Oracle/SQL Tutorial[1]

Michael Gertz
Dipartimento Sistemi Informativi e Database,
Informatica, Università della California, Davis
gertz@cs.ucdavis.edu
http://www.db.cs.ucdavis.edu

Questo seminario su Oracle/SQL fornisce una dettagliata introduzione al linguaggio per le interrogazioni SQL e al sistema di gestione di database relazionali Oracle. Ulteriori informazioni su Oracle e SQL possono essere trovate al sito web http://www.db.cs.ucdavis.edu/dbs.

Commenti, correzioni o aggiunte a queste note sono gradite. Ringraziamenti a Christina Chung per i commenti sulla versione precedente.

Pubblicazioni Raccomandate

George Koch and Kevin Loney: ORACLE8 *The Complete Reference* (l'unico volume più comprensivo per Oracle Server, include CD con la versione elettronica del libro), 1299 pagine, McGraw-Hill/Osborne, 1997, pubblicazione in inglese.

Michael Abbey and Michael Corey: ORACLE8: *A Beginner's Guide [A Thorough Introduction for First-Time Users]*, 767 pagine, McGraw-Hill/Osborne, 1997, pubblicazione in inglese.

Steven Feuerstein, Bill Pribyl, Debby Russell: ORACLE *PL/SQL Programming* (2ª edizione), O'Reilly & Associates, 1028 pagine, 1997, pubblicazione in inglese.

C.J. Date and Hugh Darwen: A Guide to the SQL Standard (4ª edizione), Addison-Wesley, 1997, pubblicazione in inglese.

Jim Melton and Alan R. Simon: *Understanding the New SQL: A Complete Guide* (2ª edizione, dicembre 2000), The Morgan Kaufmann Series in Data Management Systems, 2000.

INDICE

1 SQL – Structured Query Language

1.1 Tabelle

Nei sistemi di database relazionali (DBS) I dati sono rappresentati usando *tabelle* (*relazioni*). Una interrogazione (*query*) eseguita verso un DBS da come risultato anch'esso una tabella. Una tabella ha la seguente struttura:

Colonna 1	Colonna 2	…	Colonna n
			← Tupla (o Record)
…	…	…	…

Una tabella è definita in maniera univoca dal suo nome ed è costituita da *righe* (o tuple) che contengono i dati; ogni riga contiene esattamente una *tupla* (o *record*). Una tabella può avere una o più colonne. Una *colonna* è fatta da un nome di colonna e da un tipo di dati, e rappresenta, nella teoria matematica degli insiemi, l'attributo delle tuple. La struttura di una tabella, chiamata anche *schema relazionale* (*relation schema*), è quindi definita dai suoi attributi. Il tipo di informazione da memorizzare in una tabella è definito dal tipo di dati degli attributi al momento della creazione della tabella.

SQL usa il termine *tabella* (*table*), *riga* (*row*), e *colonna* (*column*) per *relazione* (*relation*), *tupla* (*tuple*), e *attributo* (*attribute*) rispettivamente. In questo tutorial useremo entrambe le denominazioni.

Una tabella può avere fino a 254 colonne le quali possono avere differenti o uguali tipi di dati (*data types*) e intervalli di valori (domini). Domini possibili sono dati alfanumerici (stringhe), numeri e formati di data. ORACLE offre i seguenti tipi di dati:

- **Char**(*n*): tipo di dato *carattere* a lunghezza fissa (stringa), lungo *n* caratteri. La dimensione massima per *n* è 255 bytes (2000 in ORACLE8). Ricordare che 1 byte = 1 char. Da notare che una stringa di tipo **char** è sempre riempita a destra con spazi fino alla massima lunghezza di *n* (→ può consumare molta memoria).
  Esempio: **char**(40)

- **varchar2**(*n*): stringa di caratteri a lunghezza variabile. La dimensione massima per *n* è 2000 (4000 in ORACLE8). Solo gli effettivi bytes utilizzati dalla stringa richiedono spazio.
  Esempio: **varchar2**(80)

- **number**(*p*,*s*): tipo di dati numerico per numeri interi e reali; *p* rappresenta la *precisione* (ossia il numero totale delle cifre), mentre *s* rappresenta la *scala* (ossia il numero dei decimali). I massimi valori sono p=38, s=[-84 → +127]. *Esempio:* **number**(8), **number**(5,2). Da notare che, per es., **number**(5,2) non può contenere numeri più grandi di 999,99 senza riportare errori. I tipi di dati derivati da **number** sono **int**[**eger**], **dec**[**imal**], **smallint** and **real**.

- **date**: il tipo di dati *date* è utilizzato per memorizzare data e ora. Il formato di default per la data è DD-MMM-YY. *Esempi*: '13-OCT-94', '07-JAN-98'.

- **long**: tipo di dati *char* fino alla grandezza di 2Gb. E' consentita una sola colonna **long** per tabella.

Nota: in Oracle-SQL non c'è un tipo di dati **booleano**. Può comunque essere simulato usando sia **char**(1) che **number**(1).

Finchè non esistono vincoli che restringono i possibili valori di un attributo, esso può avere il valore speciale *null* (vale a dire *sconosciuto*). Questo valore è differente dal numero 0, ed è diverso anche da una stringa vuota ''.

Ulteriori proprietà delle tabelle sono:

- l'ordine nel quale le tuple appaiono nella tabella non è rilevante (finchè una query non richiede un esplicito ordinamento).

- una tabella non ha tuple duplicate (dipende dalla query, comunque, tuple duplicate possono apparire nel risultato di una query).

Lo schema di un database è l'insieme dei suoi schemi relazionali. L'applicazione di uno schema di database in un ambiente operativo in esecuzione è chiamata *istanza di database* o semplicemente *database*.

1.1.1 Esempio di Database

Nelle seguenti discussioni ed esempi usiamo un database per gestire le informazioni riguardo a impiegati, dipartimenti e scatti di stipendio.

La tabella `EMP` viene utilizzata per memorizzare informazioni riguardo agli impiegati:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800	20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	30
…	…	…	…	…	…	…
7698	BLAKE	MANAGER		01-MAY-81	3850	30
7902	FORD	ANALYST	7566	03-DEC-81	3000	10

Per gli attributi, i seguenti tipi di dati sono definiti:

`EMPNO`:**number**(4), `ENAME`:**varchar2**(30), `JOB`:**char**(10), `MGR`:**number**(4), `HIREDATE`:**date**, `SAL`:**number**(7,2), `DEPTNO`:**number**(2).

Ogni riga (tupla) della tabella è interpretata come segue: un impiegato ha un numero, un nome, un titolo di lavoro e uno stipendio. Inoltre, per ogni impiegato, viene memorizzato anche: il numero del suo manager, la data di assunzione e il numero del dipartimento nel quale lavora.

La tabella `DEPT` memorizza informazioni sui dipartimenti (numero, nome e locazione):

DEPTNO	DNAME	LOC
10	STORE	CHICAGO
20	RESEARCH	DALLAS
30	SALES	NEW YORK
40	MARKETING	BOSTON

Infine, la tabella SALGRADE contiene tutte le informazioni circa gli scatti di stipendio; più precisamente: il minimo e il massimo stipendio per ogni grado.

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

1.2 QUERIES (Interrogazioni) – Parte I

Il linguaggio SQL viene utilizzato per ritrovare le informazioni memorizzate nel database. Restringeremo la nostra attenzione a semplici query SQL e rimanderemo la discussione di query più complesse alla sezione 1.5.

In SQL, una query ha la seguente (semplificata) forma (le parti comprese tra [] sono opzionali):

select [**distinct**] <colonna(e)>
from <tabella>
[**where** <condizione>]
[**order by** <colonna(e) [**asc**|**desc**]>]

1.2.1 Selezione delle colonne

Le colonne che devono essere selezionate da una tabella sono specificate dopo la parola chiave **select**. Questa operazione è chiamata anche *proiezione*. Per esempio, la query:

```
select LOC, DEPTNO from DEPT;
```

elenca solo i numeri e la locazione per ogni tupla della relazione DEPT. Se tutte le colonne devono essere selezionate, il simbolo di asterisco * può essere usato per indicare tutti gli attributi. La query:

```
select * from EMP;
```

elenca tutte le colonne di ogni tupla della tabella EMP. Al posto del nome di un attributo, la clausola **select** può anche contenere espressioni aritmetiche che coinvolgono operatori aritmetici, ecc.

```
select ENAME, DEPTNO, SAL*1.55 from EMP;
```

Per i differenti tipi di dati supportati da Oracle, diversi operatori e funzioni sono resi disponibili:

- Per i numeri: **abs**, **cos**, **sin**, **exp**, **log**, **power**, **mod**, **sqrt**, +, **-**, *****, /, …

- Per le stringhe: **chr**, **concat**(string1, string2), **lower**, **upper**, **replace**(string, search_string, replacement_string), **translate**, **substr**(string, m, n), **length**, **to_date**, …
- Per il tipo *date*: **add_month**, **month_between**, **next_day**, **to_char**, …

L'utilizzo di queste operazioni è descritto in dettaglio nell'help di SQL*Plus (vedi inoltre la Sezione 2).

Consideriamo la query:

```
select DEPTNO from EMP
```

che recupera il numero del dipartimento per ogni tupla. Tipicamente, alcuni numeri appariranno più di una volta nel risultato della query, dal momento che le tuple duplicate nel risultato non vengono automaticamente eliminate. L'inserimento della parola chiave **distinct**, dopo la parola chiave **select**, comunque, forza l'eliminazione dei duplicati dal risultato della query.

E' anche possibile specificare un metodo di ordinamento nel quale le risultanti tuple di una query devono essere visualizzate. A questo scopo viene utilizzata la clausola **order by**, che può avere uno o più attributi fra quelli elencati dopo la parola chiave **select**. L'opzione **desc** specifica un ordine decrescente, mentre **asc** specifica un ordine crescente (condizione di default). Per esempio, la query:

```
select ENAME, DEPTNO, HIREDATE
from EMP
order by DEPTNO, HIREDATE desc;
```

visualizzerà il risultato in ordine crescente per l'attributo DEPTNO. Se due tuple hanno lo stesso valore di attributo per DEPTNO, il criterio di ordinamento diventa decrescente per il valore dell'attributo HIREDATE. Per la query sopra, il risultato sarebbe:

ENAME	DEPTNO	HIREDATE
FORD	10	03-DEC-81
SMITH	20	17-DEC-80
BLAKE	30	01-MAY-81
WARD	30	22-FEB-81
ALLEN	30	20-FEB-81

1.2.2 Selezione di tuple

Fino ad ora abbiamo focalizzato l'attenzione solamente sulla selezione di (alcuni) attributi di tutte le tuple di una tabella. Se uno è interessato ad alcune tuple che soddisfano certe condizioni, può utilizzare la clausola **where**. In una clausola **where** semplici condizioni basate sulla comparazione con operatori può essere combinata attraverso l'uso di operatori logici (**and**, **or**, **not**) per formare condizioni più complesse. Le condizioni possono includere verifiche di appartenenza ad un insieme e anche sotto-query, o query nidificate (Sezione 1.5).

Esempio: elencare il titolo di lavoro e il salario per quegli impiegati i cui manager hanno il numero 7698 o 7566 e che guadagnano più di 1500.

```
select JOB, SAL
```

from EMP
where (MGR=7698 **or** MGR=7566) **and** SAL > 1500;

Per tutti i tipi di dati, gli operatori di comparazione =, != o <>, <, >, <=, => sono consentiti in una condizione di una clausola **where**.

Ulteriori operatori di confronto sono:

- Condizioni di insieme: <colonna> [**not**] **in** (<lista dei valori>)
  Esempio: **select** * **from** DEPT **where** DEPTNO **in** (20,30);
- Valore *null*: <colonna> **is** [**not**] **null**,
  Per esempio, per delle tuple da selezionare che devono (o no) avere un valore definito per questa colonna.
  Esempio: **select** * **from** EMP **where** MGR **is not null**;
  Nota: le operazioni di confronto con un valore *null* non sono definite!
- Condizioni di dominio: <colonna> [**not**] **between** <valore inferiore> **and** <valore superiore>.
  Esempio: **select** EMPNO, ENAME, SAL **from** EMP
  where SAL **between** 1500 **and** 2500;

  select ENAME **from** EMP
  where HIREDATE **between** '02-APR-81' **and** '08-SEP-81'

1.2.3 Operazioni di stringa

Al fine di confrontare gli attributi con una stringa, è obbligatorio racchiudere la stringa tra i simboli di apostrofo, per es., **where** LOCATION='DALLAS'. Un potente operatore di insieme è l'operatore **like**. Insieme a questo operatore, vengono usati due caratteri speciali: il segno percentuale **%** (chiamato anche carattere jolly), e la linea bassa (underline) **_**, chiamata anche marcatore di posizione. Per esempio, se si è interessati a tutte le tuple della tabella DEPT che contiengono due C nel nome del dipartimento, la condizione potrebbe essere **where** DNAME **like** '%C%C%'. Il segno percentuale indica che qualsiasi (sub)stringa è permessa al suo posto, anche una stringa vuota. Al contrario, linea bassa (underline) _ sta ad indicare un esatto carattere. Quindi la condizione **where** DNAME **like** '%C_C%' richiede che esattamente un solo carattere appaia tra le due C. Per le condizioni di ineguaglianza, si interpone l'operatore **not** davanti a **like**.

Ulteriori operazioni di stringa sono:

- **upper**(<stringa>) converte tutte le lettere della stringa in lettere maiuscole. Per es. DNAME=upper(DNAME) (il nome del dipartimento deve consistere di sole lettere maiuscole).
- **lower**(<stringa>) converte tutte le lettere della stringa in lettere minuscole.
- **initcap**(<stringa>) converte la lettera iniziale di ogni parola della stringa in una lettera maiuscola.
- **substr**(<string>, *n* [,*m*]) estrae una sottostringa dalla stringa iniziale, partendo dalla posizione *n* inclusa, per *m* caratteri (o fino alla fine se *m* è omesso).
  Per esempio, **substr**('DATABASE SYSTEMS',10,7) da come risultato la stringa 'SYSTEMS'.

1.2.4 Funzioni di Aggregazione

Le funzioni di aggregazioni sono funzioni statistiche come **count**, **min**, **max**, ecc. Sono utilizzate per calcolare un singolo valore da un insieme di valori di una colonna:

count conteggio righe

Esempio: quante righe sono memorizzate nella relazione `EMP`?
select count(`*`) **from** `EMP;`

Esempio: quanti diversi titoli di lavoro sono memorizzati nella relazione `EMP`?
select count(**distinct** `JOB`) **from** `EMP;`

max massimo valore di una colonna
min minimo valore di una colonna

Esempio: elencare il minimo e il massimo stipendio
select min(`SAL`), **max**(`SAL`) **from** `EMP;`

Esempio: calcolare la differenza tra il minimo e il massimo stipendio
select max(`SAL`) - **min**(`SAL`) **from** `EMP;`

sum calcola la somma dei valori (applicabile solo al tipo **number**)

Esempio:
```
select sum(SAL) from EMP
where DEPTNO=30;
```

avg calcola la media semplice dei valori di una colonna (applicabile solo al tipo **number**)

Nota: **avg**, **min** e **max** ignorano tuple che hanno valori *null* per l'attributo specificato, ma **count** invece le considera.

1.3 DEFINIZIONE DATI in SQL

1.3.1 Creazione di Tabelle

Il comando SQL per creare una tabella vuota ha la seguente sintassi:

create table <tabella> (
<colonna 1> <tipo dato> [**not null**] [**unique**] [<vincolo colonna>],
……..
<colonna n> <tipo dato> [**not null**] [**unique**] [<vincolo colonna>],
[<vincolo(i) tabella>]
);

Per ogni colonna, un nome e un tipo di dati devono essere specificati e il nome della colonna deve essere unico entro la definizione della tabella. Le definizioni delle colonne vengono separate da virgole. Non c'è differenza tra nomi in minuscolo e nomi in maiuscolo. Infatti, l'unica sede dove le maiuscole e le minuscole contano è il confronto tra stringhe. Un vincolo **not null** è specificato direttamente dopo il tipo di dati della colonna e richiede che i valori attribuiti per quella colonna siano diversi da *null*.

La parola chiave **unique** specifica che due tuple non possano avere lo stesso valore di attributo per quella colonna. Finchè la condizione **not null** non è specificata per la stessa colonna, allora il valore *null* è permesso, e due tuple aventi il valore *null* per questa colonna non violano il vincolo (si tratta di un'eccezione).

Esempio: il comando **create table** per la nostra tabella `EMP` ha la forma:

```
create table EMP (
        EMPNO       number(4) not null,
        ENAME       varchar2(30) not null,
        JOB         varchar2(10),
        MGR         number(4),
        HIREDATE    date,
        SAL         number(7,2),
        DEPTNO      number(2)
        );
```

Attenzione: eccetto per le colonne `EMPNO` e `ENAME`, i valori *null* sono ammessi.

1.3.2 Vincoli

La definizione di una tabella può includere le specifiche di vincoli di integrità. Sono forniti due tipi di vincoli di base: i *vincoli di colonna* sono associati con una singola colonna, mentre i *vincoli di tabella* sono tipicamente associati a più di una colonna. Comunque, qualsiasi vincolo di colonna può anche essere formulato come un vincolo di tabella. In questa sezione consideriamo solo vincoli molto semplici. Vincoli più complessi verranno discussi nella Sezione 5.1.

La sintassi di un (semplice) vincolo ha la seguente forma:

[**constraint** <nome>] **primary key** | **unique** | **not null**

Un vincolo può avere un nome. E' consigliabile dare un nome a un vincolo in modo da avere informazioni più indicative quando il vincolo è violato a causa di, per es., un inserimento di una tupla che viola il vincolo. Se non viene specificato il nome del vincolo, ORACLE automaticamente genera un nome del tipo `SYS_C`<numero>, che non è utile nella ricerca di un problema legato alla violazione di questo vincolo.

I due tipi più semplici di vincolo sono già stati discussi: **not null** e **unique**. Probabilmente il vincolo di integrità più importante in un database è il vincolo **primary key** (*chiave primaria*). Un vincolo primary key abilita un'identificazione univoca di ogni tupla in una tabella. Basandosi su una *chiave primaria*, il sistema di database si assicura che non esistano duplicati nella tabella. Per esempio, per la nostra tabella `EMP`, la specifica:

```
create table EMP (
        EMPNO number(4) constraint pk_emp primary key,
        …);
```

definisce l'attributo EMPNO come chiave primaria per la tabella. Ogni valore per l'attributo EMPNO apparirà quindi solo una volta nella tabella EMP. Una tabella, ovviamente, può avere solo una *chiave primaria*. Da notare che a diversità del vincolo **unique**, i valori *null* non sono ammessi.

Esempio:

Vogliamo creare una tabella chiamata PROJECT per memorizzare informazioni su alcuni progetti, vogliamo memorizzare il numero e il nome del progetto, il numero dell'impiegato che svolge la mansione di manager del progetto, il budget, il numero delle persone che lavorano sul progetto, la data di inizio e la data di fine progetto. Inoltre, abbiamo le seguenti condizioni:

- un progetto è identificato dal suo numero di progetto,
- il nome di un progetto deve essere unico,
- il manager e il budget devono essere definiti.

Definizione della tabella:

create table PROJECT (

PNO	**number**(3) **constraint** prj_pk **primary key**,
PNAME	**varchar2**(60) **unique**,
PMGR	**number**(4) **not null**,
PERSONS	**number**(5),
BUDGET	**number**(8,2) **not null**,
PSTART	**date**,
PEND	**date**);

Un vincolo **unique** può includere più di un attributo. In questo caso viene utilizzato l'insieme **unique**(<colonna i>, … , <colonna j>). Se è richiesto, per esempio, che due progetti non abbiano la stessa data di inizio e fine, dobbiamo aggiungere il vincolo di tabella:

constraint no_same_dates **unique**(PEND, PSTART)

Questo vincolo deve essere definito all'interno del comando **create table** dopo la definizione di entrambe le colonne PEND e PSTART. Una chiave primaria che include più di una colonna può essere specificata in modo analogo.

Al posto di un vincolo **not null** può qualche volta essere utile specificare un valore di *default* per quell'attributo, se nessun valore è dato, per es., quando una tupla viene inserita. A questo scopo, utilizziamo la clausola **default**.

Esempio:

se non viene specificata alcuna data nell'inserimento di una tupla nella tabella PROJECT, la data dovrebbe essere impostata al 1° gennaio 1995:

PSTART **date default**('01-JAN-95')

Nota: a differenza dei vincoli di integrità, non è possibile dare un nome per un **default**.

1.3.3 Checklist per la creazione di tabelle

Ciò che segue fornisce una piccola lista di controllo per le problematiche che necessitano più spesso di essere considerate prima della creazione di una tabella:

- Quali sono gli attributi delle tuple che devono essere memorizzati? Quali sono i tipi di dati necessari per questi attributi? Dovrebbe **varchar2** essere usato al posto di **char**?
- Quali colonne costituiscono la chiave primaria?
- Quali colonne (non) permettono valori *null*? Quali colonne (non) permettono duplicati?
- Ci sono dei valori di default per certe colonne che permettono valori *null*?

1.4 MODIFICA DEI DATI in SQL

Dopo che una tabella è stata creata usando il comando **create table**, le tuple devono essere inserite nella tabella, o tuple devono essere cancellate o modificate.

1.4.1 Inserimenti

Il modo più semplice per inserire una tupla in un database è quello di usare il comando **insert**

insert into <tabella> [(colonna i, …, colonna j>]
values (<valore i, …, valore j>);

Per ogni colonna elencata, un corrispondente valore (dello stesso tipo dichiarato) deve essere specificato. Quindi un inserimento non deve necessariamente seguire l'ordine degli attributi specificato nel comando **create table**. Se una colonna viene omessa, gli viene attribuito il valore *null*. Se non viene data nessuna lista di colonne, comunque, per ogni colonna come definito nel comando **create table** un valore deve essere inserito.

Esempi:

insert into PROJECT(PNO, PNAME, PERSONS, BUDGET, PSTART)
values(313, 'DBS', 4, 150000.42, '10-OCT-94');

oppure

insert into PROJECT
values(313, 'DBS', 7411, **null**, 150000.42, '10-OCT-94', **null**);

Se sono già presenti alcuni dati in altre tabelle, questi dati possono essere usati per inserimenti in una nuova tabella. Per questo, scriviamo una query il cui risultato è un insieme di tuple da inserire. Un comando **insert**, in questo caso, ha la forma:

insert into <table> [(<colonna i, …, colonna j>)] <query>

Esempio: supponiamo di dover definire la seguente tabella:

create table OLDEMP (

```
ENO        number(4) not null,
HDATE      date);
```

possiamo ora usare la tabella EMP per inserire le tuple in questa nuova relazione:

```
insert into OLDEMP (ENO, HDATE)
        select EMPNO, HIREDATE from EMP
        where HIREDATE < '31-DEC-60';
```

1.4.2 Aggiornamenti

Per modificare i valori degli attributi di (alcune) tuple in una tabella, usiamo il comando **update**:

update <tabella> **set**
<colonna i> = <espressione i>, …, <colonna j> = <espressione j>
[**where** <condizione>];

Un espressione consiste di una costante (nuovo valore), un operazione aritmetica o di stringa o una query SQL. Da notare che il nuovo valore assegnato alla <colonna i> deve corrispondere al tipo di dato dichiarato.

Un comando **update** senza una clausola **where** da come risultato un cambiamento dei rispettivi attributi di tutte le tuple nella tabella specificata. Tipicamente, comunque, solo una (piccola) porzione della tabella richiede un aggiornamento.

Esempi:

- L'impiegato JONES è trasferito al dipartimento 20 come manager e il suo stipendio è incrementato di 1000:

```
update EMP set
        JOB = 'MANAGER', DEPTNO=20, SAL = SAL + 1000
where ENAME='JONES';
```

- Tutti gli impiegati che lavorano al dipartimento 10 e 30 avranno un aumento dello stipendio del 15%.

```
update EMP set
        SAL = SAL  *  1.15
where DEPTNO in (10,30);
```

Analogamente al commando insert, altre tabelle possono essere usate per ricavare valori che possono essere utilizzati in un aggiornamento. In questo caso avremo una <query> al posto di una <espressione>.

Esempio: tutti gli agenti di vendita (salesman) che lavorano al dipartimento 20 avranno lo stesso stipendio dei manager con lo stipendio più basso tra tutti i manager.

```
update EMP set
        SAL = (select min(SAL) from EMP
        where JOB = 'MANAGER')
where JOB = 'SALESMAN' and DEPTNO = 20;
```

Spiegazione: la query recupera prima di tutto il minimo stipendio di tutti i manager Questo valore viene quindi assegnato a tutti gli agenti di vendita (salesman) che lavorano al dipartimento 20.

E' anche possibile specificare una query che recupera più di un valore (ma sempre una tupla!). In questo caso la clausola **set** assume la forma **set**(<colonna i, …, colonna j>) = <query>. E' importante che l'ordine dei tipi di dati e i valori della riga selezionata corrispondano esattamente alla lista delle colonne nella clausola **set**.

1.4.3 Cancellazioni

Tutte le tuple o solo una parte possono essere cancellate da una tabella usando il comando **delete**.

delete from <tabella> [**where** <condizione>];

Se la clausola where viene omessa, tutte le tuple sono cancellate dalla tabella. Un comando alternativo per cancellare tutte le tuple da una tabella è il comando **truncate table** <tabella>. Comunque, in questo caso, la cancellazione non può essere annullata (vedi Sezione 1.4.4 e successive).

Esempio:

Cancellare tutti i progetti (tuple) che sono sono stati terminati prima della data attuale (sysdate):

```
delete from PROJECT where PEND < sysdate;
```

sysdate è una funzione in SQL che restituisce la data di sistema. Un'altra importante funzione è **user**, che restituisce il nome dell'utente collegato con la sessione corrente di ORACLE.

1.4.4 Commit e Rollback

Una sequenza di modifiche a un database, per es., una sequenza di comandi **insert**, **update** e **delete**, viene chiamata *transazione*. Le modifiche delle tuple vengono temporaneamente memorizzate nel DBMS (Database Management System). Diventano permanenti solo dopo aver eseguito il comando **commit**;.

Fintantoché l'utente non ha eseguito il comando **commit**, è possibile annullare tutte le modifiche fino al precedente **commit**. L'annullamento delle modifiche viene eseguito con il comando **rollback**;.

E' consigliabile completare ogni modifica del database con un comando **commit** (finchè si è certi che le modifiche sono quelle desiderate). Da notare che qualsiasi comando di definizione dei dati come **create table** attiva automaticamente un comando **commit**. Un comando **commit** è inoltre implicitamente eseguito quando l'utente termina una sessione ORACLE.

1.5 QUERIES (Parte II)

Nella Sezione 1.2 abbiamo focalizzato l'attenzione su query che si riferivano esattamente ad una tabella. Inoltre, le condizioni in una clausola where erano ristrette a semplici confronti. Una grande caratteristica dei database relazionali, comunque, è quella di combinare (join) tuple memorizzate in differenti tabelle in modo da visualizzare informazioni più utili e complete in una sola volta. In SQL il comando **select** viene utilizzato per questo tipo di queries che uniscono le relazioni:

select [**distinct**] [<alias a_k>.] <colonna i>, …, [<alias a_l>.]<colonna j>
from <tabella 1> [<alias a_1>], …, <tabella n> [<alias a_n>]
[**where** <condizione>]

La dichiarazione degli alias di tabella nella clausola **from** è necessaria per far riferimento alle colonne che hanno lo stesso nome in tabelle differenti. Per esempio, la colonna `DEPTNO` esiste sia nella tabella `EMP` che nella tabella `DEPT`. Se vogliamo far riferimento a queste colonne nella clausola **where** o **select**, un alias di tabella deve essere specificato subito prima del nome della colonna. In sostituzione del nome della tabella può anche essere usato il nome completo della relazione come `DEPT.DEPTNO`, ma questo può incidere sulla lunghezza della formulazione della query.

1.5.1 Joining delle tabelle

La comparazione nella clausola **where** è usata per combinare righe di tabelle elencate nella clausola **from**.

Esempio: nella tabella `EMP` solo i numeri del dipartimento vengono memorizzati, non il loro nome. Per ogni agente di vendita (salesman), vogliamo recuperare il suo nome e il numero e il nome del dipartimento in cui lavorano:

```
select ENAME, E.DEPTNO, DNAME
    from EMP  E, DEPT  D
    where E.DEPTNO = D.DEPTNO
    and JOB = 'SALESMAN';
```

Spiegazione: `E` e `D` sono alias di tabella per `EMP` e `DEPT` rispettivamente. La compilazione della query avviene nel seguente ordine (senza ottimizzazione):

1. Ogni riga della tabella `EMP` viene combinata con ciascuna riga della tabella `DEPT` (questa operazione è il corrispondente matematico del *prodotto cartesiano*). Se `EMP` contiene *m* righe e `DEPT` contiene *n* righe, avremo come risultato una tabella di *m*n* righe.

2. Da queste righe vengono selezionate solo quelle che hanno lo stesso numero di dipartimento (**where** `E.DEPTNO = D.DEPTNO`).
3. Da questo risultato, alla fine, sono selezionate tutte le righe che soddisfano la condizione `JOB = 'SALESMAN'`.

In questo esempio la condizione di fusione delle due tabelle è basata sull'operatore di uguaglianza "=". Le colonne confrontate da questo operatore sono chiamate *colonne join* e l'operazione di join è detta *equijoin*.

Qualsiasi numero di tabelle può essere combinato in un comando **select**.

Esempio: Per ogni progetto, recuperare il suo nome, il nome del suo manager, e il nome del dipartimento dove lavora il manager.

```
select ENAME, DNAME, PNAME
from EMP  E, DEPT  D, PROJECT  P
where E.EMPNO = P.MGR
      and D.DEPTNO = E.DEPTNO;
```

E' anche possibile combinare una tabella con se stessa:

Esempio: Elencare il nome di tutti gli impiegati insieme al nome del loro manager

```
select E1.ENAME, E2.ENAME
from EMP  E1, EMP  E2
where E1.MGR = E2.  EMPNO
```

Spiegazione: le colonne join sono `MGR` per la tabella `E1` e `EMPNO` per la tabella `E2`. Il confronto equijoin è `E1.MGR = E2.EMPNO`.

1.5.2 Query nidificate (subqueries)

Fino ad ora ci siamo concentrati su semplici condizioni di confronto in una clausola **where**, per es., abbiamo comparato una colonna con una costante o abbiamo comparato due colonne. Come abbiamo già visto per il comando **insert**, le query possono essere usate per l'assegnazione a colonne. Il risultato di una query può anche essere usato in una condizione di una clausola **where**. In tal caso la query viene chiamata *subquery* o e la query completa che la contiene diventa una *query nidificata* (*nested query*).

Una relativa condizione in una clausola **where** può avere una delle seguenti forme:

1. Subquery di insiemi di valori
   <espressione> [**not**] **in** (<subquery>)
   <espressione> <operatore di confronto> [**any**|**all**] (<subquery>)
   Una <espressione> può essere una colonna o un valore calcolato.
2. Controllo sulla (non) esistenza
   [**not**] **exists** (<subquery>)

Le condizioni espresse in una clausola **where** utilizzando subqueries, possono essere arbitrariamente combinate utilizzando gli operatori logici **and** o **or**.

Esempio: Elencare il nome e lo stipendio degli impiegati del dipartimento 20 che stanno guidando un progetto partito prima del 31 dicembre 1990:

```
select ENAME, SAL from EMP
where EMPNO in
            (select PMGR from PROJECT
             where PSTART < '31-DEC-90')
and DEPTNO = 20;
```

Spiegazione: la subquery recupera l'insieme degli impiegati che guidano un progetto che è partito prima del 31 dicembre 1990. Se l'impiegato che lavora al dipartimento 20 è contenuto in questo insieme (operatore **in**), la tupla viene inserita nel risultato finale della query nidificata.

Esempio: elencare tutti gli impiegati che stanno lavorando in un dipartimento dislocato a BOSTON:

```
select * from EMP
where DEPTNO in
      (select DEPTNO from DEPT
       where LOC = 'BOSTON');
```

La subquery recupera solo un valore (il numero del dipartimento dislocato a Boston). Quindi è possibile usare "=" al posto dell'operatore **in**. Dal momento però che il risultato della subquery non è conosciuto in anticipo, ad es., che sia un valore singolo o un insieme, è consigliabile utilizzare l'operatore **in**.

Una subquery può a sua volta utilizzare un'altra subquery nella sua clausola where. Quindi le condizioni possono essere *nidificate* arbitrariamente. Una classe importante di subquery sono quelle che fanno riferimento alla (sub)query che immediatamente la circonda e alle tabelle elencate nella clausola **from**, rispettivamente. Questo tipo di query è chiamata *subquery correlata* (*correlated subquery*).

Esempio: elencare tutti gli impiegati che lavorano nello stesso dipartimento come manager (nota che i componenti racchiusi tra [] sono opzionali):

```
select * from EMP  E1
where DEPTNO in
      (select DEPTNO from EMP [E]
       where [E.]EMPNO = E1.MGR);
```

Spiegazione: la subquery in questo esempio è relativa alla query che la circonda, dal momento che fa riferimento alla colonna `E1.MGR`. Una tupla viene selezionata dalla tabella `EMP` (`E1`) come risultato, se il valore per la colonna `DEPTNO` rientra nell'insieme di valori selezionati dalla subquery. Si può pensare al processo di questa query in questi termini: per ogni tupla nella tabella `E1`, la subquery viene processata individualmente. Se la condizione where `DEPTNO` **in** … risulta vera, questa tupla viene selezionata. Nota che un alias per la tabella `EMP` nella subquery non è necessario dal momento che le colonne non precedute da un alias elencate qui fanno sempre riferimento alla query e alle tabelle più interne.

Le condizioni nella forma <espressione> <operatore di confronto> [**any**|**all**] <subquery> vengono usate per confrontare una certa <espressione> con ogni valore selezionato dalla <subquery>.

- Per la clausola **any**, la condizione risulta vera se esiste almeno una riga selezionata dalla subquery per la quale il confronto è valido. Se la subquery raccoglie un insieme vuoto come risultato, la condizione non è soddisfatta.
- Per la clausola **all**, al contrario, la condizione risulta vera se per tutte le righe selezionate dalla subquery, il confronto è valido. In questo caso la condizione risulta vera se la subquery non raccoglie nessuna riga o valore.

Esempio: Recuperare tutti gli impiegati che lavorano al dipartimento 10 e che guadagnano almeno quanto qualsiasi impiegato che lavora al dipartimento 30.

```
select * from EMP
where SAL >= any
    (select SAL from EMP
     where DEPTNO = 30)
and DEPTNO = 10;
```

Nota: anche in questa subquery non sono necessari alias dato che la colonna fa riferimento alla clausola **from** più interna.

Esempio: elencare tutti gli impiegati che non lavorano al dipartimento 30 e che guadagnano più di tutti gli impiegati che lavorano al dipartimento 30.

```
select * from EMP
where SAL > all
        (select SAL from EMP
         where DEPTNO = 30)
and DEPTNO <> 30;
```

Per all e any, valgono le seguenti corrispondenze:

in - **= any**
not in - **<> all** o **!= all**

Spesso il risultato di una query dipende dall'esistenza (o no) di certe righe in (altre) tabelle. Questi tipi di query vengono formulate utilizzando l'operatore **exists**.

Esempio: elencare tutti i dipartimenti che non hanno impiegati:

```
select * from DEPT
where not exists
        (select * from EMP
         where DEPTNO = DEPT.DEPTNO);
```

Spiegazione: per ogni tupla della tabella `DEPT`, la condizione è verificata dove esiste una tupla nella tabella `EMP` che ha lo stesso numero di dipartimento (`DEPT.DEPTNO`). Nel caso tale tupla non

esistesse, la condizione è soddisfatta per la tupla sotto considerazione ed è selezionata. Se esiste una corrispondente tupla nella tabella `EMP`, la tupla non è selezionata.

1.5.3 Operazioni sugli insiemi

Qualche volta è utile combinare i risultati di due o più query in un singolo risultato. Nella teoria matematica degli insiemi è possibile effettuare operazioni di unione, intersezione e sottrazione sugli insiemi. Un insieme è in questo caso il risultato di una query. SQL supporta tre operatori di insiemi che possono essere espressi come segue:

<query 1> <operatore di insieme> <query 2>

Gli operatori di insieme sono:

- **union** [**all**] restituisce una tabella che consiste di tutte le righe che appaiono nel risultato di <query 1> più tutte le righe che appaiono nel risultato di <query 2>. I duplicati vengono automaticamente scartati a meno che non venga specificata la clausola **all**.
- **intersect** restituisce tutte le righe che appaiono contemporaneamente nei due risultati di <query 1> e <query 2>.
- **minus** restituisce quelle righe che appaiono nel risultato di <query 1> ma non nel risultato di <query 2>.

Esempio: assumiamo di avere una tabella `EMP2` che ha la stessa struttura e colonne della tabella `EMP`:

- tutti i numeri degli impiegati e il nome da entrambe le tabelle:
```
select EMPNO, ENAME from EMP
  union
select EMPNO, ENAME from EMP2;
```
- impiegati che sono elencati in entrambe le tabelle `EMP` e `EMP2`:
```
select * from EMP
  intersect
select * from EMP2;
```
- impiegati che sono elencati solamente nella tabella `EMP`:
```
select * from EMP
  minus
select * from EMP2;
```

Ogni operatore richiede che entrambe le tabelle abbiano lo stesso tipo di dati per le colonne al quale l'operatore viene applicato.

1.5.4 Raggruppamento

Nella Sezione 1.2.4 abbiamo visto come le funzioni di aggregazione possono essere usate per calcolare un singolo valore per una colonna. Spesso le applicazioni richiedono il raggruppamento di righe che hanno certe proprietà e quindi applicano una funzione di aggregazione su una colonna per ogni gruppo separatamente. Per questo, SQL fornisce la clausola **group by** <colonna(e)_gruppo>.

Questa clausola appare dopo la clausola **where** e deve fare riferimento alle colonne della tabella elencate dopo la clausola **from**.

select <colonna(e)>
from <tabella(e)>
where <condizione>
group by <colonna(e)_gruppo>
[**having** <condizione(i)_gruppo>];

Quelle righe recuperate dalla clausola **select** che hanno lo stesso valore(i) per <colonna(e)_gruppo> vengono raggruppate. Le aggregazioni specificate nella clausola **select** vengono quindi applicate ad ogni gruppo separatamente. E' importante notare che solo quelle colonne che appaiono nella clausola <colonna(e)_gruppo> possono essere elencate senza una funzione di aggregazione nella clausola **select**!

Esempio: per ogni dipartimento, vogliamo recuperare lo stipendio minimo e massimo.

```
select DEPTNO, min(SAL), max(SAL)
from EMP
group by DEPTNO;
```

Le righe della tabella `EMP` sono raggruppate in modo che tutte le righe in un gruppo abbia lo stesso numero di dipartimento. Le funzioni di aggregazione vengono quindi applicate a tali gruppi separatamente. Abbiamo quindi il seguente risultato:

DEPTNO	MIN(SAL)	MAX(SAL)
10	1300	5000
20	800	3000
30	950	2850

Le righe che formano un gruppo possono essere ristrette nella clausola **where**. Per esempio, se aggiungiamo la condizione **where** `JOB = 'CLERK'`, solo le rispettive righe costituiranno il gruppo. La query quindi cerca di recuperare il minimo e il massimo stipendio di tutti gli impiegati per ogni dipartimento. Da notare che non è permesso specificare nessun'altra colonna tranne `DEPTNO` senza una funzione di aggregazione nella clausola **select** dato che questa è l'unica colonna elencata nella clausola **group by** (è anche facile vedere che le altre colonne non avrebbero senso).

Una volta che i gruppi sono stati formati, alcuni gruppi possono essere eliminati in base alle loro proprietà, per es., se un gruppo contiene meno di tre righe. Questo tipo di condizione è specificata utilizzando la clausola **having**. Come per una clausola **select**, anche in una clausola **having** solo <colonna(e)_gruppo> e aggregazioni possono essere usati.

Esempio: recuperare lo stipendio minimo e massimo degli impiegati per ogni dipartimento che ha più di tre impiegati:

```
select DEPTNO, min(SAL), max(SAL)
from EMP
where JOB = 'CLERK'
group by DEPTNO
having count(*) > 3;
```

Nota che è anche possibile specificare una subquery in una clausola **having**. Nella query sopra, per esempio, al posto della costante 3, può essere specificata una subquery.

Una query contenente una clausola **group by** viene processata nel seguente modo:

1. si selezionano le righe che soddisfano la condizione della clausola **where**.
2. da queste righe si formano dei gruppi in accordo con la clausola **group by**.
3. si scartano tutti i gruppi che non soddisfano la condizione nella clausola **having**.
4. vengono applicate le funzioni di aggregazione per ciascun gruppo rimasto.
5. si recuperano i valori delle colonne e le aggregazioni elencate nella clausola **select**.

1.5.5 Alcuni commenti sulle tabelle

Accesso alle tabelle di altri utenti

Ammesso che un utente abbia il privilegio di accedere alle tabelle di altri utenti (vedi anche Sezione 3), può far riferimento a queste tabelle nella formulazione delle sue query. Supponiamo che <utente> sia un utente del sistema Oracle e che <tabella> sia una tabella di proprietà di questo utente. Questa tabella può essere consultata da altri (privilegiati) utenti utilizzando il comando:

select * **from** <utente>**.**<tabella>**;**

Nel caso qualcuno faccia riferimento spesso a tabelle di altri utenti, è utile usare un *sinonimo* al posto della sintassi <utente>**.**<tabella>. In Oracle-SQL un sinonimo può essere creato utilizzando il comando:

create synonym <nome> **for** <utente>**.**<tabella>**;**

E' quindi possibile usare semplicemente <nome> in una clausola **from**. I sinonimi possono anche essere anche creati per gli stessi proprietari delle tabelle.

Aggiunta di commenti alle definizioni

Per applicazioni che includono numerose tabelle, è utile aggiungere commenti nella definizione delle tabelle, o di aggiungere commenti sulle colonne. Un commento in una tabella può essere creato usando il comando

comment on table <tabella> **is** '<testo>'**;**

Un commento su una colonna può essere creato usando il commando

comment on column <tabella>**.**<colonna> **is** '<testo>'**;**

Commenti su tabelle e colonne vengono memorizzati nel *Dizionario dei Dati* (*Data Dictionary*). Si può accedere a tali commenti utilizzando le *viste* (*view*) del data dictionary `USER_TAB_COMMENTS` e `USER_COL_COMMENTS` (vedi anche Sezione 3).

Modifica delle definizioni di tabelle e colonne

E' possibile modificare la struttura di una tabella (lo schema relazionale) anche se alcune righe sono già state inserite nella tabella. Una colonna può essere aggiunta usando il comando **alter table**:

alter table <tabella>
modify(<colonna> [<tipo di dati>] [**default** <valore>] [<vincolo di colonna>]);

Nota: in Oracle non è possibile rinominare singole colonne da una definizione di tabella. Un trucco per ovviare al problema è quello di creare una tabella temporanea e copiare la colonna da tenere, nella nuova tabella. Poi si cancella la colonna originale e si ricrea con il nuovo nome, copiando i dati dalla tabella temporanea (operazione comunque costosa in termini di elaborazione).

Cancellazione di tabelle

Una tabella e tutte le sue righe possono essere cancellate in una sola volta eseguendo il comando

drop table <tabella> [**cascade constraints**] ;

1.6 VISTE (Views)

In Oracle il comando SQL per creare una *view* (*tabella virtuale*) ha la forma :

create [**or replace**] **view** <nome_vista> [(<colonna(e)>)] **as**
<comando **select**> [**with check options** [**constraint** <nome>]];

La clausola opzionale **or replace** ri-crea la vista se questa già esiste. <colonna(e)> da un nome alle colonne della vista. Se la parte <colonna(e)> viene omessa, allora le colonne della vista assumeranno lo stesso nome degli attributi elencati nel comando **select** (se possibile).

Esempio: la seguente vista contiene il nome, il titolo di lavoro e lo stipendio annuale degli impiegati che lavorano al dipartimento 20:

```
create view DEPT20 as
        select ENAME, JOB, SAL*12 ANNUAL_SALARY from EMP
        where DEPTNO = 20;
```

Nel comando **select** l'alias di colonna `ANNUAL_SALARY` è specificato per l'espressione `SAL*12` e quindi viene considerato nella view come nome di colonna al posto dell'espressione. Una formulazione alternativa della view sopra può essere:

```
create view DEPT20 (ENAME, JOB, ANNUAL_SALARY) as
        select ENAME, JOB, SAL*12 from EMP
        where DEPTNO = 20;
```

Una vista può essere usata nello stesso modo di una tabella, vale a dire che le righe possono essere recuperate da una view come se questa si trattasse di una tabella vera e propria (anche se le righe non sono fisicamente memorizzate nel database, ma vengono recuperate nell'istante in cui viene richiamata la vista, derivando i dati dalle tabelle di origine), oppure le righe possono anche essere modificate. Una vista viene eseguita ogni volta che qualcuno vi accede. In Oracle-SQL nessuna

modifica **insert**, **update** o **delete** su vista è permessa quando uno dei seguenti costrutti viene utilizzato nella definizione della vista:

- Fusioni (joins)
- Funzioni di aggregazione come **sum**, **min**, **max**, ecc.
- Subquery basate su insiemi (**in**, **any**, **all**) o su controllo di esistenza (**exists**)
- Clausola **group by** o **distinct**

In combinazione con la clausola **with check option** qualsiasi aggiornamento o inserimento di una riga nella vista è rifiutata se la modifica non soddisfa la definizione della vista, nell' es., queste righe non potrebbero essere selezionate se basate sul comando **select**. Ad un **with check option** può essere dato un nome usando la clausola **constraint**.

Una vista può essere cancellata usando il comando **delete** <nome_vista>.

2 SQL*Plus

Introduzione

SQL*Plus è l'interfaccia utente (di basso livello) per il sistema di gestione dei database Oracle. Tipicamente, SQL*Plus è usato per formulare query e ottenere il risultato a video. Alcune caratteristiche di SQL*Plus sono:

- Un editor di linea integrato, che può essere utilizzato per correggere query sbagliate. E' possibile anche utilizzare qualsiasi altro editor installato nel computer al posto di questo editor di linea, ed è possibile richiamarlo direttamente dall'interno di SQL*Plus.
- Ci sono numerosi comandi per formattare l'output di una query.
- SQL*Plus fornisce un help in linea.
- I risultati delle query possono essere memorizzati in files che possono essere stampati o importati in altre applicazioni come Excel.

Le query che sono formulate frequentemente possono essere salvate in un file e richiamate successivamente. Le query possono essere parametrizzate ed è possibile richiamare una query salvata con un parametro.

Una Guida Utente minima

SQL*Plus può essere richiamato dall'apposita icona presente nella cartella Application Development di ogni computer in cui è presente l'installazione client, oppure richiamando l'eseguibile **sqlplus** da una sessione Dos. Il programma richiamato è lo stesso, ma le interfacce con cui si interagisce sono diverse: la prima è grafica, la seconda è a caratteri. L'utente e password richiesti sono relativi al sistema Oracle. La schermata che si ottiene sarà simile alla seguente:

```
SQL*Plus: Release 8.1.7.0.0 - Production on Ven Ago 30 17:02:57 2002
(c) Copyright 2000 Oracle Corporation.  All rights reserved.


Connesso a:

Oracle8i Enterprise Edition Release 8.1.7.0.0 - Production
With the Partitioning option
JServer Release 8.1.7.0.0 - Production

SQL>_
------------------------------------------------------------------
```

SQL> è il prompt che si presenta quando ci si connette al sistema di database Oracle. In SQL*Plus si può dividere un comando in separate linee, ciascuna delle quali è indicata con un prompt progressivo del tipo 2>, 3>, ecc. Un comando SQL deve sempre terminare con un punto e virgola (;). In mancanza di questo la query non verrà eseguita fino all'immissione del comando **run**. Lettere maiuscole e minuscole sono importanti solo nel confronto tra stringhe. Una query SQL può sempre essere interrotta usando la combinazione di tasti Control+C. Per uscire da SQL*Plus si può digitare **exit** o **quit**.

Editor dei comandi

I comandi SQL eseguiti più di recente vengono memorizzati nel *SQL buffer*, indipendentemente dalla sintassi corretta oppure no. E' possibile modificare il buffer utilizzando i seguenti comandi:

- **l[ist]** elenca tutte le linee memorizzate nel SQL buffer e imposta la linea corrente (contrassegnata da un *) all'ultima linea del buffer.
- **l**<numero> imposta la linea corrente alla linea numero <numero>
- **c[hange]**/<vecchia stringa>/<nuova stringa> rimpiazza la prima occorrenza della <vecchia stringa> con la <nuova stringa> (per la linea corrente)
- **a[ppend]**<stringa> inserisce la <stringa> in coda alla linea corrente
- **del** cancella la linea corrente
- **r[un]** esegue il contenuto del buffer corrente
- **get**<file> legge i dati dal <file> e li inserisce nel buffer
- **save**<file> scrive il contenuto del buffer nel <file>
- **edit** richiama l'editor designato e carica il buffer corrente nell'editor. Dopo essere usciti dall'editor, i comandi SQL modificati vengono memorizzati nel buffer e possono essere eseguiti (comando **r**)

L'editor può essere definito in SQL*Plus sia nell'interfaccia grafica (Modifica → Editor → Definisci Editor) che in quella a caratteri, in quest'ultimo caso tramite il comando **define _editor** = "<nome>", dove nome deve essere l'eseguibile dell'editor, un batch che lo richiama o un comando comunque riconosciuto dal sistema operativo.

SQL*Plus - Help in linea e altri comandi utili

- Per avere l'help in linea in SQL*Plus è sufficiente digitare **help** <command>, o solamente il comando **help** per avere informazioni su come usare il comando help. Per avere una lista dei comandi di SQL*Plus digitare **help index**.

- Per cambiare la password, in Oracle versione 7 viene utilizzato il comando **alter user** <utente> **identified by** <nuova_password>`;` In Oracle 8 viene utilizzato il comando **passw** <utente>, che richiede l'inserimento della vecchia e nuova password.
- Il comando **desc**[**ribe**] <tabella> elenca la struttura di una tabella, cioè la lista di tutte le colonne con i loro tipi di dati, e informazioni riguardo al valore *null* (se è permesso o no).
- Si può richiamare un comando nativo del sistema operativo digitando **host** <comando>. Ad esempio, in una sessione unix dal quale si richiama **sqlplus**, il comando **host ls –la *.sql** elenca tutti i file con estensione sql nella directory corrente.
- Si può avere un log della propria sessione SQL*Plus e di conseguenza delle query e dei risultati usando il comando **spool** <file>. Tutte le informazioni visualizzate su schermo vengono scritte su <file>. Il comando **spool off** disabilita questa funzione.
  Nota: questo comando è molto utilizzato, insieme ad altri parametri di formattazione, per creare un output importabile in un altro programma, come per es. Excel.
- Il comando **copy** può essere utilizzato per copiare un'intera tabella. Per esempio, il comando **copy from** `scott/tiger` **create** `EMPL` **using select** `*` **from** `EMP;` copia la tabella `EMP` dell'utente `scott` (password `tiger`) nella relazione `EMPL`. La relazione `EMPL` è automaticamente creata e la sua struttura viene derivata basandosi sugli attributi elencati nella clausola **select**.
- I comandi SQL salvati in un file <nome>`.sql` possono essere caricati in SQL*Plus ed eseguiti usando il comando **@**<nome>.
- I commenti possono essere inseriti facendo precedere la clausola **rem**[**ark**] (condizione permessa solo tra comandi SQL), oppure **- -** (sempre tra comandi SQL).

Formattazione dell'output

SQL*Plus fornisce numerosi comandi per la formattazione dei risultati di una query e per costruire semplici reports. A questo scopo vengono definite variabili di formattazione, che sono valide solamente durante la sessione SQL*Plus corrente. Queste ultime vengono perse dopo aver terminato la sessione SQL*Plus. E' comunque possibile salvare le impostazioni di queste variabili in un file di nome `login.sql` nella home directory dell'utente connesso. Ogni volta che si richiama SQL*Plus questo file viene automaticamente caricato.

Il comando **column** <nome colonna> <opzione 1> <opzione 2> … viene usato per formattare le colonne del risultato di una query. Le opzioni più frequentemente usate sono:

- **format A**<n> Per dati alfanumerici, questa opzione imposta la lunghezza del <nome colonna> a <n>. Per colonne che hanno tipi di dati **number**, il comando **format** può essere utilizzato per specificare il formato prima e dopo il punto decimale. Per esempio, **format** 99,999.99 specifica che se il valore ha più di tre cifre prima del punto decimale, le cifre verranno separate da una virgola, e solo due cifre vengono visualizzate dopo il punto decimale.
- L'opzione **heading** <testo> rinomina <nome colonna> e gli da una nuova intestazione.
- **null** <testo> viene usato per specificare l'output dei valori *null* (tipicamente, i valori *null* non sono visualizzati).
- **column** <nome colonna> **clear** cancella la definizione del formato per <nome colonna>.

Il comando **set linesize** <numero> può essere usato per impostare la massima lunghezza di una singola linea che può essere visualizzata a schermo. **Set pagesize** <numero> imposta il numero

totale di linee che SQL*Plus visualizza prima di visualizzare nuovamente i nomi di colonna e le intestazioni, rispettivamente, delle righe selezionate.

Diverse altre caratteristiche di formattazione sono disponibili attraverso le variabili di SQL*Plus. Il comando **show all** visualizza tutte le variabili e il loro valore corrente. Per impostare una variabile, digitare **set** <variabile> <valore>. Per esempio, **set timing on** provoca in SQL*Plus la visualizzazione delle statistiche dei tempi per ogni comando SQL eseguito. **Set pause on** [<testo>] fa sì che SQL*Plus attenda la pressione del tasto <Return> dopo che un numero di linee definite da **set pagesize** è stato visualizzato. <testo> è il messaggio che SQL*Plus visualizzerà in basso sullo schermo per indicare che attende un <Return>.

3 ORACLE DATA DICTIONARY

L'Oracle Data Dictionary è uno dei componenti più importanti del DBMS Oracle. Contiene tutte le informazioni relative alla struttura e agli oggetti del database come tabelle, colonne, utenti, file di dati, ecc. I dati memorizzati nel data dictionary sono spesso chiamati *metadata*. Anche se normalmente è dominio degli amministratori di database (DBAs), il data dictionary è una valida sorgente di informazioni per utenti finali e sviluppatori. Il data dictionary è suddiviso in due livelli: il livello interno contiene tutte le tabelle di base che sono usate dai vari componenti del software DBMS e non sono normalmente accessibili agli utenti finali. Il livello esterno fornisce numerose viste su queste tabelle di base per accedere a informazioni relative a oggetti e strutture in diversi livelli di dettaglio.

3.1 Tabelle del Data Dictionary

Un'installazione di un database Oracle include sempre la creazione di tre utenti standard:

- **SYS**: è il proprietario di tutte le tabelle e le viste del data dictionary. Questo utente ha i più alti privilegi per gestire oggetti e strutture di un database Oracle come la creazione di nuovi utenti.
- **SYSTEM**: è il proprietario di tabelle utilizzate da diverse utilità come SQL*Forms, SQL*Reports, ecc. Questo utente ha meno privilegi di **SYS**.
- **PUBLIC**: è un utente "di base" in un database Oracle. Tutti i privilegi assegnati a questo utente vengono automaticamente assegnati a tutti gli utenti conosciuti del database.

Le tabelle e le viste fornite dal data dictionary contengono informazioni relative a:

- Utenti e relativi privilegi,
- Tabelle, colonne di tabelle e indici usati dall'ottimizzatore,
- Statistiche relative a tabelle e indici usate dall'ottimizzatore,
- Privilegi concessi a oggetti del database,
- Strutture di memorizzazione del database.

Il comando SQL:

```
select * from DICT[IONARY];
```

elenca tutte le tabelle e le viste del data dictionary che sono accessibili all'utente. Le informazioni selezionate includono il nome e una breve descrizione di ogni tabella e vista. Prima di formulare questa query, controllate le definizioni di colonna di DICT[IONARY] usando **desc** DICT[IONARY] e impostate i valori appropriati per le colonne usando il comando **format**.

La query:

select * **from** TAB;

recupera i nomi di tutte le tabelle possedute dall'utente che ha formulato il comando. La query:

select * **from** COL;

restituisce tutte le informazioni relative alle colonne delle proprie tabelle.

Ogni query SQL richiede vari accessi interni alle tabelle e viste del data dictionary. Finchè il data dictionary è costituito a sua volta da tabelle, Oracle deve generare numerosi comandi SQL per controllare se il comando SQL formulato dall'utente è corretto e può essere eseguito.

Esempio: la query SQL

select * **from** EMP
where SAL > 2000;

richiede delle verifiche relative a (1) se la tabella EMP esiste, (2) se l'utente ha i privilegi necessari per accedere a questa tabella, (3) se la colonna SAL è definita per questa tabella, ecc.

3.2 Viste del Data Dictionary

Il livello esterno del data dictionary fornisce agli utenti un interfaccia per l'accesso alle informazioni più importanti inerenti all'utente. Questo livello fornisce numerose viste (in Oracle 7 approssimativamente 540) che rappresentano (una porzione di) dati dalle tabelle di base in una forma leggibile e facilmente comprensibile. Queste viste possono essere usate in query SQL come se fossero normali tabelle.

Le viste fornite dal data dictionary sono divise in tre gruppi: USER, ALL, e DBA. Il nome del gruppo costituisce il prefisso per ogni nome di vista. Per alcune viste, sono associati dei sinomini come descritto sotto.

- **USER_**: Le tuple nella vista **USER** contengono informazioni relative agli oggetti posseduti dall'utente che formula la query (utente attuale).

USER_TABLES	tutte le tabelle con i loro nomi, numero di colonne, informazioni di memorizzazione, informazioni statistiche, ecc. (TABS)
USER_CATALOG	tabelle, viste e sinonimi (CAT)
USER_COL_COMMENTS	commenti su colonne
USER_CONSTRAINTS	definizione di vincoli per le tabelle

USER_INDEXES	tutte le informazioni sugli indici creati per le tabelle (IND)
USER_OBJECTS	tutti gli oggetti di database posseduti dall'utente (OBJ)
USER_TAB_COLUMNS	colonne di tabelle e viste possedute dall'utente (COLS)
USER_TAB_COMMENTS	commenti su tabelle e viste
USER_TRIGGERS	trigger definiti dall'utente
USER_USERS	informazioni relative all'utente attuale
USER_VIEWS	viste definite dall'utente

- **ALL_**: Le righe nelle viste **ALL** includono righe delle viste **USER** e tutte le informazioni relative agli oggetti che sono accessibili all'utente attuale. La struttura di queste viste è analoga alla struttura delle viste **USER**.

ALL_CATALOG	proprietario, nome e tipo di tutte le tabelle, viste e sinonimi accessibili
ALL_TABLE	proprietario e nome di tutte le tabelle accessibili
ALL_OBJECTS	proprietario, tipo e nome di tutti gli oggetti di database accessibili
ALL_TRIGGERS	…
ALL_USERS	…
ALL_VIEWS	…

- **DBA_**: Le viste **DBA** racchiudono informazioni relative a tutti gli oggetti del database, indipendentemente dal proprietario. Solo gli utenti con privilegi **DBA** (amministratori di database) possono accedere a queste viste.

DBA_TABLES	tabelle di tutti gli utenti del database
DBA_CATALOG	tabelle, viste e sinonimi definiti nel database
DBA_OBJECTS	oggetti di tutti gli utenti
DBA_DATA_FILES	informazione sui file di dati
DBA_USERS	informazioni su tutti gli utenti conosciuti del database

4 PROGRAMMAZIONE DI APPLICAZIONI

4.1 PL/SQL

4.1.1 Introduzione

Lo sviluppo di applicazioni di database richiede tipicamente costrutti di linguaggio simili a quelli che possono essere ritrovati nella programmazione di linguaggi C, C++ o Pascal. Questi costrutti sono necessari per l'implementazione di complesse strutture di dati e algoritmi. Una delle maggiori restrizioni del linguaggio di database SQL è quella di non poter compiere diversi compiti usando solo gli elementi forniti con questo linguaggio (che è stato pensato espressamente per le interrogazioni di database).

PL/SQL (Procedural Language /SQL) è un'estensione procedurale dell'Oracle-SQL che offre costrutti di linguaggio del tutto simili a quelli dei linguaggi di programmazione per eccellenza.

PL/SQL permette agli utenti e ai designer di sviluppare complesse applicazioni di database che richiedono l'uso di strutture di controllo e elementi procedurali come procedure, funzioni e moduli.

Il costrutto di base in PL/SQL è il *block* (blocco). I blocchi permettono ai programmatori di combinare logicamente i comandi SQL in unità. In un blocco, costanti e variabili possono essere dichiarate, e le variabili possono possono essere utilizzate per memorizzare i risultati di una query. Le istruzioni in un blocco PL/SQL includono istruzioni SQL, strutture di controllo (loop), istruzioni di condizione (if-then-else), manipolazione delle eccezioni (controllo errori), e chiamate ad altri blocchi PL/SQL.

I blocchi PL/SQL che specificano procedure e funzioni possono essere raggruppati in *packages* (pacchetti). Un package è simile a un modulo e ha un'interfaccia e un'implementazione a parte. Oracle offre diversi packages predefiniti, per esempio, routines di input/output, manipolazione di files, pianificazione di jobs, ecc.

Un'altra importante caratteristica di PL/SQL è che offre un meccanismo per processare i risultati delle query in un modo "orientato alle tuple", il che vuol dire, una tupla alla volta. A questo scopo, vengono utilizzati i *cursori*. Un cursore è fondamentalmente un puntatore al risultato di una query ed è impiegato per leggere i valori degli attributi delle tuple selezionate, inserendoli in variabili. Un cursore è tipicamente usato in combinazione con un costrutto loop tale che ogni tupla letta dal cursore può essere processata individualmente.

Riassumendo, i più importanti obiettivi del PL/SQL sono:

- incrementare le possibilità delle espressioni in SQL,
- processare i risultati delle query "orientandosi alle tuple",
- ottimizzare i comandi SQL combinati,
- sviluppare applicazioni modulari per database,
- riutilizzare il codice di programma, e
- ridurre i costi per la manutenzione e la modifica delle applicazioni.

4.1.2 Struttura dei blocchi PL/SQL

PL/SQL è un linguaggio strutturato in blocchi. Ogni blocco costituisce una denominata unità di programma, in più i blocchi possono essere nidificati. I blocchi che costituiscono procedure, funzioni o package devono avere un nome. Un blocco PL/SQL può avere una sezione di dichiarazione opzionale, una parte contenente istruzioni PL/SQL e una parte opzionali per la gestione delle eccezioni (errori). Quindi la struttura di un programma PL/SQL appare come segue (le parentesi [] rachiudono parti opzionali):

[<Block Header>]
[**declare**
    <Costanti>
    <Variabili>
    <Cursori>
    <Eccezioni definite dall'utente>]
begin
    <Istruzioni PL/SQL>
    [**exception**

<gestione eccezioni>]
end;

L'intestazione del blocco specifica se è una procedura, una funzione, o un package. Se non viene specificata nessuna intestazione, il blocco viene detto *anonimo*. Ogni blocco PL/SQL è costituito ancora da istruzioni PL/SQL. Quindi il blocco può essere nidificato come i blocchi nei linguaggi di programmazione classici. Lo scopo delle variabili dichiarate (per es., nella parte di programma nel quale si può fare riferimento alla variabile) è analogo allo scopo delle variabili nei linguaggi di programmazione come C o Pascal.

4.1.3 Dichiarazioni

Costanti, variabili, cursori ed eccezioni usate in un blocco PL/SQL devono essere dichiarati nella sezione di dichiarazione di quel blocco. Variabili e costanti possono essere dichiarate come segue:

<nome variabile> [**costant**] <tipo di dati> [**not null**] [**:** = <espressione>];

Tipi di dati validi sono i tipi di dati SQL e il tipo **boolean**. Il tipo boolean può essere solamente *vero*, *falso* o *null*. La clausola **not null** richiede che la variabile dichiarata debba sempre avere un valore diverso da *null*. <espressione> è usata per inizializzare la variabile. Se non viene specificata nessuna espressione, il valore *null* viene assegnato automaticamente. La clausola **constant** indica che il valore assegnato alla variabile è costante, e non può essere cambiato (quindi la variabile diventa una costante). Esempio:

```
declare
        hire_date       date;                           /* inizializzazione implicita con null */
        job_title       varchar2(80) := 'Salesman';
        emp_found       boolean;                        /* inizializzazione implicita con null */
        salary_incr     constant number(3,2) := 1.5;    /* costante */
        …
begin … end
```

Al posto di uno specifico tipo di dati, si può fare anche riferimento al tipo di dato di una colonna di tabella (*dichiarazione ancorata*). Per esempio, `EMP.Empno%TYPE` si riferisce al tipo di dato della colonna `Empno` nella relazione `EMP`. Al posto di una singola variabile, può essere dichiarato un record che può memorizzare una tupla completa di una determinata tabella (o risultato di query). Per esempio, il tipo di dati `DEPT%ROWTYPE` specifica una record adatto a memorizzare tutti i valori degli attributi di una riga completa dalla tabella `DEPT`. Tali record sono tipicamente usati in combinazione con un cursore. Si può accedere a un campo di un record utilizzando la sintassi <nome record>**.**<nome colonna>, per esempio, `DEPT.Deptno`.

La dichiarazione di un cursore specifica un insieme di tuple (come risultato di una query) in modo che le tuple possano essere processate individualmente, una alla volta, usando l'istruzione **fetch**. La dichiarazione di un cursore ha la sintassi:

cursor <nome cursore> [(<lista parametri>)] **is** <istruzione *select*>**;**

Il nome di un cursore è un identificatore non dichiarato, diverso da qualunque nome di variabile PL/SQL. Un parametro ha la forma <nome parametro> <tipo parametro>. I possibili parametri sono **char**, **varchar2**, **number**, **date** e **boolean** e tutti i corrispondenti sotto-tipi come **integer**. I parametri sono utilizzati per assegnare valori alle variabili che sono date in un'istruzione **select**.

Esempio: Vogliamo recuperare i seguenti valori degli attributi dalla tabella `EMP` in maniera orientata alle tuple: il titolo di lavoro e il nome di quegli impiegati che sono stati assunti dopo una certa data, e che hanno un manager che lavora in un determinato dipartimento.

```
cursor employee_cur (start_date date, dno number) is
        select JOB, ENAME from EMP E where HIREDATE > start_date
        and exists (select * from EMP
                where E.MGR = EMPNO and DEPTNO = dno);
```

Se le tuple selezionate dal cursore devono essere modificate all'interno del blocco PL/SQL, deve essere aggiunta la clausola **for update** [(<colonna(e)>)] alla fine della dichiarazione del cursore. In questo caso le tuple vengono bloccate e gli altri utenti non vi possono accedere finchè non è stato eseguito un comando **commit**. Prima che un cursore dichiarato possa essere utilizzato in istruzioni PL/SQL, il cursore deve essere aperto, e dopo aver processato le tuple selezionate il cursore deve essere chiuso. Discuteremo l'utilizzo dei cursori con maggior dettaglio più sotto.

Le eccezioni vengono usate per processare errori e avvisi che si incontrano durante l'esecuzione controllata delle istruzioni PL/SQL. Alcune eccezioni sono definite internamente, come ZERO_DIVIDE (divisione per 0). Altre eccezioni possono essere specificate dall'utente alla fine di un blocco PL/SQL. Le eccezioni definite dall'utente devono essere dichiarate usando <nome eccezione> **exception**. Discuteremo la gestione delle eccezioni più dettagliatamente nella Sezione 4.1.5.

4.1.4 Elementi di linguaggio

In aggiunta alla dichiarazione delle variabili, costanti e cursori, PL/SQL offre diversi costrutti di linguaggio come assegnazione di variabili, strutture di controllo (loop, if-then-else), chiamate a procedure e funzioni, ecc. Comunque, PL/SQL non consente l'utilizzo di comandi di definizione dati SQL, come l'istruzione **create table**. Per questo, PL/SQL fornisce speciali package. Inoltre, PL/SQL usa un'istruzione **select** modificata che richiede che ogni tupla selezionata sia assegnata a un record (o a una lista di variabili).

Ci sono diverse alternative in PL/SQL per assegnare un valore a una variabile. Il modo più semplice è:

```
declare
        counter integer := 0;
        …
begin
        counter := counter + 1;
```

I valori da assegnare a una variabile possono anche essere recuperati da un database usando un'istruzione **select**

select <colonna(e)> **into** <lista di variabili corrispondenti>
from <tabella(e)> **where** <condizione>;

E' importante assicurarsi che l'istruzione **select** recuperi almeno una tupla! Altrimenti non è possibile assegnare i valori degli attributi alla lista di variabili specificata, che causerà un errore run-time. Se l'istruzione recupera più di una tupla, allora deve essere utilizzato un cursore. Inoltre, il tipo di dati delle variabili specificate deve coincidere con quello dei valori degli attributi recuperati. Per molti tipi di dati, PL/SQL esegue una conversione dei tipi automatica (per esempio, da **integer** a **real**).

Al posto di una singola lista di variabili, può essere dato un record dopo la parola chiave **into**. Anche in questo caso, l'istruzione **select** deve recuperare almeno una tupla!

```
declare
        employee_rec   EMP%ROWTYPE;
        max_sal   EMP.SAL%TYPE;
begin
        select EMPNO, ENAME, JOB, MGR, SAL, COMM, HIREDATE, DEPTNO
        into employee_rec
        from EMP where EMPNO = 5698;
        select max(SAL) into max_sal from EMP;
        …
end;
```

PL/SQL fornisce loop **while**, due tipi di loop **for** e loop continui. Questi ultimi sono utilizzati in combinazione con i cursori. Tutti i tipi di loop sono utilizzati in una sequenza di istruzioni più volte. La specificazione di un loop avviene nello stesso modo dei linguaggi di programmazione come C o Pascal.

Un loop while ha la forma

[<< <nome etichetta> >>]
while <condizione> **loop**
        <sequenza di istruzioni>;
end loop [<nome etichetta>];

Un loop può avere un nome. Dare un nome a un loop è utile quando i loop sono nidificati e i primi vengono completati incondizionatamente utilizzando l'istruzione **exit** <nome etichetta>.

Mentre il numero di iterazioni di un loop **while** è sconosciuto fino al momento in cui il loop viene completato, il numero di iterazioni di un loop **for** può essere specificato usando due interi.

[<< <nome etichetta> >>]
for <indice> **in** [**reverse**] <estremo inferiore>..<estremo superiore> **loop**
        <sequenza di istruzioni>
end loop [<nome etichetta>];

Il contatore del loop <indice> viene dichiarato implicitamente. Lo scopo del contatore del loop è solo per il loop **for**. Esso sovrascrive lo scopo di ogni variabile che ha lo stesso nome al di fuori del loop. All'interno del loop **for**, <indice> può essere utilizzato come riferimento come se fosse una

costante. <indice> può comparire in espressioni, ma non gli possono essere fatte assegnazioni. Utilizzando la parola chiave **reverse** si fa in modo che il processo di iterazione parta dall'estremo superiore fino a quello inferiore nell'insieme specificato.

Elaborazione di cursori: prima che un cursore possa essere usato, deve essere aperto utilizzando l'istruzione **open**

open <nome cursore> [(<lista di parametri>)];

L'istruzione **select** associata viene quindi processata e il cursore punta alla prima tupla selezionata. Le tuple selezionate possono essere processate una alla volta utilizzando il comando **fetch**

fetch <nome cursore> [(<lista di variabili>)];

Il comando **fetch** assegna i valori degli attributi selezionati dalla tupla corrente alla lista di variabili. Dopo un comando **fetch**, il cursore avanza alla successiva tupla nell'insieme del risultato dell'istruzione **select** ottenuto con il comando **open**.
Da notare che le variabili nella lista devono avere lo stesso tipo di dati dei valori delle tuple selezionate. Dopo che tutte le tuple sono state processate, si utilizza il comando **close** per chiudere e disabilitare il cursore.

close <nome cursore>;

L'esempio riportato sotto illustra come un cursore viene utilizzato insieme a un loop continuo:

```
declare
      cursor emp_cur is select * from EMP;
      emp_rec EMP%ROWTYPE;
      emp_sal EMP.SAL%TYPE;
begin
      open emp_cur;
      loop
            fetch emp_cur into emp_rec;
            exit when emp_cur%NOTFOUND;
            emp_sal := emp_rec.sal;
            <sequenza di istruzioni>
      end loop;
      close emp_cur;
      …
end;
```

Ogni loop può essere completato incondizionatamente usando la clausola **exit**:

exit [<etichetta blocco>] [**when** <condizione>]

Usando **exit** senza un'etichetta di blocco si provoca il completamento del loop che contiene l'istruzione **exit**. Una condizione può essere un semplice paragone di valori. In molti casi, comunque, la condizione fa riferimento a un cursore. Nell'esempio sopra, `%NOTFOUND` è un predicato che restituisce *false* se il comando **fetch** più recente ha letto una tupla. Il valore di <nome cursore>`%NOTFOUND` è *null* prima che la prima tupla venga letta. Il predicato restituisce *true* se il

comando **fetch** più recente fallisce nel restituire una tupla, nel caso contrario restituisce *false*. `%FOUND` è l'opposto logico di `%NOTFOUND`.

I loop **for** di cursore vengono utilizzati per semplificare l'utilizzo di un cursore:

[<< <nome etichetta> >>]
for <nome record> **in** <nome cursore> [(lista di parametri>)] **loop**
    <sequenza di istruzioni>
end loop;

Un record utilizzabile per memorizzare una tupla recuperata da un cursore viene implicitamente dichiarato. Inoltre, questo loop implicitamente esegue un **fetch** ad ogni iterazione, un **open** prima dell'ingresso nel loop e un **close** dopo che il loop è terminato. Se ad un'iterazione nessuna tupla viene recuperata, il loop viene automaticamente terminato senza un **exit**.

E' anche possibile specificare una query al posto di <nome cursore> in un loop for:

for <nome record> **in** (<istruzione select>) **loop**
    <sequenza di istruzioni>
end loop;

Dunque, un cursore non deve essere specificato prima dell'ingresso nel loop, ma è definito nell'istruzione **select**.

Esempio:

for `sal_rec` **in** (**select** `SAL + COMM total` **from** `EMP`) **loop**
    …;
end loop;

`total` è un alias per l'espressione calcolata nell'istruzione **select**. Quindi, ad ogni iterazione solo una tupla viene recuperata. Il record `sal_rec`, che viene implicitamente definito, contiene quindi solo un dato al quale si può accedere utilizzando `sal_rec.total`. Gli alias, ovviamente, non sono necessari se solo gli attributi devono essere selezionati, cioè, se l'istruzione **select** non contiene operatori aritmetici o funzioni di aggregazione.

Per il controllo condizionale, PL/SQL offre costrutti **if-then-else** nella forma:

if <condizione> **then** <sequenza di istruzioni>
[**elsif**] <condizione> **then** <sequenza di istruzioni>
…
[**else**] <sequenza di istruzioni> **end if**;

Partendo con la prima condizione, se risulta *true*, la corrispondente sequenza di istruzioni viene eseguita, altrimenti il controllo viene passato alla successiva condizione. Quindi, quello che sta dietro a questo tipo di istruzione PL/SQL è del tutto simile alle istruzioni if-then-else dei linguaggi di programmazione classici.

Ad eccezione dei comandi di definizione dei dati come create table, tutti i tipi di istruzione SQL può essere usata nei blocchi PL/SQL, in particolare dolete, insert, update e commit. Da notare che in PL/SQL solo l'istruzione **select** del tipo **select** <colonna(e)> **into** è permessa, cioè, i valori degli attributi selezionati possono solo essere assegnati a variabili (finchè l'istruzione select viene

utilizzata in una subquery). L'utilizzo dell'istruzione **select** come nell'SQL causa un errore di sintassi (*syntax error*). Se le istruzioni **update** o **delete** vengono utilizzate in concomitanza con un cursore, questi comandi possono essere ristretti alla tupla attualmente recuperata. In questi casi la clausola **where current** <nome cursore> viene aggiunta come mostrato nel seguente esempio:

Esempio: Il seguente blocco PL/SQL esegue le seguente modifiche: tutti gli impiegati che hanno 'KING' come loro manager avranno un aumento di stipendio del 5%.

```
declare
        manager EMP.MGR%TYPE;
        cursor emp_cur (mgr_no number) is
                select SAL from EMP
                where MGR = mgr_no
        for update of SAL;
begin
        select EMPNO into manager from EMP
        where ENAME = 'KING';
        for emp_rec in emp_cur(manager) loop
                update EMP set SAL = emp_rec.sal*1.05
                where current of emp_cur;
        end loop;
        commit;
end;
```

Attenzione: da notare che il record `emp_rec` viene implicitamente definito. Discuteremo un'altra versione di questo blocco usando i parametri nella Sezione 4.1.6.

4.1.5 Gestione Eccezioni

Un blocco PL/SQL può contenere istruzioni che specificano routines di gestione delle eccezioni. Ogni errore o avviso durante l'esecuzione di un blocco PL/SQL causa una eccezione. Si può distinguere tra due tipi di eccezioni:

- Eccezioni definite dal sistema
- Eccezioni definite dall'utente (che deve essere dichiarata dall'utente nella parte di dichiarazione di un blocco dove l'eccezione è utilizzata/implementata)

Il sistema definisce eccezioni che vengono automaticamente attivate quando i corrispondenti errori o avvisi vengono incontrati. Le eccezioni definite dall'utente, contrariamente, devono essere attivate esplicitamente in una sequenza di istruzioni utilizzando **raise** <nome eccezione>. Dopo la parola chiave **exception** alla fine di un blocco, le routines di gestione delle eccezioni definite dall'utente vengono implementate. Un'implementazione ha la forma:

when <nome eccezione> **then** <sequenza di istruzioni>;

I più comuni errori che possono avvenire durante l'esecuzione di programmi PL/SQL sono gestiti dalle eccezioni di sistema. La tabella sotto elenca alcune di queste eccezioni con il loro nome e una breve descrizione.

Nome eccezione	Numero	Descrizione
CURSOR_ALREADY_OPEN	ORA-06511	Si è cercato di aprire un cursore già aperto
INVALID_CURSOR	ORA-01001	Operazione di cursore non valida come un comando **fetch** su un cursore chiuso
NO_DATA_FOUND	ORA-01403	Un comando **select** … **into** o **fetch** non ha restituito nessuna tupla
TOO_MANY_ROWS	ORA-01422	Un comando **select** … **into** ha restituito più di una tupla
ZERO_DIVIDE	ORA-01476	E' stata tentata un'operazione di divisione per 0

Esempio:

```
declare
        emp_sal EMP.SAL%TYPE;
        emp_no EMP.EMPNO%TYPE;
        too_high_sal exception;
begin
        select EMPNO, SAL into emp_no, emp_sal
        from EMP where ENAME = 'KING';
        if emp_sal*1.05 > 4000 then raise too_high_sal
        else update EMP set SQL …
        end if;
        exception
                when NO_DATA_FOUND  - - nessuna tupla selezionata
                        then rollback;
                when too_high_sal then insert high_sal_emps values(emp_no);
                commit;
end;
```

Dopo una parola chiave **when** una lista di nomi di eccezioni (implicitamente connesse con un operatore **or**) può essere specificata. L'ultima clausola **when** nell'eccezione può contenere il nome di eccezione **others**. Questo introduce la routine di gestione delle eccezioni di default, per esempio, un **rollback**.
Se un programma PL/SQL viene eseguito da una shell SQL*Plus, le routine di gestione delle eccezioni possono contenere istruzioni che visualizzano l'errore o l'avviso sullo schermo. A questo scopo, può essere utilizzata la procedura **raise_application_error**. Questa procedura ha due parametri <numero errore> e <messaggio errore>. <numero errore> è un numero intero negativo definito dall'utente e deve essere compreso tra -20000 e -20999. <messaggio errore> è una stringa con una lunghezza fino a 2048 caratteri. L'operatore di concatenazione "||" può essere utilizzato per concatenare singole stringhe in un'unica stringa. Allo scopo di visualizzare variabili numeriche, queste variabili devono essere convertite in stringhe usando la funzione **to_char**. Se la procedura **raise_application_error** viene chiamata da un blocco PL/SQL, l'elaborazione del blocco termina e tutte le modifiche ai database vengono annullate, cioè, viene eseguito un implicito **rollback** in aggiunta alla visualizzazione del messaggio di errore.

Esempio:

```
if emp_sal*1.05 > 4000
then raise_application_error(-20010,'Aumento di stipendio per impiegato id '
||to_char(emp_no)||' è troppo alto');
```

4.1.6 Procedure e Funzioni

PL/SQL fornisce sofisticati costrutti di linguaggio per programmare procedure e funzioni come blocchi PL/SQL separati. Essi possono essere richiamati da altri blocchi PL/SQL, altre procedure e altre funzioni. La sintassi per la definizione di una procedura è

create [**or replace**] **procedure** <nome procedura> [(<lista di parametri>)] **is**
<dichiarazione>
begin
<sequenza di istruzioni>
[**exception**
<routine di gestione dell'eccezione>]
end [<nome procedura>];

Una funzione può essere specificata in modo analogo

create [**or replace**] **function** <nome funzione> [(<lista di parametri>)]
return <tipo di dati> **is**
…

La clausola opzionale **or replace** ri-crea la procedura/funzione. Una procedura può essere cancellata utilizzando il comando **drop procedure** <nome procedura> (**drop function** <nome funzione>).
Contrariamente ai blocchi PL/SQL anonimi, la clausola **declare** non può essere usata nella definizione di procedure/funzioni.

I tipi di dati validi includono tutti i tipi di dati. Comunque, per **char**, **varchar2** e **number** senza lunghezza e precisione/scala, rispettivamente, possono essere specificati. Per esempio, il parametro **number**(6) provoca un errore di compilazione e deve essere rimpiazzato da **number**. Al posto di espliciti tipi di dati, impliciti tipi nella forma `%TYPE` e `%ROWTYPE` possono essere utilizzati anche se viene fatto riferimento a dichiarazioni con vincoli. Un parametro è specificato come segue:

<nome parametro> [**IN** | **OUT** | **IN OUT**] <tipo di dato> [{ := | **DEFAULT**} <espressione>]

La clausola opzionale **IN**, **OUT**, e **IN OUT** specifica il modo nel quale il parametro è utilizzato. Il modo di default di utilizzo di un parametro è **IN**. **IN** significa che al parametro si può fare riferimento all'interno del corpo della procedura, ma non può essere modificato. **OUT** significa che un valore può essere assegnato al parametro nel corpo della procedura, ma non si può fare riferimento al valore del parametro. **IN OUT** permette sia di far riferimento al parametro che l'assegnazione di valori a quest'ultimo. Tipicamente, è sufficiente usare il modo di default dei parametri.

Esempio: la seguente procedura viene utilizzata per incrementare lo stipendio di tutti gli impiegati che lavorano nel dipartimento fornito dal parametro della procedura. La percentuale dell'aumento di stipendio è data anch'essa da un parametro.

```
create procedure raise_salary(dno number, percentage number
DEFAULT 0.5) is
        cursor emp_cur (dept_no number) is
                select SAL from EMP where DEPTNO = dept_no
```

```
            for update of SAL;
        empsal number(8);
    begin
        open emp_cur(dno); - - qui non viene assegnato nessun dno a dept_no
        loop
            fetch emp_cur into empsal;
            exit when emp_cur%NOTFOUND;
            update EMP set SAL = empsal*((100+percentage)/100)
            where current of emp_cur;
        end loop;
        close emp_cur;
        commit;
    end raise_salary;
```

Questa procedura può essere richiamata da una shell SQL*Plus utilizzando il commando

```
    execute raise_salary(10,3);
```

Se la procedura viene chiamata con il solo parametro 10, viene assunto il valore di default 0.5 come specificato nella lista di parametri nella definizione della procedura. Se la procedura viene chiamata dall'interno di un blocco PL/SQL, la parola chiave **execute** viene omessa.

Le funzioni hanno la stessa struttura delle procedure. L'unica differenza è che le funzioni restituiscono un valore il cui tipo di dato (non vincolato) deve essere specificato.

Esempio:

```
create function get_dept_salary(dno number) return number is
    all_sal number;
begin
    all_sal := 0;
    for emp_sal in (select SAL from EMP where DEPTNO= dno
                and SAL is not null) loop
        all_sal := all_sal + emp_sal.sal;
    end loop;
    return all_sal;
end get_dept_salary;
```

Per poter richiamare una funzione da una shell di SQL*Plus, è necessario prima definire una variabile al quale il risultato della funzione deve essere assegnato. In SQL*Plus, una variabile può essere definita utilizzando il comando variable <nome variabile> <tipo di dati>;, per esempio, **variable** `salary` **number**. La funziona sopra può quindi essere chiamata usando il comando **execute** `:salary := get_dept_salary(20);`. Da notare che i due punti devono essere messi davanti alla variabile.

Ulteriori informazioni riguardo procedure e funzioni possono essere ottenute usando il comando help nella shell di SQL*Plus, per esempio **help** [**create**] **function**, **help subprograms**, **help stored subprograms**.

4.1.7 Packages

E' essenziale per un buon stile di programmazione che i blocchi, procedure e funzioni logicamente correlati vengano combinati in moduli, e che ogni modulo fornisca un'interfaccia che permetta agli utenti e sviluppatori di utilizzarne le funzionalità. PL/SQL supporta il concetto di modularizzazione grazie al quale i moduli e altri costrutti possono essere organizzati in *packages*. Un package è costituito da una dichiarazione di package e da un corpo di package. La dichiarazione del package definisce l'interfaccia che è visibile ai programmatori di applicazione, e il corpo del package implementa la dichiarazione del package (simile all'header e source files nel linguaggio di programmazione C).

Sotto viene dato un package in modo che possa essere usato per combinare tutte le procedure e funzioni per gestire le informazioni sugli impiegati.

```
create package manage_employee as  - - definizione del package
        function hire_emp (name varchar2, job varchar2, mgr number, hiredate date,
                           sal number,  comm number default 0, deptno number)
                return number;
        procedure fire_emp (emp_id number);
        procedure raise_sal (emp_id number, sal_incr number);
end manage_employee;

create package body manage_employee as
        function hire_emp (name varchar2, job varchar2, mgr number, hiredate date, sal
                number, comm number default 0, deptno number)
                return number is
        -- inserisce un nuovo impiegato con un nuovo id di impiegato.
        new_empno number(10);
        begin
                select emp_sequence.nextval into new_empno from dual;
                insert into emp values(new_empno, name, job, mgr, hiredate, sal, comm,
                                deptno);
                return new_empno;
        end hire_emp;

        procedure fire_emp(emp_id number) is
        -- cancella un impiegato dalla tabella EMP
        begin
                delete from emp where empno = emp_id;
                if SQL%NOTFOUND then   -- cancella l'istruzione che si riferisce a un emp_id non
                                           valido
                raise_application_error(-20011,'Impiegato con id '||to_char(emp_id)||'
                                        inesistente.');
                end if;
        end fire_emp;

        procedure raise_sal(emp_id number, sal_incr number) is
        -- modifica lo stipendio di un determinato impiegato
        begin
```

```
            update emp set sal = sal + sal_incr
            where empno = emp_id;
            if SQL%NOTFOUND then
            raise_application_error(-20012,'Impiegato con id ',||to_char(emp_id)||'
                                    inesistente');
            end if;
      end raise_sal;
end manage_employee;
```

Attenzione: al fine di compilare ed eseguire il package sopra, è necessario creare prima la sequenza richiesta (**help sequence**):

```
create sequence emp_sequence start with 8000 increment by 10;
```

Una procedura o funzione implementata in un package può essere richiamata da altre procedure e funzioni utilizzando l'istruzione <nome package>.<nome procedura>[(<lista di parametri>)].
Se si chiama una tale procedura da una shell SQL*Plus, è richiesto il comando **execute**.

Oracle offre differenti package predefiniti e procedure che possono essere utilizzate da utenti di database e sviluppatori di applicazioni. Un insieme molto utile di procedure è implementato nel package DBMS_OUTPUT. Questo package permette agli utenti di visualizzare le informazioni allo schermo nella loro sessione SQL*Plus quando il programma PL/SQL viene eseguito. E' anche molto utile eseguire il debug dei programmi PL/SQL che sono stati compilati con successo, ma non si comportano come previsto. Sotto sono elencate alcune delle procedure più importanti di questo package:

Nome Procedura	Descrizione
DBMS_OUTPUT.ENABLE	Abilita l'output
DBMS_OUTPUT.DISABLE	Disabilita l'output
DBMS_OUTPUT.PUT(<stringa>)	Visualizza <stringa> in coda al buffer di output
DBMS_OUTPUT.PUT_LINE(<stringa>)	Visualizza <stringa> in coda al buffer di output e inserisce un marcatore di nuova linea (a capo)
DBMS_OUTPUT.NEW_LINE	Visualizza un marcatore di nuova linea (a capo)

Prima che le stringhe vengano visualizzate a video, l'output deve essere abilitato usando la procedura DBMS_OUTPUT.ENABLE oppure utilizzando il comando SQL*Plus **set serveroutput on** (prima che la procedura che produce l'output venga richiamata).

Ulteriori packages forniti da Oracle sono UTL_FILE per la lettura e scrittura di files dai programmi PL/SQL, DBMS_JOB per la pianificazione dei job, e DBMS_SQL per generare istruzioni SQL dinamicamente, cioè, durante l'esecuzione del programma. Il package DBMS_SQL è tipicamente utilizzato per creare e cancellare tabelle dall'interno di programmi PL/SQL.

4.1.8 Programmazione in PL/SQL

Solitamente, viene utilizzato un semplice editor come emacs o vi (in unix) per scrivere un programma PL/SQL. Una volta che il programma è stato memorizzato in un file <nome file> con

un'estensione `.sql`, può essere caricato in SQL*Plus utilizzando il comando **@**<nome file>. E' importante che l'ultima linea del file contenga la barra "/" (slash).

Se la procedura, funzione, o package è stata compilata con successo, SQL*Plus visualizza il messaggio **`PL/SQL procedure successfully completed`**. Se il programma contiene errori, questi vengono visualizzati nel formato `ORA-`*n* <testo messaggio>, dove *n* è un numero e <testo messaggio> è una breve descrizione dell'errore, per esempio, `ORA-1001 INVALID CURSOR`. Il comando SQL*Plus **show errors** [<**function**|**procedure**|**package**|**package body**|**trigger**> <nome>] visualizza tutti gli errori di compilazione delle più recenti funzioni create o modificate (o procedure, o package, ecc.) con maggior dettaglio. Se questo comando non visualizza alcun errore, si può tentare con **select** * **from** `USER_ERRORS`.

Sotto una shell unix si può anche usare il comando **oerr** `ORA` *n* per avere informazioni nella seguente forma:

> *descrizione errore*
> Cause: *ragione dell'errore*
> Azione: *azione suggerita*

4.2 SQL Integrato e Pro*C

I costrutti del linguaggio per interrogazioni SQL descritto nelle precedenti sezioni sono adatti per la formulazione di query *ad-hoc*, istruzioni di manipolazione dei dati e semplici blocchi PL/SQL in un'utility semplice e interattiva come SQL*Plus. Molti compiti della gestione dei dati, comunque, avvengono in sofisticate applicazioni di ingegneria e questi compiti sono troppo complessi per essere gestiti da una tale utility interattiva. Tipicamente, i dati sono generati e manipolati in complesse applicazioni di elaborazione dati che sono scritte in linguaggi di terza generazione, e quindi, di conseguenza, hanno bisogno di un'interfaccia verso il sistema di database. Inoltre, la maggioranza delle esistenti applicazioni di elaborazione intensiva di dati è stata scritta in un linguaggio di programmazione classico ed ora vuole utilizzare le funzionalità di un sistema di database. Una tale interfaccia è fornita nella forma di *SQL Integrato* (*Embedded SQL*), una integrazione di SQL in vari linguaggi di programmazione, come C, C++, Cobol, Fortran, ecc.Il SQL integrato fornisce ai programmatori di applicazione un utile modo di combinare la potenza di un linguaggio di programmazione con la ben nota capacità di gestione e manipolazione dei dati del linguaggio di query SQL.

Dato che tutte queste interfacce mostrano funzionalità paragonabili tra loro, nella seguente sezione descriveremo l'integrazione dell'SQL nel linguaggio di programmazione C. A questo scopo, baseremo la nostra discussione sull'interfaccia di Oracle per il linguaggio C, chiamata Pro*C. L'enfasi in questa sezione è collocata sulla descrizione dell'interfaccia, non sull'introduzione al linguaggio di programmazione C.

4.2.1 Concetti generali

I programmi scritti in Pro*C che includono istruzioni SQL e/o PL/SQL sono precompilati in regolari programmi C usando un precompilatore che tipicamente affianca il software di gestione del database (precompiler package). Allo scopo di rendere le istruzioni SQL e PL/SQL in un

programma Pro*C (che ha il suffisso `.pc`) riconoscibili dal precompilatore, devono essere sempre preceduti dalla parola chiave **`EXEC SQL`** e terminati da un punto e virgola "`;`". Il precompilatore Pro*C rimpiazza tali istruzioni con appropriate chiamate a funzioni implementate nella libreria runtime SQL. Il programma in C risultante può quindi essere compilato e linkato utilizzando un normale compilatore C come qualsiasi altro programma in C. Il *linker* include le appropriate librerie specifiche di Oracle. La figura 1 riassume i passi dal codice sorgente che include le istruzioni SQL al programma eseguibile.

4.2.2 Variabili host e di comunicazione

Come nel caso dei blocchi PL/SQL, anche la prima parte di un programma Pro*C ha una sezione di dichiarazione. In un programma Pro*C, in una sezione di dichiarazione vengono dichiarate le cosiddette *variabili host*. Le variabili host sono la chiave per la comunicazione tra il programma finale e il database. La dichiarazione delle variabili host può essere collocata dove le stesse normali

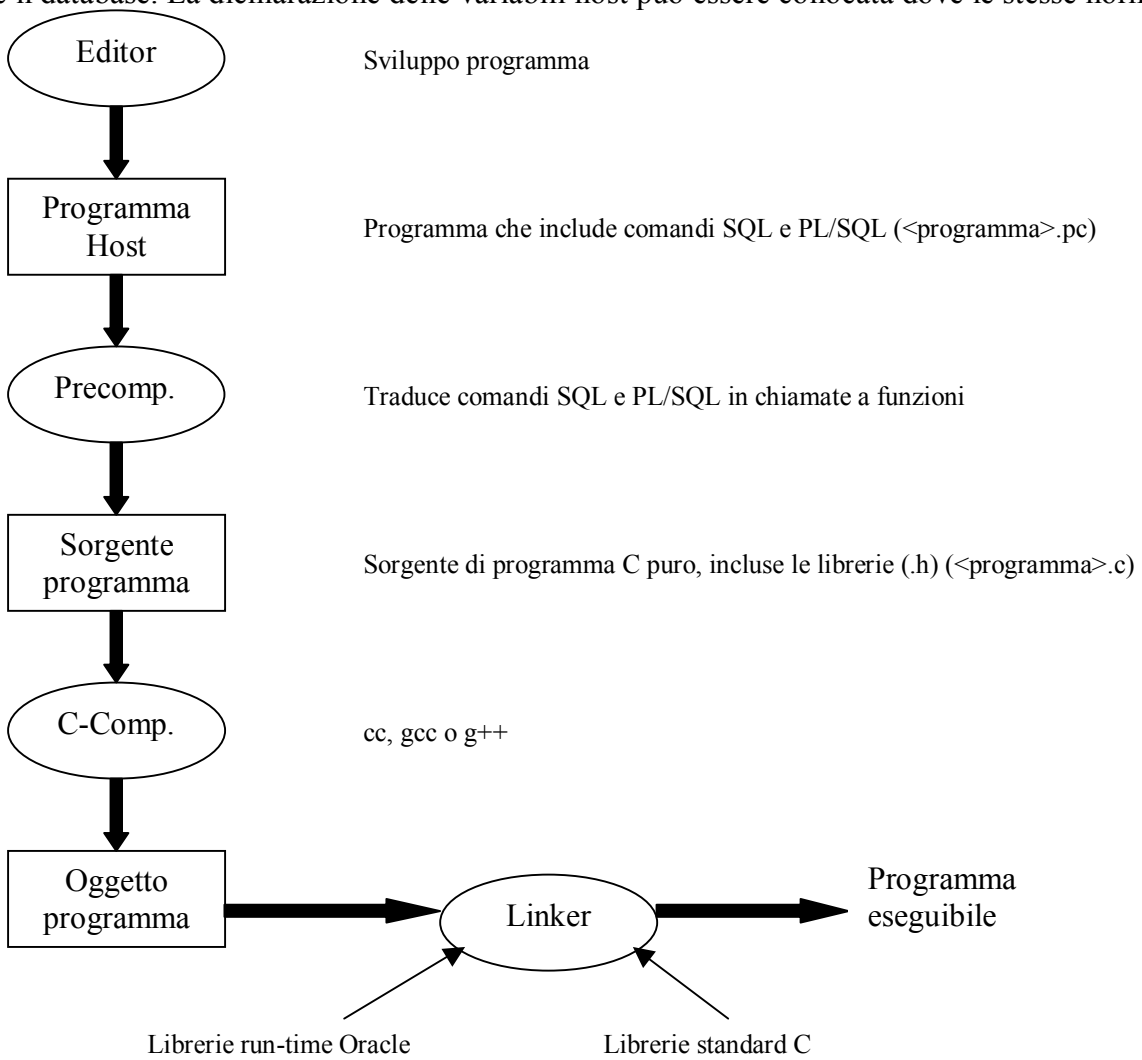

Figura 1: compilazione di un programma Pro*C

variabili del linguaggio C vengono collocate. Le variabili host vengono dichiarate seguendo la sintassi del linguaggio C. Le variabili host possono essere costituite dai seguenti tipi:

char <nome>	singolo carattere
char <nome> [*n*]	vettore di n caratteri
int	intero
float	punto decimale variabile
VARCHAR <nome> [*n*]	stringhe a lunghezza variabile

VARCHAR[2] è convertito dal precompilatore Pro*C in una struttura con un vettore di *n* caratteri (bytes) e un campo di 2-bytes per la lunghezza. La dichiarazione delle variabili host avviene in una sezione di dichiarazione che ha la seguente forma:

EXEC SQL BEGIN DECLARE SECTION
<dichiarazione delle variabili host>
/* per es. **VARCHAR userid[20];** */
/* per es. **char test_ok** */
EXEC SQL END DECLARE SECTION

In un programma Pro*C è concessa almeno una dichiarazione di questo tipo. La dichiarazione di cursori ed eccezioni avviene al di fuori della sezione di dichiarazione delle variabili host. In un programma Pro*C le variabili host a cui viene fatto riferimento in istruzioni SQL e PL/SQL devono essere precedute dai due punti ":". Da notare che non è possibile usare chiamate a funzioni C e molti dei puntatori ad espressioni come riferimenti alle variabili host.

4.2.3 L'Area di Comunicazione

In aggiunta alle variabili host che sono necessarie per passare i dati tra il database e il programma in C (e vice-versa), si potrebbe aver bisogno di alcune variabili di stato contenenti le informazioni di runtime. Le variabili vengono utilizzate per passare informazioni di stato relative all'accesso del database da parte del programma così che certi eventi possono essere gestiti nel programma in maniera corretta. La struttura contenente le variabili di stato è chiamato *SQL Communication Area* o *SQLCA*, in breve, e deve essere incluso dopo la sezione di dichiarazione utilizzando l'istruzione

EXEC SQL INCLUDE SQLCA.H

Nelle variabili definite in questa struttura, vengono mantenute informazioni sui messaggi di errore e sullo stato dei programmi:

```
struct    sqlca
     {
     /* ub1 */       char  sqlcaid[8];
     /* b4  */       long  sqlabc;
     /* b4  */       long  sqlcode;
     struct
          {
          /* ub2 */  unsigned short       sqlerrml;
          /* ub1 */  char                 sqlerrmc[70];
          } sqlerrm;
     /* ub1 */  char  sqlerrp[8];
```

[2] Nota: tutte lettere maiuscole; **varchar2** non è permessa!

```
/* b4   */ long sqlerrd[6];
/* ub1 */  char sqlwarn[8];
/* ub1 */  char sqlext[8];
};
```

I campi in questa struttura hanno il seguente significato:

`sqlcaid`	usato per identificare la SQLCA, impostato a "SQLCA"
`sqlabc`	mantiene la lunghezza della struttura SQLCA
`sqlcode`	mantiene il codice dello status delle istruzioni SQL (PL/SQL) eseguite più di recente 0 ≙ Nessun errore, istruzione completata con successo >0 ≙ Istruzione eseguita ed eccezione trovata. Situazioni tipiche sono quelle in cui **fetch** o **select into** non restituiscono righe <0 ≙ L'istruzione non è stata completata a causa di un errore; la transazione dovrebbe essere annullata (rollback) esplicitamente.
`sqlerrm`	struttura con due componenti `sqlerrml`: lunghezza del testo del messaggio in sqlerrmc, e `sqlerrmc`: messaggio di testo degli errori (fino a 70 caratteri) corrispondenti ai codici di errore registrati in sqlcode
`sqlerrp`	non utilizzato
`sqlerrd`	vettore di binari interi, ha 6 elementi: `sqlerrd[0]`, `sqlerrd[1]`, `sqlerrid[3]`, `sqlerrd[6]`, non utilizzati; `sqlerrd[2]` = numero di righe processate dall'istruzione SQL più recente; `sqlerrd[4]` = posizione di offset (scostamento) del più recente errore di istruzione SQL analizzato
`sqlwarn`	vettore con 8 elementi usati come flag di avvisi (non errori!). Il flag è impostato assegnandogli il carattere 'W'. `sqlwarn[0]`: impostato solo se un altro flag è impostato `sqlwarn[1]`: se i valori di colonna troncati sono stati assegnati a una variabile host `sqlwarn[2]`: la colonna *null* non è utilizzata nel calcolo di una funzione aggregata `sqlwarn[3]`: il n. di colonne in **select** non è uguale al n. di variabili host specificate in **into** `sqlwarn[4]`: se qualsiasi tupla è stata processata da un istruzione **update** o **delete** senza una clausola **where** `sqlwarn[5]`: compilazione del corpo di una procedura/funzione fallita a causa di un errore PL/SQL `sqlwarn[6]` e `sqlwarn[7]`: non utilizzati
`sqlext`	non utilizzato

Si può accedere ai componenti di questa struttura e verificare la durata di esecuzione, ed appropriate routine di gestione (per es. gestione delle eccezioni) possono essere eseguite per assicurare un funzionamento corretto dell'applicazione. Se alla fine del programma la variabile `sqlcode` contiene 0, allora l'esecuzione del programma è avvenuta correttamente, altrimenti è stato trovato un errore.

4.2.4 Gestione delle eccezioni

Ci sono due modi per controllare lo stato del proprio programma dopo l'esecuzione di istruzioni SQL che possono risultare in errore o in avvisi (warnings): (1) controllando esplicitamente le rispettive componenti della struttura SQLCA, o (2) eseguendo un controllo e una gestione automatica degli errori utilizzando l'istruzione **WHENEVER**. La sintassi completa di questa istruzione è:

EXEC SQL WHENEVER <condizione> <azione>;

Usando questo comando, il programma controlla automaticamente la SQLCA per la <condizione> ed esegue una determinata <azione>. La <condizione> può essere una delle seguenti:

- `SQLERROR`: `sqlcode` ha un valore negativo, cioè è avvenuto un errore
- `SQLWARNING`: in questo caso `sqlwarn[0]` è impostato per segnalare un avviso
- `NOT FOUND`: `sqlcode` ha un valore positivo, il che significa che nessuna riga che soddisfa la condizione **where** è stata trovata, o una istruzione **select into** o **fetch** non ha restituito nessuna riga.

<azione> può essere:

- STOP: il programma esce con una chiamata exit(), e tutte le istruzioni SQL che non sono state confermate (commit) vengono annullate (rollback)
- CONTINUE: se possibile, il programma cerca di continuare con l'istruzione successiva a quella che ha generato l'errore
- DO <funzione>: il programma trasferisce il processo a una funzione di gestione degli errori di nome <funzione>
- GOTO <etichetta>: l'esecuzione del programma salta all'istruzione successiva all'<etichetta> (vedi esempio)

4.2.5 Connessione al Database

All'inizio del programma Pro*C, più precisamente, all'esecuzione di istruzioni di SQL integrato o PL/SQL, è necessario connettersi al database utilizzando un account e una password validi per il database. La connessione al database avviene attraverso l'istruzione SQL integrata:

EXEC SQL CONNECT :<utente> **IDENTIFIED BY** :<password>.

Sia <utente> che <password> sono variabili host del tipo **VARCHAR** e devono essere rispettivamente specificate e gestite (vedi anche l'esempio di programma Pro*C nella Sezione 4.2.7). <utente> e <password> possono essere specificati nel programma Pro*C, ma possono anche essere inseriti durante l'esecuzione, utilizzando, per es., la funzione C **scanf**.

4.2.6 Commit e Rollback

Prima che un programma venga terminato dalla funzione C exit e se nessun errore è avvenuto, le modifiche al database attraverso le istruzioni SQL integrate **insert**, **update** e **delete** devono essere confermate (**commit**). Questo viene fatto usando l'istruzione SQL integrata

EXEC SQL COMMIT WORK RELEASE;

Se un errore di programma interviene e le precedenti modifiche al database devono essere annullate, l'istruzione SQL integrata

EXEC SQL ROLLBACK RELEASE;

deve essere specificata nella rispettiva routine di gestione dell'errore nel programma Pro*C.

4.2.7 Esempio di programma Pro*C

Il seguente esempio di programma Pro*C si connette al database usando l'account scott/tiger. Il database contiene informazioni relative agli impiegati e ai dipartimenti (vedi i precedenti esempi usati in questo tutorial). L'utente deve inserire uno stipendio che viene utilizzato per ricercare tutti gli impiegati (dalla relazione `EMP`) che guadagnano più dello stipendio minimo inserito. La ricerca e l'elaborazione delle singole tuple che risultano avviene attraverso un cursore PL/SQL in un loop while del linguaggio C.

```
/* Dichiarazioni */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* Sezione di dichiarazione per le variabili host */
EXEC SQL BEGIN DECLARE SECTION;
     VARCHAR userid[20];
     VARCHAR passwd[20];
     int empno;
     VARCHAR ename[15];
     float sal;
     float min_sal;
EXEC SQL END DECLARE SECTION;

/* Caricamento dell'SQL Communication Area */
EXEC SQL INCLUDE SQLCA.H;

main()  /* Programma principale */
{    int retval;
     /* riconosce automaticamente gli errori e salta automaticamente */
     /* alle relative routines di gestione */
     EXEC SQL WHENEVER SQLERROR GOTO error;

     /* si connette al database come scott/tiger; entrambe sono */
     /* variabili host di tipo VARCHAR; l'utente e la password sono */
     /* specificate esplicitamente */
     strcpy(userid.arr,"SCOTT");    /* userid.arr:="SCOTT" */
     userid.len=strlen(userid.arr); /* uid.len:=5 */
     strcpy(passwd.arr,"TIGER");    /* passwd.arr:="TIGER" */
     passwd.len=strlen(passwd.arr); /* passwd.len:=5 */

     EXEC SQL CONNECT :userid IDENTIFIED BY :passwd;
```

```
printf("Connesso a Oracle come: %s\n\n", userid.arr);

/* inserimento del minimo stipendio da parte dell'utente */
printf("Inserire il minimo stipendio > ");
retval = scanf("%f", &min_sal);

if(retval != 1) {
     printf("Valore errato!!\n");
     EXEC SQL ROLLBACK WORK RELEASE;
     /* Disconnessione da Oracle */
     exit(2);  /* uscita dal programma */
}

/* Dichiarazione del cursore; non può avvenire nella sezione */
/* delle dichiarazioni!! */
EXEC SQL DECLARE EMP_CUR CURSOR FOR
SELECT EMPNO, ENAME, SAL FROM EMP
     WHERE SAL >= :min_sal;

/* Stampa (a video) l'intestazione della tabella, esegue il */
/* cursore ottenendo un insieme di risultati */
printf("ID-Impiegato      Nome-Impiegato      Stipendio \n");
printf("------------      --------------      ----------\n");
EXEC SQL OPEN EMP_CUR;
EXEC  SQL FETCH EMP_CUR INTO :empno, :ename, :sal; /* recupera 1 */
                                                   /* tupla */
while(sqlca.sqlcode==0) {  /* ci sono più tuple? */
     ename.arr[ename.len] = '\0';   /* "fine stringa" */
     printf("%15d  %-17s  %7.2f\n",empno,ename.arr,sal);
     EXEC SQL FETCH EMP_CUR INTO :empno, :ename, :sal; /* recu- */
                                    /* pera la successiva tupla */
}
EXEC SQL CLOSE EMP_CUR;

/* Disconnessione dal database e fine ed uscita dal programma */
EXEC SQL COMMIT WORK RELEASE;
Printf("\nDisconnesso da Oracle\n");
exit(0);

/* Gestione degli errori: stampa (a video) dei mess. di errore */
error: printf("\nErrore: %.70s \n",sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
Exit(1);
}
```

5 VINCOLI DI INTEGRITA' E TRIGGERS

5.1 VINCOLI DI INTEGRITA'

Nella sezione 1 abbiamo discusso tre tipi di vincoli di integrità: vincoli *not null*, *chiavi primarie* e vincoli *unique*. In questa sezione introduciamo due ulteriori tipi di vincoli che possono essere

specificati all'interno dell'istruzione **create table**: vincoli *check* (per restringere i possibili valori degli attributi) e vincoli *foreign key* (per specificare le dipendenze tra relazioni).

5.1.1 Vincoli Check

Spesso le colonne in una tabella devono avere valori che rientrano in un certo insieme o che soddisfano certe condizioni. I vincoli check permettono agli utenti di restringere i possibili valori di attributo per una colonna a dei valori ammissibili per quest'ultima. Essi possono essere specificati come vincoli di colonna o vincoli di tabella. La sintassi per un vincolo **check** è:

[**constraint** <nome>] **check**(<condizione>)

Se un vincolo **check** viene specificato come vincolo di colonna, la condizione si riferisce solo a quella colonna.

Esempio: Il nome di un impiegato deve essere costituito da sole lettere maiuscole; il minimo stipendio di un impiegato è 500; i numeri dei dipartimenti devono essere compresi tra 10 e 100.

```
create table EMP
        (…,
        ENAME       varchar2(30) constraint check_name check(ENAME = upper(ENAME)),
        SAL         number(5,2) constraint check_sal check(SAL>=500),
        DEPTNO      number(3) constraint check_deptno check(DEPTNO between 10 and 100));
```

Se un vincolo **check** è specificato come vincolo di tabella, <condizione> può far riferimento a tutte le colonne della tabella. Da notare che sono consentite solo semplici condizioni. Per esempio, non è consentito far riferimento a colonne di altre tabelle o formulare query come condizioni **check**. Inoltre, la funzione **sysdate** e **user** non può essere usata nella condizione. Come principio, quindi, solo semplici confronti tra attributi e connessioni logiche come **and**, **or** e **not** sono permesse. Una condizione **check**, comunque, può includere un vincolo *not null*:

```
SAL number(5,2) constraint check_sal check(SAL is not null and SAL>=500)
```

Senza la condizione **not null**, il valore *null* per l'attributo `SAL` non causa una violazione del vincolo.

Esempio: Almeno due persone devono partecipare in un progetto, e la data di inizio del progetto deve essere antecedente alla data di fine dello stesso.

```
create table PROJECT
        (…,
        PERSONS     number(5) constraint check_pers check(PERSONS>2),
        …,
        constraint date_ok check(PEND > PSTART));
```

In questa definizione di tabella, `check_pers` è un vincolo di colonna e `dates_ok` è un vincolo di tabella.

Il sistema di database automaticamente controlla le condizioni specificate ogni volta che una modifica viene effettuata sulla relazione. Per esempio, l'inserimento

insert into `EMP` **values**(7999,'SCOTT','CLERK',7698,'31-OCT-94',450,10);

provoca una violazione del vincolo.

`ORA-02290: check_constraint (CHECK_SAL) violated`

e l'inserimento viene rifiutato.

5.1.2 Vincolo Foreign Key

Un vincolo **foreign key** (o *vincolo di integrità referenziale*) può essere specificato come vincolo di colonna o vincolo di tabella:

[**constraint** <nome>] [**foreign key** (<colonna(e)>)]
references <table>[(<colonna(e)>)]
[**on delete cascade**]

Questo vincolo specifica una colonna o lista di colonne come chiave di riferimento di una tabella esterna. La tabella dalla quale parte il riferimento è chiamata *tabella-figlia* (*child-table*), e la tabella nella quale il riferimento punta è chiamata *tabella-genitrice* (*parent-table*). In altre parole, non è possibile definire un vincolo di integrità referenziale che fa riferimento a una tabella R prima che la tabella R sia stata creata.

La clausola **foreign key** deve essere utilizzata in aggiunta alla clausola **references** se la chiave esterna include più di una colonna. In questo caso, il vincolo deve essere specificato come vincolo di tabella. La clausola **references** definisce quale colonna della tabella genitrice è referenziata. Se viene indicato solo il nome della tabella genitrice, viene assunto per default l'intera lista degli attributi che costituiscono la chiave primaria.

Esempio: Ogni impiegato nella tabella `EMP` deve lavorare in un dipartimento che è contenuto nella tabella `DEPT`:

```
create table EMP
      (EMPNO      number(4) constraint pk_emp primary key,
       …,
       DEPTNO     number(3) constraint fk_deptno references
                  DEPT(DEPTNO));
```

La colonna `DEPTNO` della tabella `EMP` (tabella figlia) costituisce la chiave esterna e fa riferimento alla chiave primaria della tabella `DEPT` (tabella genitrice). La relazione tra le due tabelle è illustrata nella figura 2. Dal momento che nella definizione della tabella sopra, il vincolo di integrità referenziale include solo una colonna, la clausola **foreign key** viene volutamente omessa. E' molto importante che una chiave esterna faccia riferimento alla completa chiave primaria della tabella

genitrice, non solo ad un sottoinsieme degli attributi che costituiscono la chiave primaria! (ricordate che una chiave primaria può essere costituita da più colonne)

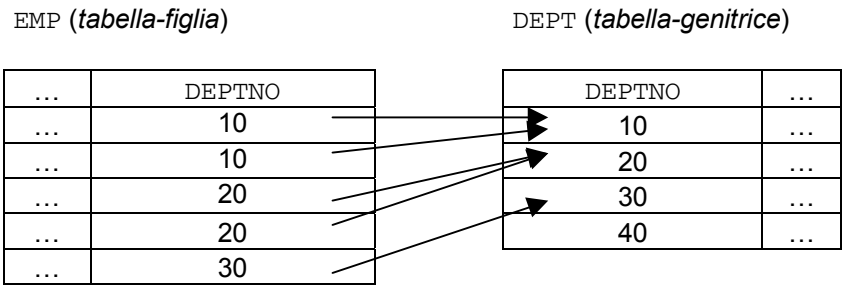

Figura 2: Vincolo di Chiave Esterna tra le tabelle EMP e DEPT

Allo scopo di soddisfare un vincolo di chiave esterna, ogni riga nella tabella-figlia deve soddisfare una due seguenti condizioni:

- Il valore di attributo (lista di valori di attributi) della chiave esterna deve comparire come chiave primaria nella tabella-genitrice, oppure
- Il valore di attributo della chiave esterna è *null* (nel caso di una chiave esterna composta, almeno un valore di attributo della chiave esterna è *null*)

In accordo con le definizioni sopra per la tabella `EMP`, un impiegato non deve necessariamente lavorare in un dipartimento, per es., per l'attributo `DEPTNO` il valore *null* è ammesso.

Esempio: Ogni manager di un progetto deve essere un impiegato:

```
create table PROJECT
        (PNO        number(3) constraint prj_pk primary key,
         PMGR       number(4) not null,
                    constraint fk_pmgr references EMP,
        ...);
```

Poiché viene dato solo il nome della tabella-genitrice (`DEPT`), la chiave primaria di questa relazione viene assunta per default. Un vincolo di chiave esterna può anche far riferimento alla stessa tabella, nell'esempio, la tabella-genitrice e la tabella-figlia sono la stessa tabella.

Esempio: Ogni manager deve necessariamente essere uno degli impiegati:

```
create table EMP
        (EMPNO      number(4) constraint emp_pk primary key,
         ...
         MGR        number(4) not null
                              constraint fk_mgr references EMP,
         ...
        );
```

5.1.3 Alcune informazioni aggiuntive sui vincoli di colonna e di tabella

Se un vincolo viene definito all'interno di un comando **create table** o aggiunto utilizzando il comando alter table (vedi Sezione 1.5.5), il vincolo viene automaticamente abilitato. Un vincolo può essere disabilitato utilizzando il comando

alter table <tabella> **disable**
constraint <nome> | **primary key** | **unique**[<colonna(e)>]
[**cascade**];

Per disabilitare una chiave primaria, è necessario disabilitare tutte le chiavi esterne che dipendono dalla chiave primaria. La clausola **cascade** disabilita automaticamente i vincoli delle chiavi esterne che dipendono dalla chiave primaria (disabilitata).

Esempio: Disabilitare la chiave primaria della tabella `DEPT` e disabilitare i vincoli delle chiavi esterne nella tabella `EMP`:

alter table `DEPT` **disable primary key cascade**;

Al fine di abilitare un vincolo di integrità, la clausola **enable** può essere utilizzata al posto di **disable**. Un vincolo può essere abilitato con successo se nessuna tupla nella tabella viola il vincolo. Altrimenti un messaggio di errore verrà visualizzato. Da notare che per abilitare/disabilitare un vincolo di integrità è importante aver dato un nome al vincolo.

Al fine di identificare quelle tuple che violano il vincolo di integrità la cui attivazione fallisce, è possibile utilizzare la clausola **exception into `EXCEPTIONS`** con l'istruzione **alter table**. `EXCEPTIONS` è una tabella che memorizza informazioni sulla violazione di tuple[3]. Ogni tupla in questa tabela è identificata dall'attributo `ROWID`. Ogni tupla in un database ha una pseudo-colonna `ROWID` che viene utilizzata per identificare le tuple. Oltre a rowid, vengono memorizzati il nome della tabella, il proprietario della tabella e il nome del vincolo violato.

Esempio: Vogliamo aggiungere un vincolo di integrità alla nostra tabella `EMP`, la quale richiede che ogni manager debba guadagnare più di 4000:

alter table `EMP` **add constraint** `manager_sal`
check(`JOB != 'MANAGER' or SAL >= 4000`)
exceptions into `EXCEPTIONS`;

Se la tabella `EMP` contiene già tuple che violano il vincolo, il vincolo non può essere attivato e informazioni sulla violazione delle tuple sono automaticamente inserite nella tabella `EXCEPTIONS`.

Dettagliate informazioni sulla violazione delle tuple possono essere ottenute combinando (joining) le tabelle `EMP` ed `EXCEPTIONS`, basandosi sull'attributo `ROWID`:

```
select EMP.*, CONSTRAINT from EMP, EXCEPTIONS
where EMP.ROWID = EXCEPTIONS.ROW_ID;
```

[3] Prima che questa tabella possa essere utilizzata, deve essere creata usando lo script SQL **utlexcept.sql** che si trova nella directory **$ORACLE_HOME/rdbms/admin**.

Le tuple contenute nel risultato di una query possono ora essere modificate (nell'esempio, incrementando lo stipendio dei managers) in modo che l'aggiunta del vincolo possa essere svolta con successo. Da notare che è importante cancellare le "vecchie" violazioni dalla relazione `EXCEPTIONS` prima che questa venga utilizzata nuovamente.

Se una tabella viene utilizzata come referenza per una chiave esterna, questa tabella può essere cancellata solamente utilizzando il comando **drop table** <tabella> **cascade constraints;**. Tutti gli altri oggetti di database che si riferiscono a questa tabella (nell'esempio, triggers, vedi sezione 5.2) rimangono nel sistema di database, ma non sono validi.

Informazioni sui vincoli di integrità, il loro stato (abilitati, disabilitati), ecc., sono memorizzate nel data dictionary, più precisamente, nella tabella `USER_CONSTRAINTS e USER_CONS_CONSTRAINTS`.

5.2 TRIGGERS

5.2.1 Panoramica

I differenti tipi di vincoli di integrità discussi finora forniscono un meccanismo di *dichiarazione* per associare "semplici" condizioni in una tabella come una chiave primaria, una chiave esterna o vincoli di dominio. Vincoli di integrità complessi che fanno riferimento a diverse tabelle e attributi (conosciuti come *asserzioni* nello standard SQL) non possono essere specificati all'interno di una definizione di tabella. I *triggers*, al contrario, forniscono una tecnica procedurale per specificare e mantenere vincoli di integrità anche complessi. I triggers permettono agli utenti di specificare vincoli di integrità più complessi dato che un trigger è essenzialmente una procedura PL/SQL. Tale procedura è associata con una tabella e viene automaticamente richiamata dal sistema quando una certa modifica (o evento) avviene all'interno della tabella. Le modifiche sulla tabella possono includere operazioni **insert**, **update**, e **delete** (Oracle7).

5.2.2 Struttura dei Triggers

La definizione di un trigger consiste nei seguenti componenti (opzionali):

- *nome trigger*
  create [**or replace**] **trigger** <nome trigger>
- *collocazione temporale del trigger*
  before | **after**
- *azione(i) del trigger*
  insert or **update** [**of** <colonna(e)>] or **delete on** <tabella>
- *tipo di trigger* (opzionale)
  for each row
- *restrizioni trigger* (solo per triggers **for each row**!)
  when (<condizione>)
- *corpo del trigger*
  <blocco PL/SQL>

La clausola **or replace** ri-crea una precedente definizione del trigger qualora questo esista e abbia lo stesso <nome trigger>. Il nome di un trigger può essere scelto arbitrariamente, ma è una buona regola di programmazione usare un nome che rifletta la tabella e l'evento(i), nell'esempio, `upd_ins_EMP`. Un trigger può essere richiamato prima (**before**) o dopo (**after**) l'evento che causa l'attivazione del trigger. L'evento che causa l'attivazione del trigger specifica prima (o dopo) quale operazione nella tabella <tabella> il trigger debba essere eseguito. Un singolo evento è un inserimento (**insert**), un aggiornamento (**update**) o una cancellazione (**delete**); gli eventi possono essere combinati usando la logica **or**. Se il trigger deve essere eseguito soltanto quando certe colonne vengono aggiornate, queste colonne devono essere specificate dopo l'evento **update**. Se un trigger viene utilizzato per mantenere un vincolo di integrità, gli eventi che causano il trigger corrispondono tipicamente alle operazioni che violano l'integrità del vincolo.

Al fine di programmare i trigger efficientemente (e correttamente), è essenziale capire la differenza tra *trigger a livello di riga* e *trigger a livello di istruzione*. Un trigger a livello di riga viene definito utilizzando la clausola **for each row**. Se questa clausola viene omessa, si assume che il trigger sia un trigger a livello di istruzione. Un trigger a livello di riga viene eseguito una volta per ogni riga dopo (o prima) dell'evento che lo ha causato. Al contrario, un trigger a livello di istruzione viene eseguito una volta dopo (o prima) dell'evento, indipendentemente da quante righe sono state interessate dall'evento. Per esempio, un trigger di riga con la specifica di evento **after update** viene eseguito una volta per ogni riga che viene interessata dall'aggiornamento (**update**). Quindi, se l'aggiornamento interessa 20 tuple, il trigger viene eseguito 20 volte, una volta per ogni riga. Al contrario, un trigger di istruzione viene eseguito una sola volta.

Quando si combinano differenti tipi di triggers, ci sono dodici possibili configurazioni che possono essere definite per una tabella:

Evento	Contesto Temporale		Tipo di Trigger	
	before	**after**	istruzione	riga
insert	**X**	**X**	**X**	**X**
update	**X**	**X**	**X**	**X**
delete	**X**	**X**	**X**	**X**

Figura 3: Tipi di Triggers

I trigger di riga hanno alcune speciali caratteristiche che non sono fornite con i trigger di istruzione:

Solo con un trigger di riga è possibile accedere ai valori degli attributi di una tupla prima e dopo la modifica (perché il trigger viene eseguito una volta per ogni tupla). Per un **update** trigger, si può accedere al vecchio valore di attributo utilizzando **:old.**<colonna> e si può accedere al nuovo attributo utilizzando **:new.**<colonna>. Per un **insert** trigger, solo **:new.**<colonna> può essere utilizzato, e per un delete trigger solo **:old.**<colonna> è valido (e si riferisce al valore dell'attributo della <colonna> della tupla cancellata). In un trigger di riga quindi è possibile specificare confronti tra il vecchio e il nuovo valore di attributo nel blocco PL/SQL, per es., "**if :old.**`SAL` < **:new.**`SAL` **then** …". Se per un trigger di riga il contesto temporale **before** viene specificato, è anche possibile modificare il nuovo valore della riga, per es., **:new.**`SAL := :new.SAL*`1.05 oppure **:new.**`SAL := :old.SAL`.
Tali modifiche non sono possibili con i trigger di riga **after**. In generale, è raccomandabile utilizzare un trigger di riga **after** se la nuova riga non viene modificata nel blocco PL/SQL. Oracle può quindi processare questi trigger più efficientemente. I trigger a livello di istruzione sono usati in generale solo in combinazione con il trigger **after**.

In una definizione di trigger la clausola **when** può essere usata solo in combinazione con un trigger **for each row**. La clausola è utilizzata per restringere ulteriormente l'attivazione del trigger. Per la specifica della condizioni nella clausola **when**, vengono mantenute le stesse restrizioni della clausola **check**. Le uniche eccezioni sono che le funzioni **sysdate** e **user** possono essere utilizzate, e che è possibile far riferimento ai vecchi/nuovi valori degli attributi della riga attuale. In quest'ultimo caso, i due punti ":" non devono essere utilizzati, per es., solo **old.**<attributo> e **new.**<attributo>.

Il corpo del trigger consiste in un blocco PL/SQL. Tutti i comandi SQL e PL/SQL eccetto le due istruzioni **commit** e **rollback** possono essere utilizzati in un blocco PL/SQL di un trigger. Inoltre, costrutti addizionali **if** permettono l'esecuzione di certe parti del blocco PL/SQL a seconda dell'evento che aziona il trigger. A questo scopo esistono tre costrutti: **if inserting**, **if updating**[('<colonna>')], e **if deleting**. Possono essere utilizzati come mostrato nel seguente esempio:

```
create or replace trigger emp_check
after insert or delete or update on EMP
for each row
begin
        if inserting then
                <blocco PL/SQL>
        end if;
        if updating then
                <blocco PL/SQL>
        end if;
        if deleting then
                <blocco PL/SQL>
        end if;
end;
```

E' importante comprendere che l'esecuzione di un blocco PL/SQL di un trigger costituisce una parte di transazione che può contenere eventi che provocano l'attivazione di altri trigger. Quindi, per esempio, un'istruzione **insert** in un blocco PL/SQL può causare l'attivazione di un altro trigger. Più triggers e modifiche quindi possono innescare un'esecuzione a cascata di triggers. Una tale sequenza di triggers termina con successo se (1) nessuna eccezione viene rilevata all'interno del blocco PL/SQL, e (2) nessuna dichiarazione di vincolo di integrità è stata violata. Se un trigger rileva un'eccezione in un blocco PL/SQL, tutte le modifiche fino all'inizio della transazione vengono annullate (rollback). Nel blocco PL/SQL di un trigger, un'eccezione può essere provocata utilizzando l'istruzione **raise_application_error** (vedi Sezione 4.1.5). Questa istruzione provoca un **rollback** implicito. In combinazione con un trigger di riga, **raise_application_error** può far riferimento a vecchi/nuovi valori della riga modificata:

```
raise_application_error(-20020,'Incremento dello stipendio da '||to_char(:old.SAL)||' to
                        '||to_char(:new.SAL)||' è troppo alto');    oppure

raise_application_error(-20030,'Id Impiegato '||to_char(:new.EMPNO)||' non esiste.');
```

5.2.3 Esempi di Triggers

Supponiamo di dover mantenere i seguenti vincoli di integrità: "Lo stipendio di un impiegato che non sia il presidente (ebbe'…) non può essere decrementato e inoltre non deve essere aumentato più del 10%. Inoltre, a seconda del titolo di lavoro, ogni stipendio deve rientrare entro un certo intervallo.

Assumiamo che una tabella SALGRADE memorizzi il minimo (MINSAL) e il massimo (MAXSAL) stipendio per ogni titolo di lavoro (JOB). Dato che la condizione esposta sopra può essere controllata per ogni impiegato individualmente, definiamo il seguente trigger di riga:

trig1.sql

```
create or replace trigger check_salary_EMP
after insert or update of SAL, JOB on EMP
for each row
when (new.JOB != 'PRESIDENT')  -- restrizione del trigger
declare
        minsal, maxsal SALGRADE.MAXSAL%TYPE;
begin
        -- recupera il minimo e il massimo stipendio per JOB
        select MINSAL, MAXSAL into minsal, maxsal from SALGRADE
        where JOB = :new.JOB;
        -- se il nuovo stipendio è stato decrementato o non è compreso entro un certo intervallo,
        -- viene provocata un'eccezione
        if (:new.SAL < minsal or :new.SAL > maxsal) then
            raise_application_error(-20225,'Stipendio non consentito');
        elsif (:new.SAL < :old.SAL) then
            raise_application_error(-20230,'Stipendio decrementato');
        elsif (:new.SAL > 1.1 * :old.SAL) then
            raise_application_error(-20235,'Incremento maggiore del 10%');
        end if;
end;
```

Utilizziamo un trigger **after** perché una riga inserita o modificata non viene cambiata all'interno del blocco PL/SQL (ad es., nel caso di una violazione del vincolo, è possibile ripristinare i vecchi valori degli attributi).

Da notare che anche le modifiche alla tabella SALGRADE può causare una violazione di vincolo. Allo scopo di mantenere la condizione completa definiamo il seguente trigger sulla tabella SALGRADE. Nel caso di una violazione da una modifica **update**, comunque, non provocheremo un'eccezione, ma ripristineremo i vecchi valori degli attributi.

trig2.sql

```
create or replace trigger check_salary_SALGRADE
before update or delete on SALGRADE
for each row
```

```
when (new.MINSAL > old.MINSAL
      or new.MAXSAL < old.MAXSAL)
      -- solo restringendo l'intervallo per lo stipendio si può provocare una violazione
declare
      job_emps number(3) := 0;
begin
      if deleting then -- esiste ancora un impiegato che possiede il lavoro cancellato?
            select count(*) into job_emps from EMP
            where JOB = :old.JOB;
            if job_emps != 0 then
                  raise_application_error(-20240,'Esiste ancora un impiegato il lavoro '
                                          ||:old.JOB);
            end if;
      end if;
      if updating then
      -- ci sono ancora impiegati il cui stipendio non rientra nel modificato intervallo?
            select count(*) into job_emps from EMP
            where JOB = :new.JOB
                        and SAL not between :new.MINSAL and :new.MAXSAL;
            if job_emps != 0 then -- rispristina il vecchio intervallo di stipendi
                  :new.MINSAL := :old.MINSAL;
                  :new.MAXSAL := :old.MAXSAL;
            end if;
      end if;
end;
```

In questo caso deve essere usato un trigger **before** per ripristinare i vecchi valori di attributo di una riga aggiornata.

Supponiamo di dover avere un ulteriore colonna BUDGET nella nostra tabella DEPT che viene utilizzata per memorizzare il budget disponibile per ogni dipartimento. Si assume che il vincolo di integrità richieda che il totale di tutti gli stipendi in un dipartimento non debba eccedere il budget del dipartimento. Le operazioni critiche sulla relazione EMP sono inserimenti nella tabella EMP e aggiornamenti sugli attributi SAL e DEPTNO.

trig3.sql

```
create or replace trigger check_budget_EMP
after insert or update of SAL, DEPTNO on EMP
declare
      cursor      DEPT_CUR is
                  select DEPTNO, BUDGET from DEPT;
      DNO         DEPT.DEPTNO%TYPE;
      ALLSAL      DEPT.BUDGET%TYPE;
      DEPT_SAL    number;
begin
      open  DEPT_CUR;
      loop
```

```
            fetch DEPT_CUR into DNO, ALLSAL;
            exit when DEPT_CUR%NOTFOUND;
            select sum(SAL) into DEPT_SAL from EMP
            where DEPTNO = DNO;
            if DEPT_SAL > ALLSAL then
                  raise_application_error(-20325,'Il totale degli stipendi nel dipartimento '||
                                          to_char(DNO)||' supera il budget');
            end if;
      end loop;
      close DEPT_CUR;
end;
```

In questo caso utilizziamo un trigger di istruzione sulla relazione EMP perché dobbiamo applicare una funzione di aggregazione sullo stipendio di tutti gli impiegati che lavorano in un particolare dipartimento. Per la relazione DEPT, dobbiamo anche definire un trigger che, comunque, può essere formulato come trigger di riga.

5.2.4 Programmazione di Triggers

Per i programmatori, i trigger di riga sono i tipi di trigger più critici perché includono diverse restrizioni. Al fine di assicurare coerenza in lettura, Oracle esegue un blocco (lock) esclusivo sulla tabella all'inizio di un'istruzione **insert**, **update** o **delete**. Cioè, gli altri utenti non possono accedere a questa tabella finchè le modifiche non sono state completate con successo. In questo caso, la tabella che è attualmente in corso di modifica viene detta tabella *mutante*. Il solo modo di accedere ad una tabella mutante in un trigger è di usare **:old.**<colonna> e **:new.**<colonna> in concomitanza con un trigger di riga.

Esempio di un trigger di riga errato:

```
create trigger check_sal_EMP
after update of SAL on EMP
for each row
declare
      sal_sum number;
begin
      select sum(SAL) into sal_sum from EMP;
      …;
end;
```

Per esempio, se un'istruzione **update** del tipo **update** EMP **set** SAL = SAL*1.1 viene eseguita sulla tabella EMP, il trigger sopra viene eseguito una volta per ogni riga modificata. Mentre la tabella è in corso di modifica dal comando **update**, non è possibile accedere a tutte le tuple della tabella usando il comando **select**, perché la tabella intera è bloccata (locked) in maniera esclusiva. In questo caso avremo il messaggio di errore:

```
ORA-04091: table EMP is mutating, trigger may not read or modify
it
ORA-06512: at line 4
ORA-04088: error during execution of trigger 'CHECK_SAL_EMP'
```

Il solo modo di accedere alla tabella, più precisamente, accedere alle tuple modificate, è quello di utilizzare la sintassi **:old.**<colonna> e **:new.**<colonna>.

E' consigliabile seguire le seguenti regole per la definizione e il mantenimento di vincoli di integrità:

identificare operazioni e tabelle che sono critiche per i vincoli di integrità
    per ogni tale tabella controllare
        se il vincolo può essere controllato a livello di riga, **quindi**
            se le righe controllate sono modificate all'interno del trigger, **quindi**
                utilizzare il trigger di riga **before**
            altrimenti utilizzare il trigger di riga **after**
        altrimenti
        utilizzare il trigger a livello di istruzione **after**

I trigger non sono esclusivamente utilizzati per il mantenimento dell'integrità. Essi possono essere usati anche per

- Scopi di monitoraggio, come il controllo dell'accesso degli utenti e le modifiche su certe tabelle sensibili
- Mantenere dei log, per es., sulle tabelle:

```
create trigger LOG_EMP
after insert or update or delete on EMP
begin
        if inserting then
                insert into EMP_LOG values(user,'INSERT',sysdate);
        end if;
        if updating then
                insert into EMP_LOG values(user,'UPDATE',sysdate);
        end if;
        if deleting then
                insert into EMP_LOG values(user,'DELETE',sysdate);
        end if;
end;
```

  usando un trigger di riga, anche i valori degli attributi delle tuple modificate possono essere memorizzati nella tabella EMP_LOG.

- Propagazione automatica di modifiche. Per esempio, se un manager viene trasferito ad un altro dipartimento, un trigger può essere definito per trasferire automaticamente l'impiegato relativo al manager al nuovo dipartimento.

5.2.5 Altro sui Triggers

Se un trigger viene specificato all'interno di una shell SQL*Plus, la definizione deve terminare con un punto "." sull'ultima linea. L'esecuzione del comando **run** provoca la compilazione del trigger

da parte di SQL*Plus. La definizione di un trigger può essere caricata da un file utilizzando il comando **@**. Da notare che l'ultima linea nel file deve essere una barra "/" (slash).

La definizione di un trigger non può essere modificata, può solo essere ricreata utilizzando la clausola **or replace**. Il comando **drop** <nome trigger> cancella un trigger.

Dopo che la definizione di un trigger è stata compilata con successo, il trigger viene automaticamente abilitato. Il comando **alter trigger** <nome trigger> **disable** viene utilizzato per disattivare un trigger. Tutti i trigger definiti su una tabella possono essere (dis)abilitati utilizzando il comando:

alter table <tabelle> **enable** | **disable all trigger***;*

Il data dictionary memorizza informazioni sui triggers nella tabella `USER_TRIGGERS`. L'informazione include il nome del trigger, il tipo, la tabella e il codice per il blocco PL/SQL.

6 ARCHITETTURA DI SISTEMA

Nelle sezioni che seguono discuteremo delle componenti principali dell'architettura DBMS Oracle, versione 7.x (Sezione 6.1) e la struttura fisica e logica del database (Sezioni 6.2 e 6.3). Inoltre vedremo sommariamente come le istruzioni SQL vengono processate (Sezione 6.4) e come gli oggetti di database vengono creati.

6.1 GESTIONE DEI PROCESSI E DELLO SPAZIO SU DISCO

Il DBMS Oracle Server è basato sulla cosiddetta *architettura Multi-Server*. Il server è responsabile della gestione di tutte le attività di database come l'esecuzione di istruzioni SQL, gestione utenti e risorse, gestione dello spazio su disco. Poiché c'è una sola copia del codice di programma per il server DBMS, ad ogni utente connesso al server viene assegnato un server logico separato. La seguente figura illustra l'architettura del DBMS Oracle, costituita da strutture di memorizzazione, processi, e files.

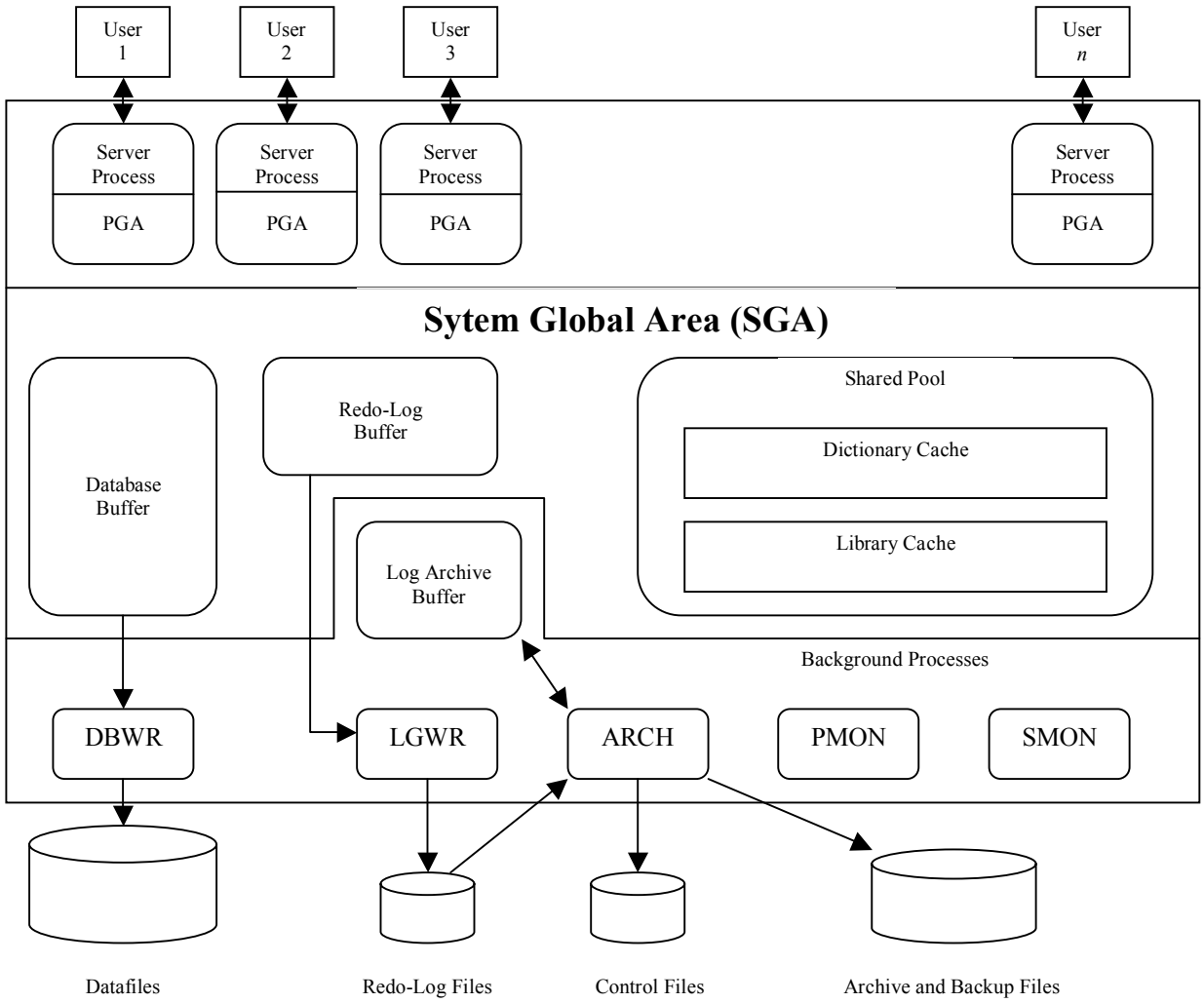

Figura 4: Architettura Oracle 7.x

Ogni volta che un database viene caricato dal server (*instanza di startup*), una porzione della memoria principale del server viene allocata, chiamata ***System Global Area*** (***SGA***). La SGA è costituita dalla *shared pool*, dal *database buffer* e dal *redo-log buffer*. Inoltre, diversi processi in background vengono fatti partire. La combinazione della SGA e dei processi è detta *istanza del database*. La memoria e i processi associati ad un'istanza sono responsabili della gestione efficiente dei dati memorizzati in un database, e permettono agli utenti di accedere al database in maniera concorrente. Il server Oracle può gestire più istanze contemporaneamente, tipicamente ogni istanza è associata con un dominio particolare di una applicazione.

La SGA è quella parte di memoria dove avvengono tutte le operazioni di database. Se molti utenti si connettono alla stessa istanza contemporaneamente, condividono tutti la stessa SGA. Le informazioni memorizzate nella SGA possono essere suddivise nelle seguenti tre aree cache:

Database Buffer. Il buffer del database è una cache nella SGA utilizzata per mantenere i blocchi di dati che vengono letti dai files di dati. I blocchi possono contenere dati di tabelle, dati di indice, ecc. I blocchi di dati vengono modificati nel buffer del database. Oracle gestisce lo spazio disponibile in questo buffer usando un algoritmo "Least Recently Used – LRU – usato meno recentemente". Quando è necessario nuovo spazio libero nel buffer, i blocchi utilizzati meno recentemente vengono scritti sui files di dati. La dimensione del buffer del database ha il maggiore impatto sulle prestazioni globali del database.

Redo-Log Buffer. Questo buffer contiene le informazioni relative alle modifiche dei blocchi di dati nel buffer del database. Mentre il buffer redo-log viene riempito durante le modifiche ai dati, il processo *log writer* scrive le informazioni relative alle modifiche nei files redo-log. Questi files vengono usati, per es., dopo un crash di sistema, allo scopo di ripristinare coerentemente il database (*database recovery*).
Shared Pool. L'area *shared pool* è una parte della SGA che viene usata da tutti gli utenti. I componenti principali di quest'area sono la *dictionary cache* e la *library cache*. Informazioni circa gli oggetti di database sono memorizzati nelle tabelle del data dictionary. Quando le informazioni sono necessarie ad un database, per esempio, per controllare se la colonna specificata in una query esiste, le tabelle del data dictionary vengono lette e i dati recuperati vengono mantenuti nella dictionary cache. Da notare che tutte le istruzioni SQL richiedono l'accesso al data dictionary. Quindi mantenere una rilevante porzione del data dictionary nella cache può incrementare le prestazioni del database. La library cache contiene informazioni sulle più recenti istruzioni SQL eseguite, l'albero di decodifica e il piano di esecuzione della query. Se la stessa istruzione viene eseguita diverse volte, non deve essere decodificata di nuovo e tutte le informazioni relative all'esecuzione dell'istruzione possono essere ritrovate nella library cache.

Ulteriori strutture di memorizzazione nella memoria principale del server sono il buffer *log-archive* (opzionale), e la *Program Global Area* (*PGA*). Il buffer log-archive è utilizzato per inserire temporaneamente nella cache voci del *redo-log* che devono essere archiviati in files speciali. La PGA è l'area in memoria che è utilizzata da un singolo processo utente di Oracle. Contiene l'area di contesto dell'utente (cursori, variabili, ecc.), e informazioni sul processo. La memoria PGA non è condivisibile.

Per ogni istanza di database, esiste un insieme di processi. Questi processi mantengono e rinforzano le relazioni tra la struttura fisica del database e la struttura nella memoria. Il numero di processi varia in dipendenza della configurazione dell'istanza. E' possibile distinguere tra processi utente e processi Oracle. I processi Oracle sono tipicamente processi di background che eseguono operazioni di I/O nel contesto del database.

DBWR. Questo processo è responsabile della gestione del contenuto del buffer del database e della dictionary cache. A questo scopo, DBWR scrive i blocchi modificati in memoria nei files di dati. Il processo scrive i blocchi nei files soltanto se i blocchi che devono essere letti superano la disponibilità attuale della memoria libera nel buffer.
LGWR. Questo processo gestisce la scrittura del contenuto del buffer redo-log nei files redo-log.
SMON. Quando un istanza di database viene avviata, il *processo di monitorizzazione del sistema* (System Monitor Process – SMON) esegue i ripristini necessari (per es., dopo un crash di sistema). Libera il database dalle transazioni abortite e dai relativi oggetti coinvolti. In particolare, questo processo è responsabile della raccolta e "fusione" di *extents* (particolari unità di memoria) contigui in extents più larghi (deframmentazione dello spazio, vedi Sezione 6.2).
PMON. Il *Processo di monitorizzazione del processo* (Process Monitor Process – PMON) si occupa della pulizia dei processi utente falliti che altrimenti rimarrebbero attivi in background occupando risorse e memoria. Analogamente a SMON, PMON si attiva periodicamente per controllare il sistema.
ARCH (opzionale). Il processo LGWR scrive i file redo-log in maniera ciclica. Una volta che l'ultimo file redo-log è riempito, LGWR sovrascrive il contenuto del primo file redo-log. E' possibile eseguire un'istanza di database in modalità *archive-log*. In questo caso il processo ARCH copia le voci dei file redo-log in file di archivio prima che queste ultime vengano sovrascritte da LGWR. E' quindi possibile ripristinare il contenuto di un database a qualsiasi momento temporale dopo che è stata attivata la modalità archive-log.

USER. Il compito di questo processo è comunicare con gli altri processi avviati da programmi come SQL*Plus. Il processo USER è quindi responsabile della trasmissione delle rispettive operazioni e richieste alla SGA o PGA. Questo include, per esempio, la lettura dei blocchi di dati.

6.2 STRUTTURA LOGICA DEL DATABASE

Nell'architettura di un database Oracle si distingue tra la struttura logica di un database e quella fisica. Le strutture logiche descrivono aree logiche di memorizzazione (chiamate spazi) dove gli oggetti come le tabelle possono essere memorizzati. Le strutture fisiche, al contrario, sono determinate dai files del sistema operativo che costituiscono il database.

Le strutture logiche del database includono:

Database. Un database è costituito da una o più divisioni di memorizzazione, chiamate *tablespace*.

Tablespace. Un *tablespace* è una suddivisione logica del database. Tutti gli oggetti del database vengono logicamente memorizzati in tablespace. Ogni database ha almeno un tablespace, il tablespace SYSTEM, che contiene il data dictionary. Altri tablespace possono essere creati e usati per differenti applicazioni o compiti.

Segmenti. Se un oggetto di database (per esempio, una tabella o un cluster) viene creato, viene allocata automaticamente una porzione di tablespace. Questa porzione è chiamata *segmento*. Per ogni tabella esiste un segmento di tabella. Per gli indici vengono allocati i cosiddetti *segmenti di indice*. Un segmento associato con un oggetto di database è relativo ad esattamente un tablespace. Cioè, un segmento non può essere collocato tra due tablespace.

Extents. Un extent è la più piccola unità logica di memorizzazione che può essere allocata per un oggetto di database, e consiste in una sequenza contigua di blocchi di dati. Se la dimensione di un oggetto di database cresce (per es., a causa di inserimenti di tuple in tabelle), un extent aggiuntivo viene allocato per quell'oggetto. Informazioni circa l'allocazione degli extents per gli oggetti di database può essere trovata nella vista del data dictionary `USER_EXTENTS`.

Un tipo speciale di segmenti sono i *segmenti di rollback*. Essi non contengono oggetti di database, ma contengono una "immagine" di come erano i dati prima di una modifica, per i quali la transazione non è stata ancora confermata (**commit**). Le modifiche vengono annullate utilizzando i segmenti di rollback. Oracle utilizza i segmenti di rollback in modo da mantenere una coerenza in lettura dei dati attraverso diversi utenti simultaneamente connessi. Inoltre, i segmenti di rollback vengono utilizzati che ripristinare l'immagine "precedente" delle tuple modificate in caso di evento esplicito di rollback in una transazione di modifica dei dati.

Tipicamente, un tablespace extra (RBS) viene utilizzato per memorizzare i segmenti di rollback. Questo tablespace può essere definito durante la creazione di un database. La dimensione di questo tablespace e dei suoi segmenti dipende dal tipo e dimensione delle transazioni che sono tipicamente eseguite dai programmi.

Un database è costituito tipicamente da un tablespace SYSTEM, che contiene il data dictionary e altre tabelle interne, procedure, ecc. e un tablespace per i segmenti di rollback. Ulteriori tablespace includono un tablespace per i dati dell'utente (USERS), un tablespace per i temporanei risultati di query e tabelle (TEMP), e un tablespace utilizzato dalle applicazioni come SQL*Forms (TOOLS).

6.3 STRUTTURA FISICA DI UN DATABASE

La struttura fisica di un database Oracle è costituita da *files* e *blocchi* di dati:

Files. Un tablespace è costituito fisicamente da uno o più files del sistema operativo che vengono memorizzati su disco. Quindi un database è essenzialmente una collezione di files di dati che può essere memorizzata su diversi dispositivi di archiviazione (nastri, dischi ottici, ecc.). Tipicamente, sono utilizzati solo i dischi magnetici. Più files di dati per un tablespace permettono al server di distribuire un oggetto di database su più dischi (dipende ovviamente anche dalla grandezza dell'oggetto).

Blocchi. Un extent è costituito da uno o più blocchi di dati Oracle contigui.Un blocco determina il più piccolo livello di granularità dove i dati possono essere memorizzati. Un blocco di dati corrisponde ad uno specifico numero di bytes di spazio fisico su disco. La dimensione di un blocco di dati è specificata per ogni database Oracle quando questo viene creato. Un database utilizza e alloca spazio libero in unità di blocchi di dati. Le informazioni sui blocchi di dati possono essere ritrovate nel data dictionary attraverso la vista USER_SEGMENT e USER_EXTENTS. Queste viste mostrano quanti blocchi sono allocati per un certo oggetto di database e quanti blocchi sono disponibili (liberi) in un segmento/extent.

Come menzionato nella Sezione 6.1, oltre ai file di dati, altri tre tipi di dati sono associati ad una istanza di database:

Redo-Log Files. Ogni instanza di database mantiene un insieme di files redo-log. Questi files sono utilizzati per memorizzare i log di tutte le transazioni. I log vengono usati per ripristinare le transazioni in un database nel relativo ordine nel caso di un crash del database (l'operazione di ripristino viene chiamata *roll forward*). Quando una transazione viene eseguita, le modifiche vengono scritte nel buffer redo-log, mentre i blocchi interessati dalle transazioni non vengono immediatamente scritti su disco, permettendo quindi l'ottimizzazione della performance attraverso operazioni di scrittura in background.
Control Files. Ogni istanza di database ha almeno un *file di controllo* (*control file*). In questo file vengono registrati il nome dell'istanza del database e la locazione (sui dischi) dei files di dati e dei files redo-log. Ogni volta che un'istanza parte ed è inizializzata, i files di dati e di redo-log vengono determinati utilizzando il control file (o i control files).
Archive/Backup Files. Se un'istanza sta lavorando in modalità archive-log, il processo ARCH archivia le modifiche dei files di redo-log in un archivio extra o in files di backup. Al contrario dei files redo-log, questi files normalmente non vengono sovrascritti.

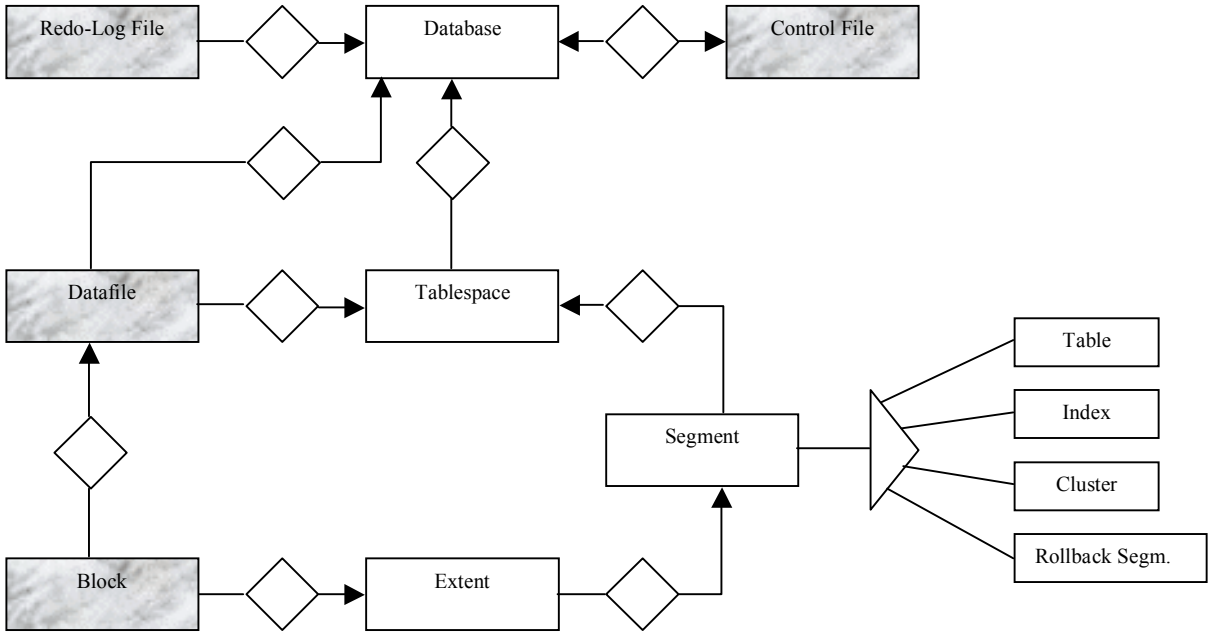

Figura 5: Relazione tra struttura fisica e logica di un database

6.4 ANALISI DEL PROCESSO DI ESECUZIONE DI UN'ISTRUZIONE SQL

Nella parte seguente analizzeremo sommariamente come viene processata un'istruzione SQL dal server Oracle e come i processi e i buffer sono coinvolti.

1. Si assuma che un utente (lavorando con SQL*Plus) esegua un'istruzione **update** nella tabella `TAB` in modo che più di una tupla venga interessata dall'aggiornamento. L'istruzione viene passata al server dal processo USER. Quindi il server (o più raramente il processore di query) controlla se l'istruzione è già contenuta nella library cache, in modo che le corrispondenti informazioni (albero di decodifica, piano di esecuzione) possano essere riutilizzate. Se l'istruzione non viene trovata, viene decodificata e dopo aver verificato l'istruzione (privilegi utente, tabelle e colonne interessate) utilizzando i dati dalla cache del data dictionary, viene generato un piano di esecuzione della query dall'ottimizzatore di query. Insieme all'albero di decodifica, questo piano viene memorizzato nella library cache.

2. Per gli oggetti interessati dall'istruzione (in questo caso la tabella `TAB`) viene controllato se i corrispondenti blocchi di dati esistono nel buffer del database. In caso negativo, il processo USER legge i blocchi di dati dai files fisici al buffer del database. Se non c'è abbastanza spazio nel buffer, i blocchi di altri oggetti utilizzati meno recentemente vengono scritti su disco dal processo DBWR.

3. Le modifiche delle tuple interessate dall'**update** avvengono nel buffer del database. Prima che i blocchi di dati vengano modificati, una "immagine" delle tuple viene scritta sui segmenti di rollback dal processo DBWR.

4. Mentre il buffer redo-log viene riempito a causa delle modifiche ai blocchi, il processo LGWR scrive sui file di redo-log le voci presenti nel redo-log buffer.

5. Dopo che tutte le tuple (o i corrispondenti blocchi di dati) sono state modificate nel buffer del database, le modifiche possono essere confermate dall'utente attraverso un comando **commit**.

6. Finchè il comando **commit** non è stato inviato dall'utente, le modifiche possono essere annullate utilizzando il commando **rollback**. In questo caso, i blocchi modificati nel buffer del database vengono sovrascritti dai blocchi originali memorizzati nei segmenti di rollback.

7. Se l'utente invia un comando **commit**, lo spazio allocato per i blocchi nel segmento di rollback può essere liberato e reso quindi disponibile per altre transazioni. Inoltre, i blocchi modificati nel buffer del database vengono sbloccati in modo che gli altri utenti possano ora leggere e modificare quei blocchi. La fine della transazione (più precisamente il **commit**) viene registrato nei files redo-log. I blocchi modificati vengono scritti su disco solo se si rende necessario altro spazio per altre transazioni.

6.5 CREAZIONE DI OGGETTI DI DATABASE

Per gli oggetti di database (tabelle, indici, clusters) che richiedono un proprio spazio di archiviazione, viene creato un segmento in un tablespace. Dato che il sistema tipicamente non conosce la dimensione che l'oggetto del database avrà, vengono utilizzati alcuni parametri di default. L'utente, comunque, ha la possibilità di specificare esplicitamente i parametri di memorizzazione utilizzando la clausola **storage** all'interno dell'istruzione **create table**. Questa specifica sovrascrive i parametri di sistema e permette all'utente di specificare la dimensione (prevista) dell'oggetto in termini di extents.

Supponiamo di avere la seguente definizione di tabella che include la clausola **storage**:

```
create table STOCKS
      (ITEM       varchar2(30),
       QUANTITY   number(4))
storage (initial 1M next 400k
         minextents 1 maxextents 20 pctincrease 50);
```

initial e **next** specificano la dimensione del primo extent e di quelli successivi, rispettivamente. Nella definizione sopra, l'extent iniziale ha la dimensione di 1Mb, e il successivo extent ha la dimensione di 400k. Il parametro **minextents** specifica il numero totale di extents allocati quando il segmento viene creato. Questo parametro permette all'utente di allocare una grande quantità di spazio quando un oggetto viene creato, anche se lo spazio disponibile non è contiguo. Il minimo valore (e default) è 1. Il parametro **maxextents** specifica il numero massimo ammissibile di extents. Il parametro **pctincrease** specifica la percentuale con la quale ogni extent dopo il secondo cresce rispetto all'extent precedente. Il valore di default è 50, il che significa che ogni extent successivo al secondo cresce del 50% rispetto all'extent precedente. Basandosi sulla definizione della tabella sopra, avremo quindi la seguente struttura logica per la tabella `STOCKS` (assumendo che 4 extents siano stati già allocati):

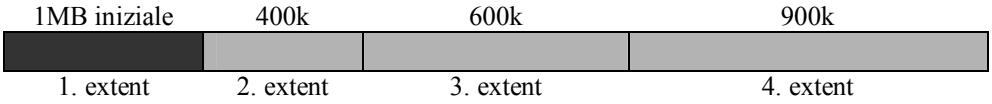

Figura 6: Struttura logica della tabella STOCKS

Se lo spazio richiesto per un oggetto di database è conosciuto prima della sua creazione, l'extent iniziale dovrebbe essere grande abbastanza per contenere l'oggetto. In questo caso, Oracle Server (più precisamente il gestore delle risorse) cerca di allocare blocchi di dati contigui su disco per questo oggetto, prevenendo quindi la deframmentazione dei blocchi associati ad un oggetto.

Per la memorizzazione degli indici, la clausola storage può essere altrettanto specificata:

```
create index STOCK_IDX on STOCKS(ITEM)
        storage (initial 200k next 100k
                minextents 1 maxextents 5);
```