :*valoreStatus*;

La variabile valoreStatus è una variabile che viene richiamata precedendone il nome con i due punti.

2.7) Assenza del cursore

Nel caso in cui sabbia la garanzia che la query restituisca un'unica tupla, si può usare dopo la SELECT la condizione INTO <*lista_da_programma*> per inserire direttamente i valori restituiti nelle variabili del programma.
Quindi ad esempio:
Declare CursoreFornitore **cursor for**
SELECT SNAME, CITY INTO :nome, :citta
FROM S
WHERE STATUS > S# = 'S1';

Questa operazione inserirà direttamente i valori di SNAME e CITY rispettivamente in :nome e :citta che sono due variabili del programma esterno.

Esempio by Rosa Meo.

Cursori in Oracle PL/SQL

- Procedura che visualizza i nomi dei fornitori richiesti dal parametro in input della procedura.

```
CREATE OR REPLACE PROCEDURE LeggeFornitori () AS
DECLARE
   nomeFornitore VARCHAR2(20); // variabile host di output cursore
   cittaFornitore VARCHAR2(15); // altra var di output del cursore
   CURSOR elencoFornitori IS
   SELECT SNAME, CITY FROM S ORDER BY CITY, SNAME;
   ..
BEGIN
  OPEN elencoFornitori;
  LOOP
    FETCH elencoFornitori INTO nomeFornitore, cittaFornitore;
    EXIT WHEN elencoFornitori%NOTFOUND;
    DBMS_OUTPUT.put_line('Nome '||nomeFornitore||' citta` '||cittaFornitore);
  END LOOP;
  CLOSE elencoFornitori;
EXCEPTION
   WHEN elencoFornitori%NODATAFOUND … codice PL/SQL in assenza fornitori
   WHEN OTHERS THEN ….codice PL/SQL di gestione degli altri errori
END;
```

Variabile di ritorno sul risultato della precedente istruzione FETCH elencoFornitori

Variabile di ritorno sul risultato della OPEN elencoFornitori

2.8) Cursori in SQL dinamico

Se vogliamo eseguire un cursore sulla base del contenuti di una variabile host (ad esempio restituire il nome dei fornitori di una città basandosi su una città passata come parametro) allora dobbiamo definire un cursore in SQL dinamico. L'istruzione verrà eseguita a run-time e non a compile-time.

Esempio by Rosa Meo.