Curso de SQL avanzado y PL/SQL básico para Oracle 10g (10.2)

Manual del alumno

Ing. Cedric Simon – Tel: 2268 0974 – Cel: 8888 2387 – Email: cedric@solucionjava.com – Web: www.solucionjava.com

SolucionJava.com

 Versión 1.1

Índice

Índice....2
1 Introducción al curso....4
1.1 Objetivo de este curso....4
1.2 Manual del alumno....4
1.3 Requisitos para atender a este curso....4
1.4 Soporte después del curso....4
1.5 Herramienta de desarrollo....4
2 DML - Sentencias de manipulación de datos....5
2.1 Objetivo del capítulo....5
2.2 Insert....5
2.3 Update....5
2.4 Delete....5
2.5 Commit y rollback....5
2.6 Savepoint....6
2.7 Select....6
2.8 Where....7
2.8.1 Operadores SQL....7
2.8.2 La ausencia de valor: NULL....8
2.9 Count....8
2.10 Sum, avg, min, max....9
2.11 Distinct....9
2.12 Order by....9
2.13 Uniones....9
2.14 Subconsultas....9
2.15 Agrupaciones....10
2.16 Rollup....10
2.17 Cube....10
2.18 Rendimiento....11
2.18.1 Indices....11
2.18.2 Diseño de la base de datos y denormalización....11
2.18.3 Uso del EXPLAIN PLAN....12
2.18.4 Uso de HINTS....12
3 Creación de objetos....14
3.1 Indices....14
3.1.1 Creación....14
3.1.2 Modificación....15
3.1.3 Eliminación....16
3.1.4 Ejercicios....16
3.2 Vistas....17
3.2.1 Creación....17
3.2.2 Modificación....17

3.2.3 Eliminación....17
3.2.4 Ejercicios....17
3.3 Vistas materializadas....17
3.4 Secuencias....18
3.4.1 Creación....18
3.4.2 Modificación....18
3.4.3 Eliminación....18
3.4.4 Ejercicios....18
3.5 Sinónimos....18
3.5.1 Creación....18
3.5.2 Eliminación....19
3.5.3 Ejercicios....19
3.6 DB Link....19
4 PL/SQL básico....20
4.1 Introducción....20
4.2 Bloque anónimo PL/SQL....20
4.3 Identificadores....20
4.4 Variables....20
4.4.1 Tipos de variables....21
4.4.2 Variables locales....21
4.4.3 Variables globales....21
4.5 Control de flujo....21
4.5.1 If..elsif...else....21
4.5.2 Case....22
4.5.3 Goto....23
4.6 Bucles....23
4.6.1 LOOP....23
4.6.2 WHILE....24
4.6.3 FOR....24
4.7 Exit....24
4.8 NULL....24
4.9 EXECUTE IMMEDIATE....24
4.10 Cursores....25
4.11 Excepciones....26
4.11.1 Excepciones predefinidas....27
4.11.2 Excepciones definidas por el usuario....29
4.11.3 RAISE_APPLICATION_ERROR....30
4.12 Procedimientos, funciones, paquetes, disparadores....31
4.12.1 Funciones....31
4.12.2 Procedimientos....36
4.12.3 Paquetes....37
4.12.4 Disparadores....38
5 Ejercicios....43
6 Diseño de la base de datos....44

1 Introducción al curso

1.1 Objetivo de este curso

Este curso brindará al alumno el conocimiento necesario para hacer programación en PL/SQL en una base de datos Oracle 10g.

Primero vamos a dar una repasadita al lenguaje SQL y ver alguna sentencias avanzadas del SQL. Luego veremos el PL/SQL.

1.2 Manual del alumno

Este manual del alumno es una ayuda para el alumno, para tenga un recuerdo del curso. Este manual contiene un resumen de las materias que se van a estudiar durante el curso, pero el alumno debería de tomar notas personales para completas este manual.

1.3 Requisitos para atender a este curso

Se requiere un conocimiento del lenguaje SQL.

1.4 Soporte después del curso

Si tienes preguntas sobre la materia del curso en tus ejercicios prácticos, puedes escribir tus preguntas a cedric@solucionjava.com .

1.5 Herramienta de desarrollo

Existen muchas herramientas de desarrollo para el PL/SQL. Lo mas básico es un editor de texto y SQL plus.

Las herramientas mas populares son TOAD, SQL Navigator, PL/SQL Developper, SQL Developper.

SQL Developper, desarrollado por Oracle es gratis, los demás necesitan comprar una licencia.

Cualquier herramienta se puede usar en el curso. Lo ideal es usar la con la cual tiene mas experiencia (y licencia), o usar SQL Developper que es gratis.

2 DML - Sentencias de manipulación de datos

2.1 Objetivo del capítulo

Al fin de este capítulo el alumno será capaz de hacer encuestas de la base de datos. No vamos a ver todas las opciones, ni las encuestas de otros objetos de la base de datos (vistas, funciones, secuencias,...) porque eso sale del cuadro de este curso de iniciación.

Existen muchas opciones (top, exists, cube,...) para cada tipo de acción, pero estas opciones dependen de la base de datos utilizadas y/o de su versión. Solo vamos a ver las sentencias básicas.

Existen muchos entornos que simplifican las encuestas sobre los datos.

2.2 Insert

La sentencia 'Insert' permite de insertar datos en una tabla.

```
INSERT INTO <nombre_de_tabla> (<campo_1>,<campo_2>,<...>) VALUES
(<valor_campo_1>,<valor_campo_2>,<valor_...>);
```

También existe:
```
INSERT INTO <nombre_de_tabla> (<campo_1>,<campo_2>,<...>) <SELECT STATEMENT>;
```

2.3 Update

La sentencia 'Update' permite de modificar el valor de uno o varios datos en una tabla.

```
UPDATE <nombre_de_tabla> SET <campo_1>=<valor_campo_1>,<campo_2>=<valor_campo_2>,<...>;
```

De costumbre se limita el cambio a ciertos registros, mencionados utilizando una cláusula WHERE.

```
UPDATE <nombre_de_tabla> SET <campo_1>=<valor_campo_1>,<campo_2>=<valor_campo_2>,<...>
WHERE <cláusula_where>;
```

2.4 Delete

La sentencia 'Delete' permite de borrar un uno o varios registros en una tabla.

```
DELETE FROM <nombre_de_tabla> ;
```

De costumbre se limita el borrado a ciertos registros, mencionados utilizando una cláusula WHERE.

```
DELETE FROM <nombre_de_tabla>  WHERE <cláusula_where>;
```

2.5 Commit y rollback

Si la base de datos permite la gestión de transacciones, se puede utilizar 'Commit' para confirmar una 'Insert', 'Update', o 'Delete', o `Rollback` para cancelarlos. Ciertas base de datos pueden ser configuradas

para autocommit, que hace un commit automaticamente despues de cada instrucción, a menos que se ha iniciado una transacción de manera explicita (con 'begin transaction xxx;).

Hasta que el 'Commit' está ejecutado, las modificaciones no están inscritas de manera permanente en la base de datos, y sólo son visible para la sesión en curso del usuario autor de las acciones. Después del 'Commit', los cambios son definitivos y visible para todos.

Cuidado que ciertos objetos pueden quedar bloqueados (bloqueando otros usuarios) hasta que el commit sea hecho.

El commit/rollback permite confirmar o de hacer un lote de transacción, para que si una falle, todas las anteriores se anulan también. Cuando se necesita una integridad de transacción, se utiliza en commit/rollback.

```
Ejemplo:
SELECT emp_no,job_grade FROM employee where emp_no=4;
update employee set job_grade=6 where emp_no=4;
SELECT emp_no,job_grade FROM employee where emp_no=4;
rollback;
SELECT emp_no,job_grade FROM employee where emp_no=4;
update employee set job_grade=6 where emp_no=4;
SELECT emp_no,job_grade FROM employee where emp_no=4;
commit;
SELECT emp_no,job_grade FROM employee where emp_no=4;
```

2.6 Savepoint

Un savepoint permite identificar un punto en una transacción al cual se podrá eventualmente regresar (rollback).

```
SELECT emp_no,job_grade FROM employee where emp_no=4;
START TRANSACTION;
update employee set job_grade=5 where emp_no=45;
SELECT emp_no,job_grade FROM employee where emp_no=4;
savepoint vale_cinco;
SELECT emp_no,job_grade FROM employee where emp_no=4;
update employee set job_grade=4 where emp_no=45;
SELECT emp_no,job_grade FROM employee where emp_no=4;
rollback to savepoint vale_cinco;
SELECT emp_no,job_grade FROM employee where emp_no=4;
rollback;
```

2.7 Select

El 'Select' permite de seleccionar datos en la base de datos, y visualizarlos.

Se puede utilizar un alias para que el campo se pueda llamar con otro nombre.

```
SELECT <campo_1>,<campo_2>,<...> FROM <nombre_tabla>;
SELECT <campo_1> as <alias1>,<campo_2>,<...> FROM <nombre_tabla>;
```

Para seleccionar todos los campos de la tabla, se utiliza el asterisco en vez de los nombres de campo.

```
SELECT * FROM <nombre_tabla>;

Ejemplo:
SELECT emp_no,job_grade as nivel FROM employee;
SELECT * FROM employee;
```


2.8 Where

La cláusula 'Where' permite de limitar la encuesta a ciertos datos.

Se utiliza evaluando un campo versus una condición. Se pueden utilizar varias condiciones, con el uso de 'Or', 'And', y/o paréntesis.

Para compara números, se utiliza el signo '=', o '<', o'>', o'<=', o '>=', o 'between ... and ...'.
Para comparar caracteres se utiliza la palabra 'like'. El wildcard es '%'.
Para compara fecha, se utiliza el signo '=', o '<', o'>', o'<=', o '>=', o 'between ... and ...'.
Para

```
SELECT * FROM <nombre_tabla>
WHERE <campo_1> <operation> <condición> AND <campo_2> <operation> <condición>;

Ejemplo:
SELECT emp_no,job_grade FROM employee where emp_no>45;
SELECT emp_no,job_grade FROM employee where emp_no=46 or  emp_no=61;
SELECT * FROM employee where emp_no between 1 and 2;
SELECT * FROM employee where last_name like 'P%';
```

2.8.1 Operadores SQL

Ya hemos visto anteriormente qué tipos de datos se pueden utilizar en Oracle. Y siempre que haya datos, habrá operaciones entre ellos, así que ahora se describirán qué operaciones y con qué operadores se realizan:

Los operadores se pueden dividir en TRES conjuntos:

- Aritméticos: utilizan valores numéricos
- Lógicos (o booleanos o de comparación): utilizan valores booleanos o lógicos.
- Concatenación: para unir cadenas de caracteres.

Operadores arítméticos Retornan un valor numérico:

Símbo lo	Significado	Ejemplo
+	Operación suma	1 + 2
-	Operación resta	1 - 2
*	Operación multiplicación	1 * 2
/	Operador división	1 / 2

Operadores lógicos Retornan un valor lógico (verdadero o falso)

Símbolo	Significado	Ejemplo
=	Igualdad	1 = 2
!= <> ^=	Desigualdad	1 != 2 1 <> 2 1 ^= 2
>	Mayor que	1 > 2
<	Menor que	1 < 2
>=	Mayor o igual que	1 >= 2
<=	Menor o igual que	1 <= 2
IN (RS)	Igual a algún elemento del arreglo de resultados.	1 IN (1,2)

		[TRUE]
ANY SOME	a algún elemento del arreglo de resultados (derecha). Debe ser estar precedido por =, !=, <, <=, >, >= Hace un OR lógico entre todos los elementos.	10 >= ANY (1,2,3,10) [TRUE]
ALL	a todos los elementos del arreglo de resultados (derecha), Debe ser estar precedido por =, !=, <, <=, >, >= Hace un AND lógico entre todos los elementos.	10 <= ALL (1,2,3,10) [TRUE]
BEETWEEN x AND y	Operando de la izquierda entre x e y. Equivalente a op >= x AND op <= y	10 BETWEEN 1 AND 100
EXISTS	Si la retorna al menos una fila	EXISTS(SELECT 1 FROM DUAL)
LIKE(*)	Es como	'pepe' LIKE 'pe%'
IS NULL	Si es nulo	1 IS NULL
IS NOT NULL	Si es No nulo	1 IS NOT NULL
NOT cond.	Niega la condición posterios	NOT EXISTS… NOT BETWEEN NOT IN NOT =
cond AND cond	Hace un Y lógico entre dos condiciones	1=1 AND 2 IS NULL
Cond OR cond	Hace un O lógico entre dos condiciones	1=1 OR 2 IS NULL

Existen los siguientes comodines:

- %: Conjunto de N caracteres (de 0 a ∞)
- _: Un solo carácter

Concatenación

Símbolo	Significado	Ejemplo
||	Concatena una cadena a otra	'juan' || 'cito' ['Juancito']

Oracle puede hacer una conversión automática cuando se utilice este operador con valores numéricos: 10 || 20 = '1020'

Este proceso de denomina CASTING y se puede aplicar en todos aquellos casos en que se utiliza valores numéricos en puesto de valores alfanuméricos o incluso viceversa.

2.8.2 La ausencia de valor: NULL

Todo valor (sea del tipo que sea) puede contener el valor NULL que no es más que la ausencia de valor. Así que cualquier columna (NUMBER, VARCHAR2, DATE…) puede contener el valor NULL, con lo que se dice que la columna *está a NULL*. Una operación retorna NULL si cualquiera de los operandos es NULL. Para comprobar si una valor es NULL se utiliza el operador IS NULL o IS NOT NULL.

2.9 Count

Para contar un numero de registros, se utiliza la palabra 'Count'.


```
SELECT COUNT(<campo_1>) FROM <nombre_tabla>;
```

Ejemplo:
```
SELECT count(*) FROM employee where job_grade=4;
```

2.10Sum, avg, min, max

Para una suma, min, max,... de un campo, se utilizan la palabras 'Sum', 'Min', 'Max', 'Avg'.

```
SELECT SUM(<campo_1>) FROM <nombre_tabla>;

Ejemplo:
SELECT avg(salary) FROM employee where job_grade=2;
```

2.11Distinct

Para tener la lista de valores distingas de un campo, se utiliza la palabra 'Distinct'.

```
SELECT DISTINCT(<campo_1>) FROM <nombre_tabla>;

Ejemplo:
SELECT distinct(job_grade) FROM employee;
```

2.12Order by

Para ordenar los registros regresados, hay que utilizar la palabre 'Order by'.

```
SELECT * FROM <nombre_tabla>
ORDER BY <campo_1>,<....>;

Ejemplo:
SELECT first_name,last_name FROM employee order by first_name,last_name;
```

2.13Uniones

Uniones permiten de unir los resultados de dos consultas. Para poder unirlas, tienen que tener los mismos campos.

```
SELECT <campo_1>,<campo_2>,<...> FROM <nombre_tabla_1>
UNION
SELECT <campo_1>,<campo_2>,<...> FROM <nombre_tabla_2>;

Ejemplo:
select t.first_name,t.last_name from employee t where job_grade=5
union
select t2.fname,t2.lname from patient t2;
```

2.14Subconsultas

Subconsultas son consultas sobre otras consultas. La subconsulta se puede utilizar el la cáusula 'From', o el la condición de la clásula 'Where'. La subconsulta se pone entre paréntesis. En MySQL, las subconsultas deben tener sus propios alias.

```
SELECT t3.<campo_1>, t3.<campo_2> FROM (SELECT t.<campo_1>, t.<campo_2>  FROM <nombre_tabla > t
<where cluase>) t3
WHERE t3.<campo_1> IN (SELECT t2.<campo_1> FROM <nombre_tabla_2> t2);

Ejemplo: SELECT t3.first_name,t3.last_name FROM
(
```



```
select t.first_name,t.last_name from employee t where job_grade=5
union
select t2.fname,t2.lname from patient t2
) t3 where t3.last_name like 'RAMIREZ%';

SELECT t3.first_name,t3.last_name, t3.job_country FROM employee t3
where t3.job_country IN
(select t.country from country t where t.currency='Euro');
```

2.15Agrupaciones

Las agrupaciones permiten agrupar datos y saber cuantos datos hay de cada valor.

```
SELECT <campo_1>,<campo_2>, COUNT(*) FROM <nombre_tabla>
GROUP BY <campo_1>,<campo_2>;
```

Las agrupaciones se pueden filtrar utilizando la clausula HAVING.

```
Ejemplo:
SELECT job_grade, count(*) FROM employee
where emp_no>45
group by job_grade;
SELECT job_grade, sum(salary) FROM employee
where emp_no>45
group by job_grade
having sum(salary)<1000000;
```

2.16Rollup

Rollup de usa en un group by para agregar el total del sub grupo.

Ejemplo:

```
select
    dept_no,
   job_code,
   count(*),
   sum(salary)
FROM
   employee
GROUP BY
   rollup(dept_no,job_code);

select
    dept_no,
   job_code,
   count(*),
   sum(salary)
FROM
   employee
GROUP BY dept_no,
   rollup(job_code);
```

2.17Cube

Cube es parecido al rollup pero da los totales de todos los grupos posibles.

Ejemplo:

```
select
    dept_no,
   job_code,
   count(*),
   sum(salary)
FROM
   employee
GROUP BY
   CUBE(dept_no,job_code);
```


2.18Rendimiento

Un problema común en las encuesta a base de datos es el rendimiento.

Las causas de problema de rendimiento son numerosas.

Las más comunes son:

- Instrucción sin o con mala clausula WHERE
- Falta de indice sobre un campo utilizado como filtro
- Mal diseño de la base de datos
- Problema de hardware (falta de memoria, disco ocupado, cpu ocupado por otra aplicación,...)
- Mala configuración del servidor (mal uso de la memoria, disco, cpu,...)
- Mala programación en el cliente. Falta de commit, conexión no cerrada, ...
- Red sobrecargada o muy lenta

Cuando se enfrenta a un problema de rendimiento hay que probar primero de identificar la causa y los síntomas. Servidor sobrecargado en CPU, disco, memoria? Un cliente afectado o todos? Cuando aparece el problema?

Para ayudar a investigar estos problemas existen herramientas. Algunos vienen con la base de datos, otros están desarrollados aparte.

2.18.1 Indices

Los indices permiten mejorar el rendimiento de las consultas, y entonces del servidor.

Un indice es un pequeño archivo que contiene los valores de uno o varios campos de manera ordenada, junto con la información de la ubicación física del registro.

Cuando la consulta usa el indice, primero lee el archivo del indice, y luego los datos en la tabla. Si todos los campos pedidos están en el indice, solo lee el indice.

Los indices se usan cuando se busca un valor exacto (= o like '...'), un rango de valores (between … and …, >, <,...) , o los valores que 'inician con' (like '...%').

Si se usan funciones (upper(campo), …) no se usara el indice (a menos que existe un indice basado en función).

Los indices tan poco se usan con clausula is null, is not null, in (…), not …, !=, like '%...',....

2.18.2 Diseño de la base de datos y denormalización

Si por lo general es mejor tener una base de datos normalizada (sin información repetida), a veces hay que denormalizar por razón de rendimiento, es decir copiar un valor en una tabla que no le pertenece directamente. Denormalizar tiene un costo a nivel de programación (mantener la integridad de los datos), pero permite mejorar (en ciertos casos) el rendimiento.

2.18.3 Uso del EXPLAIN PLAN

Oracle tiene un motor interno que, basándose en su conocimiento de los datos, su repartición (cardinality), y los indices que existen, decide cual es, según el, la mejor manera de recoger los datos: cuales indices usar, cual tablas, en cual orden, como filtrar, etc...

El 'explain plan' es una herramienta de Oracle que permite conocer cual camino Oracle va usar para recoger los datos de una cierta consulta, en una cierta base de datos, en un cierto momento.

La decisión de Oracle depende muchos factores, entre otros la configuración lógica y física de la base de datos, como de los indices disponibles, de la validez de las estadísticas, de la cantidad de registros,...

Lo ideal es de pasar todas las consultas que se van hacer en producción primero por un explain plan (en el ambiente de desarrollo), para detectar temprano posible problemas de rendimiento.

Los entornos de desarrollo para SQL traen por lo general un GUI para ejecutar/mostrar el explain plan.

2.18.4 Uso de HINTS

Los hints son indicadores que permiten influir en la decisión del motor de Oracle cuanto a la mejor manera de recuperar datos. Permiten 'forzar' el uso de un indice, o cambiar la manera de calculo de Oracle.

Los HINTs se usan con los comandos SELECT, UPDATE, y DELETE, y se mencionan como un comentario con signo +.

```
SELECT /*+ hint or text */
       cuerpo de la consulta
-- o --
SELECT --+ hint or text
       cuerpo de la consulta
```

Notas sobre el uso de HINTS: Si no estan bien formado, serán ignorado (= comentario). No generarán error.

Cuando usar los Hints: NUNCA. O lo menos posible. Son la última opción para mejorar el rendimiento de una consulta. En el 99,9% de los casos, Oracle escoge el mejor camino por si solo. El uso de los hints interfiere con el motor de Oracle, y por eso hay que usarlo con muchas precaución.

2.18.4.1 Optimizer hint

Los Optimizer hints afectan la manera de calcular el mejor camino de recuperación de datos.

`/*+ ALL_ROWS */` - Fuerza a que se utilice el optimizador basado en costes y optimiza el plan de ejecución de la sentencia DML para que devuelva todas las filas en el menor tiempo posible. Es la opción por defecto del optimizador basado en costes y es la solución apropiada para procesos en masa e informes, en los que son necesarias todas las filas para empezar a trabajar con ellas.

`/*+ FIRST_ROWS (n) */` - También fuerza a que se utilice el optimizador basado en costes y optimiza el plan de ejecución de la sentencia DML para que devuelva las "n" primeras filas en el menor tiempo posible. Esto es idóneo para procesos iterativos y bucles, en los que podemos ir trabajando con las primeras filas mientras se recuperan el resto de resultados. Obviamente este hint no será considerado por el optimizador si se utilizan funciones de grupo como MAX, SUM, AVG, etc.

2.18.4.2 Hint de método de acceso a tablas

`/*+ FULL (nombre_tabla) */` - Fuerza a que se realice la búsqueda accediendo a todos los registros de la tabla indicada. Cuando las tablas tienen un número reducido de registros puede resultar bueno para el rendimiento de una sentecia DML el forzar un escaneado completo de la tabla en lugar de que el optimizador decida acceder a dicha tabla mediante un índice, ya que, en estos casos, el acceso por índice suele ser más lento.

`/*+ ROWID (nombre_tabla) */` - Fuerza a que se acceda a la tabla utilizando el ROWID (identificativo único de los registros de una tabla). Este tipo de hint, por si solo, no es muy útil.

`/*+ INDEX (nombre_tabla [nombre_índice] ...) */` - Fuerza a que se acceda a la tabla utilizando, en sentido ascendente, el índice indicado. Muchos problemas de rendimiento vienen causados por el hecho de que el optimizador Oracle decide acceder a una tabla utilizando un índice incorrecto. Mediante este hint podemos indicarle al optimizador que utilice el índice que nosotros consideremos adecuado.

`/*+ INDEX_DESC (nombre_tabla [nombre_índice] ...) */` - Idéntico al anterior hint pero en este caso el acceso a través del índice se hace en sentido descendente.

`/*+ AND_EQUAL (nombre_tabla [nombre_índice] ...) */` - Este hint se utiliza para forzar el uso de más de un índice (se utilizarían los índices indicados como parámetros) y, después, fusionar los índices quedándose con los registros encontrados en todas las búsquedas por índice realizadas.

`/*+ INDEX_FFS (nombre_tabla [nombre_índice] ...) */` - Fuerza el acceso a los datos de la tabla mediante una búsqueda (Scan) rápida (Fast) y total (Full) sobre el índice indicado. Es parecido a utilizar el hint FULL pero sobre un índice en lugar de una tabla, lo cual, difícilmente, puede ser bueno para el rendimiento de una sentencia DML.

`/*+ NO_INDEX (nombre_tabla [nombre_índice] ...) */` - Indica al optimizador que no se utilicen los índices indicados. Puede ser útil cuando no tengamos claro cual es el mejor índice que debe ser utilizado para acceder a una tabla pero, por contra, sepamos que podemos tener problemas de rendimiento si se accede a la tabla por un determinado índice y queramos evitar que esto ocurra.

Ejemplo:

```
select /*+ FULL(patient) */ fname from patient where fname like 'ROBERTO'
```

2.18.4.3 Otros hints

Desde la versión 10g existen más de 70 hints diferentes documentados, y otros (mas de 50) no documentados.

Ver la documentación de Oracle para más detalles sobre los hints.

3 Creación de objetos

3.1 Indices

Los indices permiten aumentar el rendimiento en caso de búsqueda de datos.

3.1.1 Creación

```
Table Index
   CREATE [UNIQUE|BITMAP] INDEX [schema.]index_name
      ON [schema.]table_name [tbl_alias]
         (col [ASC | DESC]) index_clause index_attribs

Bitmap Join Index
   CREATE [UNIQUE|BITMAP] INDEX [schema.]index_name
      ON [schema.]table_name [tbl_alias]
         (col_expression [ASC | DESC])
            FROM [schema.]table_name [tbl_alias]
               WHERE condition [index_clause] index_attribs

Cluster Index
   CREATE [UNIQUE|BITMAP] INDEX [schema.]index_name
      ON CLUSTER [schema.]cluster_name index_attribs

index_clauses:

LOCAL STORE IN (tablespace)

LOCAL STORE IN (tablespace)
  (PARTITION [partition
       [LOGGING|NOLOGGING]
       [TABLESPACE {tablespace|DEFAULT}]
       [PCTFREE int]
       [PCTUSED int]
       [INITRANS int]
       [MAXTRANS int]
       [STORAGE storage_clause]
       [STORE IN {tablespace_name|DEFAULT]
       [SUBPARTITION [subpartition [TABLESPACE tablespace]]]])

LOCAL (PARTITION [partition
       [LOGGING|NOLOGGING]
       [TABLESPACE {tablespace|DEFAULT}]
       [PCTFREE int]
       [PCTUSED int]
       [INITRANS int]
       [MAXTRANS int]
       [STORAGE storage_clause]
       [STORE IN {tablespace_name|DEFAULT]
       [SUBPARTITION [subpartition [TABLESPACE tablespace]]]])

GLOBAL PARTITION BY RANGE (col_list)
   ( PARTITION partition VALUES LESS THAN (value_list)
       [LOGGING|NOLOGGING]
       [TABLESPACE {tablespace|DEFAULT}]
       [PCTFREE int]
       [PCTUSED int]
       [INITRANS int]
       [MAXTRANS int]
       [STORAGE storage_clause] )

INDEXTYPE IS indextype [PARALLEL int|NOPARALLEL] [PARAMETERS ('ODCI_Params')]
 {This for table index only, not bitmap join Index}

index_attribs:
any combination of the following

    NOSORT|SORT
```

```
    REVERSE
    COMPRESS int
    NOCOMPRESS
    COMPUTE STATISTICS
    [NO]LOGGING
    ONLINE
    TABLESPACE {tablespace|DEFAULT}
    PCTFREE int
    PCTUSED int
    INITRANS int
    MAXTRANS int
    STORAGE storage_clause
    PARALLEL parallel_clause

If the PARALLEL clause is used it should be the last option.

For example:
To create a function-based index which allows case-insensitive searches.

CREATE INDEX idx_case_ins ON my_table(UPPER(empname));

SELECT * FROM my_table WHERE UPPER(empname) = 'KARL';
```

3.1.2 Modificación

```
   ALTER INDEX [schema.]index options

Options:
The options used with this command can be any combination of the following

    ENABLE
    DISABLE
    COALESCE
    UNUSABLE
    RENAME TO new_index_name

    [NO]LOGGING
    PCTFREE int
    PCTUSED int
    INITRANS int
    MAXTRANS int
    STORAGE storage_clause

    ALLOCATE EXTENT [SIZE int K | M]
    ALLOCATE EXTENT [DATAFILE 'filename']
    ALLOCATE EXTENT [INSTANCE int]

    DEALLOCATE UNUSED
    DEALLOCATE UNUSED KEEP int K | M

    [NO]MONITORING USAGE
    UPDATE BLOCK REFERENCES

    NOPARALLEL
    PARALLEL int

    MODIFY PARTITION partition COALESCE
    MODIFY PARTITION partition UNUSABLE
    MODIFY PARTITION partition UPDATE BLOCK REFERENCES
    MODIFY PARTITION partition PARAMETERS ('alter_partition_params')
    MODIFY PARTITION partition partition_options

    partition_options:
        ALLOCATE EXTENT [SIZE int K | M]
        ALLOCATE EXTENT [DATAFILE 'filename']
        ALLOCATE EXTENT [INSTANCE int]
        DEALLOCATE UNUSED
        DEALLOCATE UNUSED KEEP int K | M
        [NO]LOGGING
        PCTFREE int
        PCTUSED int
        INITRANS int
```

```
        MAXTRANS int
        STORAGE storage_clause

    RENAME [SUB]PARTITION old_name TO new_name

    DROP PARTITION partition

    SPLIT PARTITION partition AT (value_list)
       [INTO (ptn_descr1, ptn_descr2)] [NOPARALLEL|PARALLEL int]

        ptn_descr:
           PARTITION [partition attrib_options]

    MODIFY DEFAULT ATTRIBUTES [FOR PARTITION partition] attrib_options

    attrib_options:
        TABLESPACE {tablespace|DEFAULT}
        [NO]LOGGING
        PCTFREE int
        PCTUSED int
        INITRANS int
        MAXTRANS int
        STORAGE storage_clause

    MODIFY SUBPARTITION subpartition UNUSABLE
    MODIFY SUBPARTITION subpartition sub_partition_options

    sub_partition_options:
        ALLOCATE EXTENT [SIZE int K | M]
        ALLOCATE EXTENT [DATAFILE 'filename']
        ALLOCATE EXTENT [INSTANCE int]
        DEALLOCATE UNUSED
        DEALLOCATE UNUSED KEEP int K | M

    REBUILD [rebuild_options]
    REBUILD NOREVERSE [rebuild_options]
    REBUILD REVERSE [rebuild_options]
    REBUILD [SUB]PARTITION partition [rebuild_options]

    rebuild_options:
        ONLINE
        COMPUTE STATISTICS
        TABLESPACE tablespace_name
        NOPARALLEL
        PARALLEL int
        [NO]LOGGING
        COMPRESS int
        NOCOMPRESS
        PCTFREE int
        PCTUSED int
        INITRANS int
        MAXTRANS int
        PARAMETERS ('odci_parameters')
        STORAGE storage_clause

More than one ALLOCATE EXTENT option should be specified in the same clause e.g.
ALLOCATE EXTENT SIZE 200K Datafile 'MyFile.idx'
```

3.1.3 Eliminación

```
DROP INDEX [schema.]index [FORCE]

FORCE can be used to drop domain indexes that are marked as IN_PROGRESS or are reporting errors.
```

3.1.4 Ejercicios

1. Llena una tabla con miles de filas
2. Haga una consulta sobre un campo que no tenga indice. Nota el tiempo de ejecución.
3. Repite la consulta.

4. Crea un indice sobre el campo de búsqueda, y repite la consulta.
5. crea un indice bitmap sobre un campo que tenga pocas valores diferentes.

3.2 Vistas

La vista es una sentencia de selección de datos preparada. Permite facilitar las consultas futuras, especialmente cuando se juntan varias tablas. Permite también limitar el acceso a datos (seguridad).

3.2.1 Creación

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW
     [schema.]view
       [(alias,...) inline_constraint(s)]
       [out_of_line_constraint(s)]
       [XMLType_view_clause]
          AS subquery options

options:
   WITH READ ONLY
   WITH CHECK OPTION [CONSTRAINT constraint]
```

3.2.2 Modificación

```
ALTER VIEW [schema.]view COMPILE;
ALTER VIEW [schema.]view ADD out_of_line_constraint;
ALTER VIEW [schema.]view MODIFY CONSTRAINT constraint {RELY | NORELY};
ALTER VIEW [schema.]view DROP CONSTRAINT constraint;
ALTER VIEW [schema.]view DROP PRIMARY KEY
ALTER VIEW [schema.]view UNIQUE (column [,column,...])
```

When a constraint is in NOVALIDATE mode, Oracle does not enforce it and does not take it into account for query rewrite. If you specify RELY Oracle will still not enforce the constraint but will take it into account for query rewrite.

An alternative to ALTER VIEW COMPILE is the built-in pl/sql package DBMS_UTILITY

3.2.3 Eliminación

```
DROP VIEW [schema.]view [CASCADE CONSTRAINTS]
```

3.2.4 Ejercicios

1. Crea una vista sobre 2 tablas ligadas.
2. Usa la vista en una consulta.

3.3 Vistas materializadas

Una vista materializada es un conjunto de datos de una o varias tablas (como una vista) pero del cual el dato se guarda físicamente. Aumenta el rendimiento de las consultas en comparación de la vista normal, pero disminuye el rendimiento de las consultas DML sobre las tablas ligadas a la vista materializada, ya que tiene que mantener la vista materializada ademas de la tabla fuente.

Debido al impacto que tiene sobre el rendimiento general de la base de datos debido a los recursos que usa, evalúa con su DBA la pertinencia de crear una vista materializada.

Un Snapshot es sinónimo de una vista materializada.

Para más información sobre las vistas materializadas, ver en
http://www.softics.ru/docs/oracle10r2/server.101/b10759/statements_6002.htm

3.4 Secuencias

Una secuencia es un contador que se incrementa automáticamente y permite generar numero únicos.

3.4.1 Creación

```
CREATE SEQUENCE [schema.]sequence_name option(s)

Options:
      INCREMENT BY int
      START WITH int
      MAXVALUE int | NOMAXVALUE
      MINVALUE int | NOMINVALUE
      CYCLE | NOCYCLE
      CACHE int | NOCACHE
      ORDER | NOORDER
```

3.4.2 Modificación

```
ALTER SEQUENCE [schema.]sequence_name option(s)

Options:
      INCREMENT BY int
      MAXVALUE int | NOMAXVALUE
      MINVALUE int | NOMINVALUE
      CYCLE | NOCYCLE
      CACHE int | NOCACHE
      ORDER | NOORDER
```

3.4.3 Eliminación

```
DROP SEQUENCE [schema.]sequence_name
```

3.4.4 Ejercicios

1. Crea una secuencia
2. Altera la secuencia para que el próximo numero generado sea 20

3.5 Sinónimos

Un sinónimo es un nombre de objeto que refiere a un objeto con otro nombre, o que se pueda encontrar en otro esquema.

Un sinónimo public es disponible para todos los usuarios (según sus privilegios).

3.5.1 Creación

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema.]synonym
      FOR [schema.]object [@dblink]

'PUBLIC' will create a public synonym, accessible to all users (with the appropriate privileges.)

Unlike Views, Synonyms do not need to be recompiled when the underlying table is redefined.

There is a small performance hit when accessing data through a public synonym.

Oracle will resolve object names in the following order:
```

```
current user
private synonym
public synonym

An alternative method to access data in another schema is to use:

ALTER SESSION set current_schema = Other_Schema
```

3.5.2 Eliminación

```
DROP [PUBLIC] SYNONYM [schema.]synonym [FORCE]

FORCE will drop the synonym even there are dependent tables or user-defined types.
```

3.5.3 Ejercicios

1. Crea un sinónimo publico para un objeto. Luego conecta te como otro usuario y consulta el objeto via el sinónimo público.

3.6 DB Link

Un DB Link permite llamar a objetos que se encuentran en otra base de datos (onlcuido en un servidor remoto).

Hay que tener cuidado son el rendimiento ligado a consultas remotas, ya que puede impactar de manera significativa en servidor remoto.

Evalúa con su DBA la pertinencia de crear un DB Link.

4 PL/SQL básico

4.1 Introducción

El PL/SQL es un lenguaje de programación de Oracle, que permite que el código sea ejecutado en el servidor Oracle mismo. Para códigos que necesitan muchas manipulaciones de datos, el PL/SQL es muy eficiente ya que se ejecuta localmente, con acceso directo a los datos.

El PL/SQL no es CASE-SENSITIVE, es decir, no diferencia mayúsculas de minúsculas como otros lenguajes de programación como C o Java. Sin embargo debemos recordar que ORACLE es CASE-SENSITIVE en la búsquedas de texto (hasta la versión 10.2R2 que lo permite mediante nuevas valores para variables NLS_SORT y NLS_COMP).

Las instrucciones PL/SQL terminan con un punto-coma (;), excepto algunas instrucciones con `BEGIN`, `DECLARE`, etc..

Los comentarios de una línea se inician con – y los de varias líneas inician con /* y terminan con */

4.2 Bloque anónimo PL/SQL

Un bloque PL/SQL es un conjunto de instrucciones PL/SQL relacionadas entre ellas.

Un simple bloque PL/SQL inicia con `BEGIN` y termina con `END;`

```
Ejemplo:
SET SERVEROUTPUT ON;
BEGIN
dbms_output.put_line('Hola');
END;
/
```

4.3 Identificadores

Un identificador es el nombre de un objeto PL/SQL, como por ejemplo una constante, variable, excepción, paquete, función, procedimiento, tabla, cursor,...

Un identificador puede tener hasta 30 caracteres, debe iniciar con una letra. Puede contener los signos $ o #, pero no puede contener operadores (+-%=/*...) ni ser igual a una palabra reservada.

La lista de palabras reservada se encuentra en la vista V$RESERVED_WORDS.

4.4 Variables

Las variables se declaran antes de la palabra BEGIN, y se preceden de la palabra DECLARE.

La sintaxis es : *nombre_de_variable* [CONSTANT] *tipo* [NOT NULL] [DEFAULT *valor_por_defecto*];

Ejemplo:

```
DECLARE
counter   BINARY_INTEGER := 0;
priority  VARCHAR2(8)    DEFAULT 'LOW';
...
```


4.4.1 Tipos de variables

Los tipos de variables que se pueden usar en PL/SQL son básicamente los mismos que se usan para definir campos en una tabla, plus algunos específicos del PL/SQL.

PLS_INTEGER: tipo numérico del PL/SQL, equivalente a NUMBER, pero más eficiente si se necesita hacer cálculos aritméticos.

%TYPE: refiere de manera dinámica a otro objeto.
%ROWTYPE: refiere de manera dinámica a una fila de una tabla.

Ejemplo:
```
set serveroutput on;
DECLARE
v_fname employee.first_name%TYPE;
begin
select first_name into v_fname from employee where emp_no=2;
dbms_output.put_line('Nombre='||v_fname);
end;
/
```

4.4.1.1 Tipos personalizados

Es posible crear sus propios tipo de datos. Los tipos personalizados están fuera del alcance de este curso ya que se miran en un curso de PL/SQL avanzado.

4.4.2 Variables locales

Por defecto, una variable solo está disponible en el bloque PL/SQL donde se encuentra, y los sub bloques dentro de este bloque.

Ejemplo:
```
DECLARE
 v_cnt_emp number;
BEGIN
 select count(*) into v_cnt_emp
 from employee
 where salary>1000;
  BEGIN
   dbms_output.put_line ('Empleados con salario>1000 : '||v_cnt_emp);
  END;
END;
/
```

4.4.3 Variables globales

Las variables globales son variables que se pueden llamar desde otro código.

Solo están disponibles en paquetes, pero están fuera del alcance de este curso ya que se miran en un curso de PL/SQL avanzado.

4.5 Control de flujo

4.5.1 If..elsif...else

Se usa el IF..ELSIF..ELSE para controlar el flujo en PL/SQL.

```
IF (expresion) THEN
```

```
           -- Instrucciones
ELSIF (expresion) THEN
           -- Instrucciones
ELSE
           -- Instrucciones
END IF;
```

Ejemplo:

```
DECLARE
 v_cnt_emp number;
 v_tot_emp number;
BEGIN
 select count(*) into v_tot_emp
 from employee;
 dbms_output.put_line ('Hay un total de '||v_tot_emp||' empleados.');
 select count(*) into v_cnt_emp
 from employee
 where salary>1000;
 dbms_output.put_line ('Hay '||v_cnt_emp||' empleados que ganan mas de 1000 $.');
 dbms_output.put_line ('Resultado de la analisis.');
 IF (v_tot_emp=v_cnt_emp) then
   dbms_output.put_line ('Los empleados ganan todos mas de 1000 $');
 ELSIF (v_tot_emp<v_cnt_emp*2) then
   dbms_output.put_line ('Mas de la mitad empleados ganan todos mas de 1000 $');
 ELSE
   dbms_output.put_line ('Menos de la mitad empleados ganan todos mas de 1000 $');
 END IF;
END;
/
```

4.5.2 Case

Es parecido al IF...elsif...else. Se menciona la variable a evaluar y luego los valores posibles con su código apropiado.

Ejemplo:

```
DECLARE
v_salary employee.salary%TYPE;
begin
select salary into v_salary from employee where emp_no=65;
dbms_output.put_line('Salario:'|| v_salary);
case
  when v_salary=0 THEN
    dbms_output.put_line('Gratis!');
  when v_salary<10000 then
    dbms_output.put_line('Salado!');
  when v_salary<90000 then
    dbms_output.put_line('Mas o menos');
  when v_salary>=90000 then
    dbms_output.put_line('Correcto');
end case;
end;
/
DECLARE
v_job_grade employee.job_grade%TYPE;
begin
select job_grade into v_job_grade from employee where emp_no=2;
dbms_output.put_line('job_grade:'|| v_job_grade);
case v_job_grade
  when 1 THEN
    dbms_output.put_line('Jefe!');
  when 2 then
    dbms_output.put_line('Jefecito');
  when 3 then
    dbms_output.put_line('Empleado regular');
  ELSE
    dbms_output.put_line('Esclavo o dios');
end case;
end;
/
```

4.5.3 Goto

El GOTO reenvia de manera indondicional hacia una etiqueta (otra parte del código). No aconsejo el uso del GOTO ya que complica el flujo y puede llavar a bucles infinitas.

Al meno una instrucción debe seguir el GOTO (NULL se puede usar).

Ejemplo:
```
BEGIN
   GOTO mas_adelante;
   DBMS_OUTPUT.PUT_LINE('Nunca se ejecutará.');

   <<mas_adelante>>
   DBMS_OUPUT.PUT_LINE('Aqui estamos.');
END;
```

Un GOTO tiene algunas restriciones:
- Puede salir de un IF, LOOP, o sub-bloque
- No puede entrar en un IF, LOOP, o sub-bloque
- No puede mover de una sección de un IF hacia otra sección del IF
- No puede entrar ni salir de un sub-programa
- No puede pasar de una sección ejecutable a una sección de excepción, ni vice versa.

4.6 Bucles

Una bucle permite repetir una acción un sin número de veces. Hay que tener cuidado en no crear bucles infinitas.

En PL/SQL tenemos a nuestra disposición los siguientes iteradores o bucles:
* LOOP
* WHILE
* FOR

4.6.1 LOOP

El bucle LOOP, se repite tantas veces como sea necesario hasta que se fuerza su salida con la instrucción EXIT. Su sintaxis es la siguiente

```
LOOP
   -- Instrucciones
   IF (expresion) THEN
         -- Instrucciones
         EXIT;
   END IF;
END LOOP;
```

Ejemplo:
```
DECLARE
 v_cnt PLS_INTEGER DEFAULT 0;
BEGIN
LOOP
  v_cnt:=v_cnt+1;
 dbms_output.put_line ('Contador = '||v_cnt);
 IF (v_cnt>=10) then
   dbms_output.put_line ('Ya merito!');
   exit;
 END IF;
 END LOOP;
END;
```


4.6.2 WHILE

El bucle WHILE, se repite mientras que se cumpla expresion.

```
WHILE (expresion) LOOP
   -- Instrucciones
END LOOP;
```

Ejemplo:

```
DECLARE
 v_cnt PLS_INTEGER DEFAULT 0;
BEGIN
v_cnt:=10;
 WHILE (v_cnt>0) LOOP
 dbms_output.put_line ('Contador = '||v_cnt);
  v_cnt:=v_cnt-1;
 END LOOP;
END;
```

4.6.3 FOR

El bucle FOR, se repite tanta veces como le indiquemos en los identificadores inicio y final.

```
FOR contador IN [REVERSE] inicio..final LOOP
   -- Instrucciones
END LOOP;
```

En el caso de especificar REVERSE el bucle se recorre en sentido inverso.

Ejemplo:

```
DECLARE
 v_cnt PLS_INTEGER DEFAULT 0;
BEGIN
 FOR v_cnt IN REVERSE 1..10 LOOP
 dbms_output.put_line ('Contador = '||v_cnt);
 END LOOP;
END;
```

4.7 *Exit*

Permite salir de una bucle. Se puede mencionar una condición de salida con WHEN.

Ejemplo:

```
DECLARE
v_cnt integer :=0;
begin
loop
 dbms_output.put_line(v_cnt);
 v_cnt:=v_cnt+1;
 exit when v_cnt=10;
 end loop;
end;
```

4.8 *NULL*

NULL es un comando que no hace nada. Sirve por ejemplo a documentar una condición que no hace nada, o rellenar una condición que se implementará más tarde.

4.9 *EXECUTE IMMEDIATE*

El comando EXECUTE IMMEDIATE permite ejecutar un comando SQL de manera dinámica.

Hay que tratar de usarlo lo menos posible ya que el SQL dinámico se tiene que recompilar a cada ejecución.

4.10Cursores

PL/SQL utiliza cursores para gestionar las instrucciones SELECT. Un cursor es un conjunto de registros devuelto por una instrucción SQL. Técnicamente los cursores son fragmentos de memoria que reservados para procesar los resultados de una consulta SELECT.

Podemos distinguir dos tipos de cursores:

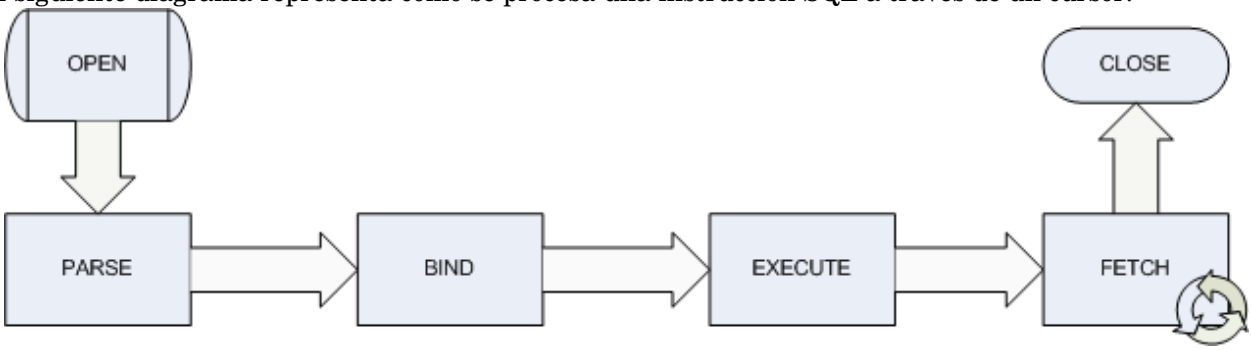
- Cursores implicitos. Este tipo de cursores se utiliza para operaciones SELECT INTO, que se usa cuando la consulta devuelve un único registro. También se pueden usar mencionando el SELECT en el FOR LOOP, devolviendo un conjunto de registros, o cuando se hace un insert/ uodate/delete.
- Cursores explicitos. Son los cursores que son declarados y controlados por el programador. Se utilizan cuando la consulta devuelve un conjunto de registros. Ocasionalmente también se utilizan en consultas que devuelven un único registro por razones de eficiencia. Son más rápidos.

Un cursor se define como cualquier otra variable de PL/SQL y debe nombrarse de acuerdo a los mismos convenios que cualquier otra variable. Los cursores implicitos no necesitan declaración.

Para procesar instrucciones SELECT que devuelvan más de una fila, son necesarios cursores explicitos combinados con un estructura de bloque.

Un cursor admite el uso de parámetros. Los parámetros deben declararse junto con el cursor.

El siguiente diagrama representa como se procesa una instrucción SQL a través de un cursor.



4.10.1.1 Atributos de cursores

SQL%ISOPEN : True si el cursor esta abierto (entre open y close),
SQL%FOUND :

- NULL antes de ejecutar
- TRUE si uno o mas registros fueron inserted, merged, updated, o deleted o si solo 1 registro fue seleccionado.
- FALSE si ningún registro fue seleccionado, merged, updated, inserted, o deleted.

SQL%NOTFOUND

- NULL antes de ejecutar
- TRUE si ningún registro fue seleccionado, merged, updated, inserted, o deleted.
- FALSE si uno o mas registros fueron inserted, merged, updated, deleted o seleccionado.

SQL%ROWCOUNT Cantidad de registros afectados por el cursor.

Ejemplo de cursor implicito:

```
DECLARE
v_sal NUMBER;
BEGIN
SELECT sal INTO v_sal FROM employee WHERE empno=7369;
  dbms_output.put_line('El empleado numero 7369 tiene un salario de '||v_sal||' $');
end;
/


DECLARE
c employee%ROWTYPE;
begin
for c in (select * from employee) loop
 dbms_output.put_line(c.first_name);
 end loop;
end;
/
```

Ejemplo de cursor explicito:

```
DECLARE
CURSOR c_emp IS /*CURSOR*/
select ename, sal from employee;
BEGIN
FOR fila IN c_emp LOOP     /*no es necesario definir la variable fila, será de tipo %ROW */
  dbms_output.put_line(fila.ename||' tiene un salario de '||fila.sal);
END LOOP;
END;
/
```

4.10.1.2 SELECT ... FOR UPDATE / WHERE CURRENT OF ...

Normalmente Oracle bloquea los registros al momento de modificarlos.

Usando el select... for update genera un bloqueo sobre todos los registros de la consulta.

Where current of permite modificar el registro corriente.
Ejemplo:

```
DECLARE
   emp_rec employee%ROWTYPE;
   cursor c_emp is select * from employee for update;
BEGIN
   OPEN c_emp;
   LOOP
      FETCH c_emp into emp_rec;
      EXIT WHEN c_emp%NOTFOUND
               OR
               c_emp%ROWCOUNT > 10;

      UPDATE employee
         SET salary = salary+5
         where current of c_emp;
   END LOOP;
   CLOSE c_emp;
END;
```

Existen otros tipo de curso explicitos. Se mira en el curso PL/SQL avanzado.

- BULK COLLECT permite cargar a alta velocidad los datos (en una tabla PL/SQL).
- Usando FORALL mejora el rendimiento de cursores que hacen insert/update/delete.

4.11Excepciones

En PL/SQL una advertencia o condición de error es llamada una excepción.

Los bloques de excepciones permiten atrapar errores de ejecución y darles eventualmente un tratamiento para evitar que se pare el programa de manera anormal.

Las excepciones se controlan dentro de su propio bloque. La estructura de bloque de una excepción se muestra a continuación.

```
DECLARE
   -- Declaraciones
  BEGIN
    -- Ejecucion
  EXCEPTION
    -- Excepcion
  END;
```

Cuando ocurre un error, se ejecuta la porción del programa marcada por el bloque EXCEPTION, transfiriéndose el control a ese bloque de sentencias.

El siguiente ejemplo muestra un bloque de excepciones que captura las excepciones NO_DATA_FOUND y ZERO_DIVIDE. Cualquier otra excepcion será capturada en el bloque WHEN OTHERS THEN.

```
DECLARE
  -- Declaraciones
 BEGIN
   -- Ejecucion
 EXCEPTION
 WHEN NO_DATA_FOUND THEN
   -- Se ejecuta cuando ocurre una excepcion de tipo NO_DATA_FOUND
 WHEN ZERO_DIVIDE THEN
   -- Se ejecuta cuando ocurre una excepcion de tipo ZERO_DIVIDE
 WHEN OTHERS THEN
   -- Se ejecuta cuando ocurre una excepcion de un tipo no tratado
   -- en los bloques anteriores
 END;
```

Como ya hemos dicho cuando ocurre un error, se ejecuta el bloque EXCEPTION, transfiriéndose el control a las sentencias del bloque. Una vez finalizada la ejecución del bloque de EXCEPTION no se continua ejecutando el bloque anterior.

Si existe un bloque de excepción apropiado para el tipo de excepción se ejecuta dicho bloque. Si no existe un bloque de control de excepciones adecuado al tipo de excepción se ejecutará el bloque de excepción WHEN OTHERS THEN (si existe!). WHEN OTHERS debe ser el último manejador de excepciones.

Las excepciones pueden ser definidas en forma interna o explícitamente por el usuario. Ejemplos de excepciones definidas en forma interna son la división por cero y la falta de memoria en tiempo de ejecución. Estas mismas condiciones excepcionales tienen sus propio tipos y pueden ser referenciadas por ellos: ZERO_DIVIDE y STORAGE_ERROR.

Las excepciones definidas por el usuario deben ser alcanzadas explícitamente utilizando la sentencia RAISE.

Con las excepciones se pueden manejar los errores cómodamente sin necesidad de mantener múltiples chequeos por cada sentencia escrita. También provee claridad en el código ya que permite mantener las rutinas correspondientes al tratamiento de los errores de forma separada de la lógica del negocio.

4.11.1 Excepciones predefinidas

PL/SQL proporciona un gran número de excepciones predefinidas que permiten controlar las condiciones de error más habituales.

Las excepciones predefinidas no necesitan ser declaradas. Simplemente se utilizan cuando estas son lanzadas por algún error determinado.

4.11.1.1 Excepciones asociadas a los cursores implícitos.

Los cursores implicitos sólo pueden devolver una fila, por lo que pueden producirse determinadas excepciones. Las más comunes que se pueden encontrar son no_data_found y too_many_rows. La siguiente tabla explica brevemente estas excepciones.

NO_DATA_FOUND Se produce cuando una sentencia SELECT intenta recuperar datos pero ninguna fila satisface sus condiciones. Es decir, cuando "no hay datos"
TOO_MANY_ROWS Dado que cada cursor implicito sólo es capaz de recuperar una fila , esta excepcion detecta la existencia de más de una fila.

Ejemplo:

```
DECLARE
v_sal NUMBER;
BEGIN
BEGIN
SELECT salary INTO v_sal FROM employee WHERE emp_no=1;
  dbms_output.put_line('El empleado numero 1 tiene un salario de '||v_sal||' $');
EXCEPTION
  WHEN NO_DATA_FOUND THEN
  dbms_output.put_line('El empleado numero 1 no existe!');
END;
BEGIN
SELECT emp_no INTO v_sal FROM employee WHERE salary>1000;
  dbms_output.put_line('El empleado que tiene un salario>1000 $ es el empleado numero'||v_sal);
EXCEPTION
  WHEN too_many_rows THEN
  dbms_output.put_line('HAY MAS DE UN EMPLEADO QUE GANA MAS DE 1000 $!');
END;
END;
```

4.11.1.2 Lista de excepciones predefinidas

ACCESS_INTO_NULL El programa intentó asignar valores a los atributos de un objeto no inicializado -6530
COLLECTION_IS_NULL El programa intentó asignar valores a una tabla anidada aún no inicializada -6531
CURSOR_ALREADY_OPEN El programa intentó abrir un cursor que ya se encontraba abierto. Recuerde que un cursor de ciclo FOR automáticamente lo abre y ello no se debe especificar con la sentencia OPEN -6511
DUP_VAL_ON_INDEX El programa intentó almacenar valores duplicados en una columna que se mantiene con restricción de integridad de un índice único (unique index) -1
INVALID_CURSOR El programa intentó efectuar una operación no válida sobre un cursor -1001
INVALID_NUMBER En una sentencia SQL, la conversión de una cadena de caracteres hacia un número falla cuando esa cadena no representa un número válido -1722
LOGIN_DENIED El programa intentó conectarse a Oracle con un nombre de usuario o password inválido -1017
NO_DATA_FOUND Una sentencia SELECT INTO no devolvió valores o el programa referenció un elemento no inicializado en una tabla indexada 100
NOT_LOGGED_ON El programa efectuó una llamada a Oracle sin estar conectado -1012
PROGRAM_ERROR PL/SQL tiene un problema interno -6501
ROWTYPE_MISMATCH Los elementos de una asignación (el valor a asignar y la variable que lo contendrá) tienen tipos incompatibles. También se presenta este error cuando un parámetro pasado a un subprograma no es del tipo esperado -6504

SELF_IS_NULL El parámetro SELF (el primero que es pasado a un método MEMBER) es nulo -30625
STORAGE_ERROR La memoria se terminó o está corrupta -6500
SUBSCRIPT_BEYOND_COUNT El programa está tratando de referenciar un elemento de un arreglo indexado que se encuentra en una posición más grande que el número real de elementos de la colección -6533
SUBSCRIPT_OUTSIDE_LIMIT El programa está referenciando un elemento de un arreglo utilizando un número fuera del rango permitido (por ejemplo, el elemento "-1") -6532
SYS_INVALID_ROWID La conversión de una cadena de caracteres hacia un tipo rowid falló porque la cadena no representa un número -1410
TIMEOUT_ON_RESOURCE Se excedió el tiempo máximo de espera por un recurso en Oracle -51
TOO_MANY_ROWS Una sentencia SELECT INTO devuelve más de una fila -1422
VALUE_ERROR Ocurrió un error aritmético, de conversión o truncamiento. Por ejemplo, sucede cuando se intenta calzar un valor muy grande dentro de una variable más pequeña -6502
ZERO_DIVIDE El programa intentó efectuar una división por cero -1476

4.11.2 Excepciones definidas por el usuario

PL/SQL permite al usuario definir sus propias excepciones, las que deberán ser declaradas y lanzadas explícitamente utilizando la sentencia RAISE.

Las excepciones deben ser declaradas en el segmento DECLARE de un bloque, subprograma o paquete. Se declara una excepción como cualquier otra variable, asignandole el tipo EXCEPTION. Las mismas reglas de alcance aplican tanto sobre variables como sobre las excepciones.

```
DECLARE
 -- Declaraciones
MyExcepcion EXCEPTION;
BEGIN
  -- Ejecucion
EXCEPTION
  -- Excepcion
END;
```

4.11.2.1 Reglas de Alcance

Una excepcion es válida dentro de su ambito de alcance, es decir el bloque o programa donde ha sido declarada. Las excepciones predefinidas son siempre válidas.

Como las variables, una excepción declarada en un bloque es local a ese bloque y global a todos los sub-bloques que comprende.

4.11.2.2 La sentencia RAISE

La sentencia RAISE permite lanzar una excepción en forma explícita. Es posible utilizar esta sentencia en cualquier lugar que se encuentre dentro del alcance de la excepción.

```
DECLARE
   -- Declaramos una excepcion identificada por VALOR_NEGATIVO
  VALOR_NEGATIVO EXCEPTION;
  valor NUMBER;
  BEGIN
    -- Ejecucion
  valor := -1;
  IF valor < 0 THEN
  RAISE VALOR_NEGATIVO;
  END IF;
  EXCEPTION
    -- Excepcion
  WHEN VALOR_NEGATIVO THEN
  dbms_output.put_line('El valor no puede ser negativo');
  END;
```


Con la sentencia RAISE podemos lanzar una excepción definida por el usuario o predefinida, siendo el comportamiento habitual lanzar excepciones definidas por el usuario.

Recordar la existencia de la excepción OTHERS, que simboliza cualquier condición de excepción que no ha sido declarada. Se utiliza comúnmente para controlar cualquier tipo de error que no ha sido previsto. En ese caso, es común observar la sentencia ROLLBACK en el grupo de sentencias de la excepción o alguna de las funciones SQLCODE – SQLERRM, que se detallan en el próximo punto.

4.11.2.3 Uso de SQLCODE y SQLERRM

Al manejar una excepción es posible usar las funciones predefinidas SQLCode y SQLERRM para aclarar al usuario la situación de error acontecida.

SQLcode devuelve el número del error de Oracle y un 0 (cero) en caso de exito al ejecutarse una sentencia SQL.

Por otra parte, SQLERRM devuelve el correspondiente mensaje de error.

Estas funciones son muy útiles cuando se utilizan en el bloque de excepciones, para aclarar el significado de la excepción OTHERS.

Estas funciones no pueden ser utilizadas directamente en una sentencia SQL, pero sí se puede asignar su valor a alguna variable de programa y luego usar esta última en alguna sentencia.

```
DECLARE
  err_num NUMBER;
  err_msg VARCHAR2(255);
  result  NUMBER;
BEGIN
  SELECT 1/0 INTO result
  FROM DUAL;
EXCEPTION
WHEN OTHERS THEN
  err_num := SQLCODE;
  err_msg := SQLERRM;
 DBMS_OUTPUT.put_line('Error:'||TO_CHAR(err_num));
  DBMS_OUTPUT.put_line(err_msg);
END;
```

También es posible entregarle a la función SQLERRM un número negativo que represente un error de Oracle y ésta devolverá el mensaje asociado.

```
DECLARE
  msg VARCHAR2(255);
BEGIN
  msg := SQLERRM(-1403);
  DBMS_OUTPUT.put_line(MSG);
END;
```

4.11.3 RAISE_APPLICATION_ERROR

En ocasiones queremos enviar un mensaje de error personalizado al producirse una excepción PL/SQL. Para ello es necesario utilizar la instruccion RAISE_APPLICATION_ERROR;

La sintaxis general es la siguiente:

RAISE_APPLICATION_ERROR(<error_num>,<mensaje>);

Siendo:

* error_num es un entero negativo comprendido entre -20001 y -20999
* mensaje la descripcion del error

Ejemplo:

```
DECLARE
   v_div NUMBER;
 BEGIN
      SELECT 1/0 INTO v_div FROM DUAL;
 EXCEPTION
   WHEN OTHERS THEN
   RAISE_APPLICATION_ERROR(-20001,'No se puede dividir por cero');
 END;
```

4.12Procedimientos, funciones, paquetes, disparadores

Son objetos PL/SQL residentes en el servidor PL/SQL.

4.12.1 Funciones

Una función es un código compilados en el servidor, que se ejecutan en local, y que pueden aceptar parámetros de entrada y tiene un solo parámetro de salido. Una función SIEMPRE debe regresar un valor.

Una función se puede usar en otro código PL/SQL, o en SQL ya sea en un SELECT, una clausula WHERE, CONNECT BY, START WITH, ORDER BY, GROUP BY, como VALUES en un INSERT, o como SET en un UPDATE.

4.12.1.1 Funciones predefinidas

PL/SQL tiene un gran número de funciones incorporadas, sumamente útiles. A continuación vamos a ver algunas de las más utilizadas.

SYSDATE

Devuelve la fecha del sistema:

SELECT SYSDATE FROM DUAL;

NVL

Devuelve el valor recibido como parámetro en el caso de que expresión sea NULL,o expresión en caso contrario.

```
NVL(<expresion>, <valor>)
```

El siguiente ejemplo devuelve 0 si el precio es nulo, y el precio cuando está informado:

```
SELECT CO_PRODUCTO, NVL(PRECIO, 0) FROM PRECIOS;
```

DECODE

Decode proporciona la funcionalidad de una sentencia de control de flujo if-elseif-else.

```
DECODE(<expr>, <cond1>, <val1>[, ..., <condN>, <valN>], <default>)
```

Esta función evalúa una expresión "<expr>", si se cumple la primera condición "<cond1>" devuelve el valor1 "<val1>", en caso contrario evalúa la siguiente condición y así hasta que una de las condiciones se cumpla. Si no se cumple ninguna condición se devuelve el valor por defecto.

Es muy común escribir la función DECODE identada como si se tratase de un bloque IF.

```
SELECT DECODE (co_pais, /* Expresion a evaluar */
               'ESP', 'ESPAÑA', /* Si co_pais = 'ESP' ==> 'ESPAÑA' */
               'MEX', 'MEXICO', /* Si co_pais = 'MEX' ==> 'MEXICO' */
               'PAIS '||co_pais)/* ELSE ==> concatena */
FROM PAISES;
```

TO_DATE

Convierte una expresión al tipo fecha. El parámetro opcional formato indica el formato de entrada de la expresión no el de salida.

```
TO_DATE(<expresion>, [<formato>])
```

En este ejemplo convertimos la expresion '01/12/2006' de tipo CHAR a una fecha (tipo DATE). Con el parámetro formato le indicamos que la fecha está escrita como día-mes-año para que devuelve el uno de diciembre y no el doce de enero.

```
SELECT TO_DATE('01/12/2006',
               'DD/MM/YYYY')
FROM DUAL;
```

Este otro ejemplo muestra la conversión con formato de día y hora.

```
SELECT TO_DATE('31/12/2006 23:59:59',
               'DD/MM/YYYY HH24:MI:SS')
FROM DUAL;
```

Parameter	***Explanation***
YEAR	Year, spelled out
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 month.
MONTH	Name of month, padded with blanks to length of 9 characters.
RM	Roman numeral month (I-XII; JAN = I).
WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
IW	Week of year (1-52 or 1-53) based on the ISO standard.
D	Day of week (1-7).
DAY	Name of day.
DD	Day of month (1-31).
DDD	Day of year (1-366).
DY	Abbreviated name of day.
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	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, 'FF4'.
AM, A.M., PM, or P.M.	Meridian indicator
AD or A.D	AD indicator
BC or B.C.	BC indicator
TZD	Daylight savings information. For example, 'PST'
TZH	Time zone hour.
TZM	Time zone minute.
TZR	Time zone region.

TO_CHAR

Convierte una expresión al tipo CHAR. El parámetro opcional formato indica el formato de salida de la expresión.

```
TO_CHAR(<expresion>, [<formato>])

    SELECT TO_CHAR(SYSDATE, 'DD/MM/YYYYY')
     FROM DUAL;
```

TO_NUMBER

Convierte una expresion alfanumérica en numerica. Opcionalmente podemos especificar el formato de salida.

```
TO_NUMBER(<expresion>, [<formato>])

    SELECT TO_NUMBER ('10')
    FROM DUAL;
```

TRUNC

Trunca una fecha o número.

Si el parámetro recibido es una fecha elimina las horas, minutos y segundos de la misma.

```
SELECT TRUNC(SYSDATE)FROM DUAL;
```

Si el parámetro es un número devuelve la parte entera.

```
SELECT TRUNC(9.99)FROM DUAL;
```

LENGTH

Devuelve la longitud de un tipo CHAR.

```
SELECT LENGTH('HOLA MUNDO')FROM DUAL;
```

INSTR

Busca una cadena de caracteres dentro de otra. Devuelve la posicion de la ocurrencia de la cadena buscada.

Su sintaxis es la siguiente: `INSTR(<char>, <search_string>, <startpos>, <occurrence> )`

```
SELECT INSTR('AQUI ES DONDE SE BUSCA', 'BUSCA', 1, 1 )
FROM DUAL;
```


REPLACE

Reemplaza un texto por otro en un expresion de busqueda.

```
REPLACE(<expresion>, <busqueda>, <reemplazo>)
```

El siguiente ejemplo reemplaza la palabra 'HOLA' por 'VAYA' en la cadena 'HOLA MUNDO'.

```
SELECT REPLACE ('HOLA MUNDO','HOLA', 'VAYA')-- devuelve VAYA MUNDO
FROM DUAL;
```

SUBSTR

Obtiene una parte de una expresion, desde una posición de inicio hasta una determinada longitud.

```
SUBSTR(<expresion>, <posicion_ini>, <longitud> )
   SELECT SUBSTR('HOLA MUNDO', 6, 5) -- Devuelve MUNDO
   FROM DUAL;
```

UPPER

Convierte una expresion alfanumerica a mayúsculas.

```
SELECT UPPER('hola mundo') -- Devuelve HOLA MUNDO
FROM DUAL;
```

LOWER

Convierte una expresion alfanumerica a minúsculas.

```
SELECT LOWER('HOLA MUNDO') -- Devuelve hola mundo
FROM DUAL;
```

ROWIDTOCHAR

Convierte un ROWID a tipo caracter.

```
SELECT ROWIDTOCHAR(ROWID)
FROM DUAL;
```

RPAD

Añade N veces una determinada cadena de caracteres a la derecha una expresión. Muy util para generar ficheros de texto de ancho fijo.

```
RPAD(<expresion>, <longitud>, <pad_string> )
```

El siguiente ejemplo añade puntos a la expresion 'Hola mundo' hasta alcanzar una longitud de 50 caracteres.

```
SELECT RPAD('Hola Mundo', 50, '.')
FROM DUAL;
```

LPAD

Añade N veces una determinada cadena de caracteres a la izquierda de una expresión. Muy util para generar ficheros de texto de ancho fijo.

```
LPAD(<expresion>, <longitud>, <pad_string> )
```

El siguiente ejemplo añade puntos a la expresion 'Hola mundo' hasta alcanzar una longitud de 50 caracteres.

```
SELECT LPAD('Hola Mundo', 50, '.')
FROM DUAL;
```


RTRIM

Elimina los espacios en blanco a la derecha de una expresion

```
SELECT RTRIM ('Hola Mundo    ')
FROM DUAL;
```

LTRIM

Elimina los espacios en blanco a la izquierda de una expresion

```
SELECT LTRIM ('       Hola Mundo')
FROM DUAL;
```

TRIM

Elimina los espacios en blanco a la izquierda y derecha de una expresion

```
SELECT TRIM ('       Hola Mundo        ')
FROM DUAL;
```

MOD

Devuelve el resto de la división entera entre dos números.

```
MOD(<dividendo>, <divisor> )
    SELECT MOD(20,15) -- Devuelve el modulo de dividir 20/15
    FROM DUAL
```

4.12.1.2 Funciones definidas por el usuario

La sintaxis para construir funciones es la siguiente:

```
CREATE [OR REPLACE]
FUNCTION <fn_name>[(<param1> IN <type>, <param2> IN <type>, ...)]
RETURN <return_type>
IS
  result <return_type>;
BEGIN
  return(result);
[EXCEPTION]
  -- Sentencias control de excepcion
END [<fn_name>];
```

El uso de OR REPLACE permite sobreescribir una función existente. Si se omite, y la función existe, se producirá, un error.

La sintaxis de los parámetros es la misma que en los procedimientos almacenado, exceptuando que solo pueden ser de entrada.

Ejemplo:

```
create table emp as select * from employee;

CREATE OR REPLACE
    FUNCTION fn_Obtener_Salario(p_empno NUMBER)
    RETURN NUMBER
    IS
      result NUMBER;
    BEGIN
      SELECT salary INTO result
      FROM employee
      WHERE emp_no = p_empno;
      return(result);
    EXCEPTION
    WHEN NO_DATA_FOUND THEN
      return 0;
    END ;
```

Si el sistema nos indica que el la función se ha creado con errores de compilación podemos ver estos errores de compilacion con la orden SHOW ERRORS en SQL *Plus.

Las funciones pueden utilizarse en sentencias SQL de manipulación de datos (SELECT, UPDATE, INSERT y DELETE):

```
SELECT emp_no, fname,
        fn_Obtener_Salario( emp_no)
 FROM employee;
```

4.12.2 Procedimientos

Los procedimientos son códigos compilados en el servidor, que se ejecutan en el servidor, y que pueden aceptar parámetros de entrada y/o de salida. Un procedimiento se puede usar en un select, pero no en un where clause.

Un procedimiento tiene un nombre, un conjunto de parámetros (opcional) y un bloque de código.

La sintaxis de un procedimiento almacenado es la siguiente:

```
CREATE [OR REPLACE]
 PROCEDURE <procedure_name> [(<param1> [IN|OUT|IN OUT] <type>,
                              <param2> [IN|OUT|IN OUT] <type>, ...)]
 IS
   -- Declaracion de variables locales
 BEGIN
   -- Sentencias
 [EXCEPTION]
   -- Sentencias control de excepcion
 END [<procedure_name>];
```

El uso de OR REPLACE permite sobreescribir un procedimiento existente. Si se omite, y el procedimiento existe, se producirá, un error.

La sintaxis es muy parecida a la de un bloque anónimo, salvo porque se reemplaza la sección DECLARE por la secuencia PROCEDURE ... IS en la especificación del procedimiento.

Debemos especificar el tipo de datos de cada parámetro. Al especificar el tipo de dato del parámetro no debemos especificar la longitud del tipo.

Los parámetros pueden ser de entrada (IN), de salida (OUT) o de entrada salida (IN OUT). El valor por defecto es IN, y se toma ese valor en caso de que no especifiquemos nada.

```
create or replace
PROCEDURE Actualiza_Salario(p_empno NUMBER,
                            p_new_salario NUMBER)
    IS
      -- Declaracion de variables locales
    BEGIN
      -- Sentencias
      UPDATE employee
       SET salary = p_new_salario,
              hire_date = SYSDATE
      WHERE emp_no = p_empno;
    END Actualiza_Salario;
```

También podemos asignar un valor por defecto a los parámetros, utilizando la clausula DEFAULT o el operador de asiganción (:=) .

```
create or replace
PROCEDURE Actualiza_Salario(p_empno NUMBER,
                            p_new_salario NUMBER DEFAULT 500)
    IS
...
    END Actualiza_Salario;
```


Una vez creado y compilado el procedimiento almacenado podemos ejecutarlo. Si el sistema nos indica que el procedimiento se ha creado con errores de compilación podemos ver estos errores de compilación con la orden SHOW ERRORS en SQL *Plus.

Existen dos formas de pasar argumentos a un procedimiento almacenado a la hora de ejecutarlo (en realidad es válido para cualquier subprograma). Estas son:

* Notación posicional: Se pasan los valores de los parámetros en el mismo orden en que el procedure los define.

```
BEGIN
   Actualiza_Salario(7369,2500);
        COMMIT;
END;
```

* Notación nominal:Se pasan los valores en cualquier orden nombrando explicitamente el parámetro.

```
BEGIN
   Actualiza_Salario(p_empno => 7369,p_new_salario  => 2500);
   COMMIT;
END;
```

4.12.3 Paquetes

Un paquete es un conjunto de funciones y/o procedimiento. Permite facilitar la administración de los códigos (agrupaciones), y la seguridad (a nivel de paquete en vez de por función/procedimiento). En el paquete se pueden definir variable de alcance de todo el paquete (global variables).

Lo primero que debemos tener en cuenta es que los paquetes están formados por dos partes: la especificación y el cuerpo. La especificación del un paquete y su cuerpo se crean por separado.

La especificación es la interfaz con las aplicaciones. En ella es posible declarar los tipos, variables, constantes, excepciones, cursores y subprogramas disponibles para su uso posterior desde fuera del paquete.

En la especificación del paquete sólo se declaran los objetos (procedures, funciones, variables ...), no se implementa el código. Los objetos declarados en la especificación del paquete son accesibles desde fuera del paquete por otro script de PL/SQL o programa.

Para crear la especificación de un paquete la sintaxis general es la siguiente:

```
CREATE [OR REPLACE] PACKAGE <pkgName>
IS
  -- Declaraciones de tipos y registros públicas
  {[TYPE <TypeName> IS <Datatype>;]}
  -- Declaraciones de variables y constantes publicas
  -- También podemos declarar cursores
  {[<ConstantName> CONSTANT <Datatype> := <valor>;]}
  {[<VariableName> <Datatype>;]}
  -- Declaraciones de procedimientos y funciones públicas
  {[FUNCTION <FunctionName>(<Parameter> <Datatype>,...)
    RETURN <Datatype>;]}
  {[PROCEDURE <ProcedureName>(<Parameter> <Datatype>, ...);]}
END <pkgName>;
```

El cuerpo es la implementación del paquete. El cuerpo del paquete debe implementar lo que se declaró inicialmente en la especificación. Es el donde debemos escribir el código de los subprogramas.

En el cuerpo de un package podemos declarar nuevos subprogramas y tipos, pero estos serán privados para el propio package.

La sintaxis general para crear el cuerpo de un paquete es muy parecida a la de la especificación, tan solo se añade la palabra clave BODY, y se implementa el código de los subprogramas.

```
CREATE [OR REPLACE] PACKAGE BODY <pkgName>
IS
  -- Declaraciones de tipos y registros privados
  {[TYPE <TypeName> IS <Datatype>;]}
  -- Declaraciones de variables y constantes privadas
  -- También podemos declarar cursores
  {[<ConstantName> CONSTANT <Datatype> := <valor>;]}
  {[<VariableName> <Datatype>;]}

  -- Implementacion de procedimientos y funciones
  FUNCTION <FunctionName>(<Parameter> <Datatype>,...)
  RETURN <Datatype>
  IS
    -- Variables locales de la funcion
  BEGIN
    -- Implementeacion de la funcion
    return(<Result>);
  [EXCEPTION]
    -- Control de excepciones
  END;

  PROCEDURE <ProcedureName>(<Parameter> <Datatype>, ...)
  IS
   -- Variables locales de la funcion
  BEGIN
    -- Implementacion de procedimiento
  [EXCEPTION]
    -- Control de excepciones
  END;

END <pkgName>;
```

Es posible modificar el cuerpo de un paquete sin necesidad de alterar por ello la especificación del mismo.

Los paquetes pueden llegar a ser programas muy complejos y suelen almacenar gran parte de la lógica de negocio.

Ejemplo:

4.12.3.1 Paquetes de Oracle

Oracle incluye varios paquetes 'preparados'. Se instalan automáticamente cuando de instala una nueva base de datos con los script regulares. Algunos paquetes solo son disponibles mediante la ejecución de script adicionales o la instalación de herramientas adicionales.

Ejemplo:

```
dbms_output.put_line();
```

4.12.4 Disparadores

Los disparadores (triggers). Un disparador es un codigo que dispara cada vez que se ha modificado el dato de una tabla. Puede disparar a nivel de la consulta, o a nivel de cada línea afectada por la consulta. también puede disparar antes o después de la consulta, y solo por ciertos tipos de consulta (insert/update/delete), y eventualmente solo cuando cierto(s) campo(s) estan afectados(s).

Un trigger es un bloque PL/SQL asociado a una tabla, que se ejecuta como consecuencia de una determinada instrucción SQL (una operación DML: INSERT, UPDATE o DELETE) sobre dicha tabla.

La sintaxis para crear un trigger es la siguiente:

```
CREATE [OR REPLACE] TRIGGER <nombre_trigger>
{BEFORE|AFTER}
              {DELETE|INSERT|UPDATE [OF col1, col2, ..., colN]
          [OR {DELETE|INSERT|UPDATE [OF col1, col2, ..., colN]...]}
ON <nombre_tabla>
[FOR EACH ROW [WHEN (<condicion>)]]
DECLARE
  -- variables locales
BEGIN
  -- Sentencias
[EXCEPTION]
  -- Sentencias control de excepcion
END <nombre_trigger>;
```

El uso de OR REPLACE permite sobreescribir un trigger existente. Si se omite, y el trigger existe, se producirá, un error.

Los triggers pueden definirse para las operaciones INSERT, UPDATE o DELETE, y pueden ejecutarse antes o después de la operación. El modificador BEFORE AFTER indica que el trigger se ejecutará antes o despues de ejecutarse la sentencia SQL definida por DELETE INSERT UPDATE. Si incluimos el modificador OF el trigger solo se ejecutará cuando la sentencia SQL afecte a los campos incluidos en la lista.

El alcance de los disparadores puede ser la fila o de orden. El modificador FOR EACH ROW indica que el trigger se disparará cada vez que se realizan operaciones sobre una fila de la tabla. Si se acompaña del modificador WHEN, se establece una restricción; el trigger solo actuará, sobre las filas que satisfagan la restricción.

La siguiente tabla resume los contenidos anteriores.

INSERT, DELETE, UPDATE Define qué tipo de orden DML provoca la activación del disparador.
BEFORE , AFTER Define si el disparador se activa antes o después de que se ejecute la orden.
FOR EACH ROW Los disparadores con nivel de fila se activan una vez por cada fila afectada por la orden que provocó el disparo. Los disparadores con nivel de orden se activan sólo una vez, antes o después de la orden. Los disparadores con nivel de fila se identifican por la cláusula FOR EACH ROW en la definición del disparador.

La cláusula WHEN sólo es válida para los disparadores con nivel de fila.

Dentro del ambito de un trigger disponemos de las variables OLD y NEW . Estas variables se utilizan del mismo modo que cualquier otra variable PL/SQL, con la salvedad de que no es necesario declararlas, son de tipo %ROWTYPE y contienen una copia del registro antes (OLD) y después (NEW) de la acción SQL (INSERT, UPDATE, DELTE) que ha ejecutado el trigger. Utilizando esta variable podemos acceder a los datos que se están insertando, actualizando o borrando.

El siguiente ejemplo muestra un trigger que inserta un registro en la tabla EMP_AUDIT cada vez que modificamos el salario de un registro en la tabla emp:

```
Create table emp_audit (emp_no number not null, fecha date not null, msg varchar2(500) not null);

create or replace TRIGGER TR_EMP_AU
```



```
  AFTER UPDATE ON EMPLOYEE
  FOR EACH ROW
  WHEN (OLD.salary<>NEW.salary)
DECLARE
  -- local variables
BEGIN
  INSERT INTO  emp_audit
  (emp_no,fecha,msg)
  VALUES
  (:NEW.emp_no,SYSDATE,'Salario modificado de '||:old.salary||' a '||:new.salary);
END ;
```

El trigger se ejecutará cuando sobre la tabla EMP se ejecute una sentencia UPDATE que modifica el salario.
Ejemplo:

```
BEGIN
       Actualiza_Salario(p_empno => 7369,p_new_salario  => 2500);
       COMMIT;
END;
```

4.12.4.1 Orden de ejecución de los triggers

Una misma tabla puede tener varios triggers. En tal caso es necesario conocer el orden en el que se van a ejecutar.

Los disparadores se activan al ejecutarse la sentencia SQL.

* Si existe, se ejecuta el disparador de tipo BEFORE (disparador previo) con nivel de orden.
* Para cada fila a la que afecte la orden:
  - Se ejecuta si existe, el disparador de tipo BEFORE con nivel de fila.
  - Se ejecuta la propia orden.
  - Se ejecuta si existe, el disparador de tipo AFTER (disparador posterior) con nivel de fila.
* Se ejecuta, si existe, el disparador de tipo AFTER con nivel de orden.

4.12.4.2 Restricciones de los triggers

El cuerpo de un trigger es un bloque PL/SQL. Cualquier orden que sea legal en un bloque PL/SQL, es legal en el cuerpo de un disparador, con las siguientes restricciones:

* Un disparador no puede emitir ninguna orden de control de transacciones: COMMIT, ROLLBACK o SAVEPOINT. El disparador se activa como parte de la ejecución de la orden que provocó el disparo, y forma parte de la misma transacción que dicha orden. Cuando la orden que provoca el disparo es confirmada o cancelada, se confirma o cancela también el trabajo realizado por el disparador.
* Por razones idénticas, ningún procedimiento o función llamado por el disparador puede emitir órdenes de control de transacciones.
* El cuerpo del disparador no puede contener ninguna declaración de variables LONG o LONG RAW

4.12.4.3 Utilización de :OLD y :NEW

Dentro del ambito de un trigger disponemos de las variables OLD y NEW . Estas variables se utilizan del mismo modo que cualquier otra variable PL/SQL, con la salvedad de que no es necesario declararlas, son de tipo %ROWTYPE y contienen una copia del registro antes (OLD) y despues(NEW) de la acción SQL (INSERT, UPDATE, DELTE) que ha ejecutado el trigger. Utilizando esta variable podemos acceder a los datos que se están insertando, actualizando o borrando.

La siguiente tabla muestra los valores de OLD y NEW.

ACCION SQL	OLD	NEW
INSERT	No definido; todos los campos toman valor NULL.	Valores que serán insertados cuando se complete la orden.
UPDATE	Valores originales de la fila, antes de la actualización.	Nuevos valores que serán escritos cuando se complete la orden.
DELETE	Valores, antes del borrado de la fila.	No definidos; todos los campos toman el valor NULL.

Los registros OLD y NEW son sólo válidos dentro de los disparadores con nivel de fila.

Podemos usar OLD y NEW como cualquier otra variable PL/SQL.

Utilización de predicados de los triggers: INSERTING, UPDATING y DELETING

Dentro de un disparador en el que se disparan distintos tipos de órdenes DML (INSERT, UPDATE y DELETE), hay tres funciones booleanas que pueden emplearse para determinar de qué operación se trata. Estos predicados son INSERTING, UPDATING y DELETING.

Su comportamiento es el siguiente:
INSERTING TRUE si la orden de disparo es INSERT; FALSE en otro caso.
UPDATING TRUE si la orden de disparo es UPDATE; FALSE en otro caso.
DELETING TRUE si la orden de disparo es DELETE; FALSE en otro caso.

Sintaxis completa:

```
   CREATE [OR REPLACE] TRIGGER [schema.]trigger
      BEFORE event
           [WHEN (condition)]
             {pl_sql_block | call_procedure_statement}

   CREATE [OR REPLACE] TRIGGER [schema.]trigger
      AFTER event
           [WHEN (condition)]
             {pl_sql_block | call_procedure_statement}

   CREATE [OR REPLACE] TRIGGER [schema.]trigger
      INSTEAD OF event
           [WHEN (condition)]
             {pl_sql_block | call_procedure_statement}

event can be one or more of the following (separate multiple events with OR)

   DELETE event_ref referencing_clause
   INSERT event_ref referencing_clause
   UPDATE event_ref referencing_clause
   UPDATE OF column, column... event_ref
   db/ddl_event ON [schema.object]
   db/ddl_event ON DATABASE

event_ref:
   ON [schema.]table
   ON [schema.]view
   ON [NESTED TABLE nested_table_column OF] [schema.]view

referencing_clause:
   FOR EACH ROW
   REFERENCING OLD [AS] old [FOR EACH ROW]
   REFERENCING NEW [AS] new [FOR EACH ROW]
   REFERENCING PARENT [AS] parent [FOR EACH ROW]

db/ddl_event:
```

```
   ALTER
   ANALYSE
   ASSOCIATE STATISTICS
   AUDIT
   COMMENT
   CREATE
   DDL
   DISASSOCIATE STATISTICS
   DROP
   GRANT
   LOGON
   LOGOFF
   NOAUDIT
   RENAME
   REVOKE
   TRUNCATE
   SERVERERROR
   STARTUP
   SHUTDOWN
   SUSPEND

  Multiple db/ddl_events can be separated with OR

Multiple OLD, NEW and PARENT correlation names can be defined in one REFERENCING clause.

Database constraints are a factor of 8x faster than triggers.
```

5 Ejercicios

1. Crear una función que agrega 10 al valor pasado en parámetro y regresa el resultado
2. Crear un procedimiento que inserta un nuevo empleado
3. Crear un paquete con procedimiento para modificar un empleado, y una función para obtener la fecha de nacimiento del empleado
4. Usar los procedimientos/funciones en select
5. Crear un trigger que genera la clave primaria del empleado basado en una secuencia

6 Diseño de la base de datos