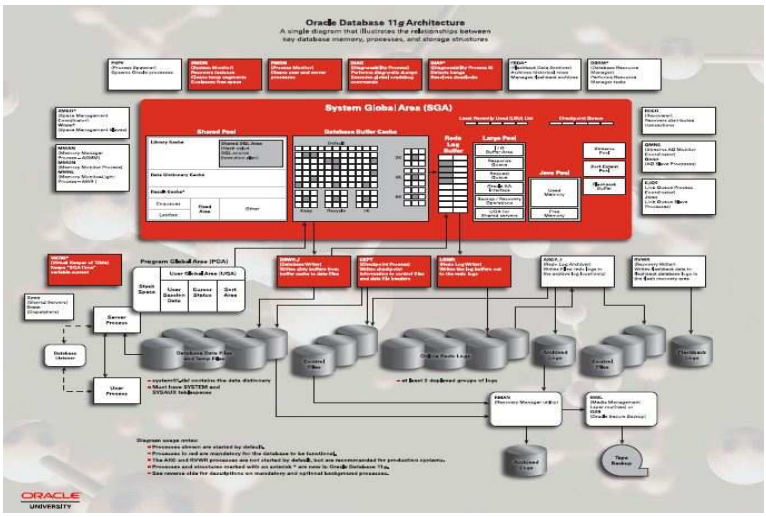
UNIVERSITA' DEGLI STUDI DI PISA
FACOLTA' DI INGEGNERIA
Corso di Laurea in Ingegneria Biomedica
Informatica Medica – Strumenti informatici per la medicina

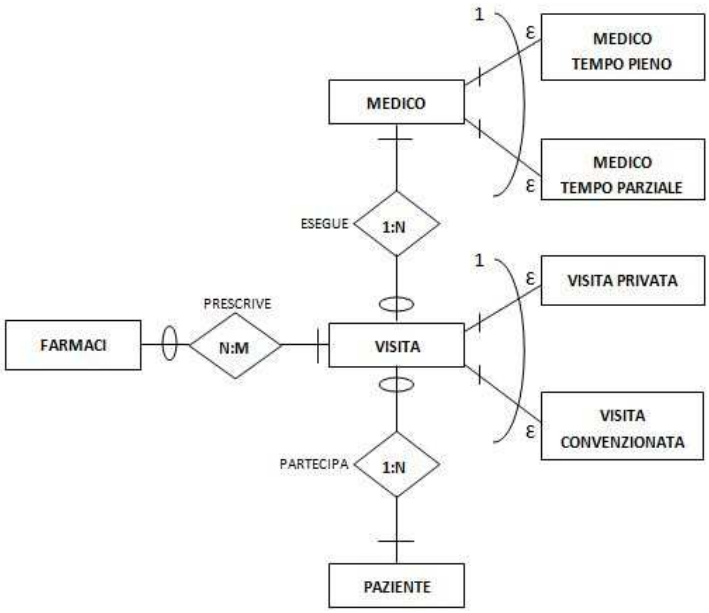
Dispensa

Stato del documento: in lavorazione

Oracle - Un sistema operativo



per la gestione di basi di dati



Maurizio Mangione

Sommario

1. INTRODUZIONE

Il termine **database, DB,** (o **base di dati, BD:** entrambi i termini verranno utilizzati indifferentemente nei presenti appunti) è entrato nell'uso comune in Informatica (intorno al 1964) per denotare una raccolta di grandi quantità di dati archiviati su supporto magnetico secondo regole precise e generali. Esso rappresenta l'evoluzione di archivi "cartacei" permettendo all'utente di accedere velocemente, attraverso l'utilizzo dei calcolatori, ai dati di interesse. Per estensione, il termine database viene riferito sia alla raccolta dei dati che al sistema software per la generazione /consultazione della struttura in cui i dati sono contenuti.

1.1 ORACLE: DBMS

1.2 Database ed istanza

Il sistema per la gestione di basi di tati Oracle è composto fondamentalmente da due strutture, **il database e l'istanza**. Con il termine database (DB) si indicano i file fisici in cui sono memorizzati i dati, mentre per istanza si intende l'insieme delle aree di memoria e dei processi di background necessari ad accedere ai dati, ovvero al DB.

L'architettura del server è complessa: ogni area di memoria nell'istanza contiene dati che non sono contenuti in altre aree di memoria ed i processi di background hanno compiti ben precisi, tutti diversi fra loro.

Ogni DB deve obbligatoriamente fare riferimento almeno ad un'istanza, è anche possibile avere più di un'istanza per database.

Ciascun DB consiste di strutture logiche di memorizzazione, per immagazzinare e gestire i dati, (tabelle, indici, etc.) e di strutture fisiche di memorizzazione che contengono le strutture logiche.

I servizi offerti dalle strutture logiche del server sono indipendenti dalle strutture fisiche che le contengono. Questo perché le strutture logiche possano essere progettate nello stesso modo indipendentemente dall'hardware e dal sistema operativo impiegati.

Quindi sia che si abbia un'installazione di server Oracle su sistema operativo Microsoft, Linux o Solaris non troveremmo alcuna differenza nella progettazione delle strutture logiche di memorizzazione.

1.3 Strutture logiche di memorizzazione

1.4 Tablespace

Al vertice della scala gerarchica troviamo i Tablespace, strutture che raggruppano, a loro volta, quelle di livello più basso. È ammissibile suddividere i dati in differenti tablespace in base ad uno specifico significato logico dettato dalla nostra applicazione che interagirà con il database. Per esempio si può creare un tablespace per ciascun utente che andrà a memorizzare propri dati nel database.

Questa suddivisione logica ha l'enorme vantaggio di consentire l'amministrazione di una porzione limitata del DB, ovvero di un tablespace, senza intaccare la funzionalità delle parti rimanenti. Ogni database deve avere uno o più tablespace. Come requisito minimo per la creazione di un DB Oracle crea sempre una serie di tablespace di sistema (ad esempio crea il tablespace denominato SYSTEM).

1.5 Block

Un blocco dati è la più piccola unità di memorizzazione in Oracle, pertanto indivisibile, e corrisponde ad un numero di byte scelto dal DBA durante la creazione del database. È sempre multiplo della capacità del blocco usato dal sistema operativo su cui si opera.

1.6 Extent

È composto di un numero specifico di blocchi contigui di dati (block).

1.7 Segment

È formato da un insieme di extent. Ne sono un esempio le tabelle o gli indici. Ogni qualvolta si crea un segment Oracle alloca al suo interno almeno un extent che, a sua volta, contiene almeno un block. Tutto questo è sempre deciso dal DBA (data base administrator). Un segment può essere associato esclusivamente ad un tablespace.

Figura 1. Relazioni tra tablespace, segment, extent e block

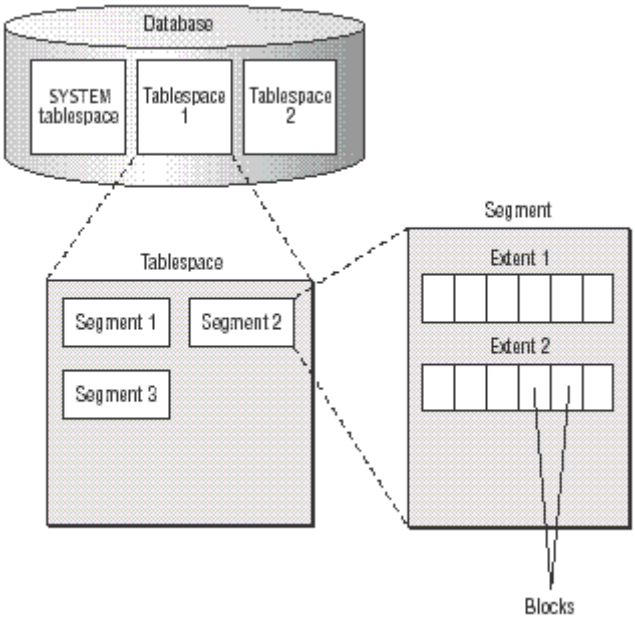


Figura 1 Strutture fisiche di memorizzazione

1.8 Data file

Cosi come si può evincere dal loro stesso nome i data files sono File che contengono tutti i dati del DB. Ogni database Oracle deve avere uno o più data file. Ciascun data file è associato esclusivamente ad un tablespace, ma un tablespace può essere formato anche da più di un data file. In fase di installazione ORACLE crea i data files di dafault (almeno uno per il tablespace SYSTEM).

1.9 Redo log file

Registrano tutte le modifiche occorse ai dati. Ogni DB possiede almeno due file di redo log, perché Oracle scrive in questi file in maniera circolare: quando un file di redo log è pieno allora Oracle scrive in quello successivo, quando l'ultimo file di redo log è pieno allora Oracle ricomincia dal primo, assicurandosi però di memorizzare le informazioni nei data file prima di sovrascriverle.

Se una qualsiasi anomalia non permette la scrittura delle modifiche occorse al database nei rispettivi data file, allora possiamo tranquillamente ottenere tali modifiche dai redo log file. Le modifiche ai dati, pertanto, non sono mai perse.

I redo log file rappresentano la cronologia delle modifiche ai dati e sono di vitale importanza per l'integrità del DB. Oracle permette di avere più di una copia per ciascun redo log file: questa importante caratteristica è denominata **multiplexing dei redo log file**.

1.10 Control file

Ogni database ha almeno un control file che contiene informazioni circa la struttura fisica del DB, ovvero il nome, la data e l'ora di creazione e il percorso completo di ciascun data file e redo log file. È di vitale importanza e si consiglia di configurarne più di una copia: anche in questo caso parleremo di multiplexing di control file.

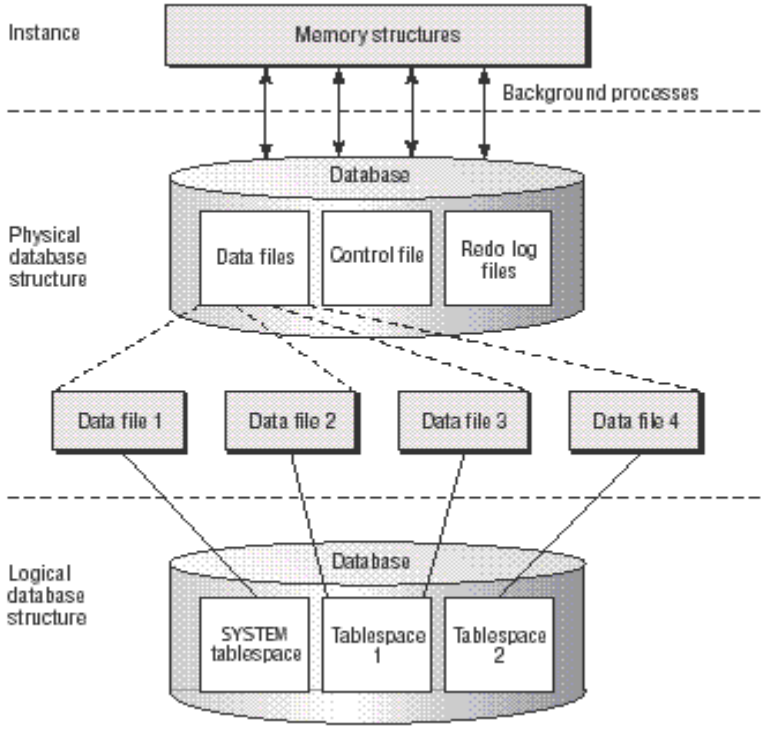


Figura 2 Relazioni fra le strutture fisiche e logiche di un DB

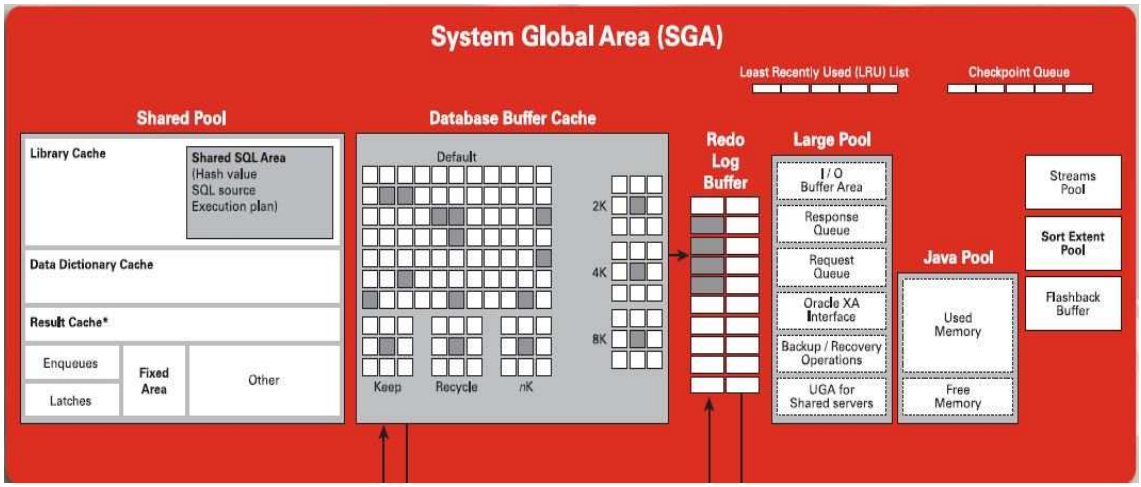
1.11 Aree di memoria in Oracle

Le aree di memoria di un server Oracle sono usate per contenere i dati, le informazioni del dizionario dei dati, i comandi SQL, il codice PL/SQL e tanto altro ancora. Tali aree sono allocate nell'istanza Oracle ogni qualvolta questa è fatta partire e deallocata, vale a dire rilasciata, quando la stessa è "terminata".

Le due maggiori strutture di memoria sono la System Global Area (SGA) e la Program Global Area (PGA), quest'ultima non sarà trattata in questa guida.

1.12 System Global Area

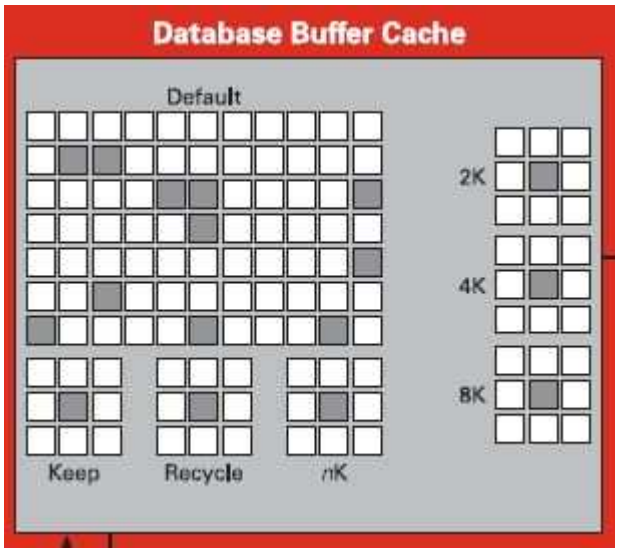
È un'area di memoria condivisa, il che significa che tutti gli utenti del DB ne condividono le informazioni. È composta a sua volta di più aree di memoria ed ha uno spazio allocato all'avvio di un'istanza definito dal parametro SGA_MAX_SIZE (consultare la documentazione per



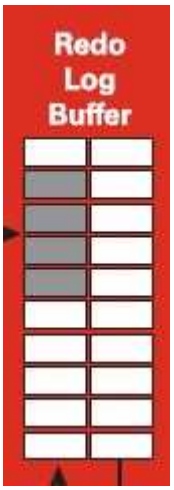
maggiori dettagli).

L'SGA e i processi di background formano l'istanza Oracle. Come si può vedere in figura quest'area è formata:

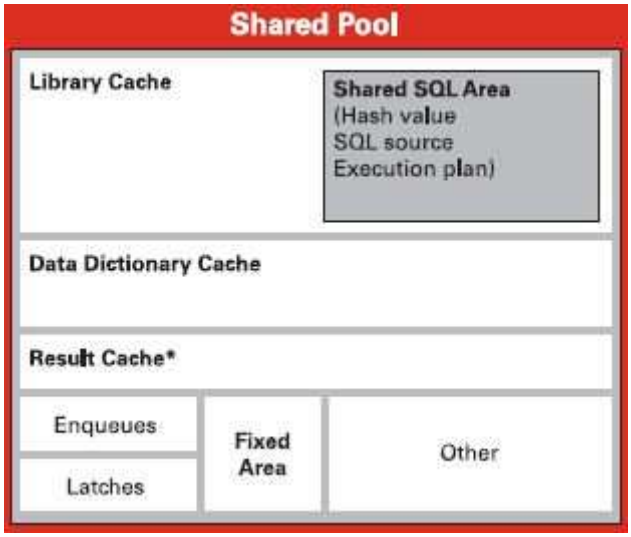
dal Database Buffer Cache,



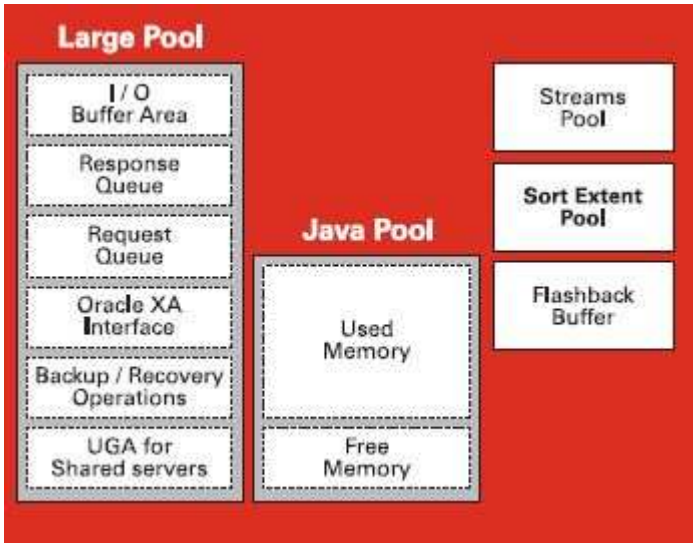
dal Redo Log Buffer



dallo Shared Pool



dal Large Pool e dal Java Pool



due code di memoria per ciascuna istanza.

1.13 PGA (Program Global Area)

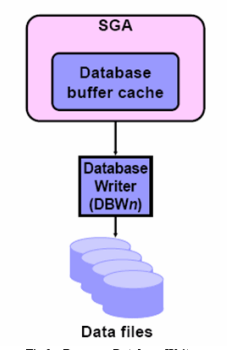
La PGA è un'area di memoria che contiene i dati di un singolo processo utente ed è allocata da Oracle quando un utente si collega al database e una sessione viene creata.

1.14 I processi di background di Oracle

Un processo è un algoritmo sotto forma diprogramma, eseguito dal sistema operativo per svolgere alcune attività. Oracle inizializza diversi processi di background per ogni istanza. Ogni processo di background è responsabile nel portare a termine specifici compiti. Non tutti i processi sono presenti in un'istanza Oracle: la maggior parte di essi sono obbligatori, la presenza di altri dipende dal tipo di architettura Oracle desiderata.

Di seguito si elencano i processi necessari allo start-up di un istanza del database.

DBWR (DataBaseWRiter): è un processo di background che scrive i blocchi che vengono modificati nella SGA nei tablespace; solitamente vengono scritti prima i blocchi usati meno di recente;



quando si esegue un COMMIT (comando SQL per rendere effettive le modifiche su database) i cambiamenti sono memorizzati entro i file del redo log e quindi resi persistenti, ma non necessariamente vengono subito scritti anche nei tablespace;

Il processo del server registra i cambiamenti e i data block nel buffer di cache del database. DBWn scrive i buffer rovinati dal database buffer cache ai data file e assicura un sufficiente numero di buffer liberi che sono disponibili al database buffer cache.

I processi del server effettuano cambiamenti solo nel buffer cache del database aumentando così le performance del database.

Checkpoint e Redo

Checkpoint e Redo sono operazioni legate strettamente tra loro e i processi che eseguono sono a loro volta coordinati tra di loro.

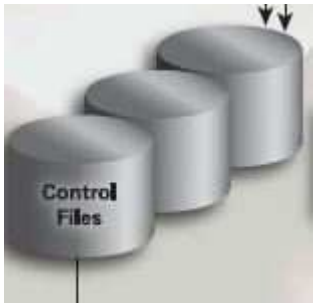
Gli obiettivi sono:

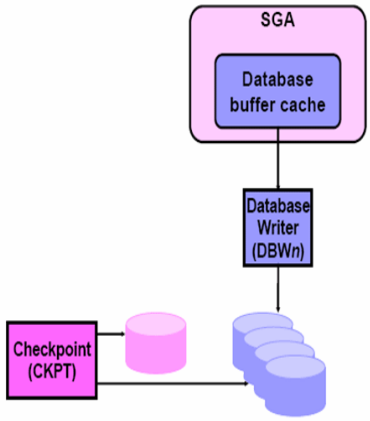
per il Checkpoint:
• Trasferire i dati modificati dalla System Global Area (SGA) al disco, memorizzare i Data Block che sono stati trasferiti nei dischi e richiedere il redo per il recovery.
• Creare buffer disponibili per i data block. In questo caso i buffer vengono segnati come liberi.
• Controllare il tempo medio di recupero (MTTR – Mean Time To Recovery). Il numero dei buffer cambiati ma non ancora copiati nel disco determina il tempo di recupero dell'istanza.

per il Redo:
• Fornire i dati non committati sul disco per le operazioni di rollback

• Fornire i sorgenti per completare il recovery. Se si sta usando la modalità ARCHIVELOG, si terrà traccia del contenuto dei redo log file (RLF), così che si attiva il processo ARCn che esegue una copia degli RLF prima che vengano sovrascritti.Lo stato di default del database è NOARCHIVELOG

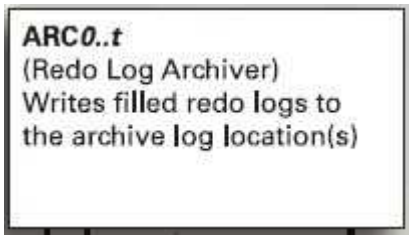
Processo **CKPT** (Checkpoint):gestisce l' evento del database durante il quale viene salvato, sui file di pertinenza, tutto quello che c'è in memoria. Questa operazione è utilissima, in quanto dopo un crash, Oracle esegue il recovery automatico partendo dall'ultimo checkpoint effettuato (checkpoint position). Quindi più spesso si verifica l'evento checkpoint e minore sarà il tempo necessario per eseguire il recovery. Il checkpoint consuma risorse, pertanto se si verifica troppo spesso, il database perderà in performance. Di solito viene effettuato almeno ogni 3 secondi. L'obiettivo di un checkpoint è di identificare la posizione nel Redo log file dove l'istanza di recovery è iniziata (chiamata "checkpoint position"). In caso di uno switch del log, il processo di CKPT scrive anche le informazioni di questo checkpoint nell'intestazione dei data file



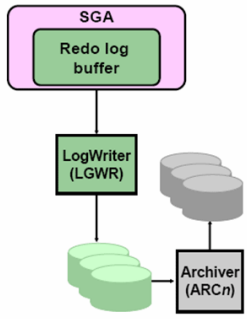


Le funzionalità dei Checkpoint sono:-

- Controllare che i data block modificati nella memoria vengano scritti nel disco regolarmente, così che i dati non vengano persi in caso di un guasto, provocato sia da parte del sistema sia da parte del database.
- Ridurre il tempo di richiesta per il recovery di un'istanza.
- Controllare che tutti i dati salvati vengano scritti nei data file durante lo spegnimento.



Archiver (**ARCn**) è un processo opzionale che gira in background. e gestisce la copia automatica dei file del Redo Log in un'apposita area di salvataggio E' fondamentale per recuperare un db dopo la perdita di un disco. Come i redo log file vengono popolati online, cosi l'istanza di Oracle inizia a scrivere nel successivo redo log file. Si definisce "switch di un log " il processo che permette di passare da un redo log file ad un'altro. Una delle decisioni importanti che un DBA deve prendere è configurare il database per operare nella modalità ARCHIVELOG o NOARCHIVELOG.



• NOARCHIVELOG, i redo log file online vengono sovrascritti ogni volta che avviene un cambiamento del log

• ARCHIVELOG, i gruppi inattivi dei redo log file online pieni devono essere archiviati prima che essi possano essere riusati.

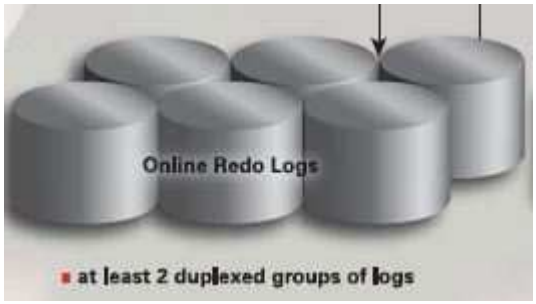
LGWR (LoG WRiter): è un processo di background che scrive i cambiamenti dal redo log buffer della SGA entro i file del redo log; durante un COMMIT i cambiamenti sono memorizzati entro i file del redo log; solo quando tutti i cambiamenti sono stati salvati nel file, viene segnalato il successo del COMMIT;

L'insieme dei redo log file online per una particolare istanza viene chiamata redo thread e funziona con un buffer ciclico.

Il redo è indicizzato dal Redo Block Address (RBA). L'RBA checkpoint è l'indirizzo dal quale l'applicazione inizia a riprendere il redo nell'evento di un crash dell'istanza. Questo indirizzo è chiamato anche checkpoint position.

Redo Architecture

Redo è stato creato per avere impatti minimi sulle performance.Redo logging è una caratteristica del DB Oracle. Redo è progettato in relazione ai checkpoint per fornire un meccanismo performante per proteggere il database dalla corruzione o dalla perdita dei dati da un fallimento di un'istanza. I checkpoint memorizzano lo starting point nei log di redo per il recovery dell'istanza, i record di redo contengono le modifiche necessarie per essere applicate ai data file e per ricostruire le informazioni dall'ultima transazione committata prima del failure dell'istanza.I Redo Record sono piccoli vettori che contengo le righe che sono cambiate e non l'intero blocco di dati. I processi del server effettuano la scrittura "memory to memory" di questi record.



LGWR scrive blocchi pieni se è possibile dal buffer di log (memory) al file di log (disk) per fare spazio a più record di redo nel buffer log. Quando la modalità ARCHIVELOG è attivata, ad ogni variazione del log, l'Archiver Process (ARCn) copierà i redo log file in un unico archivio di file di log.

Gli Archiver Process effettuano copie disco a disco e scrivono una serie di tre tipi di processi. Se gli Archiver Process non possono pulire un file di log abbastanza velocemente, LGWR attende fino a quando l'archiver non avrà completato. LGWR non può sovrascrivere un log file fino a quando non è stato archiviato. Quando il LGWR non può pulire lo spazio per il log buffer necessario ai processi del server, che scrivono i record di redo, l'utente in sessione è costretto a rimanere in attesa.

- **SMON** (System MONitor): è un processo di background che gestisce l'istanza; tale processo ha il compito di risolvere e segnalare le condizioni di deadlock fra le transazioni, provvedere al rilascio dei segmenti temporanei non più in uso, provvedere al ricompattamento dei tablespace, riunendo insieme le aree libere frammentate;

- **PMON** (Process MONitor): è un processo di background che gestisce i processi utente; in caso di fallimento di un processo durante una transazione provvede al rollback (meccanismo di ripristino dei vecchi dati), ripristinando per i dati coinvolti nella transazione la situazione preesistente all'inizio della transazione stessa;

- **LCK** (Lock) compie l'azione del locking nelle transazioni intra-istanza;
- **MMON** (Memory MONitor) controlla lo status della SGA;
- **MMAN** (Memory MANager) gestisce dinamicamente la SGA ridistribuendo la memoria tra le sue componenti quando necessario;
- il Recoverer (**RECO**): è un processo di background che risolve i fallimenti di transazioni distribuite;
- il Dispatcher (**Dnnn**)
- lo Shared Server (**Snnn**)

ecc.

2. ESERCITAZIONE

2.1 Installazione di Oracle Server

2.2 Installazione della parte Client

2.3 Creazione di un database

2.4 Fondamenti di amministrazione DB (compreso db backup)

2.5 Creazione di un Utente ed uno schema

2.6 Oggetti del database (tables, views, triggers, synonyms, ...).

3. PL/SQL

3.1 Cenni Storici

PL/SQL è stato introdotto per la prima volta nel **1991** con Oracle 6.0. La prima versaione di PL/SQL, PL/SQL 1.0, aveva un numero molto limitato di funzionalità procedurali. Uno dei punti di forza fu la capacità di processare insieme sequenze multiple di Statement SQL e costrutti procedurali.

PL/SQL ha le origini in ADA, un linguaggio di programmazione ad alto livello.

Ad esempio il PL/SQL riprende il concetto di blocchi di strutture di ADA mediante il blocco di istruzioni SQl racchiuse da un BEGIN…END, oppure altre caratteristiche, come la sintassi "=" utilizzate per il confronto e ": =" utilizzato per l'assegnazione, la gestione delle eccezioni, e la sintassi dichiarativa di definire sottoprogrammi memorizzati, ecc.

Nel corso degli anni, Oracle ha rilasciato PL / SQL 2.0, che ha trasformato il database ORACLE in un database Oracle attivo (in un sistema per la gestione di basi di dati) con la capacità di memorizzare nel database oltre ai dati anche la logica di business dell'applicazione, sotto forma di stored procedure, funzioni, e package. Ha inoltre definito la possibilità di dichiarare i record definiti dal programmatore e matrici in forma di PL / SQL tabelle. Negli anni successivi si sono susseguite le versioni con conseguenti innovazioni introducendo tra le altre anche le seguenti caratteristiche:

- l'uso delle funzioni memorizzate nelle istruzioni SQL ed ha introdotto per la prima volta l'uso di SQL dinamico (con il pacchetto DBMS_SQL);
- la capacità di definire i wrapper binari per PL / SQL sottoprogrammi memorizzati, nascondendo così il codice di altri sviluppatori;
- di accedere al file system all'interno del database. (utilizzando il pacchetto UTL_FILE);
- di includere come nativo SQL dinamico, Java stored procedure, i trigger;
- l'introduzione di compilazione nativa di codice PL/SQL, la valorizzazione di PL/SQL cursori in forma di espressione, nuovi tipi di dati, una vera eredità tra gli oggetti, ed altro ancora;
- PL/SQL incorpora un linguaggio di terza generazione (3GL) non disponibile con il solo SQL (Structured Query Language SQL è un linguaggio di quarta generazione (4GL), nel senso che utilizza costrutti e gli elementi che specificano il "cosa fare" senza dover specificare "come fare".

3.2 I vantaggi del PL/SQL

PL / SQL è un linguaggio a blocchi strutturato che consente di incapsulare la logica di business in un unico luogo. Il codice PL / SQL viene eseguito sul lato server, fornendo in tal modo l'interazione diretta con il database e il motore SQL.

Il principale vantaggio di PL / SQL è la sua capacità di definire e attuare 3GL con istruzioni SQL "embedded". Con l'utilizzo di PL/SQL, è possibile implementare le funzionalità standard 3GL, come il supporto per i tipi di dati BOOLEAN; sequenziale, condizionale, la logica iterativa, matrici e sottoprogrammi, caratteristiche object-oriented. PL / SQL è il linguaggio primario procedurale per Oracle ed i suoi strumenti client-side come Oracle Forms e Oracle Reports.

Portabilità I programmi scritti in PL/SQL sono indipendenti dal hardware e dai sistemi operativi. Essi sono altamente portatili e funzionano bene su qualsiasi piattaforma in cui sono installati un server Oracle e di un ambiente di sviluppo integrato (IDE).

Modularità Il PL/SQL permette di scrivere programmi come moduli indipendenti che si possono integrare. È possibile implementare questa modularità, utilizzando funzioni come procedure, funzioni, e pacchetti.

Performance PL / SQL offre prestazioni migliori per tre motivi principali:

- non è necessaria la conversione del tipo di dati su operazioni di input e output. Quando i dati vengono recuperati in variabili PL/SQL o quando i dati vengono immessi in Oracle con variabili PL/SQL, il PL/SQL si occupa della conversione appropriata tra il server Oracle ed il PL/SQL per mezzo dello stesso formato del database interno. Altri programmi 3GL scritti in C++ o in Java per accedere a un database Oracle necessitano di una conversione dei dati da e verso il formato Oracle.
- Generalmente, le applicazioni di database eseguite in un ambiente client-server vengono impiegate sia in un' architettura a due livelli (Oracle RDBMS sul server e il codice PL / SQL applicazione sul client) o in una tre livelli (un database server, un application server, e un livello di presentazione su un client). Con il PL/SQL, è possibile raggruppare un insieme di istruzioni SQL (insieme con la logica dell'applicazione) in un blocco PL/SQL e si può presentare l'intero blocco sul server Oracle. Questo riduce il traffico di rete in una struttura di applicazione a due livelli o tre livelli e migliora quindi le prestazioni.
- PL/SQL 9i introduce la compilazione nativa di codice PL/SQL, con la successiva conversione in C e lo stoccaggio in codice macchina. Questo si traduce in "esecuzioni più veloci".

Object Oriented PL / SQL fornisce funzionalità object-oriented come l'incapsulamento, l'astrazione (la possibilità di definire tipi di dati astratti), riusabilità, ereditarietà, polimorfismo ecc.

3.3 Costruzione di un Blocco di istruzioni PL/SQL (BEGIN END)

L'elemento di base di codifica del PL/SQL è il blocco. Un blocco PL/SQL imita il concetto di blocco di strutturazione, intrinseca nei linguaggi di programmazione strutturata come Pascal. Un tipico blocco PL/SQL inizia con una istruzione BEGIN e termina con una istruzione END seguito da un punto e virgola (;).
Ci sono tre tipi di blocchi PL / SQL:

- blocco "Anonimo ": Questa è una sezione di codice racchiuso all'interno di istruzioni BEGIN e END. Non ha un nome ad esso associato.
- blocco "*Etichettato* ": Questo è un blocco di istruzioni PL/SQL identificato da un'etichetta. Si inizia con una etichetta di PL/SQL seguita da un istruzione BEING ed un'istruzione END.
- blocco "*Denominato* ": Questo è un blocco di istruzioni PL/SQL memorizzati nel database come un sottoprogramma e identificato da un nome univoco. Questo blocco è attuato mediante sottoprogrammi e trigger.

Un blocco PL / SQL è costituito dalle seguenti sezioni definite all'interno di esso:

- sezione "Dichiarazione": Questa sezione inizia con la parola chiave DECLARE e contiene le dichiarazioni di variabili, costanti, i cursori e sottoprogrammi locali. Questa sezione è facoltativa per un blocco.
- sezione "Eseguibile": Questa sezione inizia con una dichiarazione BEGIN e si compone di logica procedurale e istruzioni SQL. Si conclude con un'istruzione END.
- sezione "Gestione delle eccezioni": Questa sezione si apre con l'eccezione quando le parole chiave EXCEPTION e contiene la logica per la gestione di PL / SQL e / o errori del server Oracle.

Esempio:

```
SET serveroutput ON;

CREATE TABLE items_tab (item_code varchar2(6) PRIMARY KEY,
                        item_descr varchar2(20) NOT NULL);

DECLARE
   v_item_code varchar2 (6);
   v_item_descr varchar2 (20);
BEGIN
   v_item_code := 'ITM101';
   v_item_descr := 'Spare parts';
   INSERT INTO items_tab VALUES (v_item_code, v_item_descr);
EXCEPTION WHEN OTHERS THEN
   dbms_output.put_line(SQLERRM);
END;
```

Eventuali errori che si verificano nell'esecuzione processo genereranno un uscita al buffer dello schermo con un messaggio di errore. Questo viene realizzato mediante una chiamata a una procedura speciale di nome dbms_output.put_line, una procedura che confeziona le visualizzazioni a schermo *SQL Plus con un corrispondente messaggio passato come input ad esso.

Esempio di un blocco annidato

```
DECLARE
   v_item_code          VARCHAR2(6);
   v_item_descr         VARCHAR2(20);
   v_num                NUMBER(1);
BEGIN
   v_item_code := 'ITM101';
   v_item_descr:= 'Cotone idrofilo';

   BEGIN
        SELECT          1
        INTO            v_num
        FROM            items_tab
        WHERE           item_code = v_item_code;
   EXCEPTION
        WHEN NO_DATA_FOUND THEN
             v_num := 0;
        WHEN OTHERS THEN
               dbms_output.put_line('Error in SELECT: '||SQLERRM);
       RETURN;
   END;

   IF (v_num = 0) THEN
     INSERT INTO items_tab VALUES (v_item_code, v_item_descr);
   END IF;
   dbms_output.put_line(Operazione completata con successo' ) ;
EXCEPTION WHEN OTHERS THEN
   dbms_output.put_line(SQLERRM);
END;
```

Tenere presente quanto segue in merito a questo esempio di blocco PL/SQL:

- C'è un blocco PL/SQL (interno) nidificato in blocco PL/SQL (esterno).
- Ogni blocco PL/SQL inizia con l'istruzione BEGIN e termina con l'istruzione END.
- le istruzioni SQL sono incorporate all'interno di costrutti 3GL.
- Ogni blocco PL/SQL può avere una sezione facoltativa dichiarazione.
- Ogni blocco PL/SQL ha la sua gestione dell'eccezione. E 'compito del programma gestire le eccezioni in ogni blocco. Le eventuali eccezioni non rilevate nel blocco nidificato vengono propagate al gestore di eccezioni nel blocco immediatamente superiore (esterno). Se un'eccezione non viene rilevata affatto in uno dei blocchi, il risultato è un'eccezione non gestita, ed il sistema trasferisce il controllo all'esterno del programma.
- Ogni statement (istruzione) all'interno di un blocco PL/SQL deve terminare con unpunto e virgola(;)

3.4 I Commenti

All'interno di un blocco PL/sql possono essere presenti dei commenti. Se i commenti sono solo su una sola riga dovranno essere preceduti da -- (due trattini) su più righe da /* */ :

```
-- Commento su una riga
/* Commento su più righe */
```

3.5 Dichiarazioni di variabili

```
DECLARE
        ……..; - - dichiarazione di variabile
BEGIN
    Statement;
    DECLARE
        Statement;
    BEGIN
        Statement;
    EXCEPTION
        Statement;
    END;
EXCEPTION
    Statement;
    DECLARE
        Statement;
    BEGIN
        Statement;
    EXCEPTION
        Statement;
    END;
END;
```

3.6 Tipi di variabili e costanti

Regole per la dichiarazione di Variabili e costanti:

- Le variabili e le costanti devono essere dichiarati nella sezione DECLARE del blocco Pl/sql.
- Possono essere scritte sia con lettere maiuscole che minuscole.
- Devono iniziare con un carattere
- Non possono contenere spazi.
- Possono essere lunghe massimo 30 caratteri.
- Non possono essere parole riservate.
- La dichiarazione termina con un punto e virgola (;)

Per facilitare la lettura del codice di seguito si propone:

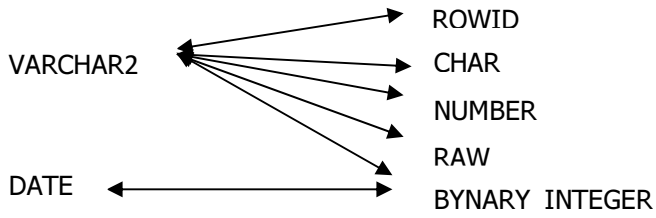
- di iniziare tutti i nomi di variabile con la lettera v

Esempio:

```
DECLARE
    v_Anagrafica        Tipo (il tipo sta per tipo variabile); --punto e virgola finale
    v_Contatore         Tipo; --punto e virgola finale
    v_Totale            Tipo; -- punto e virgola finale
BEGIN
    ……..;
END;
```

TIPI	SOTTOTIPI	DESCRIZIONE
NUMBER (precisione, scala)	DEC, DECIMAL, DOUBLE_PRECISION, FLOAT, INT, INTEGER NUMERIC, REAL, SMALLEST	Include solo le cifre da 0 a 9, un punto decimale e un segno meno, se necessario.
BINARY_INTEGER	NATURAL, POSITIVE	Interi con segno da –2 elevato alla 31 –1 e da 2 elevato alla 31 –1
CHAR (dimensione)	CHARACTER, STRING	Stringhe a lunghezza fissa, massimo 32767
VARCHAR2 (dimensione)	VARCHAR	Stringhe a lunghezza variabile, massimo 32767 byte, non può eccedere la lunghezza di 2000 caratteri
DATE		Date, ore, minuti e secondi
BOOLEAN		Valori logici True o False
RECORD		Tipi di record definiti dall'utente
TABLE		Tabelle Pl/sql

Conversione tra Tipi di Dato



Il pl/sql può effettuare automaticamente delle conversioni di dato, quando viene fatta l'assegnazione di una variabile.

```
DECLARE

   v_Id                 NUMBER(2);
   v_Nome               CHAR(50);
   v_Data               DATE;
   v_Descrizione   VARCHAR2(250);
   ……..;
BEGIN
   ……..;
   ……..;
END;
```

Le variabili possono essere inizializzate con un valore attraverso **:=** o la keyword default:

```
v_Nome          CHAR(50) := 'Maurizio';          (l'apice ' indica che è una  Stringa)
v_Cont          NUMBER(2) default 4;   (l'apice non serve poiché il tipo di variabile è numerico)
v_Data          Date;
v_Descrizione   Varchar2(250) default 'mio nome';
   oppure
v_Descrizione   Varchar2(250) := 'mio nome';
```

3.7 Le Costanti

La costante è un particolare identificatore, il quale assume un valore quando viene dichiarato che non cambierà mai.
Le costanti vengono dichiarate come le variabili, unica differenza è quella della presenza della keyword CONSTANT posizionata dopo il nome e prima del tipo dato:

Per facilitare la lettura del codice di seguito si propone:

- **di iniziare tutti i nomi di costanti con la lettera c**

```
c_nome_costante CONSTANT tipo_dato :=  valore;
```

Esempio:

```
DECLARE

   c_Id              CONSTANT NUMBER(2) :=  1;
   c_Nome            CONSTANT CHAR(50)     default 'Sconosciuto';
   v_Data            DATE;
   v_Descrizione     VARCHAR2(250);
```

3.8 Costanti tipo

DATE data corrente
Istruzioni SQL per ottenere la data corrente:

```
SELECT TO_CHAR(CURRENT_DATE, 'DD-MON-YYYY HH:MI:SS') FROM DUAL;
SELECT TO_CHAR(SYSDATE, 'DD-MON-YYYY HH:MI:SS') FROM DUAL;
```

<date> + <integer> SELECT SYSDATE + 1 FROM DUAL; (aggiunge un giorno)
<date> - <integer> SELECT SYSDATE - 1 FROM DUAL; (sottre un giorno)

Formati

Day	Month	Year
D	MM	YY
DD	MON	YYYY
DDTH		RR
DAY		RRRR

ADD_MONTHS

ADD_MONTHS(<date>, <number of months_integer> Add A Month To A Date

Esempio:

```
SELECT add_months(SYSDATE, 2) FROM DUAL;
SELECT add_months(TO_DATE('27-JAN-2007'), 1) FROM DUAL;
SELECT add_months(TO_DATE('28-JAN-2007'), 1) FROM DUAL;
SELECT add_months(TO_DATE('29-JAN-2007'), 1) FROM DUAL;
SELECT add_months(TO_DATE('30-JAN-2007'), 1) FROM DUAL;
SELECT add_months(TO_DATE('31-JAN-2007'), 1) FROM DUAL;
SELECT add_months(TO_DATE('01-FEB-2007'), 1) FROM DUAL;
```

GREATEST

GREATEST(<date>, <date>, <date>, ...) Return the Latest Date

Esempio:

```
CREATE TABLE t (datecol1 DATE, datecol2 DATE, datecol3 DATE)
INSERT INTO t VALUES (SYSDATE+23, SYSDATE-10, SYSDATE-24);
INSERT INTO t VALUES (SYSDATE-15, SYSDATE, SYSDATE+15);
INSERT INTO t VALUES (SYSDATE-7, SYSDATE-18, SYSDATE-9);
COMMIT;
SELECT * FROM t;
SELECT GREATEST(datecol1, datecol2, datecol3) FROM t;
```

INTERVAL

INTERVAL '<integer>' <unit> Interval to adjust date-time

Esempio:

```
SELECT TO_CHAR(SYSDATE, 'HH:MI:SS') FROM DUAL;
SELECT TO_CHAR(SYSDATE + INTERVAL '10' MINUTE, 'HH:MI:SS') FROM DUAL;
SELECT TO_CHAR(SYSDATE - INTERVAL '10' MINUTE, 'HH:MI:SS') FROM DUAL;
```

LAST_DAY

LAST_DAY(<date>) Returns The Last Date Of A Month

Esempio:

```
SELECT * FROM t;
SELECT LAST_DAY(datecol1) FROM t;
```

LEAST

LEAST(<date>, <date>, <date>, ...) Return the Earliest Date

Esempio:

```
SELECT * FROM t;
SELECT LEAST(datecol1, datecol2, datecol3) FROM t;
```

LENGTH

LENGTH(<date>) Returns length in characters

Esempio

```
SELECT LENGTH(last_ddl_time) FROM user_objects;
```

MONTHS_BETWEEN

MONTHS_BETWEEN(<latest_date>, <earliest_date>) Returns The Months Separating Two Dates

Esempio:

```
SELECT MONTHS_BETWEEN(SYSDATE+365, SYSDATE-365) FROM DUAL;
SELECT MONTHS_BETWEEN(SYSDATE-365, SYSDATE+365) FROM DUAL;
```

ROUND

ROUND(<date_value>, <format>) Returns date rounded to the unit specified by the format model. If you omit the format, the date is rounded to the nearest day

Esempio:

```
SELECT ROUND(TO_DATE('27-OCT-00'),'YEAR') NEW_YEAR FROM DUAL;
```

Uso del TO_CHAR

Esempio:

```
SELECT TO_CHAR(TO_DATE('10:30:18', 'HH24:MI:SS'), 'HH24SP:MISP:SSSP') FROM DUAL;
SELECT TO_CHAR(TO_DATE('01-JAN-2008', 'DD-MON-YYYY'), 'DDSP-MONTH-YYYYSP') FROM DUAL;
SELECT TO_CHAR(TO_DATE('01-JAN-2008', 'DD-MM-YYYY'), 'DDSP-MMSP-YYYYSP') FROM DUAL;
SELECT TO_CHAR(TO_DATE(sal,'J'), 'JSP') FROM emp;
```

SYSDATE

SYSDATE Returns the current date and time set for the operating system on which the database resides

Esempio:

```
SELECT SYSDATE FROM DUAL;
```

TO_DATE

TO_DATE(<string1>, [format_mask], [nls_language]) In Oracle/PLSQL, the to_date function converts a string to a date.

string1 is the string that will be converted to a date.

The format_mask parameter is optional. It is the format that will be used to convert string1 to a date.

nls_language is optional. The nls_language parameter sets the default language of the database. This language is used for messages, day and month names, symbols for AD, BC, a.m., and p.m., and the default sorting mechanism. This parameter also determines the default values of the parameters NLS_DATE_LANGUAGE and NLS_SORT.

The following table shows options for the format_mask parameter. These parameters can be used in various combinations

Parameter	Explanation
YEAR	Year, spelled out alphabetically
YYYY	4-digit year
YYY YY Y	Last 3, 2, or 1 digit(s) of year.
IYY IY I	Last 3, 2, or 1 digit(s) of ISO year.
IYYY	4-digit year based on the ISO standard
RRRR	Accepts a 2-digit year and returns a 4-digit year. A value between 0-49 will return a 20xx year. A value between 50-99 will return a 19xx year.
Q	Quarter of year (1, 2, 3, 4; JAN-MAR = 1).

MM	Month (01-12; JAN = 01).
MON	Abbreviated name of the month.
MONTH	The name of month, padded with blanks to length of 9 characters.
RM	Roman numeral month (I-XII; JAN = I).
WW	The week of the year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	The week of the month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
IW	The week of year (1-52 or 1-53) based on the ISO standard.
D	Day of the week (1-7). Sunday is day 1 when nls_territory is set to 'AMERICA' but differs if another nls_territory is set (i.e. 'UNITED KINGDOM' or 'GERMANY' - in these cases Monday is 1.
DAY	Name of the day.
DD	The day of month (1-31).
DDD	The day of year (1-366).
DY	Abbreviated name of the day. (Mon, Tue, Wed, etc)
J	Julian day; the number of days since January 1, 4712 BC.
HH	Hour of day (1-12).
HH12	Hour of day (1-12).
HH24	Hour of day (0-23).
MI	Minute (0-59).
SS	Second (0-59).
SSSSS	Number of seconds past midnight (0-86399).
FF	Fractional seconds. Use a value from 1 to 9 after FF to indicate the number of digits in the fractional seconds. For example, 'FF5'.
AM, A.M., PM, or P.M.	Meridian indicator
AD or A.D	AD indicator
BC or B.C.	BC indicator
TZD	Daylight savings identifier. For example, 'PST'
TZH	Time zone hour.
TZM	Time zone minute.
TZR	Time zone region.

TRUNC

TRUNC(<date_time>) Convert a date to the date at midnight

Esempio:

```
CREATE TABLE t (datecol DATE);
INSERT INTO t (datecol) VALUES (SYSDATE);
INSERT INTO t (datecol) VALUES (TRUNC(SYSDATE));
INSERT INTO t (datecol) VALUES (TRUNC(SYSDATE, 'HH'));
INSERT INTO t (datecol) VALUES (TRUNC(SYSDATE, 'MI'));
COMMIT;
SELECT TO_CHAR(datecol, 'DD-MON-YYYY HH:MI:SS')FROM t;
```

TRUNC(<date_time>, '<format>') Selectively remove part of the date information

Esempio:

```
SELECT TO_CHAR(SYSDATE, 'DD-MON-YYYY HH:MI:SS') FROM DUAL;
-- first day of the month
SELECT TO_CHAR(TRUNC(SYSDATE, 'MM'), 'DD-MON-YYYY HH:MI:SS') FROM DUAL;
SELECT TO_CHAR(TRUNC(SYSDATE, 'MON'), 'DD-MON-YYYY HH:MI:SS') FROM DUAL;
SELECT TO_CHAR(TRUNC(SYSDATE, 'MONTH'), 'DD-MON-YYYY HH:MI:SS') FROM DUAL;
-- first day of the year
SELECT TO_CHAR(TRUNC(SYSDATE, 'YYYY'), 'DD-MON-YYYY HH:MI:SS') FROM DUAL;
SELECT TO_CHAR(TRUNC(SYSDATE, 'YEAR'), 'DD-MON-YYYY HH:MI:SS') FROM DUAL;
```

Esempi d'uso delle date nelle condizioni di Join (Dates in WHERE Clause Joins)

```
SELECT SYSDATE FROM DUAL;
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT SYSDATE FROM DUAL;
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY';
CREATE TABLE t (datecol DATE);
INSERT INTO t (datecol) VALUES (SYSDATE);
SELECT * FROM t;
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT * FROM t;
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY';
SELECT * FROM t;
SELECT SYSDATE FROM DUAL;
SELECT * FROM t WHERE datecol = SYSDATE;
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT * FROM t;
SELECT SYSDATE FROM DUAL;
SELECT TRUNC(SYSDATE) FROM DUAL;
SELECT * FROM t WHERE TRUNC(datecol) = TRUNC(SYSDATE);
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY';
```

3.9 Assegnazione di variabili

Per assegnare un valore ad una variabile si usa l'operatore di assegnazione := o default
L'assegnazione di una variabile può avvenire in qualsiasi parte del blocco pl/sql. Indispensabile è l'assegnazione delle variabili numeriche prima che esse vengano usate, in quanto altrimenti esse conterrebbero un valore nullo.

L'assegnazione delle variabili può essere fatta anche attraverso delle espressioni

Le espressioni usate per l'assegnazione sono:

- Le stringhe
- Le variabili
- Le Costanti
- Le funzioni pl/sql
- Operatori Algebrici

Tipi di dato composti in PL/SQL

- tipo RECORD
- tipo TABLE

3.10 Record

Il record è un tipo di dato composto, formato da campi che hanno un proprio nome e possono avere tipi di dato diversi.

$Campo_1$	$Campo_2$	$Campo_3$	$Campo_n$

$Campo_1$ Varchar2
$Campo_2$ Number
$Campo_3$ Char
$Campo_n$ ……

```
Type r_nome IS RECORD
      (campo1 Varchar2,
       campo2 Number,
       campo3 Char,….);
```

Esempio:

```
Declare
    Type r_Anagrafica IS RECORD     (codice Varchar2,  nominativo   Number,
    data_nas  Char,….);
       r_mia_anagrafica      r_anagrafica;
       contatore             Number(2) := 1;
       ……..
    Begin
       ……..
    End;
```

Come accedere al campo del record? Si usa:

nome_record.nome_campo

Quindi per assegnare un valore ad un campo del record:

r_mia_anagrafica.codice := '02';

oppure

```
Select     codice,nominativo,data_nas
  Into   mio_record
 From   t_clienti
Where  codice='02';
```

Da ricordare: Anche in questo caso la SELECT deve ritornare un'unica riga.
Quando un record viene dichiarato con la stessa struttura della riga di una tabella del database si può utilizzare l'attributo %ROWTYPE.
Esempio:
nome_record nome_tabella%ROWTYPE;

Esempio:
r_mia_anagrafica t_clienti.%ROWTYPE;

3.11 Tabella (TABLE)

TABLE è un tipo di dato composto simile alle tabelle del database.
Fondamentalmente una TABLE è un array unidimensionale composto da record.
La table contiene almeno una colonna definita dall'utente, di qualsiasi tipo purché scalare, che non ha un nome esplicito. La table non ha una grandezza definita, cresce in modo dinamico.
Tutti gli elementi della table devono essere indicizzati da una colonna(chiave primaria) che permette l'accesso alle righe di tipo vettoriale.

Campo1	Campo2	Campo3	Campon

$Campo_1$ Varchar2
$Campo_2$ Number
$Campo_3$ Char
$Campo_n$ ……

```
TYPE    Tipo_tabella IS TABLE OF Tabella.colonna%TYPE INDEX BY BYNARY_INTEGER;

Nome_tabella   Tipo_tabella;
```

Esempio:

```
Declare        TYPE tipo_clienti IS TABLE OF r_mia_anagrafica.codice%TYPE INDEX BY
BYNARY_INTEGER;
t_clienti              tipo_clienti;
```

3.12 Attributi per la dichiarazione delle variabili

Quando si dichiara una variabile che deve assumere un valore prelevato da una tabella del database Oracle, può essere utilizzato l'attributo %TYPE.
Con tale attributo possiamo dire che la nostra variabile è dello stesso tipo della colonna dove si trova un nostro dato.

```
Sintassi :        Nome_variabile        Colonna.Tabella%TYPE;
      Declare
        Identificativo   nominativo.clienti%type          (colonna.Tabella%TYPE)
        Contatore              Number(2) := 1;
        ……                  …….
      Begin
          Select        nominativo
          Into          identificativo
          From          clienti
          Where         codice=01;
      End;
```

3.13 Caratteristiche di SQL disponibili in PL/SQL

È possibile utilizzare **TIMESTAMP** con due varianti, come TIMESTAMP WITH TIME ZONE e TIMESTAMP WITH TIME ZONE LOCALE. TIMESTAMP offre un alto livello di granularità della componente in aggiunta alla capacità di trattare con fusi orari. È possibile utilizzare **INTERVAL** con due varianti: INTERVAL YEAR TO MONTH e INTERVAL DAY TO SECOND. L'istruzione **MERGE** consente di combinare le funzioni di INSERT e UPDATE in una sola operazione.

Construtti procedurali in PL/SQL

Le funzionalità 3GL disponibili in PL/SQL sono i seguenti:

- supporto del tipo di dati booleano, riferimento a cursore (REF CURSOR) tipi e sottotipi definiti dall'utente
- nuovi tipi di dati introdotti in Oracle9i come TIMESTAMP e INTERVAL, così come i tipi definiti dall'utente e costrutti condizionali come IF THEN ELSE
- dichiarazioni CASE ed espressioni
- costrutti iterativi come cicli (LOOP ... END LOOP, FOR… LOOP, WHILE… LOOP), array unidimensionali, e record
- tipi di Raccolta (tabelle nidificate e varray)
- i sottoprogrammi come procedure, funzioni, e pacchetti
- caratteristiche object-oriented

3.14 Costrutto condizionale

Un costrutto condizionale esegue un insieme di istruzioni in base al verificarsi o meno di una condizione.
Ecco la sintassi del costrutto IF:

```
IF (condition1) THEN
  [action1]
ELSIF (condition2) THEN
  [action2]
ELSIF (condition3) THEN
  [action3]
 ... ...
ELSE
  [actionN]
END IF;
```

Nel codice precedente, condizione1, condizione2 e condition3 sono dei BOOLEAN che possono assumere i valori TRUE o FALSE. Se la condizione restituisce true, viene eseguita l'azione correlata. Se una condizione restituisce true, le altre condizioni della catena IF non vengono controllati.
L'ordine in cui sono scritti gli IF o i rami ELSIF è importante, perchè determina quale delle azioni deve essere eseguita.

3.15 L'istruzione CASE

Oracle9i introduce l'istruzione CASE come la migliore alternativa alla dichiarazione IF ... ... ELSIF. L'utilizzo di istruzioni CASE rende il codice più compatto, aumenta la facilità di scrittura del codice da parte del programmatore, e consente una maggiore leggibilità da parte del revisore.
Ecco la sintassi della dichiarazione CASE:

```
CASE   selector
 WHEN value1 THEN action1;
 WHEN value2 THEN action2;
 ... ...
 ELSE      actionN;
END CASE;
```

Nel codice precedente, la selezione è una variabile o espressione, e $valore_1$, $valore_2$, e $value_3$ sono i valori del selettore. Il selettore è valutato una sola volta. L'ordine in cui sono scritte i rami WHEN è importante, poiché determina quale delle azioni deve essere eseguita.

Di seguito un esempio dell'istruzione CASE:

```
CASE v_report_choice
   WHEN 1 THEN p_proc_report1;
   WHEN 2 THEN p_proc_report2;
   WHEN 3 THEN p_proc_report3;
   WHEN 4 THEN p_proc_report4;
   ELSE dbms_output.put_line('Invalid option.');
END CASE;
```

Nell'esempio precedente, a seconda del valore della variabile v_report_choice, la procedura corrispondente viene eseguita. Se avessimo scritto lo stesso codice come una catena IF ... ELSIF avremmo dovuto ripetere la stessa condizione per ogni scelta, come illustrato di seguito:

```
IF (v_report_choice=1) THEN
  p_proc_report1;
ELSIF (v_report_choice=2) THEN
  p_proc_report2;
ELSE
   dbms_output.put_line('Invalid option.');
END IF;
```

L'istruzione "IF" è più soggetta ad errori di scrittura dell'istruzione "CASE".

3.16 Espressione CASE

È possibile utilizzare il costrutto CASE in un'espressione e assegnarlo ad una variabile per generare un valore. Ecco la sintassi dell'espressione CASE:

```
var := CASE selector
   WHEN value1 THEN assigned_value1
   WHEN value2 THEN assigned_value2
   ... ...
   ELSE assigned_valueN;
END CASE;
```

Nel codice precedente, var è una variabile dichiarata che riceve un valore. valore1, valore2, e value3 sono i valori del selettore, e assigned_value1, assigned_value2, assigned_value3 e assigned_valueN sono i valori che vengono assegnati alla variabile a seconda del valore del selector. Aa esempio, si consideri il seguente segmento di codice, che fa parte di un programma che converte i numeri alle parole:

```
temp := CASE i
            WHEN 0 THEN 'Zero'
              ……..……
            WHEN 9 THEN 'Nove'
            ELSE
               NULL
            END;
```

3.17 Costrutto iterativo

Costrutti iterativi sono specificati per mezzo di cicli. Ci sono tre tipi di cicli che eseguono il flusso ripetitivo di controllo, vale a dire il semplice LOOP, il ciclo numerico FOR, e il ciclo WHILE LOOP. Ognuno di questi cicli si conclude con una dichiarazione finale del loop END LOOP.

Di seguito la sintassi del LOOP

```
LOOP
   [statement1]
   [statement2]
   ... ...
   EXIT WHEN (condition);
END LOOP;
```

Nel codice precedente, *statement1, statement2*, e così via, costituiscono il corpo del ciclo, *condition* si riferisce alla condizione di uscita per il ciclo. Se la condizione d'uscita non fosse specificata, l'esecuzione si tradurrebbe in un ciclo infinito.

Di seguito la sintassi del FOR LOOP

```
FOR index IN initialval..finalval LOOP
   [statement1]
   [statement2]
   ... ...
END LOOP;
```

Nel codice precedente, l'indice si riferisce a una variabile che incrementa il conteggio per la durata del ciclo, e initialval e finalval fanno riferimento ai valori iniziali e finali che la variabile indice assumerà durante il ciclo. La differenza tra questi due valori più 1 riporta il numero di volte che il controllo esegue ripetutamente le istruzioni nel corpo del ciclo, cioè tra il FOR LOOP …… END LOOP.

Di seguito la sintassi del WHILE LOOP

```
WHILE (condition) LOOP
   [statement1]
   [statement2]
   ... ...
END LOOP;
```

L'indice del ciclo WHILE se usato come condizione, deve essere dichiarato, inizializzato e dovrebbe essere incrementato nel corpo del ciclo. La condizione viene valutata per prima, e se risulta essere TRUE, solo il corpo del ciclo viene eseguito. Non c'è bisogno di un EXIT per un uscita esplicita dal ciclo. Dal ciclo si esce quando la condizione assume il valore FALSE.

LOOP

```
DECLARE
   line_length NUMBER := 50;
   separator   VARCHAR2(1) := '=';
   actual_line VARCHAR2(150);
   i NUMBER := 1;
 BEGIN
   LOOP
      actual_line := actual_line || separator;
      EXIT WHEN i = line_length;
      i:= i + 1;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE(actual_line);
END;
```

FOR LOOP

```
DECLARE
   line_length NUMBER := 50;
   separator   VARCHAR2(1) := '=';
   actual_line VARCHAR2(150);
BEGIN
   FOR idx IN 1..line_length LOOP
     actual_line := actual_line || separator;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE(actual_line);
END;
```

WHILE LOOP

```
DECLARE
   line_length NUMBER := 50;
   separator   VARCHAR2(1) := '=';
   actual_line VARCHAR2(150);
   idx NUMBER := 1;
BEGIN
  WHILE (idx<=line_length) LOOP
      actual_line := actual_line || separator;
      idx := idx +1 ;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(actual_line);
END;
/
```

3.18 Cursori

3.19 Introduzione

Un cursore è un puntatore ad un'area di lavoro che contiene il risultato di una query SQL (una o più righe). Oracle apre una zona di lavoro per tenere in memoria quest'array di record. Un cursore dà a questa zona di lavoro un nome e può essere utilizzato per elaborare le righe restituite dalla query SQL. Ci sono due tipi di cursori: explicit e implicit.

3.20 Corsori espliciti

Nella definizione di un cursore esplicito, il nome del cursore è esplicitamente associato ad una istruzione SELECT. Questo viene fatto usando il costrutto PL/SQL SELECT CURSOR ... IS ... dichiarazione. Un cursore esplicito può essere associato ad una sola istruzione SELECT.

3.21 Definizione di un cursore esplicito

```
CURSOR cursor_name IS
    SELECT_statement ;
```

dove cursor_name è il nome del cursore e select_statement è qualsiasi istruzione SQL SELECT senza la clausola INTO.
Quando si utilizza un blocco PL / SQL, è necessario dichiarare un cursore esplicito nella sezione dichiarazione dopo la parola chiave DECLARE. Il seguente è un esempio di un cursore esplicito:

```
DECLARE
   CURSOR csr_org IS
       SELECT        h.hrc_descr,
                     o.org_short_name
       FROM          org_tab o,
                     hrc_tab h
       WHERE o.hrc_code = h.hrc_code ORDER by 2;
        v_hrc_descr VARCHAR2(20);
        v_org_short_name VARCHAR2(30);
 BEGIN
    /* ... <Process the cursor resultset> ... */
    null;
 END;
/
```

L'istruzione SELECT associata a un cursore non può contenere una clausola INTO. Si possono tuttavia usare le clausole ORDER e GROUP BY , così come gli operatori insieme come UNION, INTERSECT e MINUS.

4.4 Utilizzo di un cursore Esplicito

Una volta definito un cursore, è possibile utilizzarlo per l'elaborazione delle righe contenute nel set di risultati. Ecco i passaggi:

- Aprire il cursore.
- Recuperare i risultati in una PL / SQL record o singoli PL / SQL variabili.
- Chiudere il cursore.

Ci sono due modi per utilizzare un cursore esplicito una volta che è stata definito:
con OPEN, FETCH e CLOSE, o utilizzando un CURSOR FOR LOOP. Potete farlo nella sezione eseguibile di un blocco PL / SQL tra BEGIN ed END.

4.5 Utilizzo OPEN, FETCH e CLOSE

Dopo aver dichiarato il cursore, è necessario aprirlo come segue:

OPEN cursor_name;

dove cursor_name è il nome del cursore dichiarato.

Ecco un esempio che illustra l'apertura del csr_org cursore dichiarato in precedenza:

```
DECLARE
   CURSOR csr_org IS
       SELECT h.hrc_descr, o.org_short_name
       FROM   org_tab o, hrc_tab h
       WHERE o.hrc_code = h.hrc_code
        ORDER by 2;
        v_hrc_descr VARCHAR2(20);
        v_org_short_name VARCHAR2(30);
 BEGIN
    OPEN csr_org;
          /* ... <Process the cursor resultset> ... */
 END;
```

Non aprire un cursore già aperto.

Ciò genererebbe l'eccezione predefinita CURSOR_ALREADY_OPEN.

Il passo successivo è quello di recuperare I record contenuti dal cursore in variabili PL/SQL. Si possono recuperare quindi singole righe di dati e valorizzare delle variabili PL / SQL.

Si possono usare i record(riche) mediante l'istruzione FETCH, che ha quattro forme. Ecco la sintassi:

FETCH cursor_name INTO var1, var2, ... , varN; var1, var2, and varN sono variabili PL/SQL che hanno lo stesso tipo di dati dei campi contenuti da un record del cursore

FETCH cursor_name INTO cursor_name%ROWTYPE;

SELECT columns. cursor_name%ROWTYPE rappresenta un tipo di record PL/SQL con attribute implicitamente definiti che sono identici ai tipi di dati restituiti dal cursore. Il tipo record usato in questo cursore va definito esplicitamente

FETCH cursor_name INTO table_name%ROWTYPE;

table_name%ROWTYPE rappresenta un tipo di record simile ma che ha sia gli attribute (tipi didato) che nomi delle colonne uguali a quelli delcursore; le colonne della table_name dovranno essere indentiche in numero e tipo di dato al numero di campi contenuti da un record del cursore considerato.

FETCH cursor_name INTO record_name; il numero di tipi di dato e gli attribute individuali del record dovranno combaciare uno ad uno con le colonne del cursore.

Per esempio:

```
DECLARE
   CURSOR csr_org IS
       SELECT h.hrc_descr, o.org_short_name FROM   org_tab o, hrc_tab h
       WHERE o.hrc_code = h.hrc_code ORDER by 2;
        v_hrc_descr VARCHAR2(20);
        v_org_short_name VARCHAR2(30);
 BEGIN
    OPEN csr_org;
    FETCH csr_org INTO v_hrc_descr, v_org_short_name;
 END;
```

Alternativamente si può dichiarare un tipo record come variabile di istanza di cursore

type cursor_name%ROWTYPE ed inserire le righe in esso.

Questo metodo è il più comodo perché evita la definizione di molte variabili. Per esempio:

```
DECLARE
   CURSOR csr_org IS
       SELECT h.hrc_descr, o.org_short_name
       FROM   org_tab o, hrc_tab h
       WHERE o.hrc_code = h.hrc_code
        ORDER by 2;
        v_org_rec csr_org%ROWTYPE;
  BEGIN
     OPEN csr_org;
     FETCH csr_org INTO v_org_rec;
    -- inserisce una riga del cursore nel record
 END;
/
```

In questo caso noi potremo accedere alle colonne del record usando gli stessi nomi delle colonne del cursore.

L'istruzione FETCH prende un solo record per volta.

A single FETCH always fetches only one row (the current row) from the active set. To fetch multiple rows, use the FETCH statement in a loop.

Il cursore, dopo esser stato usato, si deve chiudere con l'istruzione CLOSE

```
CLOSE cursor_name;
```

Di seguito un esempio complete di tutto quanto ditto inprecedenza:

```
DECLARE
/* Declare a cursor explicitly */
   CURSOR csr_org IS
       SELECT h.hrc_descr, o.org_short_name
       FROM   org_tab o, hrc_tab h
       WHERE o.hrc_code = h.hrc_code
       ORDER by 2;

        v_org_rec csr_org%ROWTYPE;

  BEGIN
     OPEN csr_org;
      /* Format headings */
…….--etc
      LOOP
         FETCH csr_org INTO v_org_rec;
         EXIT WHEN csr_org%NOTFOUND;
        dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' '|| rpad(v_org_rec.org_short_name,30,' '));
      END LOOP;
      CLOSE csr_org;
 END;
/
```

Dal programma precedente si può notare quale sia la condizione di uscita dal loop. Il verificarsi o meno dell'esistenza di un'ennesima riga. Questo si può sapere mediante l'uso del %not_found che è un'attributo di oracle.

```
EXIT WHEN csr_org%NOTFOUND;
```

%NOTFOUND restituisce un booleano a valore TRUE quando l'ultima riga è stata letta dal cursore e non ce ne sono più da elaborare.

Stesso esempio del precedente ma riscritto usando un WHILE fetch loop:

```
DECLARE
   CURSOR csr_org IS
       SELECT h.hrc_descr, o.org_short_name
       FROM   org_tab o, hrc_tab h
       WHERE o.hrc_code = h.hrc_code
        ORDER by 2;
       v_org_rec csr_org%ROWTYPE;
  BEGIN
     OPEN csr_org;
     FETCH csr_org INTO v_org_rec;
     WHILE (csr_org%FOUND) LOOP
         dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' '|| rpad(v_org_rec.org_short_name,30,' '));
          FETCH csr_org INTO v_org_rec;
     END LOOP;
     CLOSE csr_org;
 END;
/
```

- Il primo FETCH prima di iniziare il WHILE LOOP è necessario per garantire che la condizione sia a TRUE.Poi si usa la condizione %FOUND su un attributo del cursore che viene vaslutata a TRUE se esiste almeno una riga attiva nel cursore.

- Se il cursore non contiene righe, nonviene eseguito il WHILE LOOP. Differentemente dal LOOP … END LOOP, per i quail prima si entra nel LOOP e poi si valuta se uscire o restare.

- Non sarà necessaria una istruzione EXIT dopo il secondo FETCHall'interno del LOOP.

3.22 Utilizzo del FOR LOOP su un cursore

Si può usare un cursore anche all'interno di un FOR LOOP invece di un esplicito OPEN, FETCH, and CLOSE. Esempio:

```
FOR idx in cursor_name LOOP
 ...
 ...
END LOOP;
```

cursor_name è il nome del cursore e idx è l'indice del cursore FOR LOOP ed è di tipo ROWTYPE%cursor_name .

E 'ancora un cursore esplicito e deve essere dichiarata esplicitamente.

Ecco l'esempio del cursore csr_org modificato utilizzando un ciclo FOR:

```
1.  DECLARE
2.      CURSOR csr_org IS
3.          SELECT h.hrc_descr, o.org_short_name
4.          FROM   org_tab o, hrc_tab h
5.          WHERE o.hrc_code = h.hrc_code
6.           ORDER by 2;
7.  BEGIN
8.        FOR idx IN csr_org LOOP
9.            dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '||   rpad(idx.org_short_name,30,' '));
10.       END LOOP;
11. END;
12. /
```

I seguenti punti sono degni di nota:
• L'indice del cursore FOR LOOP non è dichiarato. E 'implicitamente dichiarato dal PL / SQL compilatore come ROWTYPE% csr_org . Mai dichiarare l'indice di un cursore per LOOP
• È possibile accedere alle singole colonne del cursore SELECT utilizzando la notazione di accedere ad attributi tipo di record utilizzando il nome dell'indice con un punto "." seguito dal nome della colonna del cursore SELECT.
• Non vi è alcuna necessità del OPEN, FETCH, CLOSE cursore.

3.23 Evitare la dichiarazione esplicita di un cursore nel FOR LOOP

Nell'esempio precedente, anche se è stato utilizzato il cursore FOR LOOP, il csr_org cursore è stato dichiarato nella sezione dichiarazione del blocco PL / SQL. Tuttavia, è possibile specificare il cursore SELECT nella specificazione del FOR LOOP stesso invece di una dichiarazione esplicita. Questo migliora la leggibilità ed è meno soggetto a errori. Ecco il cursore csr_org riscritto in questo modo:

```
BEGIN
        FOR idx in (SELECT h.hrc_descr, o.org_short_name
                        FROM   org_tab o, hrc_tab h
                        WHERE o.hrc_code = h.hrc_code ORDER by 2) LOOP
           dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '||   rpad(idx.org_short_name,30,'
'));
        END LOOP;
END;
/
```

3.24 Attributi di un cursore esplicito

Ogni cursore esplicito ha quattro attributi associata ad essa che è possibile utilizzare per determinare se un cursore è aperto o meno, se un'operazione di recupero ha prodotto una riga o no, e quante righe sono state recuperate finora. La Tabella 2-1 elenca questi attributi.

Attributo	**Uso**
%FOUND	Indica se un FETCH ha prodotto una riga o no
%ISOPEN	Indica se un cursore in stato OPEN o no
%NOTFOUND	Indic ache una FETCH è fallita oppure che non ci sono più record da prendere (FETCH)
%ROWCOUNT	indica il numero di righe recuperate finora

3.25 Tabella 2-1: Attributi di un cursore esplicito

Per usare questi quattro attributi bisogna anteporre il nome del cursore.

Gli attribute %FOUND, %ISOPEN, e %NOTFOUND restituiscono un valore booleano TRUE o FALSE, e l'attributo %ROWCOUNT restituisce un valore numerico.

3.25.1.1.1 %FOUND

Si utilizza% FOUND per determinare se un FETCH restituisce una riga o no. Si consiglia di usarlo dopo che un cursore è stato aperto, e restituisce un valore TRUE se il FETCH immediato ha prodotto una riga e un valore di FALSE se l'immediato FETCH non prendere qualsiasi riga. Usando% FOUND prima di aprire un cursore o dopo la chiusura di un cursore genera l'errore "ORA-01001: invalid cursor" o la INVALID_CURSOR eccezione.

Esempio:

```
DECLARE
   CURSOR csr_org IS
       SELECT h.hrc_descr, o.org_short_name
       FROM   org_tab o, hrc_tab h
       WHERE o.hrc_code = h.hrc_code
        ORDER by 2;
       v_org_rec csr_org%ROWTYPE;
BEGIN
    OPEN csr_org;
    dbms_output.put_line('Organization Details with Hierarchy');
    dbms_output.put_line('------------------------');
    dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||  rpad('Organization',30,' '));
    dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
     FETCH csr_org INTO v_org_rec;
     WHILE (csr_org%FOUND) LOOP
         dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' '||rpad(v_org_rec.org_short_name,30,' '));
          FETCH csr_org INTO v_org_rec;
     END LOOP;
     CLOSE csr_org;
END;
```

I seguenti punti sono degni di nota per quanto riguarda la dichiarazione

```
    WHILE (csr_org% FOUND) LOOP
```

• La dichiarazione appare dopo la prima istruzione FETCH, e dovrebbe sempre apparire dopo una istruzione FETCH. Se NOTFOUND% fa riferimento prima della prima FETCH, restituisce NULL.

• La condizione csr_org% FOUND restituisce TRUE se la prima FETCH restituisce una riga, altrimenti restituisce FALSE e il ciclo while non viene mai eseguito.

3.25.1.1.2 %ISOPEN

Si utilizza ISOpen% per controllare se un cursore è già aperto o no. Lo si usa per evitare di aprire un cursore già aperto o di chiudere un cursore già chiuso. Esso restituisce un valore TRUE se il cursore a cui si fa riferimento è aperto, altrimenti restituisce FALSE. Ecco l'esempio precedente modificato per utilizzare l'attributo ISOpen%:

```
DECLARE
   CURSOR csr_org IS
       SELECT h.hrc_descr, o.org_short_name
       FROM   org_tab o, hrc_tab h
       WHERE o.hrc_code = h.hrc_code
        ORDER by 2;
       v_org_rec csr_org%ROWTYPE;
  BEGIN
    IF (NOT csr_org%ISOPEN) THEN  OPEN csr_org; END IF;
    dbms_output.put_line('Organization Details with Hierarchy');
    dbms_output.put_line('------------------------');
    dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||  rpad('Organization',30,' '));
    dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
     FETCH csr_org INTO v_org_rec;
     WHILE (csr_org%FOUND) LOOP
        dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' '|| rpad(v_org_rec.org_short_name,30,' '));
          FETCH csr_org INTO v_org_rec;
     END LOOP;
     IF (csr_org%ISOPEN) THEN CLOSE csr_org; END IF;
END;
```

Note the following points about %ISOPEN:

- csr_org%ISOPEN is negated in the beginning to check that the cursor isn't already open.
- At the end, the cursor csr_org is closed only if it's open.
- %ISOPEN can be referenced after a cursor is closed, and it returns FALSE in this case.

3.25.1.1.3 %NOTFOUND

You use %NOTFOUND to determine if a FETCH resulted in no rows (i.e., the FETCH failed) or there are no more rows to FETCH. It returns a value of TRUE if the immediate FETCH yielded no row and a value of FALSE if the immediate FETCH resulted in one row. Using %NOTFOUND before opening a cursor or after a cursor is closed raises the error "ORA-01001: invalid cursor" or the predefined exception INVALID_CURSOR. I presented an example of using %NOTFOUND during the discussion of using the simple LOOP to fetch multiple rows. Here's the same example repeated for illustration:

```
DECLARE
   CURSOR csr_org IS
       SELECT h.hrc_descr, o.org_short_name
       FROM   org_tab o, hrc_tab h
       WHERE o.hrc_code = h.hrc_code
        ORDER by 2;
       v_org_rec csr_org%ROWTYPE;
  BEGIN
     OPEN csr_org;
    dbms_output.put_line('Organization Details with Hierarchy');
    dbms_output.put_line('------------------------');
    dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||rpad('Organization',30,' '));
    dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
     LOOP
        FETCH csr_org INTO v_org_rec;
        EXIT WHEN csr_org%NOTFOUND;
      dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' '|| rpad(v_org_rec.org_short_name,30,' '));
     END LOOP;
     CLOSE csr_org;
 END;
/
```

The following points are worth noting:

- Note the statement
- EXIT WHEN csr_org%NOTFOUND;.

It appears after the first FETCH statement, and it should always appear after a FETCH statement. If %NOTFOUND is referenced before the first FETCH or after a cursor is opened, it returns NULL.

- The condition csr_org%NOTFOUND is used as the EXIT condition for the loop. It evaluates to TRUE if the first FETCH didn't return a row and the loop is exited. If the first FETCH resulted in at least one row, it evaluates to FALSE and the loop is executed until the last row is fetched. After the last row is fetched, %NOTFOUND evaluates to TRUE and the loop is exited.

3.25.1.1.4 %ROWCOUNT

You use %ROWCOUNT to determine the number of rows fetched from a cursor. It returns 1 after the first fetch and is incremented by 1 after every successful fetch. It can be referenced after a cursor is opened or before the first fetch and returns zero in both cases. Using %ROWCOUNT before opening a cursor or after closing a cursor raises the error "ORA-01001: invalid cursor" or the predefined exception INVALID_CURSOR. The best use of this attribute is in a cursor FOR LOOP to determine the number of rows returned by the cursor. Since a cursor FOR LOOP is used to process *all* the rows of the cursor unconditionally, the value of this attribute after the cursor FOR LOOP is executed gives the total number of rows returned by the cursor.

In the following example, I've modified the cursor FOR LOOP presented earlier to include %ROWCOUNT:

```
DECLARE
   CURSOR csr_org IS
       SELECT h.hrc_descr, o.org_short_name
       FROM   org_tab o, hrc_tab h
       WHERE o.hrc_code = h.hrc_code
        ORDER by 2;
        num_total_rows NUMBER;
  BEGIN
      dbms_output.put_line('Organization Details with Hierarchy');
      dbms_output.put_line('------------------------');
      dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||  rpad('Organization',30,' '));
      dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
       FOR idx IN csr_org LOOP
          dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '||    rpad(idx.org_short_name,30,' '));
             num_total_rows := csr_org%ROWCOUNT;
       END LOOP;
       IF num_total_rows > 0 THEN
       dbms_output.new_line;
       dbms_output.put_line('Total Organizations = '||to_char(num_total_rows));
       END IF;
END;
/
```

Here's the output of this program:

```
Organization Details with Hierarchy
-------------------------------------------
Hierarchy         Organization
------------------  ------------------------
CEO/COO             Office of CEO ABC Inc.
CEO/COO             Office of CEO DataPro Inc.
CEO/COO             Office of CEO XYZ Inc.
VP                      Office of VP Mktg ABC Inc.
VP                      Office of VP Sales ABC Inc.
VP                      Office of VP Tech ABC Inc.
Total Organizations = 6
```

%ROWCOUNT is an incremental count of the number of rows, and hence you can use it to check for a particular value. In this example, the first three lines after the BEGIN and before the cursor loop are displayed, irrespective of the number of rows returned by the cursor. This is true even if the cursor returned no rows. To prevent this, you can use the value of %ROWCOUNT to display them only if the cursor returns at least one row. Here's the code to do so:

```
DECLARE
   CURSOR csr_org IS
       SELECT h.hrc_descr, o.org_short_name
       FROM   org_tab o, hrc_tab h
       WHERE o.hrc_code = h.hrc_code
        ORDER by 2;
      num_total_rows NUMBER;
BEGIN
      FOR idx IN csr_org LOOP
          IF csr_org%ROWCOUNT = 1 THEN
            dbms_output.put_line('Organization Details with Hierarchy');
            dbms_output.put_line
            ('------------------------');
            dbms_output.put_line(rpad('Hierarchy',20,' ')||' '|| rpad('Organization',30,' '));
            dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
          END IF;
          dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '||  rpad(idx.org_short_name,30,'
'));
           num_total_rows := csr_org%ROWCOUNT;
     END LOOP;
     IF num_total_rows > 0 THEN
      dbms_output.new_line;
      dbms_output.put_line('Total Organizations = '||to_char(num_total_rows));
     END IF;
END;
/
```

The following points are worth noting:

- The %ROWCOUNT is checked inside the cursor FOR LOOP.
- After the first row is fetched, the value of %ROWCOUNT is 1 and the headings are displayed. Successive fetches increment the value of %ROWCOUNT by 1 so that %ROWCOUNT is greater than 1 after the first fetch.
- After the last fetch, the cursor FOR LOOP is exited and the value of %ROWCOUNT is the total number of rows processed.

3.26 Parameterized Cursors

An explicit cursor can take parameters and return a data set for a specific parameter value. This eliminates the need to define multiple cursors and hard-code a value in each cursor. It also eliminates the need to use PL/SQL bind variables.

In the following code, I use the cursor example presented earlier in the section to illustrate parameterized cursors:

```
DECLARE
   CURSOR csr_org(p_hrc_code NUMBER) IS
       SELECT h.hrc_descr, o.org_short_name
       FROM   org_tab o, hrc_tab h
       WHERE o.hrc_code = h.hrc_code
            AND h.hrc_code = p_hrc_code
        ORDER by 2;
       v_org_rec csr_org%ROWTYPE;
  BEGIN
     OPEN csr_org(1);
     dbms_output.put_line('Organization Details with Hierarchy 1');
     dbms_output.put_line('------------------------');
     dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||  rpad('Organization',30,' '));
     dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
     LOOP
         FETCH csr_org INTO v_org_rec;
         EXIT WHEN csr_org%NOTFOUND;
         dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' '||
rpad(v_org_rec.org_short_name,30,' '));
     END LOOP;
     CLOSE csr_org;
     OPEN csr_org(2);
      dbms_output.put_line('Organization Details with Hierarchy 2');
      dbms_output.put_line('------------------------');
      dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||   rpad('Organization',30,' '));
      dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
      LOOP
          FETCH csr_org INTO v_org_rec;
          EXIT WHEN csr_org%NOTFOUND;
          dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' '||
rpad(v_org_rec.org_short_name,30,' '));
     END LOOP;
     CLOSE csr_org;
 END;
/
```

You define the cursor parameters immediately after the cursor name by including the name of the parameter and its data type within parentheses. These are referred to as the *formal parameters.* The actual parameters (i.e., the actual data values for the formal parameters) are passed via the OPEN statement as shown in the previous example. Notice how the same cursor is used twice with different values of the parameters in each case.

You can rewrite the same example using a cursor FOR LOOP. In this case, the actual parameters are passed via the cursor name referenced in the cursor FOR LOOP. Here's the code:

```
DECLARE
   CURSOR csr_org(p_hrc_code NUMBER) IS
        SELECT h.hrc_descr, o.org_short_name
        FROM   org_tab o, hrc_tab h
        WHERE o.hrc_code = h.hrc_code
            AND h.hrc_code = p_hrc_code
         ORDER by 2;
        v_org_rec csr_org%ROWTYPE;
  BEGIN
       dbms_output.put_line('Organization Details with Hierarchy 1');
       dbms_output.put_line('------------------------');
       dbms_output.put_line(rpad('Hierarchy',20,' ')||   ' '||rpad('Organization',30,' '));
       dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
       FOR idx in csr_org(1) LOOP
          dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '|| rpad(idx.org_short_name,30,'
'));
      END LOOP;
       dbms_output.put_line('Organization Details with Hierarchy 2');
       dbms_output.put_line('------------------------');;
       dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||  rpad('Organization',30,' '));
       dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
       FOR idx in csr_org(2) LOOP
          dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '|| rpad(idx.org_short_name,30,'
'));
      END LOOP;
 END;
/
```

Parameterized cursors are very useful in processing nested cursor loops in which an inner cursor is opened with data values passed to it from an outer opened cursor.

3.27 Implicit Cursors

Here's an example of an implicit cursor:

```
BEGIN
     DELETE sec_hrc_org_tab WHERE hrc_code = 1;
     INSERT INTO sec_hrc_org_tab
          SELECT   h.hrc_code, h.hrc_descr, o.org_id, o.org_short_name, o.org_long_name
          FROM     org_tab o, hrc_tab h
          WHERE   o.hrc_code = h.hrc_code
             AND  h.hrc_code = 1;
     IF (SQL%FOUND) THEN
        dbms_output.put_line(TO_CHAR(SQL%ROWCOUNT)|| ' rows inserted into secondary
table for hierarchy 1');
     END IF;
     COMMIT;
 END;
```

3.28 Implicit Cursor Attributes

Although an implicit cursor is opened and closed automatically by the PL/SQL engine, the four attributes associated with an explicit cursor are also available for an implicit cursor. You can reference these attributes by prefixing the keyword SQL with the particular attribute. Table 2-2 lists the four attributes of the implicit cursor.

Table 2-2: Implicit Cursor Attributes

ATTRIBUTE	USE
SQL%FOUND	Indicates whether an INSERT, UPDATE, or DELETE affected any row(s) or not.
SQL%ISOPEN	Indicates whether the cursor is OPEN or not. This is FALSE always, as the implicit cursor is closed after the DML statement is executed.
SQL%NOTFOUND	Indicates if a DML statement failed to modify any rows.
SQL%ROWCOUNT	Indicates the number of rows affected by the DML statement.

Note that the name of the cursor in this case is "SQL" instead of a programmer-defined cursor name.

The SQL%FOUND, SQL%ISOPEN, and SQL%NOTFOUND attributes return a boolean TRUE or FALSE, and the SQL%ROWCOUNT attribute returns a numeric value. The following sections describe these attributes in detail.

3.28.1.1.1 SQL%FOUND

You use SQL%FOUND to determine whether an INSERT, UPDATE, or DELETE affected any row(s) or not, or a SELECT … INTO returned a row or not. You should use it immediately after the DML statement, and it returns a value of TRUE if the INSERT, UPDATE, or DELETE affected one or more rows, or the SELECT … INTO fetched a row. Otherwise, it returns a value of FALSE. Using SQL%FOUND before defining any DML statement yields NULL.

An example of using SQL%FOUND during the discussion of implicit cursors:

```
BEGIN
      DELETE sec_hrc_org_tab WHERE hrc_code = 1;
      INSERT INTO sec_hrc_org_tab
           SELECT h.hrc_code, h.hrc_descr,
                     o.org_id, o.org_short_name, o.org_long_name
           FROM   org_tab o, hrc_tab h
           WHERE o.hrc_code = h.hrc_code
              AND h.hrc_code = 1;
      IF (SQL%FOUND) THEN
        dbms_output.put_line(TO_CHAR(SQL%ROWCOUNT)|| ' rows inserted into secondary
table for hierarchy 1');
      END IF;
      COMMIT;
 END;
```

The following points are worth noting:

- The statement IF (SQL%FOUND) THEN appears immediately after the INSERT statement and it always should. If SQL%FOUND is referenced before the INSERT statement, it returns NULL.

- The condition SQL%FOUND evaluates to TRUE if the INSERT succeeded in creating one or more rows; otherwise, it evaluates to FALSE and the code inside the IF is never executed.

3.28.1.1.2 SQL%ISOPEN

SQL%ISOPEN is always FALSE because the implicit cursor is closed after the DML statement is executed. Hence, it's not useful to check this attribute for the same.

3.28.1.1.3 SQL%NOTFOUND

You use SQL%NOTFOUND to determine if an INSERT, UPDATE, or DELETE failed to modify any rows. It returns a value of TRUE if no rows were modified by the INSERT, UPDATE, or DELETE, and a value of FALSE if at least one row was modified. Using SQL%NOTFOUND before executing any DML statement yields a NULL value. Here's an example of using SQL%NOTFOUND:

```
DECLARE
  v_num_rows NUMBER;
BEGIN
      DELETE sec_hrc_org_tab WHERE hrc_code = 1;
      INSERT INTO sec_hrc_org_tab
           SELECT h.hrc_code, h.hrc_descr, o.org_id, o.org_short_name, o.org_long_name
           FROM   org_tab o, hrc_tab h
           WHERE o.hrc_code = h.hrc_code
              AND h.hrc_code = 1;
     v_num_rows := SQL%ROWCOUNT;
     IF (SQL%FOUND) THEN
       UPDATE sec_hrc_audit
       SET num_rows = v_num_rows
       WHERE hrc_code = 1;
       IF (SQL%NOTFOUND) THEN
        INSERT INTO sec_hrc_audit(hrc_code, num_rows) VALUES (1, v_num_rows);
       END IF;
     END IF;
     COMMIT;
 END;
```

This code first deletes all rows from the sec_hrc_org_tab table where the hrc_code matches 1. It then inserts new rows into the same table. Now the question is, did the INSERT succeed? That is, did it insert zero or more rows? This is determined by the implicit cursor attribute SQL%FOUND, which is defined in the statement

```
IF (SQL%FOUND) THEN
```

SQL%FOUND returns a boolean true when at least one row has been inserted into the sec_hrc_org_tab. When this happens, the code inside the IF condition is executed and the UPDATE statement against the sec_hrc_audit table is executed.

3.28.1.1.4 SQL%ROWCOUNT

You use %ROWCOUNT to determine the number of rows affected by a DML statement. It returns a value greater than zero if the DML statement succeeded; otherwise, it returns zero. It's a good alternative to SQL%NOTFOUND. Since %NOTFOUND returns TRUE if the DML statement failed, it's equivalent to use

```
  IF (SQL%ROWCOUNT = 0) THEN ...
```

instead of

```
 IF (SQL%NOTFOUND) THEN ...
```

Here's the previous example modified to use %ROWCOUNT:

```
DECLARE
  v_num_rows NUMBER;
```

```
BEGIN
      DELETE sec_hrc_org_tab WHERE hrc_code = 1;
      INSERT INTO sec_hrc_org_tab
           SELECT h.hrc_code, h.hrc_descr, o.org_id, o.org_short_name, o.org_long_name
           FROM   org_tab o, hrc_tab h
           WHERE o.hrc_code = h.hrc_code
              AND h.hrc_code = 1;
     v_num_rows := SQL%ROWCOUNT;
     IF (SQL%FOUND) THEN
       UPDATE sec_hrc_audit
       SET num_rows = v_num_rows
       WHERE hrc_code = 1;
       IF (SQL%ROWCOUNT=0) THEN
        INSERT INTO sec_hrc_audit(hrc_code, num_rows) VALUES (1, v_num_rows);
       END IF;
     END IF;
     COMMIT;
END;
```

- The first SQL%ROWCOUNT returns the number of rows affected by the very first INSERT statement—that is, the number of rows inserted into the sec_hrc_org_tab table.

- The second SQL%ROWCOUNT returns the number of rows affected by the UPDATE statement against the table sec_hrc_audit.

Always check for the attributes SQL%FOUND, SQL%NOTFOUND, and SQL%ROWCOUNT immediately after the DML statement.

3.29 Records

A *record* is a composite data type consisting of individual elements that are logically related. Each element of the record is called a *field,* has a value associated with it, and can be assessed as an individual element of the record. Each field in a record has a name and is usually of the scalar data type. However, a field in a record can be another record. Using record types enables you to make use of data abstraction because you work with data as a group instead of using individual elements. This also means less code, which in turn means easy maintainability. As an example, consider that an organization can be defined as a record with fields composed of logically related information about it, such as org id, org short name, and org long name.

PL/SQL lets you define three types of records: explicitly defined records, database table-oriented records, and cursor-oriented records. I begin with a discussion of explicitly defined records and then touch on the other record types in the later part of this section.

3.30 Defining a Record

Unlike scalar variables, which you define by directly declaring the variables of the particular type, you define a record variable by first defining a record type and then declaring a variable of that type. These steps are outlined in the following sections.

Defining the Record Type

You define a record type using the TYPE statement. Here's the syntax:

```
TYPE record_type_name IS RECORD
  ( field_name1 datatype [NOT NULL] [DEFAULT value1 | := assign1],
    field_name2 datatype [NOT NULL] [DEFAULT value2 | := assign2],
    ... ...
    field_nameN datatype [NOT NULL] [DEFAULT valueN | := assignN]
);
```

where record_type_name is a name that identifies the structure of the new record containing individual fields, with the names field_name1 to field_nameN. Notice that each field has a data type specified (which can be a scalar or user-defined subtype, or another record type), an optional NOT NULL constraint, and a DEFAULT clause with initial values specified by value1 through valueN. Alternatively, initial values can be specified by an initial assignment using the := syntax followed by a value. Here, assign1 through assignN are expressions that each evaluate to a value with the same data type as the specified data type of the corresponding field.

Here's an example of a record type declaration:

```
TYPE hrc_org_rec IS RECORD
      (hrc_org_id NUMBER,
       hrc_descr VARCHAR2(20),
       org_short_name VARCHAR2(30));
```

where hrc_org_rec is a record type that has a three-field structure.

Declaring a Variable of the Record Type

Once you've defined a record type with a specific structure, you need to declare the actual records of that type. Here's the syntax:

```
<record_var_name> <record_type_name>;
```

In this line, record_type_name is the record type defined using the TYPE … RECORD specification, and record_var_name is an arbitrary variable name with a data type of this record type. Here's an example to show this:

v_example_rec hrc_org_rec;

A TYPE … RECORD definition is only abstract, and as such, it can't be used by itself.
Defining a TYPE … RECORD declaration doesn't occupy any memory until you declare variables of that type.

A complete example combining the two preceding steps is as follows:

```
DECLARE
   TYPE hrc_org_rec IS RECORD
         (hrc_org_id NUMBER,
         hrc_descr VARCHAR2(20),
         org_short_name VARCHAR2(30));
   v_example_rec hrc_org_rec;
BEGIN
   /* Do some processing */
   null;
END;
/
```

3.31 Using a Record Type

Once you've defined a record type and declared variables of that type, the next step is to use the record for processing data. This step usually consists of accessing the individual record elements, storing data in the record, and performing comparison operations on the record for equality.

You access the individual elements of a record using the dot notation. Here's the syntax:

Accessing Individual Record Elements

<record_var_name>.<field_name>

Note that the actual record variable name, not the record type name, is used. This syntax is similar to the syntax you use when accessing a column in a database table using the <table_name>.<column_name> syntax. This brings to light the analogy between database tables and records. However, a difference between the two is that the former are stored in a database, whereas the latter are not and cannot be stored in a database.

When you access the individual elements of a record, use the record variable name and not the record type name. Record types are PL/SQL specific, and as such, they aren't available in SQL. Also, they can't be stored in a database.

Here's the example for the hrc_org_rec record defined previously:

```
DECLARE
   TYPE hrc_org_rec IS RECORD
        (hrc_org_id NUMBER,
         hrc_descr VARCHAR2(20),
         org_short_name VARCHAR2(30));
   v_example_rec hrc_org_rec;
BEGIN
   v_example_rec.hrc_org_id := 1001;
   v_example_rec.hrc_descr := 'CEO/COO';
   v_example_rec.org_short_name := 'Office of CEO/COO ABC Inc.';
   dbms_output.put_line('An example record:');
```

```
   dbms_output.new_line;
   dbms_output.put_line(to_number(v_example_rec.hrc_org_id)||' '||
                                  v_example_rec.hrc_descr||' '||
                                  v_example_rec.org_short_name);
END;
/
```

In this example, the individual record elements are assigned values and then they're accessed to display these values. Notice how the record elements are referenced using the dot notation when both populating the record variable and referencing the fields in the record variable.

A PL/SQL record variable is a read-write variable.You can use it on both sides of an assignment operator.

In addition to accessing individual elements of a record, you can access the **entire** record itself in some cases—for example, when you initialize a record with another record or when you pass a record as a parameter to a subprogram. Here's an example to illustrate this:

```
DECLARE
   TYPE hrc_org_rec IS RECORD
        (hrc_org_id NUMBER,
         hrc_descr VARCHAR2(20),
         org_short_name VARCHAR2(30));
   v_example_rec1 hrc_org_rec;
   v_example_rec2 hrc_org_rec;
BEGIN
   v_example_rec1.hrc_org_id := 1001;
   v_example_rec1.hrc_descr := 'CEO/COO';
   v_example_rec1.org_short_name := 'Office of CEO/COO ABC Inc. ';
   v_example_rec2 := v_example_rec1;
   dbms_output.put_line('An example record: ');
   dbms_output.new_line;
   dbms_output.put_line(to_number(v_example_rec2.hrc_org_id)|| ' '||
                                  v_example_rec2.hrc_descr||' '||
                                  v_example_rec2.org_short_name);
END;
/
```

Notice the assignment statement

```
  v_example_rec2 := v_example_rec1;
```

where the first record as a whole is referenced using the record name and used as input to a second record that's also referenced as a whole.

Testing for Equality of Records

In the previous program, two example records were defined with the same structure. The first record was populated using explicit assignment of its individual fields, and this whole record was assigned to the second record. Now, how do you test these records for equality? This section provides the answer to that question.

You test for equality of each of the individual fields one-to-one in both records. Testing for equality using the record names as a whole won't work. That means the following code results in an error:

```
IF (v_example_rec1 = v_example_rec2) THEN
  ... ...
END IF;
```

Instead, you have to use the following:

```
IF ((v_example_rec1.hrc_org_id = v_example_rec2.hrc_org_id) AND
  ((v_example_rec1.hrc_descr = v_example_rec2.hrc_descr) AND
  ((v_example_rec1.org_short_name = v_example_rec2.org_short_name) THEN
 ... ...
END IF;
```

Here's a complete example:

```
DECLARE
 TYPE hrc_org_rec IS RECORD
         (hrc_org_id NUMBER,
        hrc_descr VARCHAR2(20),
        org_short_name VARCHAR2(30));
   v_example_rec1 hrc_org_rec;
   v_example_rec2 hrc_org_rec;
BEGIN
   v_example_rec1.hrc_org_id := 1001;
   v_example_rec1.hrc_descr := 'CEO/COO';
   v_example_rec1.org_short_name := 'Office of CEO/COO ABC Inc. ';
   v_example_rec2.hrc_org_id := 1002;
   v_example_rec2.hrc_descr := 'VP';
   v_example_rec2.org_short_name := 'Office of VP ABC Inc. ';
  IF ((v_example_rec1.hrc_org_id = v_example_rec2.hrc_org_id) AND
      (v_example_rec1.hrc_descr = v_example_rec2.hrc_descr) AND
      (v_example_rec1.org_short_name = v_example_rec2.org_short_name)) THEN
      dbms_output.put_line('Both example records are identical.');
  ELSE
     dbms_output.put_line('The two example records are different. ');
  END IF;
END;
/
```

3.32 Record Initialization

A record type is a composite data type, and as such, the rules for initializing record type variables are different from those for initializing scalar type variables. To initialize a scalar variable, you simply provide an initial value in its declaration using the DEFAULT or := syntax. To initialize a record variable, you have to assign a second record variable that's compatible with the record variable you're initializing. However, you still have to use the DEFAULT or := syntax.

Record initialization is useful when you're passing records as parameters to subprograms. You might pass a record as parameter, declare a local record variable of the same type as that of the parameter, and then initialize this local variable with the parameter you're passing.

A compatible record *is a record based on the same record type as the target record*

Initializing a record to NULL is only possible by initializing each of its individual fields to NULL. You can't assign the record as a whole to NULL, because NULL is treated by PL/SQL as a scalar value.

3.33 Record Assignment

In the simplest terms, *record assignment* means assigning a record with values. This in turn implies two things: populating the individual fields of a record and assigning a record to another record. The latter is called *aggregate assignment.* PL/SQL 9*i* allows you to assign a record with values in the following four ways:

1. Individual field assignment
2. Populating a record with a SELECT INTO (using an implicit cursor)
3. Populating a record with FETCH INTO (using an explicit cursor)
4. Assigning a record using a second record (aggregate assignment)

I discuss each of the methods in the following sections.

Individual Field Assignment

This method consists of assigning each of the individual fields in the record with data values. The fields are referenced using the record_name.field_name notation on the left side of the assignment operator and assigned with values. In the hrc_org_rec record example presented in the section "Accessing Individual Record Elements," you learned how to assign the individual fields with values. Here's the same example reproduced for illustration:

```
DECLARE
  TYPE hrc_org_rec IS RECORD
          (hrc_org_id NUMBER,
         hrc_descr VARCHAR2(20),
         org_short_name VARCHAR2(30));
    v_example_rec hrc_org_rec;
BEGIN
    v_example_rec.hrc_org_id := 1001;
    v_example_rec.hrc_descr := 'CEO/COO';
    v_example_rec.org_short_name := 'Office of CEO/COO ABC Inc. ';
   dbms_output.put_line('An example record: ');
   dbms_output.new_line;
   dbms_output.put_line(to_number(v_example_rec.hrc_org_id)|| ' '||
                                v_example_rec.hrc_descr||' '||
                                v_example_rec.org_short_name);
END;
/
```

You must explicitly assign each field of the record with a value. You can't leave out any field

Populating a Record with SELECT INTO (Using an Implicit Cursor)

A second way to populate a record is with a SELECT INTO statement. The INTO clause can specify the entire record name or it can specify the individual fields of the records using the dot notation. However, the structure of the SELECT column list must exactly match the record structure. This means the data type of each column and the number of columns in the SELECT list should exactly match that of the record fields. The individual column names in the SELECT list can be different. Here's an example of this:

```
DECLARE
 TYPE hrc_org_rec IS RECORD
```

```
      (hrc_org_id NUMBER,    hrc_descr VARCHAR2(20),    org_short_name VARCHAR2(30));
   v_example_rec hrc_org_rec;
BEGIN
   SELECT hrc_org_seq.nextval, h.hrc_descr, o.org_short_name
   INTO    v_example_rec
   FROM   org_tab o, hrc_tab h
   WHERE o.hrc_code = h.hrc_code
        AND o.org_id = 1001;
   dbms_output.put_line('An example record: ');
   dbms_output.new_line;
   dbms_output.put_line(to_number(v_example_rec.hrc_org_id)||' '||
                              v_example_rec.hrc_descr||' '||
                              v_example_rec.org_short_name);
END;
```

Populating a Record with FETCH INTO (Using an Explicit Cursor)

A third way to populate a record is with a FETCH INTO statement resulting from an explicit cursor. As with the implicit cursor, the INTO clause can specify the entire record name or it can specify the individual fields of the records using the dot notation. However, the structure of the cursor SELECT column list must exactly match the record structure. This means the data type of each column and the number of columns in the cursor SELECT list should exactly match that of the record fields. However, the individual column names in the cursor SELECT list can be different. Here's an example of this:

```
DECLARE
 TYPE hrc_org_rec IS RECORD
      (hrc_org_id NUMBER,    hrc_descr VARCHAR2(20),    org_short_name VARCHAR2(30));
   v_example_rec hrc_org_rec;
   CURSOR csr_hrc_org IS
   SELECT hrc_org_seq.nextval, h.hrc_descr, o.org_short_name
   FROM   org_tab o, hrc_tab h
   WHERE o.hrc_code = h.hrc_code
        AND h.hrc_code = 1;
BEGIN
   OPEN csr_hrc_org;
   dbms_output.put_line('An example output: ');
    dbms_output.new_line;
   LOOP
      FETCH csr_hrc_org INTO v_example_rec;
      EXIT WHEN csr_hrc_org%NOTFOUND;
     dbms_output.put_line(to_number(v_example_rec.hrc_org_id)||' '||
                              v_example_rec.hrc_descr||' '||
                              v_example_rec.org_short_name);
  END LOOP;
  CLOSE csr_hrc_org;
END;
/
```

Cursor-Oriented Records

A *cursor-oriented record* is a record that resembles the structure of a PL/SQL cursor. In this case, the individual fields are composed of the columns of the PL/SQL cursor SELECT. The fields in this record correspond one-to-one in name and structure to the columns of the PL/SQL cursor on which the record is based.

You create a cursor-oriented record with the %ROWTYPE operator. Here's the syntax:

record_var_name cursor_name%ROWTYPE;

where record_var_name is the record variable that has the same record structure as a row in the cursor identified by cursor_name. As with table-oriented records, you directly declare cursor-oriented records without having to define the record type. The record type is inherited from the cursor row structure.

You still access the individual fields in a cursor-oriented record using the dot notation.

Here's a complete example of defining and using a cursor-oriented record:

```
DECLARE
   CURSOR csr_hrc IS
       SELECT * FROM hrc_tab ORDER BY 1;
   hrc_rec csr_hrc%ROWTYPE;
BEGIN
   OPEN csr_hrc;
  dbms_output.put_line('Hierarchy records: ');
  dbms_output.new_line;
  LOOP
      FETCH csr_hrc INTO hrc_rec;
      EXIT WHEN csr_hrc%NOTFOUND;
     dbms_output.put_line(to_char(hrc_rec.hrc_code)|| ' '||hrc_rec.hrc_descr);
  END LOOP;
   CLOSE csr_hrc;
END;
/
```

DML Operations (Specifically, INSERT, UPDATE, and SELECT) Involving Entire PL/SQL Records

PL/SQL 9*i* Release 2 allows INSERT and UPDATE operations involving entire PL/SQL records. Instead of specifying a list of individual record attributes, you can insert records into the database using a single variable of type RECORD or %ROWTYPE. The same is allowed for the UPDATE operation. Also, you can do bulk-binding operations involving SELECT, FETCH, INSERT, UPDATE, and RETURNING INTO using a single variable of type as that of a PL/SQL table of records, instead of specifying individual PL/SQL tables for each SQL column.

The INSERT and UPDATE statements are extended in PL/SQL to enable INSERT and UPDATE operations involving entire records. Here's the syntax for the INSERT statement:

INSERT INTO table_name VALUES record_variable;

where table_name is the database table into which a new record is inserted and record_variable is the name of the variable of type RECORD or %ROWTYPE that holds the data to be inserted. The number of fields in the record must be same as the number of columns in the table, and the corresponding record fields and table columns must have data types that match one-to-one. Here's an example to illustrate this:

```
DECLARE
   TYPE hrc_rec IS RECORD
        (hrc_code NUMBER,
         hrc_descr VARCHAR2(20));
   v_example_rec hrc_rec;
BEGIN
   v_example_rec.hrc_code := 99;
```

```
   v_example_rec.hrc_descr := ' Web Analyst';
   INSERT INTO hrc_tab VALUES v_example_rec;
   COMMIT;
END;
/
```

Here's the syntax for the UPDATE statement involving entire records:

UPDATE table_name SET ROW = record_variable [WHERE ...]

where table_name is the database table into which a new record is inserted, record_variable is the name of the variable of type RECORD or %ROWTYPE that holds the data, and the keyword ROW represents an entire row of the table being updated. As in the case of the INSERT statement, the number of fields in the record must be the same as the number of columns in the table, and the corresponding record fields and table columns must have data types that match one-to-one. Here's an example to illustrate this:

```
DECLARE
   TYPE hrc_rec IS RECORD
        (hrc_code NUMBER,
         hrc_descr VARCHAR2(20));
   v_example_rec hrc_rec;
BEGIN
   v_example_rec.hrc_code := 99;
   v_example_rec.hrc_descr := ' Web Analyst Sr.';
   UPDATE hrc_tab SET ROW = v_example_rec WHERE hrc_code = 99;
   COMMIT;
END;
/
```

Using entire records in INSERT and UPDATE statements provides ease of use and the code becomes easily maintainable. However, there are some restrictions for using entire records. Those restrictions are as follows:

- In the case of INSERT statement, you can use a record variable only in the VALUES clause. Also, if the VALUE clause contains a record variable, it's the only variable allowed and you shouldn't give any other variable or value in the VALUES clause.

- In the case of UPDATE statement, a record variable is allowed only on the right side of the SET clause. Also, the keyword ROW is allowed only on the left side of the SET clause, and you can't use it with a subquery. You should specify only one SET clause if you use ROW.

- The record variable you use must not be a nested record type or a return value of a function returning a record type.

4. ERROR MESSAGE HANDLING OVERVIEW

The three main functions of writing programs using any programming language are writing the program, compiling the program, and executing the program. Of these, the compilation and execution stages often result in errors that need to be corrected. Error message handling is part and parcel of any programming language, and PL/SQL is no exception.

4.1 PL/SQL Exceptions: Types and Definition

In PL/SQL, a runtime error is called an *exception.* In its simplest definition, an exception is an error raised at runtime either by Oracle or by the programmer. Once raised, an exception has to be properly handled.

Here are the three basic steps in dealing with exceptions:

1. Declare an exception (implicitly or explicitly).
2. Raise the exception (implicitly or explicitly).
3. Handle the exception (necessary).

Before I deal with the techniques of the performing these steps, I'll cover some basic concepts related to PL/SQL error messaging.

All PL/SQL exceptions are characterized by

- *Error type:* This indicates whether the error is an ORA error or a PLS error.
- *Error code:* This is a number indicating the error number.
- *Error text:* This is the text of the error message, including the error code.

Both compilation and runtime errors have an error type, an error code, and error text. All compilation errors have the error type as PLS and runtime errors have the error type as ORA. Sometimes a PLS error is embedded in an outer ORA error. This occurs when you compile anonymous PL/SQL blocks or execute dynamic PL/SQL blocks of code. In the following section, I describe each of these error types.

4.2 Error Type

PL/SQL raises two types of errors: ORA errors and PLS errors. ORA errors are mostly runtime errors and should be handled by the program. PLS errors are mostly compile time errors and should be corrected before you execute the program.

4.3 Error Code

This is a negative number indicating the error code of the particular error.

4.4 Error Text

Error text is an error message specifying the text of the particular error that occurred. The maximum length of an Oracle error message is 512 bytes.

I give examples of both compilation and runtime errors in the following sections, and I also present an example in which a PLS error is embedded within an ORA error.

Here's an ***example of a compilation error***. Consider the following PL/SQL procedure:

```
SQL> create or replace procedure
  2   p1 is
  3  begin
  4   null
  5  end;
  6  /
```

Here's the output of this code:

Warning: Procedure created with compilation errors.

To see the actual error, use the SQL*Plus command show errors at the SQL prompt. Here are the results:

```
SQL> show errors
Errors for PROCEDURE P1:
LINE/COL ERROR
-------- -----------------------------------------------------------------
5/1      PLS-00103: Encountered the symbol "END" when expecting one of the
         following:
         ;
         The symbol ";" was substituted for "END" to continue.
```

Here the compilation error is:
PLS-00103: Encountered the symbol "END" when expecting one of the following:;
The symbol ";" was substituted for "END" to continue.The error type is PLS, the error code is –00103, and the error text is the entire message. This error occurred because a semicolon (;) was missing after the null statement in line 4 of the procedure code.

In this section I present an example of a ***runtime error***. Consider the following procedure:

```
SQL> create or replace procedure p2
  2  is
  3    v_descr VARCHAR2(20);
  4  begin
  5    SELECT hrc_descr
  6    INTO   v_descr
  7    FROM   hrc_tab
  8    WHERE hrc_code = 10;
  9    dbms_output.put_line(v_descr);
 10  end;
 11  /

Procedure created.
```

Here's the output of executing the procedure:

```
SQL> exec p2
BEGIN p1; END;

*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "PLSQL9I.P2", line 5
ORA-06512: at line 1
```

The runtime error is

```
ORA-01403: no data found
```

The error type is ORA, the error code is –1403, and the error text is the entire message:

```
ORA-01403: no data found
```

Finally, I show an example of a ***PLS error embedded within an ORA error.*** Consider the code of procedure p1. When I convert it into an anonymous PL/SQL block, I get the following:

```
SQL> begin
  2    null
  3  end;
  4  /
end;
*
ERROR at line 3:
ORA-06550: line 3, column 1:
```

PLS-00103: Encountered the symbol "END" when expecting one of the following:
;
The symbol ";" was substituted for "END" to continue.

This is still a compilation error because the PL/SQL compiler first compiles and then executes an anonymous PL/SQL block. However, the corresponding PLS error is embedded within the ORA error –06550.

The following is a list of ***declarations/statements*** used in exception handling:

- *EXCEPTION declaration:* You use this declaration when you declare a user- defined exception.
- *RAISE statement:* You use this statement to raise an exception.
- *PRAGMA EXCEPTION_INIT directive:* You use this directive to associate an Oracle error with a user-defined exception.
- *SAVE EXCEPTIONS clause:* This is new in Oracle9*i* and is used to continue processing after a row-wise failure during bulk binding.

You use the following two ***functions*** to capture information about Oracle errors occurring in PL/SQL:

- *SQLCODE:* This function returns the Oracle error code of the error in question.
- *SQLERRM:* This function returns the Oracle error message text of the error in question.

You use the following ***procedure*** to define custom error messages in PL/SQL:

- *RAISE_APPLICATION_ERROR:* You use this procedure to define custom error messages and halt program execution at the point it is used.

4.5 Exception Handlers

As Chapter 1 outlined, a PL/SQL block is composed of three sections: a declaration section, an executable section, and an exception handling section. The exception handling section is where runtime errors raised implicitly or explicitly are handled. It's also called an *exception handler.*

Although an exception handling section isn't a mandatory requirement to code a PL/SQL block, it's necessary for any PL/SQL program, and it's highly recommended that you code an exception handling section in each PL/SQL program.

An exception handler is specified by the EXCEPTION WHEN … clause. Here's the syntax:

```
EXCEPTION WHEN exception_name THEN
  ... /* code to handle the error */
```

You can specify multiple exceptions as follows:

```
EXCEPTION
   WHEN exception_name1 THEN
      ... /* code to handle the error */
   WHEN exception_name2 THEN
      ... /* code to handle the error */
... ...
WHEN OTHERS THEN
      ... /* code to handle the error */
```

4.6 Types of PL/SQL Exceptions

Basically, exceptions in PL/SQL are classified into the following broad categories:

- Predefined exceptions
- Nonpredefined Oracle errors
- User-defined exceptions
- User-defined PL/SQL error messages

You can track all these types of exceptions either by using the default PL/SQL error-messaging capabilities or by augmenting the default PL/SQL error- messaging capabilities with customized code.

4.7 Handling PL/SQL Exceptions

Recall the three steps you need to perform in exception processing:

1. Define the exception.
2. Raise the exception.
3. Handle the exception.

All types of PL/SQL exceptions, whether predefined or not, are handled by means of exception handlers.

I repeat the “exception handlers” definition here for clarity. Here's the syntax:

```
EXCEPTION WHEN exception_name THEN
   ... /* code to handle the error */
```

You can specify multiple exceptions as follows:

```
EXCEPTION
    WHEN exception_name1 THEN
       ... /* code to handle the error */
    WHEN exception_name2 THEN
       ... /* code to handle the error */
 ... ...
WHEN OTHERS THEN
      ... /* code to handle the error */
```

The WHEN OTHERS clause acts like a bucket, catching any error not explicitly handled. It specifies to handle all other exceptions that are raised whose names aren't listed (above it) in the EXCEPTION handling section. This means it can detect any unhandled errors in the PL/SQL program. If only WHEN OTHERS is specified, it will trap any raised exception.

You have two ways to handle an exception:

- Continue program execution after handling the error
- Stop program execution after handling the error

You can do both using exception handlers. Here's the sequence of steps in the case of an exception occurrence:

1. The exception is raised either implicitly or explicitly.

2. Control transfers to the defined exception handler for the raised exception in the current block. If no exception handler is defined in the current block, control transfers to the exception handler in an outer block defined for the raised exception. In case there is no exception handler defined for the particular exception raised, control transfers to the WHEN OTHERS part of the exception handler either in the current block or in an outer (enclosing) block.

3. The code in the exception handler is executed and program execution resumes or terminates accordingly.

4. If no exception handler has been defined in the entire program, program execution terminates at the point the exception was raised and no code is executed after that point.

4.8 Handling Predefined Exceptions

PL/SQL has defined certain commonly occurring Oracle errors as exceptions. These are called *predefined exceptions,* and each such predefined exception has a unique name associated with it. Every Oracle error has an error type, an error number, and error message text, and these predefined exceptions are no exception. Table 4-1 lists the predefined PL/SQL exceptions along with their error codes and descriptions.

Table 4-1: Predefined PL/SQL Exceptions

EXCEPTION	**ORACLE ERROR CODE/SQL CODE**	**DESCRIPTION**
ACCESS_INTO_NULL	ORA-06530 –6530	Occurs when populating attributes of an uninitialized (atomically null) object.
CASE_NOT_FOUND	ORA-06592 –6592	Occurs in case of a CASE statement defined without an ELSE clause when the selector doesn't match any of the values specified in the WHEN clauses.
COLLECTION_IS_NULL	ORA-06531 –6531	Occurs when populating elements of an uninitialized PL/SQL collection or invoking methods on an uninitialized PL/SQL collection.
CURSOR_ALREADY_OPEN	ORA-06511 –6511	Occurs when trying to open an already opened PL/SQL cursor.
DUP_VAL_ON_INDEX	ORA-00001 –1	Occurs when inserting a record that violates a primary key or unique key, or a unique index.
INVALID_CURSOR	ORA-01001 –1001	Occurs when referencing an unopened cursor.
INVALID_NUMBER	ORA-01722 –1722	Occurs in a SQL statement when the conversion of a character string into a number fails or when the LIMIT clause expression in a bulk FETCH doesn't result in a positive value. In PL/SQL, VALUE_ERROR is raised.

Table 4-1: Predefined PL/SQL Exceptions

EXCEPTION	ORACLE ERROR CODE/SQL CODE	DESCRIPTION
LOGIN_DENIED	ORA-01017 –1017	Occurs when an invalid username/password is specified while logging in.
NO_DATA_FOUND	ORA-01403 +100	Occurs in three cases: first, when a SELECT… INTO implicit cursor returns more than one row; second, when an undefined element is referenced in an index-by table or a deleted element is referenced in a nested table; and third when accessing beyond an EOF while performing PL/SQL file I/O.
NOT_LOGGED_ON	ORA-01012 –1012	Occurs when accessing an Oracle database without logging in.
PROGRAM_ERROR	ORA-06501 –6501	This refers to an internal PL/SQL error.
ROWTYPE_MISMATCH	ORA-06504 –6504	Occurs when the types of a host cursor variable and PL/SQL cursor variable don't match, like a mismatch of the actual and formal parameters of a procedure or function.
SELF_IS_NULL	ORA-30625 –30625	Occurs when a MEMBER method is invoked on a null instance of an object.
STORAGE_ERROR	ORA-06500 –6500	Occurs when there's insufficient memory for processing or there's a memory corruption.
SUBSCRIPT_BEYOND_COUNT	ORA-06533 –6533	Occurs when referencing an element of a PL/SQL collection with an index greater than the largest existing index.
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532 –6532	Occurs when referencing an element of a PL/SQL collection with an index outside the valid range.
SYS_INVALID_ROWID	ORA-01410 –1410	Occurs when the conversion of a character string into a ROWID fails.
TIMEOUT_ON_RESOURCE	ORA-00051 –51	Occurs when a time-out happens when Oracle is waiting for a resource.
TOO_MANY_ROWS	ORA-01422 –1422	Occurs when a SELECT … INTO implicit query returns multiple rows.
VALUE_ERROR	ORA-06502	Occurs when an arithmetic, conversion, truncation, or size- constraint error occurs in

Table 4-1: Predefined PL/SQL Exceptions

EXCEPTION	ORACLE ERROR CODE/SQL CODE	DESCRIPTION
	–6502	PL/SQL statements.
ZERO_DIVIDE	ORA-01476 –1476	Occurs when an attempt is made to divide by zero.

PL/SQL defines these exceptions in a package called STANDARD that's available to all schemas.

In the case of predefined exceptions, the process of exception handling boils down to one step: handling the exception. There's no need to define and raise the predefined exception. PL/SQL automatically does these steps for you.

Here's an example to illustrate handling of predefined exceptions:

```
DECLARE
   v_descr VARCHAR2(20);
BEGIN
    SELECT hrc_descr
    INTO      v_descr
    FROM    hrc_tab
    WHERE  hrc_code = 10;
   dbms_output.put_line(' The hierarchy description for code 10 is: '||v_descr);
EXCEPTION WHEN NO_DATA_FOUND THEN
    dbms_output.put_line('ERR: Invalid Hierarchy Code 10');
END;
/
```

The following points are worth noting:

- The SELECT … INTO doesn't return any row for hrc_code 10, and as a result the NO_DATA_FOUND predefined exception is raised implicitly.
- Control transfers to the exception handler specified by
- EXCEPTION WHEN NO_DATA_FOUND THEN
- The code in this exception handler is executed.
- The code following the SELECT statement isn't executed at all.

Here's the output of this program:

```
ERR: Invalid Hierarchy Code 10

PL/SQL procedure successfully completed.
```

You can write the same program using WHEN OTHERS. Here's the code:

```
DECLARE
   v_descr VARCHAR2(20);
BEGIN
    SELECT hrc_descr
    INTO      v_descr
    FROM    hrc_tab
```

```
    WHERE  hrc_code = 10;
   dbms_output.put_line(' The hierarchy description for code 10 is: '||v_descr);
EXCEPTION WHEN OTHERS THEN
    dbms_output.put_line('ERR: Invalid Hierarchy Code 10');
END;
/
```

The output of this example is the same as the output of the previous example.

The following points are worth noting:

- The SELECT … INTO raises the predefined exception NO_DATA_FOUND, which is caught by the exception handler WHEN OTHERS. However, in this case, it can't be known that the NO_DATA_FOUND exception was raised.

- Even if NO_DATA_FOUND wasn't raised and any other exception was raised, the output would have been the same. This is because WHEN OTHERS traps all possible exceptions occurring in the program.

WHEN OTHERS traps all possible exceptions occurring in a PL/SQL program.

To better improve the error-processing capabilities, it's recommended that you include a WHEN OTHERS handler in addition to the handlers for any specific exceptions that might possibly be raised, such as NO_DATA_FOUND.

```
DECLARE
   v_descr VARCHAR2(20);
BEGIN
    SELECT hrc_descr
    INTO     v_descr
    FROM    hrc_tab
    WHERE  hrc_code = 10;
   dbms_output.put_line(' The hierarchy description for code 10 is: '||v_descr);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
         dbms_output.put_line('ERR: Invalid Hierarchy Code 10');
    WHEN OTHERS THEN
         dbms_output.put_line('ERR: An error occurred');
END;
/
```

SQLCODE returns the error code of the Oracle error that most recently occurred, and ***SQLERRM*** returns the corresponding error message text. Every Oracle error, whether it's a predefined exception or not, has a SQLCODE and a SQLERRM associated with it. Also, SQLERRM always begins with the error type ORA followed by the error code and the error message text. SQLCODE need not necessarily be the error code. A distinctive example of this is the error code –1403, which corresponds to the predefined exception NO_DATA_FOUND. The best use of these two functions would be in the WHEN OTHERS handler. Here's the previous program written in a more meaningful way using SQLCODE and SQLERRM:

```
DECLARE
   v_descr VARCHAR2(20);
BEGIN
    SELECT hrc_descr
    INTO     v_descr
    FROM    hrc_tab
    WHERE  hrc_code = 10;
   dbms_output.put_line(' The hierarchy description for code 10 is: '||v_descr);
```

```
EXCEPTION
   WHEN NO_DATA_FOUND THEN
        dbms_output.put_line('ERR: Invalid Hierarchy Code 10');
   WHEN OTHERS THEN
        dbms_output.put_line('ERR: An error occurred with info :'||
                                TO_CHAR(SQLCODE)||' '||SQLERRM);
END;
/
```

Continuing with Program Execution After Handling the Exception

The previous programs illustrate cases in which the program execution terminates after an exception occurs. However, there may be requirements when the program needs to resume execution after handling an exception. To resume execution after handling an exception, place the particular PL/SQL statement in a nested block with its own exception handlers. Here's an example to illustrate this:

```
DECLARE
  v_descr VARCHAR2(20);
BEGIN
  BEGIN
     SELECT hrc_descr
     INTO     v_descr
     FROM    hrc_tab
     WHERE  hrc_code = 10;
 dbms_output.put_line(' The lowest hierarchy available is: Code 10 '||v_descr);
  EXCEPTION WHEN NO_DATA_FOUND THEN
     INSERT INTO hrc_tab VALUES (10, 'Assistant');
    COMMIT;
  END;
  BEGIN
     SELECT hrc_descr
     INTO     v_descr
     FROM    hrc_tab
     WHERE  hrc_code = 1;
     dbms_output.put_line(' The highest hierarchy available is: Code 1 '||
                                v_descr);
  EXCEPTION WHEN NO_DATA_FOUND THEN
     dbms_output.put_line('ERR: Invalid Data for Hierarchy');
  END;
EXCEPTION
   WHEN OTHERS THEN
      dbms_output.put_line('ERR: An error occurred with info :'||
                                TO_CHAR(SQLCODE)||' '||SQLERRM);
END;
/
```

The following points are worth noting:

- The first SELECT … INTO raises NO_DATA_FOUND and the corresponding handler executes the INSERT statement. This is because this SELECT statement is included in a nested block with its own exception handler.

- Execution resumes after processing the previous exception and the output of the second SELECT … INTO is displayed. Note that this too is included in its own nested block.

- The WHEN OTHERS handler in the outermost block handles any other exceptions occurring in any of the two nested blocks and the ones in the outer block. This is shown in the following code. Any other exceptions in the nested blocks between the lines 4 and 15, and between the lines 16 and 26, is handled by the exception handler in lines 27 through 30.

4.9 User-Defined PL/SQL Error Messages

So far you've dealt with errors that were defined by Oracle SQL and/or PL/SQL and user-defined exceptions. In the former case, the error code and error message text were defined by Oracle and these errors were handled by exception handlers using the predefined exceptions or a WHEN OTHERS exception handler, or by associating user-defined exceptions with Oracle errors. In the latter case, exceptions were defined by the programmer, raised explicitly, and handled using the corresponding exception names.

In addition to Oracle SQL and/or PL/SQL errors and user-defined exceptions, PL/SQL provides you with the flexibility to define your own error messages customized toward the specific application you're coding. This section highlights the methods for defining user-defined error messages and handling them.

4.10 Defining User-Defined Error Messages in PL/SQL

To define customized error messages, PL/SQL provides the procedure RAISE_APPLICATION_ERROR. Here's the signature of this procedure:

RAISE_APPLICATION_ERROR(error_no IN NUMBER, error_message IN VARCHAR2, [keep_errors IN BOOLEAN]);

where error_no is any number between –20000 and –20999, error_message is the customized error message of the error being raised of a length that doesn't exceed 512 bytes, and keep_errors is a boolean flag. Here's an example of using this procedure:

RAISE_APPLICATION_ERROR(-20000, 'Organization 1010 does not exist.', FALSE);

The preceding statement means that the customized error message is specific to a particular application and has the error number –20000 and the error message text

Organization 1010 does not exist.

The keep_errors boolean flag is optional. It has a default value of FALSE. If a value of TRUE is passed for this flag, the new error defined in this way is added to the list of already raised errors (if one exists). A value of FALSE will replace the current list of errors with the new error defined in this way.

Note that in all the earlier cases, when trapping errors, messages were given using DBMS_OUTPUT.PUT_LINE, and as such, these messages were being displayed to the screen output in tools such as SQL*Plus. But PL/SQL is a procedural language widely used in many kinds of development and execution environments—for example, client-server environments such as Oracle Developer, embedded SQL and PL/SQL applications, and so forth. Also, PL/SQL, being the major procedural language for Oracle, is used to code subprograms and database triggers that encapsulate business logic in the database. Imagine an error-definition and handling mechanism in such situations. This is where RAISE APPLICATION_ERROR comes in as a handy mechanism to raise customized server-side errors and propagate them to the client side.

4.11 Handling User-Defined Error Messages in PL/SQL

Whenever a user-defined error message is defined with a call to RAISE_APPLICATION_ERROR, it behaves as if an exception has been raised. Execution of the PL/SQL program stops at this point and the program either terminates with the defined error or returns the customized error number and message text to the calling environment.

The procedure RAISE_APPLICATION_ERROR can perform two functions:

- Define customized error messages.
- Propagate a server-side customized error number and message to a client program in a client-server environment that can be then detected by exception handlers such as WHEN_OTHERS.

I describe the method of handling such errors in each case in the sections that follow.

Here's a complete example to demonstrate the use of RAISE_APPLICATION_ERROR:

```
CREATE OR REPLACE PROCEDURE org_proc
                    (p_flag_in              VARCHAR2,
                     p_hrc_code             NUMBER,
                     p_org_id               NUMBER,
                     p_org_short_name       VARCHAR2,
                     p_org_long_name        VARCHAR2)
IS
  v_error_code NUMBER;
BEGIN
      IF (p_flag_in = 'I') THEN
        BEGIN
           INSERT INTO org_tab VALUES
                (p_hrc_code, p_org_id, p_org_short_name, p_org_long_name);
        EXCEPTION WHEN OTHERS THEN
            v_error_code := SQLCODE;
            IF v_error_code = -1 THEN
              RAISE_APPLICATION_ERROR(-20000, 'Organization '||  TO_CHAR(p_org_id)||
         ' already exists. Cannot create a duplicate with the same id.');
            ELSIF v_error_code = -2291 THEN
RAISE_APPLICATION_ERROR(-20001, 'Invalid Hierarchy Code '|| TO_CHAR(p_hrc_code)||
                          ' specified. Cannot create organization.');
          END IF;
       END;
ELSIF (p_flag_in = 'C') THEN
        BEGIN
           UPDATE org_tab
               set org_short_name = p_org_short_name,
                   org_long_name = p_org_long_name
           WHERE hrc_code = p_hrc_code
               AND org_id = p_org_id;
           IF SQL%NOTFOUND THEN
             RAISE_APPLICATION_ERROR(-20002, 'Organization '||  TO_CHAR(p_org_id)||
                  ' does not exist. Cannot change info for the same.');
           END IF;
        END;
ELSIF (p_flag_in = 'D') THEN
```

```
        BEGIN
           DELETE org_tab
           WHERE hrc_code = p_hrc_code
                AND org_id = p_org_id;
           IF SQL%NOTFOUND THEN
              RAISE_APPLICATION_ERROR(-20003, 'Organization '||  TO_CHAR(p_org_id)||
                   ' does not exist. Cannot delete info for the same.');
           END IF;
        EXCEPTION WHEN OTHERS THEN
           v_error_code := SQLCODE;
           IF v_error_code = -2292 THEN
              RAISE_APPLICATION_ERROR(-20004, 'Organization '||    TO_CHAR(p_org_id)||
           'has site details defined for it. Cannot perform delete operation.');
          END IF;
       END;
END IF;
END;
/
```

The following points are worth noting:

- This procedure calls the RAISE_APPLICATION_ERROR in several places with different error numbers and error messages that are customized. These aren't part of the ORA errors defined by Oracle.

- At each execution of RAISE_APPLICATION_ERROR, processing stops and the program is terminated with the customized error message as output.

Here's the output of several instances of executing this procedure:

```
SQL> exec org_proc('I',1,1001,'Office of CEO ABC Inc.','Office of CEO ABC Inc.');
BEGIN org_proc('I',1,1001,'Office of CEO ABC Inc.','Office of CEO ABC Inc.');
END;
*
ERROR at line 1:
ORA-20000: Organization 1001 already exists. Cannot create a duplicate with the same id.
ORA-06512: at "PLSQL9I.ORG_PROC", line 16
ORA-06512: at line 1

SQL> exec org_proc('I',6,1011,'Office of Mgr ABC Inc.','Office of Mgr ABC Inc.');
BEGIN org_proc('I',6,1011,'Office of Mgr ABC Inc.','Office of Mgr ABC Inc.');
END;
*
ERROR at line 1:
ORA-20001: Invalid Hierarchy Code 6 specified. Cannot create organization.
ORA-06512: at "PLSQL9I.ORG_PROC", line 18
ORA-06512: at line 1

SQL> exec org_proc('D',1, 1001, null, null);
BEGIN org_proc('D',1, 1001, null, null); END;
*
ERROR at line 1:
ORA-20004: Organization 1001 has site details defined for it. Cannot perform delete
operation.
ORA-06512: at "PLSQL9I.ORG_PROC", line 41
ORA-06512: at line 1
```

4.12 Tips for PL/SQL Error Message and Exception Handling

The following is a short list of some of the **recommended practices** in error message handling.

- Always use a WHEN OTHERS handler in the outermost level of the PL/SQL program to handle previously unhandled error messages.
- Log an error in an error table with information such as the error code and error message text.
- Use customized error messages tailored to the specific application and categorize these messages based on their severity level.
- Don't override predefined exception names when defining user-defined exceptions.
- Don't define duplicate user-defined exceptions when dealing with nested blocks.
- Write error-handling routines that separate error processing from business logic processing.

5. STORED SUBPROGRAMS (PROCEDURES, FUNCTIONS, AND PACKAGES)OVERVIEW

The universal concept of a PL/SQL program is that the PL/SQL block could be one of three types: an anonymous block, a labeled block, or a named block. The PL/SQL block is quite handy in coding PL/SQL programs tailored toward a specific function and employing other PL/SQL programming constructs. However, it has a disadvantage: You can't store it in the database like you can a table. If you need to share a block within another application, then you must rewrite it for that specific application. How easy and efficient would it be if the code were shareable? PL/SQL provides you with a mechanism, the *stored subprogram,* that permits you to share code between applications. A stored subprogram has the major advantage of being stored in the database and it's therefore shareable.

A stored subprogram is a named PL/SQL block that's stored in the database. PL/SQL 9*i* supports three types of stored subprograms:

- Procedures
- Functions
- Packages

This brings up the concept of procedures and functions in 3GL languages such as Pascal, C, C++, and Java. PL/SQL extends the 3GL capabilities by providing the stored subprograms feature.

A stored subprogram in PL/SQL has the following features:

- It's uniquely named.
- It's stored in the database.
- It's available as metadata in the data dictionary.
- Parameters can be passed to it, and values (one value or multiple values) can be returned from it.
- It can be executed at the top level or called from other PL/SQL programs by using its name.

In addition to the stored subprograms' capability to be stored in the database, procedures, functions, and packages offer many advantages. Their primary benefits are as follows:

- *Modularity:* Stored subprograms introduce the concept of *modular programming,* with a single, large program being broken down into logically independent modules and each module coded separately. The main program can then be written by calling these individual procedures with any additional code that's required. This enables you to write APIs tailored toward a specific application.
- *Encapsulation:* Stored subprograms enable the encapsulation of business logic, such as business rules and application logic, into the database.
- *Centralization of code:* Because the stored subprograms are stored in the database and run on the server side, they're easy to maintain. The code is available in a single location and multiple applications can use it.
- *Reusability:* As mentioned previously, multiple programs can reuse the same subprogram, and, when necessary, with different sets of data.

- *Better performance:* A subprogram allows you to group together sets of SQL statements, which results in better performance with fewer database calls and reduced network traffic in two-tier and three-tier application environments. Also, as of PL/SQL 9*i*, stored subprograms can make use of native compilation, subsequent conversion to C code, and then storage in machine code as opposed to interpreted code. This enables faster execution.

5.1 Creating and Using Procedures and Functions

Procedures and functions are two of the ways in which the 3GL modular programming capabilities are implemented in PL/SQL in the form of stored objects. This section discusses the method of creating and executing procedures and functions. It also outlines the method of defining parameters to procedures and functions. The concept of native compilation of stored subprograms is highlighted.

A procedure or function consists of four parts:

- A signature or header
- The keyword IS or AS
- Local declarations (optional)
- The body of the procedure (including exception handlers) between BEGIN and END

5.2 Creating and Using a Procedure

You create a procedure with the CREATE OR REPLACE PROCEDURE statement. Here's the syntax:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name1 parameter_mode datatype,
  ... ...
parameter_nameN parameter_mode datatype)]
[AUTHID {DEFINER | CURRENT_USER}]
{IS | AS}
  [PRAGMA AUTONOMOUS_TRANSACTION;]
  [local declaration section]
BEGIN
  executable section
[EXCEPTION
  exception handling section]
END [procedure_name];
```

where procedure_name is the name of the procedure being created, parameter1 through parameterN are the names of the procedure parameters, parameter_mode is one of [{IN | OUT [NOCOPY] | IN OUT [NOCOPY]}], and datatype is the data type of the associated parameter. The local declaration section is specified without the DECLARE keyword and the BEGIN … END section resembles that of a regular PL/SQL block. The AUTHID clause specifies the execution privileges of the procedure and is explained in the section "Definer and Invoker Rights" later in this chapter. The PRAGMA refers to autonomous transactions and is explained in Chapter 8. For the purposes of this section, we ignore the AUTHID clause and the PRAGMA while defining procedures or functions.

Here's an example of a simple procedure:

```
CREATE OR REPLACE PROCEDURE show_line  (ip_line_length IN NUMBER,
                                        ip_separator IN VARCHAR2)
IS
  actual_line VARCHAR2(150);
BEGIN
    FOR idx in 1..ip_line_length LOOP
      actual_line := actual_line || ip_separator;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE(actual_line);
EXCEPTION WHEN OTHERS THEN
   dbms_output.put_line(SQLERRM);
END;
/
```

Once you've created the show_line procedure, you can call it from a PL/SQL block as follows:

```
BEGIN
  show_line(50, '=');
END;
/
```

The following points are worth noting:

- The signature or the header of the procedure is

  show_line (ip_line_length IN NUMBER, ip_separator IN VARCHAR2).

This item is called a signature or header because it identifies the procedure uniquely and is used to execute the procedure or call it from another procedure or a PL/SQL block, or from the client side. Here, the name of the procedure is show_line and it takes two input parameters named ip_line_length and ip_separator. The ip_line_length parameter is a numeric parameter and is of IN mode. The ip_separator parameter is of type VARCHAR2 and is also of IN mode. Note that the length of the parameters isn't specified. This length is determined when the actual data values are passed when the procedure is executed or called.

- The keyword IS follows the signature.
- The local declaration section is
- actual_line VARCHAR2(150);
- The procedure body is the code between BEGIN and END.
- The procedure is called as an executable statement in the PL/SQL block.

5.3 Creating and Using a Function

A *function* is a stored subprogram that returns a value. You create a function with the CREATE OR REPLACE FUNCTION statement. Here's the syntax:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name1 parameter_mode datatype,
  ... ...
parameter_nameN parameter_mode datatype)]              RETURN datatype

[AUTHID {DEFINER | CURRENT_USER}]
{IS | AS}
  [PRAGMA AUTONOMOUS_TRANSACTION;]
  [local declaration section]
BEGIN
  executable section
[EXCEPTION
  exception handling section]
END [function_name];
```

The definition of a function differs from that of a procedure in the signature. The signature of a function has an additional RETURN clause that specifies the data type of the return value of the function. It's mandatory that you specify the RETURN clause while defining a function.

Here's an example of a simple function:

```
CREATE OR REPLACE FUNCTION f_line
                (ip_line_length IN NUMBER,
                 ip_separator   IN VARCHAR2)
RETURN VARCHAR2
IS
  actual_line VARCHAR2(150);
BEGIN
    FOR idx in 1..ip_line_length LOOP
      actual_line := actual_line || ip_separator;
    END LOOP;
    RETURN (actual_line);
EXCEPTION WHEN OTHERS THEN
   dbms_output.put_line(SQLERRM);
   RETURN (null);
END;
/
```

Once you've created the f_line function, you can call it from a PL/SQL block as follows:

```
DECLARE
    v_line VARCHAR2(150);
BEGIN
    v_line := f_line(50, '=');
    DBMS_OUTPUT.PUT_LINE(v_line);
END;
/
```

The following points are worth noting:

- The signature or header of the function is
- f_line (ip_line_length IN NUMBER, ip_separator IN VARCHAR2) RETURN VARCHAR2.
- The keyword IS follows the signature.
- The local declaration section is
- actual_line VARCHAR2(150);
- The procedure body is the code between BEGIN and END.

Functions can be called from DML statements such as SELECT, INSERT, and so forth.

Calling a Function and the RETURN Statement

Notice the RETURN statement inside the body of the function. It's necessary to specify this. It returns a value to the calling environment.

You can specify more than one RETURN statement in the body of the function.

As soon as the RETURN statement is encountered, the function execution stops and the control returns to the calling environment.

The RETURN statement has two general forms:

RETURN;

You can use in procedures. It stops the execution of the procedure at that point and transfers control to the calling program. It *can't* return a value.

RETURN expression;

Must be used in functions. It not only transfers control to the calling program, but it also returns a value.

5.4 Executing a Procedure or a Function

The owner of a procedure or a function can execute it, or else the EXECUTE privilege should be granted to a user other than the owner. You can do this with the GRANT statement. Here's the syntax:

GRANT EXECUTE ON sub_program_name to user_name;

where sub_program_name is the name of the procedure or function and user_name is the schema name executing the procedure.

You can execute a procedure or function in one of three ways:

- As an executable statement or as part of an expression from a PL/SQL block.
- Using the EXECUTE command in interactive environments such as SQL*Plus.
- Using the CALL statement.

The EXECUTE command enables you to execute a procedure or function in SQL*Plus. Here's the syntax for executing procedures:

EXECUTE procedure_name(arg1, ... , argN)

where procedure_name is the name of the procedure being executed and arg1, …, argN are the arguments with a one-to-one correspondence to the parameters defined in the signature of the procedure.

Here's the show_line procedure being executed in this way:

```
SQL> set serverout on;
SQL> exec show_line(50, '=')
```

The EXECUTE command for function takes a different form. Here's the syntax:

EXECUTE :var := function_name(arg1, ... , argN)

where var is a SQL*Plus variable with the same data type as the return type of the function and is defined using the VARIABLE command in SQL*Plus; function_name is the name of the function being executed; and arg1, …, argN are the arguments with a one-to-one correspondence to the parameters defined in the signature of the function. You can see the value of the var variable using the PRINT command in SQL*Plus.

Here's the f_line function executed in this way:

```
SQL> VARIABLE var VARCHAR2(150)
SQL> EXECUTE :var := f_line(50,'=')
SQL> PRINT var
```

The third way of calling procedures and functions is using the CALL statement. This is a new SQL statement introduced in Oracle8*i*. Here's the syntax:

CALL proc_or_func_name([arg1, ... , argN]) [INTO host_variable];

where proc_or_func_name is the name of the procedure or function being executed; arg1, …, argN are the arguments with a one-to-one correspondence to the parameters defined in the signature of the procedure or function; and host_variable is a host variable to store the return value of functions. The INTO clause is used only for functions.

Here are the show_line procedure and the f_line function executed using the CALL statement:

```
SQL> CALL show_line(50, '=');
SQL> VARIABLE v_line VARCHAR2(150);
SQL> CALL f_line(50, '=') INTO :v_line;
SQL> PRINT v_line
```

The parentheses are necessary even when no arguments are specified. Here's an example:

```
CREATE OR REPLACE PROCEDURE p1
IS
BEGIN
   dbms_output.put_line('Welcome!!! ');
END;
/
```

Here's the output of executing this procedure using the CALL statement:

```
SQL> CALL p1();
Welcome!!!

Call completed.
```

Omitting the parentheses results in this error:

```
SQL> CALL p1;
CALL p1
   *
ERROR at line 1:
```

ORA-06576: not a valid function or procedure name

5.5 Specifying Procedure or Function Parameters

We discuss the following methods of specifying parameters in the following sections:

- Naming parameters
- Specifying parameter modes
- Specifying the type for parameters
- Specifying default values for parameters

Naming Parameters

Naming parameters consists of distinguishing between actual parameters and formal parameters. As you learned in the earlier sections, when a procedure or function is defined with parameters, the signature consists of parameter names and their corresponding data types. These are called *formal parameters,* and the procedure or function body references them using these names. When the subprogram is executed, the actual values corresponding to the formal parameters are passed. These are termed *actual parameters* and can be in the form of literals or constants, or as variables holding values.

As an example, consider the procedure show_line. The signature of the procedure is

show_line (ip_line_length IN NUMBER, ip_separator IN VARCHAR2)

The two parameters, ip_line_length and ip_separator, are the formal parameters of the procedure. These parameters are referenced inside the body of the procedure.

You can pass the actual parameters to a procedure or function in three ways:

1. positional notation (es.1 and es.2)

```
BEGIN
   show_line(50, '=');
END;
/
DECLARE
    v_length NUMBER := 50;
    v_separator VARCHAR2(1) := '=';
BEGIN
    show_line(v_length, v_separator);
END;
/
```

2. named notation,

Named notation uses the name of the formal parameter followed by an arrow and then the actual parameter name in the call of the procedure. Here's the show_line procedure called in this way:

```
DECLARE
    v_length NUMBER := 50;
    v_separator VARCHAR2(1) := '=';
BEGIN
    show_line(ip_line_length=>v_length, ip_separator=>v_separator);
```

```
    show_line(ip_separator=>v_separator, ip_line_length=>v_length);
END;
/
```

3. and mixed notation.

The third notation is *mixed notation,* in which you can mix positional and named notation. In this case, positional notation must precede named notation. Here's an example:

```
DECLARE
    v_length NUMBER := 50;
    v_separator VARCHAR2(1) := '=';
BEGIN
    show_line(v_length, ip_separator=>v_separator);
END;
/
```

Specifying Parameter Modes

As I mentioned earlier, a procedure can take parameters and perform an action, whereas a function can take parameters and return a single value to the calling environment. Now, two questions arise:

1. Can a procedure return a value to the calling environment?

2. Can a procedure or function return multiple values to the calling environment?

The answer to both of these questions is yes.

PL/SQL provides three parameter modes for formal parameters: IN, OUT, and IN OUT.

The IN mode acts as a read-only specifier so that the value of the called formal parameter can't be changed inside the value of the subprogram.

If you omit the mode while you define the signature, IN mode is assumed by default.

The OUT mode acts as a write specifier and allows a value to be assigned to it inside the body of the subprogram. Thus, an OUT parameter acts like an uninitialized variable that can be read from and written into. Moreover, the OUT mode enables you to return a value to the calling environment using the actual parameter. Therefore, the actual parameter corresponding to an OUT formal parameter must be a variable. The contents of the formal parameter are assigned to the actual parameter in this case, and the modified value is available to the calling environment by means of the actual parameter, provided the subprogram call ends successfully.

Here's the show_line procedure modified to include an OUT parameter:

```
CREATE OR REPLACE PROCEDURE show_line2
                (ip_line_length     IN          NUMBER,
                 ip_separator       IN          VARCHAR2,
                 op_line OUT        VARCHAR2)
IS
  actual_line VARCHAR2(150);
BEGIN
    FOR idx in 1..ip_line_length LOOP
      actual_line := actual_line || ip_separator;
    END LOOP;
```

```
   op_line := actual_line;
EXCEPTION WHEN OTHERS THEN
   dbms_output.put_line(SQLERRM);
   op_line := null;
END;
/
```

You can then call this procedure as follows:

```
DECLARE
    v_length          NUMBER  := 50;
    v_separator       VARCHAR2(1)         := '=';
    v_line  VARCHAR2(150);
BEGIN
    show_line2(v_length, v_separator, v_line);
   dbms_output.put_line(v_line);
END;
/
```

The following points are worth noting:

- The actual parameter v_line corresponds to the OUT formal parameter op_line and is defined as an uninitialized variable.
- The value of the v_line actual parameter is available to the calling program, after the call to show_line2. This is what I mean when I say the procedure returns a value.

The IN OUT mode acts as a read-write specifier and allows a value to be read from the actual parameter and a modified value to be returned to the calling program. Thus, an IN OUT parameter acts as an initialized variable inside the body of the subprogram. You must explicitly assign a value to the formal parameter in IN OUT mode. Also, the actual parameter corresponding to an IN OUT formal parameter must be a variable.

In IN mode, the parameter is passed by reference (i.e., a pointer to the value is passed),

in OUT and IN OUT modes, the parameter is passed by value. This is the default behavior.

Multiple values can be returned by a procedure using multiple OUT parameters. Functions can also be defined using OUT or IN OUT parameters. In this case, the function returns values in addition to the function return value.

It's **recommended** that you use functions without OUT or IN OUT parameters. Use a procedure instead.

Specifying the Type for Parameters

The data type of the formal parameters can be any valid PL/SQL type. However, the data type shouldn't specify any length or precision for the formal parameter. This is because the length or precision is determined by the value of the actual parameter corresponding to the formal parameter.

Now what about data types such as SUBTYPE, %TYPE, and %ROWTYPE? You can define data types such as SUBTYPE as subtypes of the primary types. In this case, you must define the actual parameter as having a data type as the subtype and not the primary type. Here's an example to illustrate this:

```
declare
```

```
  subtype st1 is number not null;
 v_value NUMBER;
  procedure p1(ip_1 st1)
  is
  begin
    dbms_output.put_line(to_char(ip_1));
  end;
begin
       p1(v_value);
end;
/
```

This raises the following error:

```
declare
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 10
```

You can define subtypes with constraints such as NOT NULL as shown in the previous code. In this case, the value of the actual parameter being passed should be non-null. Here's an example to illustrate this:

```
declare
  subtype st1 is number not null;
  procedure p1(ip_1 st1)
  is
  begin
    dbms_output.put_line(to_char(ip_1));
  end;
begin
       p1(null);
end;
/
```

Here's the output of the preceding code:

```
p1(null);
        *
ERROR at line 9:
ORA-06550: line 9, column 14:
PLS-00567: cannot pass NULL to a NOT NULL constrained formal parameter
ORA-06550: line 9, column 11:
PL/SQL: Statement ignored
```

Also, data types such as %TYPE and %ROWTYPE have length or precision limitations depending on the underlying column.

This means that declaring a formal parameter as type %TYPE or %ROWTYPE restricts its length or precision to a certain value.

If the actual parameter value has a length or precision greater than that of the formal parameter subtype as NOT NULL, you get the ORA-06502 error.

The data type of the formal parameter can also be:

- a composite PL/SQL data type such as an index-by table or a record (either user-defined or %ROWTYPE).

- an index-by table of records is also allowed.
- a REF CURSOR (strong or weak)
- a user-defined object type or collection.

Specifying Default Values for Parameters

We can specify default values for the formal parameters while you define the procedure or function, and you need not pass an actual parameter to the subprogram call, provided you follow certain rules.

Here's the syntax for providing default values for a formal parameter:

parameter_name [parameter_mode] data_type [:= | DEFAULT] default_value

where parameter_name is the name of the formal parameter; parameter_mode is either IN, OUT, or IN OUT; data_type is a predefined or user_defined data type of said parameter; and default_value is the type-compatible value of the parameter assigned by default.

Here's an example of the show_line procedure with a default value for one parameter:

```
CREATE OR REPLACE PROCEDURE show_line
                (ip_line_length IN NUMBER,
                 ip_separator IN VARCHAR2 DEFAULT '=')
IS
  actual_line VARCHAR2(150);
BEGIN
   FOR idx in 1..ip_line_length LOOP
     actual_line := actual_line || ip_separator;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE(actual_line);
EXCEPTION WHEN OTHERS THEN
   dbms_output.put_line(SQLERRM);
END;
/
```

The call to this procedure can be as follows:

```
BEGIN
   show_line(50);
END;
/
```

The following points are worth noting:

- The show_line procedure is called by passing only one actual parameter. The value for the second parameter is derived from the default value of its formal parameter.
- If the first formal parameter also has a default value, then the show_line procedure can be called with no arguments at all. However, to omit the value for the first parameter and provide an actual parameter for the second, you must use named notation.

Here's an example to illustrate the second point:

```
CREATE OR REPLACE PROCEDURE show_line
                (ip_line_length     IN NUMBER           DEFAULT 50,
```

```
                ip_separator       IN VARCHAR2          DEFAULT '=')
IS
  actual_line VARCHAR2(150);
BEGIN
    FOR idx in 1..ip_line_length LOOP
      actual_line := actual_line || ip_separator;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE(actual_line);
EXCEPTION WHEN OTHERS THEN
    dbms_output.put_line(SQLERRM);
END;
/
```

The call to this procedure can be as follows:

```
BEGIN
     show_line;
END;
/
```

However, the following call results in an error:

```
BEGIN
   show_line('_');
END;
/
```

Although the ip_line_length formal parameter has a default value, PL/SQL interprets the previous call as though '_' is being passed for the ip_line_length parameter, and hence gives a value error as shown here:

```
SQL> BEGIN
  2       show_line('_');
  3  END;
  4  /
BEGIN
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character to number conversion error
     ORA-06512: at line 2
```

To avoid this, you should use named notation as follows:

```
BEGIN
  show_line(ip_separator=>'_');
END;
/
```

6. PL/SQL PACKAGES

A *package* is a stored program unit that groups logically related PL/SQL objects and stores them in the database as a single object.

The related PL/SQL objects can be constants, variables, cursors, exceptions, procedures, and/or functions.

A PL/SQL package has two parts:

- the package specification
- the package body.

The *package specification* consists of the declarations of the related objects to be included in the package. In the case of procedures and functions, only the signature or the header of the procedure or function is included in the package specification.

The actual implementation of these procedures and functions is given in the *package body.* In addition, the package body may contain other declarations such as variables, constants, cursors, and even procedures and functions not defined in the package specification.

The declarations in the package specification are called *public declarations,* as their scope is the entire database session, and all application programs having access to said package can reference them.

The declarations and additional procedures and functions defined in the package body are termed *private* because they're accessible only to the creator of the package. In this way, you can reference them only in the code contained in the package body and not by other PL/SQL applications.

Other advantages of packages include encapsulation, modularity, and better performance.

6.1 Creating and Using a Package

Creating a package involves creating a specification and a body separately. Both are stored separately in the database. You create a package specification with the CREATE OR REPLACE PACKAGE statement. Here's the syntax:

```
CREATE [OR REPLACE] PACKAGE package_name
            [AUTHID {CURRENT_USER | DEFINER}]
                        {IS | AS}
[PRAGMA SERIALLY_REUSABLE;]
[type_definition ... ]
[constant_declaration ... ]
[variable_declaration ... ]
[exception_declaration ... ]
[cursor_declaration ... ]
[procedure_header ... ]
[function_header ... ]
END [package_name];
```

where package_name is the name of the package being created and the definitions correspond to type definitions, constants, variables, exceptions, cursors, procedure signatures, and/or function signatures. The AUTHID clause specifies the execution privileges of the package and is explained in the section "Definer and Invoker Rights" later in this chapter. PRAGMA

SERIALLY_REUSABLE refers to serially reusable packages and is explained in the section "Additional Package Features" of this chapter. Here's an example of a simple package:

```
CREATE OR REPLACE PACKAGE orgMaster
IS
  max_sites_for_an_org NUMBER;

  TYPE rc IS REF CURSOR;

  PROCEDURE createOrg (  ip_hrc_code                    NUMBER,
                         ip_org_id                      NUMBER,
                         ip_org_short_name              VARCHAR2,
                         ip_org_long_name               VARCHAR2,
                         op_retcd                OUT    NUMBER,
                         op_err_msg              OUT    VARCHAR2);

PROCEDURE updateOrg(    ip_org_id                      NUMBER,
                         ip_org_short_name              VARCHAR2,
                         ip_org_long_name               VARCHAR2,
                         op_retcd                OUT    NUMBER,
                         op_err_msg              OUT    VARCHAR2);

PROCEDURE removeOrg(ip_org_id                           NUMBER,
                        op_retcd            OUT    NUMBER,
                        op_err_msg                 OUT    VARCHAR2);
FUNCTION getOrgInfo(   ip_org_id                        NUMBER)          RETURN rc;

FUNCTION getAllOrgs(          ip_hrc_code                        NUMBER)
            RETURN rc;

PROCEDURE assignSiteToOrg(ip_org_id NUMBER,
                                ip_site_no                      NUMBER,
                                op_retcd            OUT    NUMBER,
                                op_err_msg                 OUT    VARCHAR2);
END orgMaster;
/
```

Once you've created the package specification, you should create a corresponding package body. Here's the syntax:

```
[CREATE [OR REPLACE] PACKAGE BODY package_name
{IS | AS}
  [PRAGMA SERIALLY_REUSABLE;]
[private_declarations]
[cursor_body ... ]
[public_procedure_implementation ... ]
[public_function_implementation ... ]
[BEGIN
  sequence_of_statements]
END [package_name];]
```

where package_name is the name of the package as specified in the package specification, private_declarations corresponds to declarations being included only in the package body, cursor_body is the definition of the cursor along with the associated SELECT statement, and public_procedure_implementation and public_function_implementation are the procedure and function bodies corresponding to the signatures defined in the package specification. Also,

[BEGIN
  sequence_of_statements]

is an initialization section of the package and is explained later in this section.

Here's an example of a package body for the package org_master defined previously:

```
CREATE OR REPLACE PACKAGE BODY orgMaster
IS
-- Procedure to create a new Org record in org_tab
  PROCEDURE createOrg (ip_hrc_code NUMBER,
                       ip_org_id NUMBER,
                       ip_org_short_name VARCHAR2,
                       ip_org_long_name VARCHAR2,
                       op_retcd OUT NUMBER,
                       op_err_msg OUT VARCHAR2)
 IS
     BEGIN
    INSERT INTO org_tab VALUES
   (ip_hrc_code, ip_org_id, ip_org_short_name, ip_org_long_name);
   op_retcd := 0;
 EXCEPTION WHEN DUP_VAL_ON_INDEX THEN
    op_retcd := -1;
    op_err_msg := 'Organization with Id '||TO_CHAR(ip_org_id)||
                  ' already exists.';
   WHEN OTHERS THEN
       op_retcd := SQLCODE;
       op_err_msg := SQLERRM;
 END createOrg;

-- Procedure to update the short and long names of an Org in org_tab
-- based on input org_id
  PROCEDURE updateOrg(ip_org_id NUMBER,
                      ip_org_short_name VARCHAR2,
                      ip_org_long_name VARCHAR2,
                      op_retcd OUT NUMBER,
                      op_err_msg OUT VARCHAR2)
  IS
  BEGIN
     UPDATE org_tab
     SET org_short_name = ip_org_short_name,
           org_long_name = ip_org_long_name
     WHERE org_id = ip_org_id;
     IF (SQL%NOTFOUND) THEN
        op_retcd := -1;
        op_err_msg := 'Organization with Id '||TO_CHAR(ip_org_id)||
                      ' does not exist.';
        RETURN;
       END IF;
       op_retcd := 0;
  EXCEPTION WHEN OTHERS THEN
        op_retcd := SQLCODE;
        op_err_msg := SQLERRM;
   END updateOrg;
```

```
-- Procedure to delete a record in org_tab
 PROCEDURE removeOrg(ip_org_id NUMBER,
                              op_retcd OUT NUMBER,
                              op_err_msg OUT VARCHAR2)
  IS
  BEGIN
      DELETE org_tab WHERE org_id = ip_org_id;
     IF (SQL%NOTFOUND) THEN
         op_retcd := -1;
         op_err_msg := 'Organization with Id '||TO_CHAR(ip_org_id)||
                       ' does not exist.';
         RETURN;
       END IF;
       op_retcd := 0;
   EXCEPTION WHEN OTHERS THEN
         op_retcd := SQLCODE;
         op_err_msg := SQLERRM;
 END removeOrg;

-- Function to return a row in org_tab for a given org_id.
-- It returns a resultset of type REF CURSOR defined in the package specification
 FUNCTION getOrgInfo(ip_org_id NUMBER) RETURN rc
 IS
   v_rc rc;
 BEGIN
        OPEN v_rc FOR SELECT * FROM org_tab WHERE org_id = ip_org_id;
   RETURN (v_rc);
 EXCEPTION WHEN OTHERS THEN
     RAISE_APPLICATION_ERROR(-20001, SQLERRM);
END getOrgInfo;
-- Function to return all rows in org_tab.
-- It returns a resultset of type REF CURSOR defined in the package specification
    FUNCTION getAllOrgs(ip_hrc_code NUMBER) RETURN rc
 IS
   v_rc rc;
 BEGIN
        OPEN v_rc FOR SELECT * FROM org_tab WHERE hrc_code = ip_hrc_code;
   RETURN (v_rc);
EXCEPTION WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20002, SQLERRM);
END getAllOrgs;

-- Procedure to insert a row into org_site_tab based on
-- input org_id and site_no
 PROCEDURE assignSiteToOrg(ip_org_id NUMBER,
                  ip_site_no NUMBER,
                  op_retcd OUT NUMBER,
                  op_err_msg OUT VARCHAR2)
IS
  v_num NUMBER;
BEGIN
   BEGIN
      SELECT 1
      INTO     v_num
```

```
      FROM     org_site_tab
      WHERE    org_id = ip_org_id
            AND site_no = ip_site_no;
      IF (v_num = 1) THEN
         op_retcd := 0;
         RETURN;
      END IF;
    EXCEPTION WHEN NO_DATA_FOUND THEN
        INSERT INTO org_site_tab VALUES (ip_org_id, ip_site_no);
   END;
   op_retcd := 0;
  EXCEPTION WHEN OTHERS THEN
     op_retcd := SQLCODE;
     op_err_msg := SQLERRM;
  END assignSiteToOrg;
 END orgMaster;
/
```

The following points are worth noting:

- The package specification and the package body have the same name. This is mandatory.

- There is no BEGIN statement at the beginning of the package. Don't confuse this with the BEGIN sequence_of_statements section included at the end. This is an initialization section and doesn't act as a BEGIN for the package.

- The DECLARE keyword isn't used while declaring constants, variables, type definitions, exceptions, and/or cursors.

- The CREATE OR REPLACE clause isn't used while defining procedures and/or functions.

- The public declarations can appear in any order as long they're declared before they're referenced. Also, these shouldn't be repeated in the package body.

- In the case of cursors, the specification can contain the cursor name with its return type. In this case, the cursor is to be defined completely in the package body. If it isn't specified with a return type, the entire cursor along with its associated SELECT statement should be specified in the specification and this doesn't need to be repeated in the package body.

- The procedure and/or function signatures must be repeated in the package body, while giving the corresponding implementation details.

Referencing Package Objects

The individual objects defined in a package specification are global in scope in that they're available for the entire session and accessible by all application programs having the required privileges. Hence, they're termed *public objects*. You can reference public objects in application programs (i.e., outside of the package body) with the following syntax:

package_name.object_name

The following points are worth noting:

- The packaged procedure createOrg is called like a stand-alone stored procedure but it's prefixed with the package name and a dot.

- The procedure is executed and control is transferred to the calling PL/SQL block just like a stand-alone procedure.

However, inside a package body, you can reference the objects defined in its specification without the dot notation.

Private Objects

What about *private objects* defined inside of the package body? These objects are accessible only to the particular package body and you can also reference them without the dot notation.

Consider the org_master package defined previously. The procedure removeOrg removes an organization based on the input org_id. Now, what about the org site details existing for said organization? These first have to be removed, otherwise the deletion of records in the org_tab table will fail. Also, this deletion is transparent to other applications accessing the removeOrg procedure. Hence, you can code a separate procedure inside the package body for deleting the org site details before you delete the org itself. I'll call this procedure removeOrgSites. With this change in place, the package body for org_master looks like this:

```
CREATE OR REPLACE PACKAGE BODY orgMaster
IS
-- Procedure to remove rows from org_site_tab table for a given org_id
-- This is necessary before deleting rows from org_tab.
-- This procedure is called from removeOrg procedure
PROCEDURE removeOrgSites(ip_org_id NUMBER,
                         op_retcd OUT NUMBER,
                         op_err_msg OUT VARCHAR2)
 IS
 BEGIN
    DELETE org_site_tab WHERE org_id = ip_org_id;
    op_retcd := 0;
 EXCEPTION WHEN OTHERS THEN
       op_retcd := SQLCODE;
       op_err_msg := SQLERRM;
 END removeOrgSites;
-- Procedure to create a new Org record in org_tab
  PROCEDURE createOrg (ip_hrc_code NUMBER,
              ip_org_id NUMBER,
              ip_org_short_name VARCHAR2,
              ip_org_long_name VARCHAR2,
              op_retcd OUT NUMBER,
              op_err_msg OUT VARCHAR2)
 IS
     BEGIN
     INSERT INTO org_tab VALUES
    (ip_hrc_code, ip_org_id, ip_org_short_name, ip_org_long_name);
    op_retcd := 0;
 EXCEPTION WHEN DUP_VAL_ON_INDEX THEN
    op_retcd := -1;
    op_err_msg := 'Organization with Id '||TO_CHAR(ip_org_id)||
                  ' already exists.';
   WHEN OTHERS THEN
       op_retcd := SQLCODE;
       op_err_msg := SQLERRM;
```

```
 END createOrg;
-- Procedure to update the short and long names of an Org in org_tab
-- based on input org_id
  PROCEDURE updateOrg(ip_org_id NUMBER,
                ip_org_short_name VARCHAR2,
                ip_org_long_name VARCHAR2,
                op_retcd OUT NUMBER,
                op_err_msg OUT VARCHAR2)
  IS
  BEGIN
     UPDATE org_tab
     SET org_short_name = ip_org_short_name,
          org_long_name = ip_org_long_name
     WHERE org_id = ip_org_id;
     IF (SQL%NOTFOUND) THEN
       op_retcd := -1;
       op_err_msg := 'Organization with Id '||TO_CHAR(ip_org_id)||
                        ' does not exist.';
       RETURN;
      END IF;
      op_retcd := 0;
  EXCEPTION WHEN OTHERS THEN
        op_retcd := SQLCODE;
        op_err_msg := SQLERRM;
  END updateOrg;
-- Procedure to delete a record in org_tab
  PROCEDURE removeOrg(ip_org_id NUMBER,
                op_retcd OUT NUMBER,
                op_err_msg OUT VARCHAR2)
  IS
  BEGIN
    removeOrgSites(ip_org_id, op_retcd, op_err_msg);
    IF (op_retcd <> 0) then
      RETURN;
    END IF;
     DELETE org_tab WHERE org_id = ip_org_id;
    IF (SQL%NOTFOUND) THEN
        op_retcd := -1;
        op_err_msg := 'Organization with Id '||TO_CHAR(ip_org_id)||
                         ' does not exist.';
        RETURN;
      END IF;
     op_retcd := 0;
  EXCEPTION WHEN OTHERS THEN
       op_retcd := SQLCODE;
       op_err_msg := SQLERRM;
  END removeOrg;
-- Function to return a row in org_tab for a given org_id.
-- It returns a resultset of type REF CURSOR defined in the package specification
  FUNCTION getOrgInfo(ip_org_id NUMBER) RETURN rc
  IS
    v_rc rc;
  BEGIN
        OPEN v_rc FOR SELECT * FROM org_tab WHERE org_id = ip_org_id;
```

```
    RETURN (v_rc);
  EXCEPTION WHEN OTHERS THEN
      RAISE_APPLICATION_ERROR(-20001, SQLERRM);
END getOrgInfo;
-- Function to return all rows in org_tab.
-- It returns a resultset of type REF CURSOR defined in the package specification
  FUNCTION getAllOrgs(ip_hrc_code NUMBER) RETURN rc
  IS
    v_rc rc;
  BEGIN
        OPEN v_rc FOR SELECT * FROM org_tab WHERE hrc_code = ip_hrc_code;
    RETURN (v_rc);
 EXCEPTION WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20002, SQLERRM);
END getAllOrgs;
-- Procedure to insert a row into org_site_tab based on
-- input org_id and site_no
  PROCEDURE assignSiteToOrg(ip_org_id NUMBER,
                    ip_site_no NUMBER,
                    op_retcd OUT NUMBER,
                    op_err_msg OUT VARCHAR2)
IS
  v_num NUMBER;
BEGIN
   BEGIN
      SELECT 1
      INTO     v_num
      FROM     org_site_tab
      WHERE    org_id = ip_org_id
           AND site_no = ip_site_no;
      IF (v_num = 1) THEN
         op_retcd := 0;
         RETURN;
      END IF;
    EXCEPTION WHEN NO_DATA_FOUND THEN
        INSERT INTO org_site_tab VALUES (ip_org_id, ip_site_no);
   END;
   op_retcd := 0;
  EXCEPTION WHEN OTHERS THEN
    op_retcd := SQLCODE;
    op_err_msg := SQLERRM;
  END assignSiteToOrg;
END orgMaster;
/
```

The following points are worth noting:

- The private procedure removeOrgSites is available only inside the package body of orgMaster.
- The private procedure removeOrgSites is called from the removeOrg procedure. In fact, the only place it can be called from is the package body of orgMaster.

Package Instantiation and Initialization

The very first time a packaged variable is referenced, or a packaged procedure or function is called, the package is loaded into memory (shared pool) from the disk and remains there for the duration of the session. This process is called *package instantiation.* For each user, a memory location is assigned in the shared pool for the package, a copy of the packaged variables is made in the session memory, and the packaged variables persist for the duration of the session. These variables constitute the package runtime state. Also, any initialization of variables or code to be executed one time is done at this time. You can specify this initialization process by means of an initialization section in the package body after the implementation details of the package. Here's the syntax:

[BEGIN
 sequence_of_statements ...]

Here's the orgMaster package body with an initialization section included:

```
CREATE OR REPLACE PACKAGE BODY orgMaster
IS
-- Procedure to remove rows from org_site_tab table for a given org_id
-- This is necessary before deleting rows from org_tab.
-- This procedure is called from removeOrg procedure
PROCEDURE removeOrgSites(ip_org_id NUMBER,
                  op_retcd OUT NUMBER,
                  op_err_msg OUT VARCHAR2)
 IS
 BEGIN
    DELETE org_site_tab WHERE org_id = ip_org_id;
    op_retcd := 0;
 EXCEPTION WHEN OTHERS THEN
       op_retcd := SQLCODE;
       op_err_msg := SQLERRM;
 END removeOrgSites;
- Procedure to create a new Org record in org_tab
  PROCEDURE createOrg (ip_hrc_code NUMBER,
                ip_org_id NUMBER,
                ip_org_short_name VARCHAR2,
                ip_org_long_name VARCHAR2,
                op_retcd OUT NUMBER,
                op_err_msg OUT VARCHAR2)
 IS
      BEGIN
     INSERT INTO org_tab VALUES
    (ip_hrc_code, ip_org_id, ip_org_short_name, ip_org_long_name);
    op_retcd := 0;
 EXCEPTION WHEN DUP_VAL_ON_INDEX THEN
     op_retcd := -1;
     op_err_msg := 'Organization with Id '||TO_CHAR(ip_org_id)||
                    ' already exists.';
   WHEN OTHERS THEN
       op_retcd := SQLCODE;
       op_err_msg := SQLERRM;
 END createOrg;
-- Procedure to update the short and long names of an Org in org_tab
```

```
-- based on input org_id
  PROCEDURE updateOrg(ip_org_id NUMBER,
                ip_org_short_name VARCHAR2,
                ip_org_long_name VARCHAR2,
                op_retcd OUT NUMBER,
                op_err_msg OUT VARCHAR2)
  IS
  BEGIN
     UPDATE org_tab
     SET org_short_name = ip_org_short_name,
          org_long_name = ip_org_long_name
     WHERE org_id = ip_org_id;
     IF (SQL%NOTFOUND) THEN
       op_retcd := -1;
       op_err_msg := 'Organization with Id '||TO_CHAR(ip_org_id)||
                        ' does not exist.';
       RETURN;
      END IF;
      op_retcd := 0;
  EXCEPTION WHEN OTHERS THEN
        op_retcd := SQLCODE;
        op_err_msg := SQLERRM;
  END updateOrg;
-- Procedure to delete a record in org_tab
  PROCEDURE removeOrg(ip_org_id NUMBER,
                op_retcd OUT NUMBER,
                op_err_msg OUT VARCHAR2)
  IS
  BEGIN
     removeOrgSites(ip_org_id, op_retcd, op_err_msg);
     IF (op_retcd <> 0) then
       RETURN;
    END IF;
     DELETE org_tab WHERE org_id = ip_org_id;
    IF (SQL%NOTFOUND) THEN
        op_retcd := -1;
        op_err_msg := 'Organization with Id '||TO_CHAR(ip_org_id)||
                        ' does not exist.';
        RETURN;
      END IF;
      op_retcd := 0;
   EXCEPTION WHEN OTHERS THEN
        op_retcd := SQLCODE;
        op_err_msg := SQLERRM;
  END removeOrg;
-- Function to return a row in org_tab for a given org_id.
-- It returns a resultset of type REF CURSOR defined in the package specification
  FUNCTION getOrgInfo(ip_org_id NUMBER) RETURN rc
  IS
    v_rc rc;
  BEGIN
        OPEN v_rc FOR SELECT * FROM org_tab WHERE org_id = ip_org_id;
    RETURN (v_rc);
 EXCEPTION WHEN OTHERS THEN
```

```
     RAISE_APPLICATION_ERROR(-20001, SQLERRM);
END getOrgInfo;
-- Function to return all rows in org_tab.
-- It returns a resultset of type REF CURSOR defined in the package specification
 FUNCTION getAllOrgs(ip_hrc_code NUMBER) RETURN rc
 IS
    v_rc rc;
 BEGIN
        OPEN v_rc FOR SELECT * FROM org_tab WHERE hrc_code = ip_hrc_code;
   RETURN (v_rc);
EXCEPTION WHEN OTHERS THEN
     RAISE_APPLICATION_ERROR(-20002, SQLERRM);
END getAllOrgs;
-- Procedure to insert a row into org_site_tab based on
-- input org_id and site_no
 PROCEDURE assignSiteToOrg(ip_org_id NUMBER,
                    ip_site_no NUMBER,
                    op_retcd OUT NUMBER,
                    op_err_msg OUT VARCHAR2)
IS
   v_num NUMBER;
BEGIN
   BEGIN
      SELECT 1
      INTO      v_num
      FROM     org_site_tab
      WHERE    org_id = ip_org_id
           AND site_no = ip_site_no;
      IF (v_num = 1) THEN
         op_retcd := 0;
         RETURN;
      END IF;
    EXCEPTION WHEN NO_DATA_FOUND THEN
        INSERT INTO org_site_tab VALUES (ip_org_id, ip_site_no);
    END;
    op_retcd := 0;
   EXCEPTION WHEN OTHERS THEN
      op_retcd := SQLCODE;
      op_err_msg := SQLERRM;
  END assignSiteToOrg;
-- This is the initialization section that is executed
-- the first time a package sub-program is invoked
-- or a packaged variable is referenced
BEGIN
    max_sites_for_an_org := 4;
 END orgMaster;
/
```

The following points are worth noting:

- The initialization section is specified as the last part of the package body.
- There's no separate END for this BEGIN.
- The initialization section is executed only once at the time of package instantiation.

- To execute a packaged procedure or function, or to reference a packaged public variable, the schema executing the package must own the package or have the EXECUTE privilege on the package granted to it.
- Like procedures and functions, packages (both package specification and package body) are recorded in the data dictionary views USER_OBJECTS and ALL_OBJECTS, and their source code is stored in USER_SOURCE and ALL_SOURCE. In this case, the object_type is either PACKAGE for package specification or PACKAGE BODY for the package body.
- Like stored procedures and functions, a PL/SQL package can also be natively compiled and executed.

6.2 Subprograms Returning Resultsets

Stored subprograms, such as stand-alone functions or packaged functions, return a single value that can be used in the calling environment. Even stored procedures can return a value to the calling environment by means of OUT parameters. You can specify multiple OUT parameters to return multiple values. However, it may sometimes be necessary to have subprograms return resultsets or rows of data instead of scalar values. PL/SQL is a flexible language and provides this capability. In PL/SQL, subprograms can return resultsets in two ways: using REF cursors and table functions. I describe the first method in this section.

A function can return a resultset by specifying a return type of REF CURSOR. A procedure can do this by specifying an OUT parameter of type REF CURSOR. There are two ways to define the REF cursor type:

1. Use the generic (weak) REF CURSOR type SYS_REFCURSOR available in Oracle9*i*. In this case, specify the return type of the function or the data type of the OUT parameter of the procedure as of type SYS_REFCURSOR. The advantage of specifying a weak REF cursor is that you don't need to bother about the number of columns selected.

2. Define a REF CURSOR type in a package specification and use this as the return type or the data type of the OUT parameter.

Here's an example function that returns a resultset using the SYS_REFCURSOR type:

```
CREATE OR REPLACE FUNCTION getAllHierarchies
RETURN SYS_REFCURSOR
IS
  v_rc SYS_REFCURSOR;
BEGIN
   OPEN v_rc FOR SELECT * FROM hrc_tab;
   RETURN (v_rc);
EXCEPTION WHEN OTHERS THEN
   RAISE_APPLICATION_ERROR(-20002, SQLERRM);
END;
/
```

You can call this function from a PL/SQL block as follows:

```
DECLARE
   v_rc SYS_REFCURSOR;
   hrc_rec hrc_tab%ROWTYPE;
BEGIN
   v_rc := getAllHierarchies;
   LOOP
      FETCH v_rc INTO hrc_rec;
      EXIT WHEN v_rc%NOTFOUND;
      dbms_output.put_line(TO_CHAR(hrc_rec.hrc_code)||' '||hrc_rec.hrc_descr);
```

```
   END LOOP;
EXCEPTION WHEN OTHERS THEN
  dbms_output.put_line(TO_CHAR(SQLCODE)||' '||SQLERRM);
END;
/
```

The following points are worth noting:

- The function getAllHierarchies has a return type of SYS_REFCURSOR. Also, a local cursor variable is defined inside the function. This cursor variable is OPENed for a query. The set of rows returned by this query is identified by the cursor variable. In fact, a pointer to this resultset is stored.

- Outside the function, in the calling environment, the function is called and assigned to a second cursor variable of compatible type. This second cursor variable isn't opened for query. In fact, it shouldn't be. The FETCH operation takes place outside of the function. This is the power of a function returning resultsets. The second cursor variable now points to the resultset returned by the function.

The method of using a REF CURSOR type in a package and using this as a function return type was illustrated in the orgMaster package previously defined. The functions getOrgInfo and getAllOrgs both return resultsets based on a REF CURSOR type defined in this package.

6.3 Using Stored Functions in SQL Statements

As mentioned earlier, stored functions are invoked as part of a PL/SQL expression, or they can be called from procedural statements such as the IF statement. Now the question is, can they be called from SQL statements? The answer is yes. Just like built-in SQL functions, stored functions (either stand-alone or packaged) can be called from SQL statements provided they meet certain criteria. PL/SQL 2.1 and above offer this flexibility. Specifically, a stored function can be called from the following SQL statements:

- SELECT statement (SELECT column list, and WHERE, HAVING, GROUP BY, CONNECT BY, START WITH, and ORDER BY clauses)

- INSERT statement (VALUES clause)

- UPDATE statement (SET clause)

[[schema_name.]package_name.]function_name[(arg1, ... , argN)]

where schema_name is the schema in which the stored function is created, package_name is the PL/SQL package in which the function is defined (in the case of packaged functions), and arg1, …, argN are the arguments corresponding to the formal parameters of the function. The schema_name is optional as is the package_name in the case of a stand-alone function.

Here's an example of calling a stored function from a SELECT statement:

```
CREATE OR REPLACE FUNCTION f_get_formatted_org_name
                    (ip_hrc_code NUMBER,
                     ip_org_id NUMBER)
RETURN VARCHAR2
IS
  v_name VARCHAR2(120);
BEGIN
   SELECT 'Org Name: (Short) '||org_short_name||' (Long) '||org_long_name
   INTO    v_name
```

```
   FROM  org_tab
   WHERE hrc_code = ip_hrc_code
        AND org_id = ip_org_id;
   RETURN (v_name);
END f_get_formatted_org_name;
/
SELECT f_get_formatted_org_name(hrc_code, org_id) "Formatted Org Name"
FROM  org_tab
ORDER BY hrc_code, org_id;
```

Here's the output of this SELECT:

6.4 Criteria for Calling Stored Functions from SQL

As mentioned earlier in this section, a PL/SQL function must meet the following criteria to be able to be called from a SQL statement:

- The function must be a stored function, either stand-alone or part of a package.
- When you specify actual parameters to functions called from SQL, only positional notation is allowed. Also, you should specify all parameters, even if default values exist for the formal parameters. This rule doesn't apply to functions called from procedural statements.
- The function must be a row function and not a column group function.
- All the function's formal parameters must be of IN mode.
- The data type of all the function's formal parameters are valid SQL types and not PL/SQL types such as BOOLEAN RECORD, TABLE, REF CURSOR, and so forth.
- The return type of the function is a valid SQL type.

6.5 Purity of a Function Called from SQL

A function called from SQL can have certain side effects on database tables and packaged variables that it reads or writes. These side effects are specified using *purity levels.* You can specify four purity levels for a function: WNDS, RNDS, WNPS, and RNPS. These are explained in the following code:

- *WNDS—Writes no database state:* The function does not modify database tables.
- *RNDS—Reads no database state:* The function does not query database tables.
- *WNPS—Writes no package state:* The function does not change the values of packaged variables.
- *RNPS—Reads no package state:* The function does not reference the values of packaged variables.
- *TRUST:* Allows easy calling from functions that do have RESTRICT_REFERENCES declarations to those that do not.

In general, any function callable from SQL must be *pure*—that is, it must meet the following requirements in terms of purity levels:

- It must obey WNDS (i.e., it can't modify database tables). This assertion is relaxed in Oracle8*i*.

- If the function is executed in parallel or remotely, it must obey WNPS and RNPS.

- If the function is called from WHERE, HAVING, GROUP BY, CONNECT BY, START WITH, and ORDER BY clauses, it must obey WNPS.

- If the function calls any subprograms inside it, those subprograms must obey the preceding purity rules. In other words, a function is only as pure as the procedures or functions it calls.

For stored stand-alone functions, the purity levels are checked at runtime.

Starting from Oracle8*i*, the purity level of a stored function, either stand- alone or packaged, is checked at runtime when the function is called from a SQL statement. Any violations thereof result in an error. However, PL/SQL provides a way of checking the purity level of packaged functions at compile time using a PRAGMA. The purity level of a packaged function can be asserted by using a PL/SQL PRAGMA called PRAGMA RESTRICT_REFERENCES. Here's the syntax:

```
PRAGMA RESTRICT_REFERENCES ( DEFAULT|function_name, WNDS [, WNPS] [, RNDS]
[, RNPS] [, TRUST] );
```

where function_name is the name of the packaged function, and WNDS, WNPS, RNDS, and RNPS refer to the four purity levels. The DEFAULT and TRUST keywords are explained later in this section.

Prior to Oracle8*i*, packaged functions must be defined with the PRAGMA specified to be callable from SQL. Here's an example of using this PRAGMA:

```
CREATE OR REPLACE PACKAGE rfPkg
IS
 FUNCTION f_get_formatted_org_name
               (ip_hrc_code NUMBER,
                ip_org_id NUMBER)
 RETURN VARCHAR2;
 PRAGMA RESTRICT_REFERENCES(f_get_formatted_org_name, WNDS, WNPS);
END rfPkg;
/
```

Here's the corresponding package body:

```
CREATE OR REPLACE PACKAGE BODY rfPkg
IS
 FUNCTION f_get_formatted_org_name
                        (ip_hrc_code NUMBER,
                         ip_org_id NUMBER)
 RETURN VARCHAR2
 IS
    v_name VARCHAR2(120);
 BEGIN
    SELECT 'Org Name: (Short) '||org_short_name||' (Long) '||org_long_name
    INTO    v_name
    FROM  org_tab
    WHERE hrc_code = ip_hrc_code
          AND org_id = ip_org_id;
    RETURN (v_name);
 END f_get_formatted_org_name;
END rfPkg;
/
```

Here's the SELECT statement calling the packaged function:

```
SELECT rfPkg.f_get_formatted_org_name(hrc_code, org_id) "Formatted Org Name"
FROM org_tab
ORDER BY hrc_code, org_id;
```

Here's the output of this SELECT:

Any function violating the PRAGMA will result in a compilation error at the time of parsing. Here's an example to illustrate this:

```
CREATE OR REPLACE PACKAGE rfPkg2
IS
 FUNCTION f_get_formatted_org_name
                (ip_hrc_code NUMBER,
                 ip_org_id NUMBER)
 RETURN VARCHAR2;
 PRAGMA RESTRICT_REFERENCES(f_get_formatted_org_name, WNDS, WNPS);
END rfPkg2;
/
CREATE OR REPLACE PACKAGE BODY rfPkg2
IS
 FUNCTION f_get_formatted_org_name
                 (ip_hrc_code NUMBER,
                  ip_org_id NUMBER)
 RETURN VARCHAR2
 IS
    v_name VARCHAR2(120);
    v_hrc_descr VARCHAR2(20);
    v_org_short_name VARCHAR2(30);
    v_org_long_name VARCHAR2(60);
 BEGIN
    SELECT 'Org Name: (Short) '||org_short_name||' (Long) '||org_long_name
    INTO     v_name
    FROM  org_tab
    WHERE hrc_code = ip_hrc_code
        AND org_id = ip_org_id;
    SELECT hrc_descr
    INTO  v_hrc_descr
    FROM  hrc_tab
    WHERE hrc_code = ip_hrc_code;
    SELECT org_short_name, org_long_name
    INTO     v_org_short_name, v_org_long_name
    FROM  org_tab
    WHERE hrc_code = ip_hrc_code
         AND org_id = ip_org_id;
    INSERT INTO sec_hrc_org_tab VALUES
    (ip_hrc_code, v_hrc_descr, ip_org_id,
     v_org_short_name, v_org_long_name);
    RETURN (v_name);
 END f_get_formatted_org_name;
END rfPkg2;
/
```

Compiling the preceding package results in the following error:

PLS-00452: Subprogram 'F_GET_FORMATTED_ORG_NAME' violates its associated pragma

This error happens because the function body writes to the database (the INSERT INTO sec_hrc_org_tab … statement) as opposed to its associated PRAGMA declaration of WNDS.

Tip You should specify the PRAGMA RESTRICT_REFERENCES only in the package specification following the function signature. You shouldn't specify it in the package body.

Tip You can also specify the PRAGMA RESTRICT_REFERENCES in the initialization section of the package using the package name to replace the function name in the PRAGMA syntax. In this case, a package function is only as pure as the initialization section of the package.

1. Parameter Passing by Reference

Parameters to stored subprograms are of two types: actual parameters and formal parameters. You can define formal parameters with three modes: IN, OUT, and IN OUT. Actual parameters come into the picture when you call the stored subprogram, and they have a one-to-one correspondence with the formal parameters. Generally, there are two distinct ways of passing actual parameters: call by value and call by reference.

In the case of *call by value,* the value of the actual parameter is copied into the formal parameter. That is, a copy of the actual parameter is made. In the case of *call by reference,* a pointer to the actual parameter is passed to the corresponding formal parameter. Thus, no copy is made and both actual and formal parameters refer to the same memory location. Call by reference is faster because it avoids the copy. This is illustrated in a later section, "Performance Improvement of NOCOPY."

PL/SQL uses call by reference for IN mode and call by value for OUT and IN OUT modes. This preserves the exception semantics while passing OUT and IN OUT parameters. By default, if a subprogram exits normally, the values assigned to the OUT and IN OUT formal parameters inside the body of the subprogram are copied into the corresponding actual parameters. However, if the subprogram terminates with an unhandled exception, this copying isn't done. Also, call by value can impact performance with reduced execution time and excess memory consumption when large data structures are passed as OUT or IN OUT parameters. This is especially true when passing resultsets, collections, an index-by table of records, or objects.

PL/SQL 8*i* onward define a compiler hint called NOCOPY that you can specify for OUT and IN OUT parameters. Specifying NOCOPY overrides the default call by value behavior for OUT and IN OUT modes and enables parameters in these modes to be passed by reference. This eliminates the performance bottleneck mentioned earlier. However, NOCOPY is a compiler hint and not a directive, so it may not apply always—that is, even if it's specified, the parameter may still be passed by value.

Here's the syntax of using NOCOPY:

parameter_name [OUT| IN OUT] NOCOPY datatype

where parameter_name is the name of the parameter and datatype represents the data type of said parameter. You can use either the OUT or IN OUT parameter mode with NOCOPY. Here's an example:

```
CREATE OR REPLACE PROCEDURE p_nocopy
            (ip_1 IN NUMBER,
             op_2 OUT NOCOPY VARCHAR2)
```

```
IS
BEGIN
   NULL;
END;
/
```

Table 5-1 describes the behavior of specifying IN, INOUT, OUT, and NOCOPY.

Table 5-1: Behavior of Various Parameter Modes

IN	**IN OUT**	**OUT**	**IN OUT NOCOPY, OUT NOCOPY**
Actual parameter is passed by reference.	The actual parameter is passed by value.	The actual parameter is passed by value.	The actual parameter is passed by reference.
A pointer (address) to the value is passed.	A copy of the value is passed out.	A copy of the value is passed in and out.	The address of the value is passed.
N/A	The OUT value is rolled back in case of an unhandled. exception	The OUT value is rolled back in case of an unhandled exception.	You can't predict the correctness of the OUT value always, as no rollback occurs in the case of an unhandled exception.

Tip You can use NOCOPY only for parameters in OUT or IN OUT mode. NOCOPY is a compiler hint and not a directive, and hence it may be ignored sometimes. With NOCOPY, changes to formal parameter values affect the values of the actual parameters also, so if a subprogram exits with an unhandled exception, the unfinished changes are not rolled back. That is, the actual parameters may still return modified values.

7. OVERVIEW **DATABASE TRIGGERS**

In Chapter 5, I described three types of stored subprograms: procedures, functions, and packages. In addition to these subprogram types, PL/SQL enables a fourth type of stored subprogram called a *database trigger.* A database trigger is a stored subprogram that's stored in the database and executed implicitly on the occurrence of an event. The event can be a DDL operation, such as an object creation, alteration, or dropping; a DML operation, such an INSERT, an UPDATE, or a DELETE on a table or view; a system event, such as database startup and shutdown, logon, and logoff; or a user event, such as schema logon and logoff.

Database triggers enable you to perform various functions that would otherwise be tedious for you to code. The most common uses of database triggers are as follows:

- For DDL and DML auditing
- For enforcing complex validation rules to prevent inappropriate and inconsistent data from being input into the database
- For performing related actions when a particular action occurs
- For enforcing complex data-integrity relationships that you can't otherwise specify declaratively, such as a cascading update operation on a child table whenever a parent row is updated
- For automatically generating derived values
- For system event handling

After defining triggers, I go on to highlight the various types of triggers. I describe the implications of creating triggers and the concepts involved therein. I then explain the concept of creating triggers on views. Finally, I present descriptions of the new triggers introduced in Oracle8*i*.

I illustrate the concept of triggers by taking into account the organizational hierarchy system presented in Chapter 2. In addition, I use the table DDL_AUDIT to test user-event triggers. Appendix A lists the schema objects to be created.

2. PL/SQL Triggers: Types and Definition

A database trigger is a stored program executed in response to a database event. This event is called the *triggering event,* and it can be one of the following:

- A DDL operation such as CREATE, ALTER, or DROP
- A DML operation such as INSERT, UPDATE, or DELETE
- A system event such as database STARTUP, SHUTDOWN, or SERVERERROR
- A user event such as LOGON or LOGOFF

A triggering event is initiated on the execution of a triggering statement such as an INSERT, UPDATE, or DELETE statement, or a CREATE, ALTER, or DROP statement. A triggering event is also initiated on database startup and shutdown or user logon and logoff.

7.1 Types of Triggers

Triggers are classified according to the event that causes them to fire and the time of that event. You can set them to fire either before or after the particular event. Triggers fall into three main categories: DML triggers, INSTEAD-OF triggers, and triggers on system and user events.

DML Triggers

DML triggers fire when an INSERT, UPDATE, or DELETE statement is executed on a database table. They can be further classified as ROW or STATEMENT triggers. ROW and STATEMENT triggers specify how many times the trigger body should be executed.

INSTEAD-OF Triggers

INSTEAD-OF triggers fire on DML statements issued against either a relational view or an object view. I discuss these triggers further in the section "Read-Only Views, Updateable Views, and INSTEAD-OF Triggers."

Triggers on System and User Events

Triggers on system events and user events fire on the occurrence of system events such as database startup and shutdown, whenever a server error occurs, or when a user event such as user logon and logoff occurs or a DDL command is executed. I discuss these types of triggers further in the section "New Database Triggers."

A second classification of triggers is based on the time the event occurred. These are BEFORE and AFTER triggers. The trigger should fire before or after the occurrence of the triggering event. These triggers can pertain to DML triggers or triggers on system and user events.

7.2 Defining Triggers

A database trigger is defined using the CREATE OR REPLACE TRIGGER statement. Here's the general syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
[BEFORE|AFTER|INSTEAD OF] triggering_event
[referencing_clause]
[WHEN trigger_restriction]
[FOR EACH ROW]
trigger_body ;
```

where trigger_name is the name of the trigger being created, triggering_event specifies a particular event that fires the trigger, and trigger_body is a PL/SQL block specifying the action the trigger performs when fired. Instead of a PL/SQL block, the trigger body can be a CALL statement invoking a PL/SQL procedure or function or a Java stored procedure published in

PL/SQL. REFERENCING clause, WHEN trigger_restriction, and FOR EACH ROW are explained in the sections "REFERENCING Clause," "WHEN Clause," and "ROW and STATEMENT Triggers," respectively.

Here's an example:

```
CREATE OR REPLACE TRIGGER ai_org_trig
AFTER INSERT ON org_tab
FOR EACH ROW
BEGIN
      UPDATE sec_hrc_audit
       SET num_rows = num_rows+1
       WHERE hrc_code = :NEW.hrc_code;
       IF (SQL%NOTFOUND) THEN
         INSERT INTO sec_hrc_audit VALUES (:NEW.hrc_code, 1);
       END IF;
END;
/
```

As you can see from the CREATE OR REPLACE TRIGGER statement, a trigger has three distinct parts: the triggering event, the triggering restriction, and the trigger body. Depending on the type of trigger, the *triggering event* can be a DML event, a system event, or a user event that causes the trigger to fire. In the example trigger, the triggering event is the statement

```
AFTER INSERT ON org_tab
```

In this case, the event is the INSERT operation on the org_tab table, and this trigger fires whenever an INSERT operation is done on the org_tab table and after the new row is written to the org_tab table. The INSERT statement that causes the INSERT trigger event to fire is termed the *triggering statement.* The *triggering restriction* is an optional condition specified as a WHEN clause that causes the trigger body to be executed only when the condition is TRUE. The *trigger body* specifies the code to be executed (i.e., the sequence of actions to be performed) when the trigger fires. In the example trigger, the trigger body is the following:

```
BEGIN
      UPDATE sec_hrc_audit
       SET num_rows = num_rows+1
       WHERE hrc_code = :NEW.hrc_code;
      IF (SQL%NOTFOUND) THEN
        INSERT INTO sec_hrc_audit VALUES (:NEW.hrc_code, 1);
     END IF;
END;
```

In this case, the trigger updates the num_rows column by incrementing it by 1 in the sec_hrc_audit table for the newly inserted hrc_code if it exists; otherwise, it inserts a row into the sec_hrc_audit table with a count of 1.

A trigger doesn't accept parameters, includes an executable section with BEGIN and END, and can include an optional declaration section and an exception handling section. The BEGIN … END section can include SQL and/or PL/SQL statements, and calls to PL/SQL procedures and functions (either stand-alone or packaged). Also, the trigger body can be a single CALL statement invoking a PL/SQL procedure or function, or a Java stored procedure published in PL/SQL.

Tip Triggers exist in a separate namespace from database tables, so it's possible to create a database trigger with the same name as a database table. However, this isn't recommended.

Next, I discuss DML triggers. I discuss INSTEAD-OF triggers in after that, and I explain system and user event triggers in the section "New Database Triggers."

Defining DML Triggers

You define DML triggers on a particular database table and they fire during the execution of an INSERT, an UPDATE, or a DELETE.

You define DML triggers with the CREATE OR REPLACE TRIGGER statement. Here's the syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
[BEFORE|AFTER] triggering_event [OF column_name] ON table_name
[referencing_clause]
[WHEN trigger_restriction]
[FOR EACH ROW]
trigger_body ;
```

where trigger_name is the name of the DML trigger being created; triggering_event is one or a combination of INSERT, UPDATE, and/or DELETE; and table_name is the table on which the INSERT, UPDATE, and/or DELETE is being executed. The [OF column_name] clause specifies the column_name being updated in the case of an UPDATE operation as the triggering event. BEFORE or AFTER specifies the time the trigger should fire (i.e., before or after the DML statement is executed). trigger_body is the sequence of actions to be performed when the trigger is fired, and the [WHEN condition] clause specifies any condition that when evaluated to TRUE causes the trigger body to execute. I explain the REFERENCING clause later.

Tip You can specify multiple DML operations for the triggering event by separating them with the OR keyword.

The example trigger specified is an example of a DML trigger. Here's the same trigger repeated for your reference:

```
CREATE OR REPLACE TRIGGER ai_org_trig
AFTER INSERT ON org_tab
FOR EACH ROW
BEGIN
      UPDATE sec_hrc_audit
       SET num_rows = num_rows+1
       WHERE hrc_code = :NEW.hrc_code;
     IF (SQL%NOTFOUND) THEN
       INSERT INTO sec_hrc_audit VALUES (:NEW.hrc_code, 1);
     END IF;
END;
/
```

The following points are worth noting:

- The triggering event is an INSERT operation and the trigger fires after inserting into the org_tab table.

- The trigger body contains the code to be executed when this trigger fires. Here, a record in the sec_hrc_audit table is updated for the inserted hrc_code if it exists; otherwise, a new row is inserted into the sec_hrc_audit table.

Multiple Triggering Events in a Single DML Trigger

As mentioned earlier, you can specify multiple DML operations for the triggering event by separating them with the OR keyword. To distinguish between the three operations, PL/SQL provides trigger predicates. PL/SQL defines three predicates called INSERTING, UPDATING, and DELETING to account for INSERT, UPDATE, and DELETE operations, respectively. You use these predicates like BOOLEAN conditions in an IF statement in the trigger body. Here's an example:

```
IF INSERTING THEN
 ...
ELSIF UPDATING THEN
 ...
ELSIF DELETING THEN
 ...
END IF;
```

Consider the ai_org_trig trigger mentioned previously. This trigger updates the sec_hrc_audit table whenever a new row is inserted into the org_tab table. But what about when a row is deleted from the orc_tab table? The num_rows column in the sec_hrc_audit table should be decremented in this case. It suffices to write a new trigger with DELETE as the triggering event. However, with the use of trigger predicates, a single trigger can achieve the same functionality for both the INSERT and DELETE operations. Here's the example of ai_org_trig modified in this way:

```
CREATE OR REPLACE TRIGGER ai_org_trig
AFTER INSERT OR DELETE ON org_tab
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        UPDATE sec_hrc_audit
         SET num_rows = num_rows+1
         WHERE hrc_code = :NEW.hrc_code;
      IF (SQL%NOTFOUND) THEN
         INSERT INTO sec_hrc_audit VALUES (:NEW.hrc_code, 1);
       END IF;
  ELSIF DELETING THEN
      UPDATE sec_hrc_audit
       SET num_rows = num_rows-1
       WHERE hrc_code = :OLD.hrc_code;
  END IF;
END;
/
```

The following points are worth noting:

- A single trigger is defined for two triggering events, INSERT and DELETE.
- Conditional predicates are used to distinguish between INSERT and DELETE operations and separate code is executed for each.

Number and Type of DML Triggers You Can Define on a Single Table

As mentioned earlier, DML triggers can be BEFORE or AFTER and can be at ROW or STATEMENT level. Taking into account these two factors, you can define a total of four DML triggers on a table: Before Statement, Before Row, After Row, and After Statement.

Also, DML triggers can fire when an INSERT, UPDATE, or DELETE statement is issued on the table on which the trigger is defined. Taking this into consideration, a total of 12 possible trigger combinations is available. Table 6-2 summarizes the possible trigger combinations.

Table 6-2: Trigger Combinations

TRIGGER TYPE	DESCRIPTION
BEFORE INSERT (STATEMENT level)	Before writing new rows to the database as a result of an INSERT
BEFORE INSERT (ROW level)	Before writing each row affected by an INSERT, firing once per row
AFTER INSERT (ROW level)	After writing each row affected by an INSERT, firing once per row
AFTER INSERT (STATEMENT level)	After inserting all rows as a result of an INSERT
BEFORE UPDATE (STATEMENT level)	Before modifying rows as a result of an UPDATE
BEFORE UPDATE (ROW level)	Before modifying each row affected by an UPDATE, firing once per row
AFTER UPDATE (ROW level)	After writing each change affected by an UPDATE, firing once per row
AFTER UPDATE (STATEMENT level)	After modifying all rows as a result of an UPDATE
BEFORE DELETE (STATEMENT level)	Before deleting rows as a result of an INSERT
BEFORE DELETE (ROW level)	Before deleting each row affected by a DELETE, firing once per row
AFTER DELETE (ROW level)	After deleting each row affected by a DELETE, firing once per row
AFTER DELETE (STATEMENT level)	After deleting all rows as a result of a DELETE

Although there are 12 possible combinations, you can define multiple triggers of the same type on the same table. (This is possible as of PL/SQL 2.1 onward.) Another important consideration is the order in which triggers fire. The firing order is as follows:

1. Before Statement (fires once per DML statement, no matter how many rows it affects).
2. Before Row (fires once per each row affected by the DML statement).
3. Execute the triggering statement itself.
4. After Row (fires once per each row affected by the DML statement).
5. After Statement (fires once per DML statement, no matter how many rows it affects).

BIBLIOGRAFIA

- ORACLE www.oracle.com
- Oracle.PL.SQL.for.DBAs.Oct.2005
- Oracle.PL.SQL.Programming.4th.Edition.Aug.2005
- 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.