### UNIDAD DE TRABAJO 4: DEFINICIÓN de Datos

### *Conceptos*

### Introducción

### Objetos de la base de datos

### Tipos de datos

### La sentencia CREATE TABLE

### Referencias a tablas de otros usuarios

### La opción DEFAULT

### Tablas de una base de datos Oracle

### Consultas al Diccionario de Datos

### Tipos de datos

### Creación de una tabla por medio de una subconsulta

### La sentencia ALTER TABLE

### Añadir una columna

### Modificar una columna

### Eliminación de una columna

### Opción SET UNUSED

### La sentencia DROP TABLE

### Cambiar el nombre a una tabla

### Truncar una tabla

### Añadir comentarios a una tabla

### ¿Qué son las restricciones?

### Definición de restricciones

### La restricción NOT NULL

### La restricción UNIQUE Key

### La restricción PRIMARY KEY

### La restricción FOREIGN KEY

### La restricción CHECK

### Operaciones con restricciones

### Añadir una restricción

### Eliminar una restricción

### Desactivar restricciones

### Activar restricciones

### Consulta de restricciones en el Diccionario de Datos

### Vistas

### *Introducción*

### El lenguaje SQL (*Structured Query Language*) permite la comunicación con el SGBD. Fue desarrollado sobre un prototipo de SGBD relacional denominado SYSTEM R y diseñado por IBM a mediados de los años setenta. En 1979 Oracle Corporation presentó la primera implementación comercial de SQL, que estuvo disponible antes que otros productos de IBM. Por su parte, IBM desarrolló productos herederos del prototipo SYSTEM R, como DB2 y SQL/DS.

### El instituto ANSI (*American National Standard Institute*) adoptó el lenguaje SQL como estándar para la gestión de bases de datos relacionales en octubre de 1986. En 1987 lo adopta ISO (*International Standardization Organization*).

### SQL es tan conocido en bases de datos que muchos lenguajes de programación incorporan sentencias SQL como parte de su repertorio; tal es el caso de Visual Basic para acceder a bases de datos relacionales.

### Entre las principales características de SQL podemos destacar:

### Es un lenguaje para todo tipo de usuarios (administradores, desarrolladores y usuarios normales), el usuario que emplea SQL especifica qué quiere, no cómo ni dónde, permite hacer cualquier consulta de datos.

### Es un lenguaje no procedimental. Permite al usuario solicitar un resultado sin preocuparse de los medios necesarios para obtener dicho resultado.

### Permite manipular un registro o un conjunto de registros , por tanto, no son necesarias estructuras de control.

### Universal, puede utilizarse en todos los niveles de la gestión de una bbdd, administración de la bbdd, desarrollo, gestion de datos,etc

### Podemos clasificar las sentencias SQL en 2 tipos:

### Sentencias DDL (*Data Definition Language*). Con este lenguaje se crea y mantiene la estructura de la bbdd. Sirve para realizar las siguientes tareas:

### Crear un objeto de la bbdd, tablas, vistas, procedimientos...(Create)

### Eliminar un objeto de la bbdd (Drop)

### Modificar un objeto de la bbdd (Alter)

### Conceder privilegios sobre un objeto de la bbdd (Grant)

### Retirar privilegios sobre un objeto de la bbdd. (revoke)

### Sentencias DML (Data Manipulation Language) Conjunto de sentencias sirve para manipular los datos contenidos en la bbdd. Es posible hacer:

### Insert. Insertar filas en la bbdd

### Update. Actualizar filas de datos

### Delete. Eliminar filas de datos.

### Select. Recuperar filas de datos.

### Bloqueo de tablas

### Existen otro grupo de sentencias SQL no tan extendidas como las anteriores y son:

### Lenguaje de control de transacción, gestiona las modificaciones realizadas por las instrucciones DML:

### Características de la transacción

### Validación y anulación de la modificación

### COMMIT, SAVEPOINT, ROLLBACK, SET TRANSACTION

### En la presente unidad se va a contemplar el sublenguaje DDL (*Data Definition Language*) de SQL. Se mostrará cómo crear los objetos principales del modelo relacional: las tablas, así como las restricciones de los datos. Se cierra la línea que iniciamos en la unidad anterior iniciada en el diseño conceptual, seguida del diseño lógico, terminando en el diseño físico con la creación de las tablas y las restricciones en el SGBDR Oracle.

### Por último, indicar que las sentencias DDL tienen incluido el comando COMMIT, que finaliza y lleva a cabo todas las transacciones pendientes hasta el lanzamiento de una sentencia DDL.

### *Objetos de la base de datos*

### Una base de datos Oracle puede contener múltiples estructuras de datos. Cada estructura debería definirse en el diseño de la base de datos, para que pueda ser creada durante la etapa de construcción del desarrollo de la base de datos.

### Principales objetos:

### Tabla 🡪 Unidad básica de almacenamiento, compuesta de registros y columnas.

### Vista 🡪 representación lógica de un subconjunto de una o mas tablas. Puede manipularse como una tabla (tabla virtual)

### Secuencia 🡪 Genera valores para la clave primaria. (Similar a campo autonumérico de Access)

### Índice 🡪 Columna o conjunto de columnas que permiten realizar búsquedas más rápidas. Sirven para mejorar el rendimiento de algunas consultas, en las que se incluya.

### Sinónimo 🡪 Da nombres alternativos a los objetos de almacenamiento. Nombres alternativos a tablas o vistas.

### Notas sobre tablas en Oracle:

### Pueden ser creadas en cualquier momento, incluso mientras los usuarios usan la BD.

### No se necesita especificar el tamaño de ninguna tabla. El tamaño es definido por la cantidad de espacio en la BD según va creciendo. Es importante, sin embargo, estimar cuánto espacio utilizará la tabla.

### La estructura de una tabla puede ser modificada on-line.

### Reglas para los nombres de tablas columnas y demás objetos de la BD Oracle:

### Deben comenzar por una letra.

### Pueden tener una longitud de 1-30 caracteres de largo.

### Caracteres permitidos: A-Z, a-z, 0-9, \_, $, # (No valen los espacios)

### No deben duplicar el nombre de otro objeto que sea propiedad del mismo usuario.

### No debe ser una palabra reservada del servidor Oracle.

### Nota: los nombres no son sensibles a mayúsculas/minúsculas. Es lo mismo DEPT que dept que Dept.

### *Tipos de datos*

|  |  |
| --- | --- |
| CHAR(tamaño) | cadena de caracteres de longitud fija con longitud “tamaño”. Por defecto vale 1.1 >= tamaño <= 2000Si se introduce una cadena de menor longitud a la establecida lo rellena con espacios en blanco en la derecha. |
| VARCHAR2(tamaño) | cadena de caracteres de longitud variable, máximo el “tamaño”.1 >= tamaño <= 4000 Apellido Varchar2(20)Si se introduce una cadena superior al tamaño definido Oracle dará un error |
| NUMBER(P,S)NUMBER (n)NUMBER | Representa datos numericos tanto enteros como reales y con signo.Cuando especifico p,s es porque puedo poner máximo número de dígitos p (entre 1 y 38)S representa el número de dígitos de la parte fraccionaria (hasta 127)Ejemplo: Sal\_Medio NUMBER(9,2) podría guardar 4566770.86Cuando pongo n, es por que trabajo con números enteros y en decimal supone 0Salario Number(10)Cuando solo pongo NUMBER se guarda en decimal o entero según se tecleeEl carácter por defecto para separar la parte fija y la fraccionaria es el punto. Se puede modificar en NLS\_NUMERIC\_CHARACTERS(ALTER SESSION) |
| DATE | Fecha y horas. Para cada tipo Date se almacena Siglo/Año/Mes/día/Hora/Minutos/segundosEl formato de la fecha está en NLS\_DATE\_FORMAT . Por defecto es dd//mm/yyEjemplo: fecha\_edición date |
| TIMESTAMP(P) | Datos tipo Date en los que podemos indicar la precisión para las fracciones de segundo. Por defecto es 6 |
| BLOB | datos binarios, hasta 4 GBytes |
| BFILE | Datos binarios almacenados en archivos externos a la bbdd (4 GB máximo) |
| LONG | Cadenas de caracteres de longitud variable (2 GB máximo) Para almacenar textos grandes.Solo se puede definir una columna por tablaNo pueden aparecer en restricciones de integridad (constraints) ni para indexar, ni en where,etc |
| RAW(N) | Datos binarios de longitud variable no interpretable por Oracle. La diferencia con los varchar2 es que maneja cadenas de caracteres en lugar de cadenas de n bytes (n <2000 bytes) |
| LONG RAW(N) | Datos binarios de longitud variable no interpretable por Oracle Se emplea para almacenamiento de graficos. Utiliza (4 Gb como máximo) |

### *La sentencia CREATE TABLE*

### Esta sentencia forma parte del sublenguaje DDL de SQL y es utilizada para crear tablas. Tiene un efecto inmediato sobre la base de datos y graba información en el Diccionario de Datos.

### Para crear una tabla, el usuario debe tener el privilegio CREATE TABLE y un área de almacenamiento para crear objetos

### Es posible definir hasta 1000 columnas por tabla

### Sintaxis:

### CREATE [GLOBAL TEMPORARY] TABLE [schema.]nombretabla

### (columna tipo\_datos [DEFAULT expr] [NOT NULL] ) [TABLESPACE espacio\_de\_tabla] ;

### GLOBAL TEMPORARY: especifica que la tabla es temporal y que su definición es visible para todas las sesiones. Los datos en una tabla temporal son visibles sólo para la sesión que inserta datos en la tabla.

### Schema. : propietario

### DEFAULT expr: específica un valor por defecto si se omite en la sentencia insert.

### NOT NULL la columna debe contener alguna información. Nunca puede ser nula cuando insertemos una nueva fila

### Se puede indicar el TABLESPACE donde se debe almacenar la tabla. Este debe existir. Si se omite se inserta en el tablespace que tenga asignado el usuario.

### El usuario debe especificar el nombre de la tabla y los nombres y tipos de las columnas, siguiendo las reglas para la definición de nombres.

### Ejercicio: Crear la tabla alumnos con num\_mat, nombre, fecha\_nac, dirección, localidad en el tablespace USER\_DATA

### CREATE TABLE ALUMNOS(

### NUM\_MAT NUMBER(6) NOT NULL,

### NOMBRE VARCHAR2(15) NOT NULL,

### FECHA\_NAC DATE,

### DIRECCION VARCHAR2(30),

### LOCALIDAD VRCHAR2(15)

### ) TABLESPACE USER\_DATA;

### *Referencias a tablas de otros usuarios:*

### Un esquema (*schema*) es una colección de objetos. Los objetos de un esquema son las estructuras lógicas que hacen referencia directa a los datos en una base de datos. Incluyen tablas, vistas, sinónimos, secuencias, procedimientos almacenados, índices, clusters y database links.

### Las tablas propiedad de otros usuarios no pertenecen a nuestro esquema. Si deseamos hacer referencia a ellas, debemos anteponer el nombre del usuario propietario al de la tabla separando ambos vocablos con un punto.

### Podremos acceder a esas tablas si el propietario lo permite con los permisos oportunos.

### *La opción DEFAULT:*

### Usando la opción DEFAULT se puede dar un valor por defecto a una columna. Esta opción previene la introducción de valores nulos en aquellas filas que no especifican un valor para esa columna. El valor por defecto puede ser un literal, una expresión o una función SQL, pero el valor no puede ser el nombre de otra columna. La expresión por defecto debe ser del mismo tipo que el de la columna.

### Ejemplo:

### CREATE TABLE dept2

### (numdept NUMBER(2),

### nombre dept VARCHAR2(14),

### ubicacion VARCHAR2(13));

### *Tablas de una base de datos Oracle*

### Las tablas de usuario son tablas creadas por el usuario.

### Hay otro conjunto de tablas y vistas en una base de datos Oracle conocido como *Diccionario de Datos* (DD). Este conjunto de tablas es creado y mantenido por Oracle y contiene información sobre la base de datos.

### Todas las tablas del DD son propiedad del usuario SYS.

### Los usuarios raramente acceden a las tablas base porque la información que hay en ellas no es fácil de entender.

### No se puede escribir directamente sobre él.

### Los usuarios normalmente acceden a vistas del diccionario de datos porque aquí la información se presenta en un formato más fácil de entender.

### La información que se almacena en el DD incluye nombres de los usuarios de Oracle Server, privilegios concedidos a los usuarios, nombres de los objetos de la BD, restricciones de integridad e información de auditoría.

### *Consultas al Diccionario de Datos:*

### Se consultan las tablas del DD para localizar información acerca de los objetos que contiene. Algunas de las tablas del diccionario utilizadas con mayor frecuencia son:

### USER\_TABLES

### USER\_OBJECTS

### USER\_CATALOG: Tablas, vistas, sinónimos, y secuencias propiedad del usuario.

### Ejemplos:

### Tablas propiedad del usuario:

### select table\_name from user\_tables;

### En este caso me visualiza los nombres de las tablas de usuario, es decir, mis tablas

### select \* from user\_tables;

### Visualiza toda la información posible de todas mis tablas

### USER\_TABLES es una vista cuyo propietario es SYS, por eso ponemos:

### select table\_name from sys.user\_tables;

### Visualiza el nombre de las tablas creadas igual que anteriormente.

### También podemos poner:

### Desc SYS.USER\_TABLES; Me visualiza todos los campos de la vista USER\_TABLES

### Distintos objetos propiedad del usuario:

### select distinct object\_type from user\_objects;

### En el caso de scott me dice que los únicos objetos que tiene son vistas y tablas

### select object\_type, object\_name from user\_objects;

### Nos dice de cada objeto su nombre y el tipo que es.

### select object\_type, substr (object\_name, 1, 10) from user\_objects;

### Como los nombres salían en un formato poco legible cogemos del nombre solo los 10 primeros caracteres.

### Tablas, vistas, sinónimos y secuencias propiedad del usuario:

### select \* from user\_catalog;

### *Creación de una tabla por medio de una subconsulta*

### Un segundo método para crear una tabla, es aplicar una cláusula AS como subconsulta, para crear tanto la tabla, como para llenarla con los registros devueltos por la subconsulta.

### Sintaxis:

### CREATE TABLE [columna (,columna…)] AS subconsulta;

### Ejemplo:

### CREATE TABLE dept50 AS select \* from dept where deptno=50;

### La tabla se creará con los nombres de columnas especificadas y los registros recuperados por la sentencia SELECT serán insertados en la tabla. Si se dieran nombres de columnas, su número será igual al número de columnas especificadas en la sentencia SELECT de la subconsulta.

### Cuando creo una tabla así no copia las restricciones de la tabla original.

### *La sentencia ALTER TABLE*

### Después de haber creado una tabla, puede necesitar cambiar su estructura, porque tal vez omitió una columna, o la definición ha cambiado. Esto puede hacerse utilizando la sentencia ALTER TABLE.

### Se utiliza para:

### Añadir una nueva columna (o atributo)

### Modificar una columna existente

### Dar un valor por defecto a una nueva columna

### *Añadir una columna:*

### Sintaxis:

### ALTER TABLE tabla

### ADD (columna tipo\_datos [DEFAULT expr]

### [, columna tipo\_datos]…);

### Ejemplo:

### ALTER TABLE dept30

### ADD (puesto VARCHAR2(9));

### La columna añadida aparecerá la última de la tabla.

### *Modificar una columna:*

### Se utiliza para cambiar el tipo de datos de una columna, su tamaño o su valor por defecto. En este último caso, el valor por defecto se aplicará sólo a posteriores inserciones en la tabla.

### Sintaxis:

### ALTER TABLE tabla

### MODIFY (columna tipo\_datos [DEFAULT expr]

### [, columna tipo\_datos]…);

### Ejemplo:

### ALTER TABLE Dept

### MODIFY (Provincia VARCHAR2(15));

### *Eliminación de una columna:*

### Esta sentencia se utiliza para borrar una columna de una tabla cada vez. La columna podrá o no contener datos no pudiéndose recuperar una vez se haya eliminado (no se puede recuperar con ROLLBACK puesto que lleva COMMIT implícito). En la tabla debe quedar como mínimo una columna después de ALTER.

### Sintaxis:

### ALTER TABLE tabla DROP COLUMN columna;

### Ejemplo:

### ALTER TABLE Dept DROP COLUMN Provincia;

### *Renombrar una columna:*

### ALTER TABLE nombre\_tabla

### RENAME COLUMN old\_name to new\_name;

### *Opción SET UNUSED ( Se utiliza conALTER):*

### La opción SET UNUSED marca una o más columnas como “no usadas” para que puedan ser eliminadas cuando se necesiten recursos en el sistema. Esta es una prestación propia de Oracle. Especificando esta cláusula no se eliminan realmente las columnas de cada fila de la tabla, es decir, no restaura el espacio usado por estas columnas. Sin embargo, el tiempo de respuesta es mejor que si se ejecuta la cláusula de borrado DROP.

### Las columnas “unused” son tratadas como si hubieran sido borradas, aunque los datos permanecen en la tabla. Después de marcar una columna como “unused”, no tenemos acceso a ella. Una sentencia de consulta no recuperará los datos de estas columnas. Además, los nombres y tipos de datos de estas columnas no se mostrarán con DESCRIBE y podemos añadir a la tabla una nueva columna con el mismo nombre que otra columna “unused”.

### Sintaxis:

### ALTER TABLE tabla SET UNUSED (columna);

### Ejemplo:

### ALTER TABLE Dept30 SET UNUSED (ename);

### En un segundo paso, eliminaremos las columnas “unused” para conseguir el espacio en disco que ocupan. Si la tabla no contiene este tipo de columnas, la sentencia no devuelve error.

### Sintaxis:

### ALTER TABLE tabla DROP UNUSED COLUMNS;

### *La sentencia DROP TABLE*

### Sirve para eliminar las tablas (y también sus datos). Se borra la definición de la tabla en el DD y todos los índices asociados. Como toda sentencia DDL no se puede hacer ROLLBACK de la sentencia.

### Solamente el propietario de la tabla u otro usuario con el permiso DROP ANY TABLE puede eliminar una tabla.

### Sintaxis:

### DROP TABLE tabla;

### *Cambiar el nombre de un objeto*

### Para cambiar el nombre de una tabla, vista, secuencia o sinónimo, se utiliza la instrucción RENAME. Ha de ser el propietario del objeto que renombra. Es una instrucción propia de Oracle.

### Sintaxis:

### RENAME nombre1 TO nombre2;

### Ejemplo:

### RENAME dept TO departamento;

### *Truncar una tabla*

### Es propia de Oracle y se utiliza para borrar todos los registros de una tabla y liberar así todo el espacio de almacenamiento ocupado por la tabla. No se puede hacer ROLLBACK. Es una alternativa a la sentencia DELETE de DML pero ésta si puede efectuar ROLLBACK. DELETE no libera espacio.

### El usuario que trunque una tabla debe ser el propietario de la misma o tener el privilegio del sistema DELETE TABLE.

### Sintaxis:

### TRUNCATE TABLE tabla;

### *Añadir comentarios a una tabla*

### Es propio de Oracle y no de SQL. Sirve para añadir comentarios de hasta 2000 bytes sobre una columna, tabla o vista. El comentario se almacena en el Diccionario de Datos y puede ser visualizado por medio de la columna COMMENTS de una de las siguientes vistas del DD:

### Para columnas:

### ALL\_COL\_COMMENTS

### USER\_COL\_COMMENTS

### Para tablas:

### ALL\_TAB\_COMMENTS

### USER\_TAB\_COMMENTS

### Sintaxis:

### COMMENT ON TABLE tabla IS ‘comentario descriptivo’;

### COMMENT ON COLUMN tabla.col IS ‘comentario descriptivo’;

### *¿Qué son las restricciones?*

### Cuando almacenamos datos en nuestras tablas, se ajustan a una serie de restricciones:

### Integridad de datos

### Que una columna no puede almacenar valores negativos

### Que se almacenen las cadenas en mayúsculas

### Que una columna no permita 0

### La integridad hace referencia al hecho de que los datos cumplan ciertas restricciones.

### Integridad: Regla que restringe el rango de valores de una o más columnas.

### Integridad referencial

### Garantiza que los valores de una columna de una tabla dependan de los valores de otra columna de otra tabla.

### El servidor Oracle utiliza las restricciones (*constraints*) para prevenir la introducción de datos no validos en una tabla.

### Se pueden usar para

### Garantizar el cumplimiento de reglas a nivel de tablas, en el momento que una fila se inserta, se actualiza o se borra. La restricción debe ser cumplida para que la operación tenga éxito.

### Impedir la eliminación de una tabla si existen dependencias desde otras tablas.

### RESTRICCIONES DE INTEGRIDAD EN ORACLE

|  |  |
| --- | --- |
| *CONSTRAINT* | *DESCRIPCIÓN* |
| NOT NULL | Especifica que una columna no tenga valores nulos. |
| UNIQUE | Especifica que una columna o combinación de ellas, tenga valores irrepetibles. |
| PRIMARY KEY | Especifica la clave principal. |
| FOREIGN KEY | Especifica una clave foránea. |
| CHECK | Especifica que una condición debe ser verdadera. |

### Se recomienda asignarles un nombre descriptivo a las restricciones. De lo contrario, Oracle le dará uno del tipo SYS\_CX donde X es un número entero.

### Se pueden crear en:

### En el momento de crear la tabla.

### Después de que la tabla haya sido creada.

### Las restricciones se pueden definir tanto a nivel de columna como a nivel de tabla. Se pueden ver las restricciones en el Diccionario de Datos, en la tabla USER\_CONSTRAINTS.

### *Definición de restricciones*

### Las restricciones se crean normalmente a la vez que se crea la tabla pero pueden ser añadidas después de haber creado la tabla y también pueden desactivarse temporalmente.

### Sintaxis:

### CREATE TABLE tabla

### (columna1 tipo\_datos [CONSTRAINT\_NOMBRERESTRICCION][NOT NULL] [CONSTRAINT\_NOMBRERESTRICCION][UNIQUE] [CONSTRAINT\_NOMBRERESTRICCION][PRIMARY KEY]

### [CONSTRAINT\_NOMBRERESTRICCION][DEFAULT VALOR][REFERENCES NOMBRE TABLA][CHECK CONDICION]

### columna2 tipo\_datos [CONSTRAINT\_NOMBRERESTRICCION][NOT NULL][UNIQUE][PRIMARY KEY][DEFAULT VALOR][REFERENCES NOMBRE TABLA][CHECK CONDICION]

### …

### [constraint\_tabla]);

### Ejemplo (restricción de columna sin nombre explícito):

### CREATE TABLE empleados

### (empno NUMBER (4),

### ename VARCHAR2 (10),

### *deptno NUMBER (2) NOT NULL,*

### CONSTRAINT empleados\_empno\_pk PRIMARY KEY (empno));

### Ejemplo (restricción de columna con nombre explícito):

### CREATE TABLE empleados

### (empno NUMBER (4),

### ename VARCHAR2 (10),

### *deptno NUMBER (2) CONSTRAINT empleados\_deptno\_nn NOT NULL,*

### CONSTRAINT empleados\_empno\_pk PRIMARY KEY (empno));

### *La restricción NOT NULL:*

### La restricción NOT NULL asegura que en la columna no se permitirán valores nulos. Las columnas sin la restricción NOT NULL pueden contener valores nulos, por defecto.

### Esta restricción se aplica solamente a nivel de columna, no a nivel de tabla.

### *La restricción UNIQUE Key:*

### Esta restricción requiere que cada valor en una columna o conjunto de columnas sean únicas, es decir, dos registros de una tabla no tendrán valores duplicados para determinada columna o conjunto de columnas.

### Permite la entrada de nulos, a menos que se especifique también la restricción NOT NULL para las mismas columnas.

### Una columna o conjunto de columnas con restricciones UNIQUE y NOT NULL definen una clave alternativa en una tabla.

### Sintaxis:

### … CONSTRAINT nombre\_uk UNIQUE (columna [,columna]…)….

### Ejemplo:

### CREATE TABLE dept

### (deptno NUMBRE(2),

### dname VARCHAR2(14),

### loc VARCHAR2(13),

### CONSTRAINT dept\_dname\_uq UNIQUE (dname));

### *La restricción PRIMARY KEY:*

### Crea una clave principal en una tabla. Se puede crear solamente una Primay Key por tabla. Esta restricción consiste en una columna o conjunto de columnas que identifican unívocamente a cada fila de una tabla. Garantiza la unicidad de la columna (o combinación de ellas) y asegura que ninguna columna que sea parte de la Primary Key pueda contener un valor nulo.

### Puede definirse a nivel de columna o de tabla. Oracle crea automáticamente un índice para la columna (o columnas) de la clave principal.

### A nivel de tabla la sintaxis será:

### … CONSTRAINT nombre\_pk PRIMARY KEY (columna [,columna]…)….

### A nivel de columna la sintaxis será:

### CREATE TABLE nombredetabla

### (col1 tipocol [CONSTRAINT nombre] PRIMARY KEY;

### Ejemplo:

### CREATE TABLE dept

### (deptno NUMBER(2),

### dname VARCHAR2(14),

### loc VARCHAR2(13),

### CONSTRAINT dept\_dname\_uq UNIQUE (dname),

### CONSTRAINT dept\_deptno\_pk PRIMARY KEY (deptno));

### Cuando una tabla está formada por una clave principal de varias columnas, no podremos utilizar la sintaxis de columnas, tendremos que definirla a nivel de tabla.

### *La restricción FOREIGN KEY:*

### La restricción FOREIGN KEY, o de integridad referencial, designa a una columna o combinación de ellas como una clave extranjera y establece una relación con una Primary Key de la misma tabla o de otra. Se pueden definir a nivel de tabla o columna.

### El valor de la Foreign Key debe coincidir con uno de la tabla padre o ser NULL.

### Las Foreign Key están basadas en valores de datos y son puramente lógicos, no son punteros físicos.

### Podemos definir tantas claves ajenas como sea necesario.

### Sintaxis a nivel de tabla:

### … CONSTRAINT nombre\_fk FOREIGN KEY (columna [,columna]…)

### REFERENCES tabla (columna [,columna]…) ….

### Sintaxis a nivel de columna:

### Columna tipo\_columna [CONSTRAINT nombrerestriccion] REFERENCES nombretabla[(columna)][ ON DELETE CASCADE];

### Ejemplo (a nivel de tabla):

### CREATE TABLE empleados

### (empno NUMBER(4),

### ename VARCHAR2(10) NOT NULL,

### job VARCHAR2 (9),

### deptno NUMBER (3) NOT NULL,

### CONSTRAINT emp\_deptno\_fk FOREIGN KEY (deptno) REFERENCES dept (deptno));

### Ejemplo (a nivel de columna):

### CREATE TABLE empleados

### (empno NUMBER(4),

### ename VARCHAR2(10) NOT NULL,

### job VARCHAR2 (9),

### deptno NUMBER (3) CONSTRAINT emp\_deptno\_fk REFERENCES dept (deptno));

### La palabra reservada FOREIGN KEY no es necesaria en este último caso.

### A esta restricción se le puede añadir la opción ON DELETE CASCADE. Indica que cuando la fila es borrada en la tabla padre, se borrarán todas las filas dependientes en la tabla hija.

### Sin la opción ON DELETE CASCADE, la fila en la tabla padre no puede ser borrada mientras haya referencias a ella en la tabla hija. (EJEMPLO DE DEPT EN ORACLE)

### Sintaxis:

### … CONSTRAINT nombre\_fk FOREIGN KEY (columna [,columna]…)

### REFERENCES tabla (columna [,columna]…) ON DELETE CASCADE …

### Ejemplo:

### CONSTRAINT emp\_deptno\_fk FOREIGN KEY (deptno) REFERENCES dept (deptno) ON DELETE CASCADE);

### EJEMPLO:

### Crear la tabla PERSONAS con la siguiente información:

|  |  |  |
| --- | --- | --- |
| NOMBRE COLUMNA | TIPO | RESTRICCION |
| DNI | NUMBER(8) | PRIMARIA |
| NOMBRE | VARCHAR2(15) |  |
| DIRECCIÓN | VARCHAR2(15) |  |
| POBLACIÓN | VARCHAR2(15) |  |
| COD\_PROV | NUMBER(2) | AJENA |

### Crear la tabla PROVINCIAS con la siguiente información:

|  |  |  |
| --- | --- | --- |
| NOMBRE COLUMNA | TIPO | RESTRICCION |
| COD\_PROV | NUMBER(2) | PRIMARIA |
| NOMBRE | VARCHAR2(15) |  |

### CREATE TABLE PERSONAS

### (....

### FOREIGN KEY (COD\_PROV) REFERENCES PROVINCIAS

### )

### Si creamos primero la tabla PERSONAS al meter esta sentencia nos dirá que no existe la tabla PROVINCIAS, por este motivo tendremos que crear antes la tabla PROVINCIAS.

### Igualmente si queremos borrar las tablas, habrá que borrar antes PERSONAS que PROVINCIAS , pues de lo contrario Oracle dá error al borrar PROVINVIAS pues te dice que tiene claves primarias que son referenciadas en otra tabla.

### Si quisiéramos borrar alguna provincia en concreto, y automáticamente borrasemos las personas asociadas, pondríamos ON DELETE CASCADE.

### FOREIGN KEY (COD\_PROV) REFERENCES PROVINCIAS ON DELETE CASCADE.

### RESUMEN DE RESTRICCIONES:

### RESTRICCIONES DE BORRADO: Una fila no se puede borrar si es referenciada por un clave ajena. Tampoco se puede actualizar la clave primaria .

### RESTRICCIONES EN CASCADA: Si borramos una fila de la tabla maestra (PROVINCIAS), todas las filas de la tabla detalle (PERSONAS) que sean referenciadas se borraran automáticamente. Cuando salga el mensaje de n rows deleted solo se refiere a las de la tabla maestra.

### 

### *La restricción CHECK:*

### Define una condición que cada fila debe satisfacer. La condición puede usar la misma sintaxis que las condiciones de las consultas ( ej. WHERE color = red)

### Una misma columna puede tener más de una restricción CHECK. Se pueden definir a nivel de tabla o de columna.

### Sintaxis:

### … CONSTRAINT nombre\_ck CHECK (condición) …

### Ejemplo a nivel de tabla:

### CREATE TABLE emp

### (deptno NUMBER (2),

### CONSTRAINT emp\_deptno\_ck CHECK (deptno BETWEEN 10 AND 99));

### Ejemplo:

### Crear la tabla BLOQUEDEPISOS

### *Operaciones con restricciones*

### Las operaciones que pueden realizarse con las restricciones son:

### Añadir

### Eliminar

### Activar

### Desactivar

### Visualización en el Diccionario de Datos.

### *Añadir una restricción:*

### Se pueden añadir restricciones para tablas ya existentes, usando la sentencia ALTER TABLE con la cláusula ADD.

### Sintaxis:

### ALTER TABLE tabla

### ADD [CONSTRAINT restricción] tipo\_restricción (columna)

### En el caso de querer añadir una restricción del tipo NOT NULL a una columna existente, debe utilizarse la sentencia ALTER TABLE con la cláusula MODIFY. Esto sólo puede hacerse si la tabla no contiene filas.

### Ejemplo:

### ALTER TABLE empleados

### ADD CONSTRAINT empleados\_mgr\_fk FOREIGN KEY (mgr) REFERENCES empleados(empno);

### *Eliminar una restricción:*

### Se utiliza para borrar una restricción creada previamente. Cuando se borra una restricción de integridad, esa restricción ya no es parte del servidor Oracle y por tanto, no está disponible en el Diccionario de Datos.

### Sintaxis

### ALTER TABLE tabla DROP CONSTRAINT nombre\_constraint;

### Ejemplos:

### ALTER TABLE emp DROP CONSTRAINT emp\_mgr\_fk;

### ALTER TABLE dept DROP PRIMARY KEY CASCADE;

### En el último ejemplo, se borra la restricción de Primary Key de la tabla dept y también la restricción de Foreign Key asociada en la columna emp.deptno.

### *Desactivar restricciones:*

### Puede desactivarse una restricción sin borrarla. Se utiliza la sentencia ALTER TABLE con la cláusula DISABLE. Con la opción CASCADE se desactivan las restricciones de integridad dependientes.

### Puede utilizarse la cláusula DISABLE también en la sentencia CREATE TABLE

### Sintaxis:

### ALTER TABLE tabla

### DISABLE CONSTRAINT nombre\_restricción [CASCADE];

### Ejemplo:

### ALTER TABLE empleados

### DISABLE CONSTRAINT empleados\_empno\_pk CASCADE;

### *Activar restricciones:*

### Permite activar restricciones de integridad actualmente desactivadas en la definición de la tabla por medio de la cláusula ENABLE.

### Si se activa una restricción UNIQUE o PRIMARY KEY se creará automáticamente un índice para la clave UNIQUE o PRIMARY KEY.

### Puede utilizarse la cláusula ENABLE también en la sentencia CREATE TABLE

### Sintaxis:

### ALTER TABLE tabla

### ENABLE CONSTRAINT nombre\_restricción;

### Ejemplo:

### ALTER TABLE emp ENABLE CONSTRAINT emp\_empno\_pk;

### *Consulta de restricciones en el Diccionario de Datos*

### Tras crear una tabla, puede verificarse su existencia haciendo uso del comando DESCRIBE. La única restricción que puede verificar es NOT NULL. Para ver todas las restricciones sobre una tabla, hay que consultar la vista USER\_CONSTRAINTS del Diccionario de Datos.

### En la vista USER\_CONSTRAINTS se ven todos los nombres y definiciones de restricciones.

### En la vista USER\_CONS\_COLUMNS se ven todas las columnas asociadas con los nombres de las restricciones. Esta vista es especialmente útil para restricciones a las que el propio sistema ha asignado un nombre.

### Ejemplos:

### SELECT constraint\_name, constraint\_type, search\_condition

### FROM USER\_CONSTRAINTS

### WHERE table\_name = ‘EMP’;

### SELECT constraint\_name, column\_name

### FROM USER\_CONS\_COLUMNS

### WHERE table\_name = ‘EMP’;

### *Vistas*

### Una vista representa lógicamente un subconjunto de una o más tablas (llamadas tablas base). Una vista no contiene datos en sí misma pero es como una ventana a través de la cual se pueden ver o cambiar los datos de las tablas.

### La vista se almacena como una sentencia SELECT en el diccionario de datos.

### Las vistas se utilizan para:

### Restringir el acceso a los datos

### Realizar consultas complejas fácilmente. Por ejemplo, para que los usuarios consulten datos de varias tablas sin necesidad de conocer cómo escribir sentencias de unión o JOIN.

### Independencia de los datos para usuarios y aplicaciones: una vista puede ser usada para recuperar datos de diversas tablas

### Presentar diferentes vistas de los mismos datos, ofreciendo acceso a los datos a grupos de usuarios de acuerdo a su criterio particular

### Tipos de vistas:

### Simples: utilizan una sola tabla, no contienen funciones ni grupos de datos y pueden hacer DML a través de la vista.

### Complejas: utilizan una o más tablas, contienen funciones o grupos de datos y no siempre pueden realizar DML a través de la vista.

### *Sentencia CREATE VIEW*

### Podemos crear una vista embebiendo una subconsulta dentro de la sentencia CREATE VIEW.

### Es una sentencia perteneciente al sublenguaje DDL (Data Definition Language)

### Sintaxis:

### CREATE [OR REPLACE] [FORCE | NOFORCE] VIEW nombre\_vista

### [(columna[,columna]...)]

### AS subconsulta

### [WITH CHECK OPTION [CONSTRAINT restricción]]

### [WITH READ ONLY];

### OR REPLACE: recrea la vista si ya existe

### FORCE: crea la vista sin importar que la tabla base exista o no

### NOFORCE: crea la vista únicamente si la tabla base existe. Es la opción por defecto.

### alias: especifica nombres para las expresiones seleccionadas en la consulta de la vista. El número de alias de las columnas debe coincidir con el de expresiones seleccionadas por la subselect. Cuando no se ponen nombres de columnas

### subconsulta: es una sentencia SELECT completa. Se pueden usar los alias para las columnas de la lista del SELECT. No puede contener la cláusula ORDER BY.

### WITH CHECK OPTION: especifica que solamente las filas accesibles a la vista pueden ser insertadas o actualizadas.

### restricción: es el nombre asignado a la restricción CHECK OPTION

### WITH READ ONLY: asegura que ninguna operación DML pueda realizarse sobre esta vista.

### Si se suprime una tabla la vista asociada es invalida

### *Creación de vistas sencillas*

### Las vistas mas sencillas son las que unicamente tocan una tabla

### Ejemplos: Vista de los ermpleados del deparatmento 30

### CREATE VIEW empvu10

### AS SELECT empno, ename, job

### FROM emp

### WHERE deptno = 10;

### CREATE VIEW salvu30

### AS SELECT empno NUM\_EMP, ename NOMBRE, sal SALARIO

### FROM emp

### WHERE deptno = 30;

### Recuperación de datos de una vista

### Podemos recuperar datos desde una vista al igual que lo hacemos desde una tabla. Podemos visualizar el contenido de la vista al completo o sólo ver columnas y filas específicas.

### SELECT \* FROM salvu30;

### 

### *Modificación de una vista*

### La opción OR REPLACE permite que se cree una vista incluso si ya existe una con ese nombre, reemplazando de esta forma la antigua versión de la vista. Esto significa que la vista puede alterarse sin hacer DROP, volver a crear ni volver a conceder privilegios sobre el objeto.

### Ejemplo:

### CREATE OR REPLACE VIEW empvu10

### (num\_empleado, nombre\_empleado, empleo)

### AS SELECT empno, ename, job

### FROM emp

### WHERE deptno = 10;

### Nota: los alias de columna (en el ejemplo: num\_empleado, nombre\_empleado, empleo) en la cláusula CREATE VIEW deben aparecer en el mismo orden que las columnas en la subconsulta (en el ejemplo: empno,, ename, job).

### Ejemplo de vista compleja:

### CREATE VIEW deptsumvu

### (nombre\_dep, minimo\_sueldo, maximo\_sueldo, media\_sueldo)

### AS SELECT d.dname, MIN(e.sal), MAX(e.sal), AVG(e.sal)

### FROM emp e, dept d

### WHERE e.deptno = d.deptno

### GROUP BY d.dname;

### Reglas para realizar operaciones DML sobre vistas

### Cuando creamos una vista con todas las columnas de la tabla asociada, podremos insertar filas en la vista sin ningún problema.

### Podemos realizar operaciones DML sobre los datos a través de una vista siempre que esas operaciones sigan ciertas reglas.

### Se puede eliminar una fila de una vista salvo que la misma contenga:

### Funciones de grupo (SUM, AVG, COUNT, etc.)

### Una cláusula GROUP BY

### La cláusula DISTINCT

### Se pueden modificar los datos de una vista salvo que contenga una de las condiciones mencionadas y tenga columnas definidas por expresiones (por ejemplo: sal\*12).

### Se pueden agregar datos a través de una vista a menos que la misma contenga cualquiera de las condiciones anteriores y además no existan columnas NOT NULL en la tabla base no seleccionada por la vista. Todos los valores requeridos deben estar presentes en la vista. Recordar que se están agregando valores directamente en la tabla subyacente *a través* de la vista.

### La cláusula WITH CHECK OPTION

### Es posible realizar chequeos de integridad referencial a través de las vistas. Podemos reforzar las restricciones al nivel de la base de datos. La vista puede utilizarse para proteger la integridad de los datos pero el uso es muy limitado.

### La cláusula WITH CHECK OPTION especifica que los INSERTs y UPDATEs realizados a través de las vistas no pueden crear filas que la vista no pueda seleccionar y, por lo tanto, esto permite las restricciones de integridad y chequeos en la validación de los datos para imponerse sobre los datos que se están insertando o actualizando.

### Si hay un intento de realizar operaciones DML sobre las filas que la vista no ha seleccionado, se visualiza un error, con el nombre de la restricción si se ha especificado.

### Ejemplo:

### CREATE OR REPLACE VIEW empvu20

### AS SELECT \*

### FROM EMP

### WHERE deptno = 20

### WITH CHECK OPTION CONSTRAINT empvu20\_ck;

### UPDATE empvu20

### SET deptno = 22

### WHERE empno = 7788;

### SQL> ERROR: violación de la cláusula WITH CHECK OPTION. Si se intenta cambiar el número de departamento para cualquier fila, la sentencia fallará porque viola la restricción de CHECK OPTION.

### La opción WITH READ ONLY

### Podemos denegar operaciones DML sobre determinada vista creándola con la opción WITH READ ONLY.

### Ejemplo:

### CREATE OR REPLACE VIEW empvu10

### (num\_empleado, nombre\_empleado, empleo)

### AS SELECT empno, ename, job

### FROM emp

### WHERE deptno = 10

### WITH READ ONLY;

### DELETE FROM empvu10

### WHERE num\_empleado = 7782;

### SQL> ERROR: cualquier intento de quitar una fila de la vista, dará un error.

### *La sentencia DROP VIEW*

### Se utiliza la sentencia DROP VIEW para eliminar una vista. La ejecución de esta sentencia provoca la eliminación de la definición de la vista de la base de datos. La eliminación de una vista no afecta a las tablas sobre las que se basa la vista. Las vistas o las aplicaciones basadas en la vista eliminada se convierten en inválidas.

### Únicamente el creador o un usuario con el privilegio DROP ANY VIEW puede eliminar una vista.

### Sintaxis:

### DROP VIEW nombre\_vista;

### Ejemplo:

### DROP VIEW empvu10;

### Vistas inline

### Una vista inline es una subconsulta con un alias (nombre correlacionado) que podemos usar dentro de la sentencia SQL.

### Es similar a usar una subconsulta en la cláusula FROM de la consulta principal.

### Una vista inline no es un objeto del esquema.

### Ejemplo:

### SELECT a.ename, a.sal, a.deptno, b.maxsal

### FROM emp a, (SELECT deptno, max(sal) maxsal

### FROM emp

### GROUP BY deptno) b

### WHERE a.deptno = b.deptno

### AND a.sal < b.sal;

### PARTE 5 DML

# UNIDAD DE TRABAJO 5: Consulta y Manipulación de Datos

Texto

El contenido generado por IA puede ser incorrecto.

## La sentencia SELECT

Una sentencia SELECT recupera información de la base de datos. Utilizando una sentencia SELECT, podemos hacer lo siguiente:

* *Selección*: podemos seleccionar los registros de la tabla que queramos, devueltos por una consulta. Podemos utilizar varios criterios para restringir de manera selectiva los registros que queremos ver.
* *Proyección*: podemos seleccionar las columnas de la tabla que deseemos, devueltas por una consulta. Podemos seleccionar tantas tablas como queramos.
* *Unión*: para extraer los datos almacenados en diferentes tablas mediante la creación de enlaces entre una columna y las dos tablas compartidas.

Sintaxis:

**SELECT** [**DISTINCT**] {\*, columna, … }

**FROM** tabla [,tabla …]

**[WHERE** condicion];

En la forma más simple una sentencia SELECT incluye las columnas a consultar (\* indica todas) y una cláusula FROM que especifica la tabla que contiene las columnas descritas en la cláusula SELECT. Adicionalmente, puede incluir una cláusula WHERE que filtra las filas o registros que devolverá la consulta. Si no hay cláusula WHERE, se mostrarán todas las filas de la tabla.

La opción DISTINCT sirve para eliminar filas duplicadas en el resultado de una consulta.

## Expresiones aritméticas, alias de columna, concatenación y strings

Puede ser necesario modificar la forma en que se visualizan los datos o se realizan los cálculos. Esto es posible utilizando expresiones aritméticas detrás de la cláusula SELECT. Una expresión aritmética puede contener nombres de columna, valores numéricos constantes y operadores aritméticos.

Los **operadores aritméticos disponibles en SQL son: + - \* /**

Ejemplo:

SELECT ename, sal, sal+300

FROM emp;

La precedencia de estos operadores es: \* / + - Podemos sobreescribir las reglas de precedencia utilizando paréntesis y así especificar el orden en que se ejecutan los operadores.

Cuando una expresión aritmética contiene un NULL se evalúa a NULL. Un NULL es un valor inaccesible o desconocido. No representa ni un cero ni un espacio en blanco. El cero es un número y el espacio es un carácter.

**Si se desea renombrar el encabezamiento de una columna resultado se pueden utilizar alias**. Es útil especialmente en cálculos. *Sigue inmediatamente al nombre de la columna y opcionalmente puede situarse en medio la palabra clave AS. Si se desea que contenga espacios en blanco o caracteres especiales se encierra entre comillas dobles.*

Ejemplo:

SELECT sal AS salario, ename “nombre del empleado”

FROM emp;

Con el operador de concatenación || se pueden concatenar columnas con otras columnas, expresiones aritméticas o valores constantes con el fin de crear una expresión de caracteres. Las columnas de cualquiera de los lados del operador se combinan para formar una sola columna resultante.

Ejemplo:

SELECT ename || job AS “Empleados”

FROM emp;

Un literal es un carácter, expresión o número incluido en la lista de la cláusula SELECT. Por cada registro devuelto se imprime una cadena de caracteres. Las cadenas de caracteres con formato de texto libre se pueden incluir en el resultado de la consulta y se tratan como una columna de la lista SELECT. Los valores literales de tipo fecha y carácter deben estar encerrados dentro de comillas simples ‘ ‘ mientras que los literales de tipo número no.

Ejemplo:

SELECT ename || ‘ es un ‘ || job AS Empleados

FROM emp;

En Oracle, para visualizar la estructura o definición de una tabla se utiliza el comando DESCRIBE.

Ejemplo:

DESCRIBE dept

## Restricción y clasificación de los datos: condiciones en la cláusula WHERE

Podemos restringir las filas recuperadas usando la cláusula WHERE. Una cláusula WHERE contiene una condición que se debe cumplir y se escribe a continuación de la cláusula FROM.

La cláusula WHERE puede comparar los valores en las columnas, valores literales, expresiones aritméticas o funciones. La cláusula WHERE consta de tres elementos:

* Nombre de la columna
* Operador de comparación
* Nombre de columna, constante o lista de valores

Ejemplo:

SELECT ename, job, deptno

FROM emp

WHERE job=’CLERK’;

Tabla resultado devuelta:

|  |  |  |
| --- | --- | --- |
| **ENAME** | **JOB** | **DEPTNO** |
| JAMES | CLERK | 30 |
| SMITH | CLERK | 20 |
| ADAMS | CLERK | 20 |
| MILLER | CLERK | 10 |

En el ejemplo, la sentencia SELECT recupera el nombre, oficio y número de departamento de todos los empleados cuyo oficio es CLERK.

Podemos observar que el oficio CLERK ha sido especificado en mayúsculas para asegurarse de que se corresponde con el valor de la columna JOB en la tabla EMP, que están todos en mayúsculas**. Las cadenas de caracteres son, por tanto, “case sensitive”, es decir, distingue entre mayúsculas y minúsculas.**

**Cadenas de caracteres y fechas**

Las cadenas de caracteres y las fechas se encierran entre comillas simples en la cláusula WHERE. Sin embargo, los valore numéricos no se encierran entre comillas.

Si la consulta anterior se hubiera formulado como:

SELECT ename, job, deptno

FROM emp

WHERE job=’clerck’;

No se habría devuelto ningún valor en la tabla resultado porque todos los valores están almacenados en mayúsculas.

En cuanto **a las fechas, Oracle las almacena en un formato numérico interno, representado por el siglo, año, mes, día, hora, minutos y segundos. La fecha por defecto se visualiza como YYY-MM-DD.**

**Evitar Comparaciones Directas con Texto o Valores Parciales: No uses LIKE, ni valores como '1982' o 1982, ya que no coinciden con el formato completo de un tipo DATE.**

1. **Usa el formato estándar DATE 'YYYY-MM-DD' para escribir fechas literales:**

**WHERE HIREDATE = DATE '1982-06-15';**

1. **Comparar con Rangos de Fechas: Usa el operador BETWEEN o condiciones de rango para filtrar fechas:**

**SELECT ENAME, HIREDATE**

**FROM EMP**

**WHERE HIREDATE BETWEEN DATE '1982-01-01' AND DATE '1982-12-31';**

1. **Si necesitas trabajar con otro formato (como DD-MM-YYYY), debes convertirlo explícitamente usando TO\_DATE**

**Alternativa con TO\_DATE (solo si necesitas usar otro formato):**

**SELECT ENAME, HIREDATE**

**FROM EMP**

**WHERE HIREDATE BETWEEN TO\_DATE('01-01-1982', 'DD-MM-YYYY') AND TO\_DATE('31-12-1982', 'DD-MM-YYYY’)**

**Operadores de comparación**

Se utilizan en las condiciones en las que se compara una expresión con otra. Tienen el siguiente formato:

WHERE *expresion operador valor*

Los operadores de comparación clásicos son los siguientes:

= Igual a

> Mayor que

< Menor que

>= Mayor o igual que

<= Menor o igual que

<> Distinto o no igual a

Ejemplo:

SELECT ename, sal, comm

FROM emp

WHERE sal<=comm;

Otros operadores de comparación que pueden aparecer en una condición son:

* **BETWEEN … AND …**

Se utiliza para visualizar registros basados en un rango de valores. Los valores especificados como valor superior e inferior se incluyen en la operación.

Ejemplo:

SELECT ename, sal

FROM emp

WHERE sal BETWEEN 1000 AND 1500;

La cláusula WHERE se podría haber expresado también como:

WHERE sal >= 1000 AND sal <= 1500;

También se puede utilizar este operador de forma negativa precediéndola de NOT:

NOT BETWEEN … AND …

* **IN (lista …)**

Se utiliza este operador para localizar valores coincidentes con una determinada lista. Puede utilizarse con cualquier tipo de dato.

Ejemplos:

**SELECT empno, ename, sal, mgr**

**FROM emp**

**WHERE mgr IN (7902, 7566, 7788);**

SELECT empno, ename, mgr, deptno

FROM emp

WHERE ename IN (‘FORD’, ‘ALLEN’);

Admite la versión negativa del operador: NOT IN (lista ….)

* **LIKE**

**Se utiliza para realizar búsquedas en cadenas de caracteres. No siempre se conoce el valor exacto a buscar. Se pueden seleccionar filas que coincidan con un patrón de caracteres usando el operador LIKE. La operación de coincidencia se conoce como una búsqueda que incluye *comodines*:**

% representa cualquier secuencia de cero o más caracteres

\_ (guión bajo) representa un solo carácter

Ejemplos:

- Empleados cuyo nombre empieza por S:

SELECT ename

FROM emp

WHERE ename LIKE ‘S%’;

- Empleados que se contrataron en 1981:

SELECT ename, hiredate

FROM emp

WHERE hiredate LIKE ‘%1981’;

- Empleados cuyo nombre tenga una A como segunda letra:

SELECT ename

FROM emp

WHERE ename LIKE ‘\_A%’;

En el caso de necesitar una coincidencia exacta para los comodines, se utiliza la opción ESCAPE. Dicha opción especifica cual es el carácter ESCAPE. Si quisiéramos hacer una consulta sobre el departamento HEAD\_QUARTERS, lo haríamos de la siguiente manera:

SELECT \* FROM dept

WHERE dname LIKE ‘%\\_%’ ESCAPE ‘\’;

Devolverá la tabla resultado:

|  |  |  |
| --- | --- | --- |
| **DEPTNO** | **DNAME** | **LOC** |
| 50 | HEAD\_QUARTERS | ATLANTA |

La opción ESCAPE identifica la barra (\) como el carácter ESCAPE. En el patrón de búsqueda, el carácter ESCAPE precede al \_. Esto hace que Oracle interprete el \_ literalmente.

* **IS NULL**

Verifica la presencia de valores nulos. Un valor nulo significa que el valor es inaccesible, sin valor, desconocido o inaplicable. Por lo tanto, **no se puede comprobar con el símbolo =**, porque un valor nulo no puede ser igual o distinto a otro valor.

Ejemplos:

SELECT ename, mgr

FROM emp

WHERE mgr IS NULL;

SELECT ename, job, comm

FROM emp

WHERE comm IS NULL;

**Operadores lógicos**

Un operador lógico combina el resultado de dos condiciones para producir un único resultado o para invertir el resultado de una condición simple. Hay tres operadores lógicos disponibles en SQL:

AND devuelve TRUE si ambas condiciones son TRUE

OR devuelve TRUE si alguna de las condiciones es TRUE

NOT devuelve TRUE si la siguiente condición es FALSE

Los operadores lógicos AND y OR nos permiten utilizar varias condiciones en una cláusula WHERE.

Ejemplos:

SELECT empno, ename, job, sal

FROM emp

WHERE sal >= 1100 AND job = ’CLERK’;

SELECT empno, ename, job, sal

FROM emp

WHERE sal >= 1100 OR job = ’CLERK’;

SELECT ename, job

FROM emp

WHERE job NOT IN (‘CLERK’, MANAGER’, ANALYST’);

SELECT empno, ename, job,

FROM emp

WHERE comm IS NOT NULL;

**Reglas de Precedencia**

Cuando aparece más de un operador en una condición, se evalúan en el orden que digan las reglas de precedencia.

|  |  |
| --- | --- |
| **ORDEN EVALUADO** | **OPERADOR** |
| 1 | Todos los operadores de comparación |
| 2 | NOT |
| 3 | AND |
| 4 | OR |

Se pueden utilizar los paréntesis para modificar las reglas de precedencia.

Ejemplo:

SELECT ename, job, sal

FROM emp

WHERE job = ’SALESMAN’ OR job=’PRESIDENT’ AND sal>1500;

Hay 2 condiciones:

* La primera es que el oficio sea PRESIDENT y el salario mayor que 1500.
* La segunda es que el oficio sea SALESMAN.

Por lo tanto, la sentencia SELECT interpretará lo siguiente: “Seleccionar los registros de los empleados cuyo oficio sea PRESIDENT y gane más de 1500 o si su oficio es SALESMAN”.

En el caso de haber utilizado paréntesis como en el siguiente ejemplo, la condición sería totalmente distinta:

SELECT ename, job, sal

FROM emp

WHERE (job = ’SALESMAN’ OR job=’PRESIDENT’) AND sal>1500;

“Seleccionar los registros de los empleados cuyo oficio sea PRESIDENT o SALESMAN y ganen más de 1500”.

**La cláusula ORDER BY**

El orden de las filas recuperadas por el resultado de una consulta es indefinido. La cláusula ORDER BY se utiliza para ordenar las filas. Si se usa, será la última cláusula de la sentencia SELECT.

La ordenación se puede hacer de forma ascendente o de forma descendente, poniendo las palabras clave ASC (ascendente) o DESC (descendente) al final de la cláusula.

Ejemplos:

SELECT ename, job, deptno, hiredate

FROM emp

ORDER BY hiredate DESC;

Utilizando un alias de columna:

SELECT empno, ename, sal\*12 sal\_anual

FROM emp

ORDER BY sal\_anual;

Ordenando por más de una columna. Se puede ordenar por una columna no seleccionada:

SELECT ename, deptno, sal

FROM emp

ORDER BY deptno, sal DESC;

## Funciones a nivel de fila

Las funciones constituyen una característica muy poderosa de SQL y se pueden utilizar para:

* Realizar cálculos sobre datos
* Modificar ítems de datos individuales
* Manipular la salida de grupos de filas
* Modificar formatos de los datos para su presentación
* Convertir los tipos de datos de las columnas

Las funciones SQL aceptan argumentos y devuelven un valor.

Forma

El contenido generado por IA puede ser incorrecto.

Existen 2 tipos distintos de funciones:

* A nivel de fila
* A nivel de grupos de filas

Las funciones a nivel de fila operan sólo sobre filas y devuelven un único resultado por cada una de ellas:

* Funciones de caracteres
* Funciones de número
* Funciones de fecha
* Funciones de conversión

Las funciones a nivel de fila se usan para manipular ítems de datos. Aceptan uno o más argumentos y devuelven un valor por cada fila que recupera la consulta. Un argumento puede ser:

* Una constante suministrada por el usuario
* Un valor de variable
* Un nombre de columna
* Una expresión

Las funciones a nivel de fila pueden actuar sobre cada fila recuperada por la consulta, devuelven un resultado por cada fila, pueden devolver un dato de diferente tipo que el referenciado, pueden aceptar uno o más argumentos del usuario, se pueden anidar y se pueden usar en las cláusulas SELECT, WHERE y ORDER BY.

Sintaxis:

*nombre\_de\_función (columna | expresión, [arg1, arg2,…])*

**Funciones de caracteres:**

a) LOWER (columna | expresión) -> Pasa a minúsculas el campo que se le pase.

b) UPPER (columna | expresión) -> Pasa a mayúsculas el campo.

c) INITCAP (columna | expresión) -> Pasa a mayúsculas la primera letra de cada palabra y deja las demás en minúsculas.

d) CONCAT (columna1 | expresión1, columna2 | expresión2) -> Concatena las 2 cadenas. Es equivalente al operador ||

e) SUBSTR (columna | expresión, m, [n]) -> devuelve la cadena de caracteres empezando por el carácter en la posición m hasta la n. Si n se omite (opcional), devuelve los caracteres hasta el final. Si m es negativo, el contador empieza desde el final de la cadena.

f) LENGTH (columna | expresión) -> longitud de la cadena que se le pase.

g) INSTR (columna | expresión, m) -> devuelve la posición del carácter m dentro de la cadena.

h) LPAD (columna | expresión, n, ‘string’) -> rellena la cadena por la IZQUIERDA con el carácter ESPECIFICADO (‘string’) hasta alcanzar un tamaño total de n posiciones. Esto hace que los datos de una columna sean de tamaño fijo.

 **columna | expresión**: La cadena original que quieres rellenar. COLUMNA O EXPRESION

 **n**: El tamaño total que tendrá la cadena resultante (incluyendo el relleno).

 **'string'**: El carácter o los caracteres que se usarán para rellenar la cadena por la izquierda. Si no se especifica, por defecto se usa un espacio en blanco.

**trabaja con cadenas .tienes que cambiar si te da error con funciones tipo to\_char... ...**

SELECT LPAD(codigo, 6, '0') AS codigo\_fijo

FROM tabla;

RPAD SU INVERSA DERECHA

i) TRIM (parámetro, carácter\_a\_borrar FROM campo) -> sirve para eliminar caracteres del principio (leading), del final (trailing) o ambos (both).

El parámetro tiene tres valores posibles:

* leading: de una cadena de caracteres, borras el principio
* trailing: de una cadena de caracteres, borras el final
* both: de una cadena de caracteres, borras tantos el principio como el final.

Ejemplos (sacar de la tabla de empleados (emp) información como la siguiente):

* ‘El trabajo de’ nombre\_del\_empleado\_en\_mayúsculas ‘es’ ‘Puesto\_en\_minúsculas’

SELECT 'El trabajo de ' || INITCAP (ename) || ' es ' || LOWER (job) FROM emp;

* Visualizar el numero de empleado, nombre y departamento de blake:

SELECT ename, empno, deptno

FROM emp

WHERE ename = UPPER ('blake');

* Visualizar el nombre del empleado, nombre del empleado concatenado a su puesto de trabajo, longitud del nombre y posición de la letra “A” en el nombre:

SELECT ename, ename ||empno, LENGTH (ename), INSTR (ename, 'A')

FROM emp;

**Funciones numéricas:**

a) ROUND (columna | expresión, [n]) -> Redondea la columna, expresión o valor a n posiciones decimales. Si se omite n, no se redondea con lugares decimales. Si n es negativo, los números a la izquierda del punto decimal se redondean.

b) TRUNC (columna | expresión, [n]) -> Trunca la columna o valor en la n-ésima posición decimal. Si se omite n, sin lugares decimales. Si n es negativo, los números a la izquierda del punto decimal se truncan a cero.

c) MOD (m, n) -> Devuelve el resto de la división de m entre n.

Ejemplos:

ROUND (45.926, 2) ➔ 45.93

TRUNC (45.926, 2) ➔ 45.92

MOD (1600, 300) ➔ 100

* Calcular el ratio (resto) del salario respecto a la comisión de cada empleado, cuyo oficio sea “SALESMAN”:

SELECT ename, sal, comm, MOD (sal, comm)

FROM emp

WHERE job='SALESMAN';

* Visualizar el valor 45.923 redondeado a centenas, 0 y 10 posiciones decimales:

SELECT ROUND (45.923, 2), ROUND (45.923, 0), ROUND (45.923, -1)

FROM SYS.DUAL;

Tabla resultado devuelta:

|  |  |  |
| --- | --- | --- |
| **ROUND (45.923, 2)** | **ROUND (45.923, 0)** | **ROUND (45.923, -1)** |
| 45.92 | 46 | 50 |

* Visualizar el valor 45.923 truncado a centenas, 0 y 10 posiciones decimales:

SELECT TRUNC (45.923, 2), TRUNC (45.923, 0), TRUNC (45.923, -1)

FROM SYS.DUAL;

Tabla resultado devuelta:

|  |  |  |
| --- | --- | --- |
| **TRUNC (45.923, 2)** | **TRUNC (45.923, 0)** | **TRUNC (45.923, -1)** |
| 45.92 | 45 | 40 |

**Funciones de fecha:**

Internamente Oracle las almacena como números, que incluye: Siglo, año, mes, día, horas, minutos y segundos.

Va desde el 1 de enero de 4712 A.C. hasta el 31 de diciembre de 9999 D.C.

SELECT SYSDATE FROM SYS.DUAL; 🡪 para saber la fecha actual. DUAL es una tabla virtual de la BD .

Recordemos que el formato de fecha es YYY-MM-DD.

A las fechas, se les puede aplicar operadores aritméticos (+, -):

* Fecha + Número ➔ Fecha (agrega una cantidad de días a la fecha)
* Fecha – Número ➔ Fecha (resta una cantidad de días a partir de una fecha)
* Fecha – Fecha ➔ Numero de días (resta una fecha de otra)
* Fecha + Numero/24 ➔ Fecha (agrega una cantidad de horas a la fecha)

Ejemplo: Número de semanas que llevan contratados los empleados.

SELECT ename, (SYSDATE - hiredate)/7

FROM emp;

Las funciones de fecha son las siguientes:

* MONTHS\_BETWEEN (‘fecha1’, ‘fecha2’): devuelve un número de meses entre dos fechas dadas.
* ADD\_MONTHS (‘fecha’, n): devuelve una fecha en la que se añadirá ‘n’ meses.
* NEXT\_DAY (‘fecha’, ‘caracteres’): calcula la fecha posterior a la dada. Los caracteres pueden ser un número representando un día o una cadena de caracteres Ejemplo: NEXT\_DAY (04-10-04, ‘MONDAY’) ➔ ‘08-10-04’
* LAST\_DAY (‘fecha’): te devuelve el último día del mes de la fecha especificada.
* ROUND (‘fecha’, [formato de fecha]): devuelve la fecha redondeada a la unidad especificada por el formato. Ejemplos de formato: ‘month’, ‘year’.
* TRUNC (‘fecha’, [formato de fecha]): devuelve la fecha con la porción del día truncado en la unidad especificada por el modelo de formato de fecha. Si se omite el formato, la fecha se trunca en el día más próximo.
* CEIL: que redondean hacia arriba Funcionan parecido a Round y Trunc.
* FLOOR: que redondean hacia abajo Funcionan parecido a Round y Trunc. <https://blogs.oracle.com/connect/post/from-floor-to-ceiling-and-other-functional-cases> (Ejemplos de Ceil, Floor, Round Trunc)

Ejemplos:

* Comparar las fechas de contratación de todos los empleados que empezaron en 1987. Visualizar el número de empleados, fecha de contratación y el mes en que el empezaron, utilizando funciones ROUND y TRUNC.

SELECT empno, hiredate, ROUND (hiredate, 'month'), TRUNC (hiredate, 'month')

FROM emp

WHERE hiredate LIKE '%87';

* Para todos los trabajadores empleados por menos de 200 meses, visualizar número de empleado, fecha de contratación, fecha de revisión de 6 meses, primer viernes después de la fecha de contratación y el mismo día del mes en que se contrataron.

SELECT empno, hiredate, ADD\_MONTHS (hiredate, 6),

NEXT\_DAY (hiredate, 'VIERNES'), LAST\_DAY (hiredate)

FROM emp

WHERE MONTHS\_BETWEEN (sysdate, hiredate) <300;

**Tipos y conversión de tipos:**

Recordemos los tipos de datos existentes en Oracle:

* NUMBER (n, d) 🡪 n: numero máximo de dígitos
* d: numero de decimales
* dígitos enteros = n – d
* VARCHAR2 (s) 🡪 s: cadena de caracteres de tamaño máximo s (tamaño variable)
* DATE
* CHAR (s) 🡪 s: cadena de caracteres de tamaño fijo s

Las conversiones de datos pueden realizarse de 2 maneras:

1. Implícitas: las hace Oracle, sin intervención del usuario.

***De A***

VARCHAR2 o CHAR NUMBER

VARCHAR2 o CHAR DATE

NUMBER VARCHAR2

DATE VARCHAR2

1. Explícitas: las que el usuario hace explícitamente.

Forma

El contenido generado por IA puede ser incorrecto.

TO\_CHAR

1. de fecha a caracteres.

2. de número a caracteres.

1. TO\_CHAR (fecha, ‘formato’): de fecha a caracteres

Ejemplo: TO\_CHAR (hiredate, ‘MM/YY’).

Formato:

* 1. Encerrado entre comillas simples
  2. Es case sensitive Y ≠ y
  3. Elementos del formato
     1. YYYY : año completo en numero
     2. YEAR : año en letras
     3. MM : número del mes con 2 dígitos
     4. MONTH : nombre completo del mes
     5. DY : abreviatura de 3 letras del día de la semana
     6. DAY : nombre completo del día
     7. S : siglo
     8. SCC : pone AC antes de fecha
     9. Q : pone el trimestre del año
     10. RM : pone el mes en números romanos
     11. J: día juliano (el numero de días desde el 31 de diciembre del 4712 a.c.)
     12. fm : quita el relleno en blanco o los 0 a la izquierda

2. TO\_CHAR (numero, ‘formato’): de número a caracteres

Ejemplo: SELECT TO\_CHAR (sal, ’99,999L’) FROM emp;

Nota: Mirar el libro McGRAW

9: Digito

TO\_NUMBER (cadena\_de\_caracteres): convierte de caracteres a números.

TO\_DATE (cadena\_de\_caracteres [,formato]): cambia de caracteres a fecha

Ej.: SELECT TO\_DATE (‘01111996’,’DDMMYYYY’)

FROM SYS.DUAL;

Hay un formato de fecha: RR (Como YY, pero sensible a cambios de siglo)

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Fecha especificada (2 dígitos) | |
|  |  | 0-49 | 50-99 |
| Fecha actual  (2 dígitos) | 0-49 | Mismo siglo | Siglo anterior |
| 50-99 | Siglo posterior | Mismo siglo |

Algunos ejemplos que utilizan este formato RR comparado con el YY:

|  |  |  |  |
| --- | --- | --- | --- |
| **Año Actual** | **Fecha especificada** | **Formato RR** | **Formato YY** |
| 1995 | 27-OCT-95 | 1995 | 1995 |
| 1995 | 27-OCT-17 | 2017 | 1917 |
| 2001 | 27-OCT-17 | 2017 | 2017 |
| 2001 | 27-OCT-95 | 1995 | 2095 |

Funciones de Oracle

https://sites.google.com/site/josepando/home/funciones-sql/funciones-que-devuelven-una-valor-nico-para-cada-fila-de-una-consulta-o-vista/funciones-de-tipo-fecha

**Otras funciones de Oracle:**

Función NVL: Convierte un nulo a un valor concreto

* Es aplicable a cualquier tipo de datos: fechas, caracteres y números.
* Los tipos de datos de ambas expresiones deben coincidir.

Sintaxis: *NVL (expresion1, expresion2)*

Cada vez que la *expresión1* equivale a nulo, la sustituye por la *expresion2*.

Ejemplos:

* Numérico: NVL (comm, 0)
* Fecha: NVL (hiredate, ’01-ene-82’)
* Cadena de caracteres: NVL (job, ‘Sin trabajo’)
* Calcular la compensación anual de todos los empleados, es decir, multiplicar el salario mensual por 12 y añadirle la comisión:

SELECT ename, sal, comm, (sal\*12) + NVL (comm, 0)

FROM emp;

Función DECODE: hace las veces de sentencias CASE o IF-THEN-ELSE para facilitar consultas condicionales.

Esta función descifra una expresión después de compararla con cada valor de búsqueda. Si la expresión coincide con el valor de búsqueda, devuelve el resultado. Si se omite el valor por defecto (default), se devolverá un valor nulo en el caso de que la búsqueda no coincida con ninguno de los valores resultantes.

Sintaxis:

DECODE (columna | expresión, valor1, resultado1

[, valor2, resultado2, …, ]

[, default ]) ;

Interpretación:

IF (columna | expresión) = valor1

THEN resultado1

IF (columna | expresión) = valor2

THEN resultado2

IF (columna | expresión) = valor3

THEN resultado3

ELSE resultado 4

Ejemplo:

Mostrar el puesto de trabajo y el salario de los empleados de tal manera que si el puesto de trabajo (JOB) es ANALYST, el salario (SAL) se incrementa en un 10%; si JOB es CLERK, el salario se incrementa en un 15%; si JOB es MANAGER, el salario se incrementa en un 20%. El resto no percibe incremento de salario.

SELECT job, sal,

DECODE (job, 'ANALYST', sal \*1.1,

'CLERK', sal \*1.15,

'MANAGER', sal\*1.20,

sal)

FROM emp;

**Anidamiento de funciones:**

Las funciones a nivel de fila pueden ser anidadas hasta cualquier profundidad. Estas se evalúan desde el nivel más interno hasta el más externo.

F3 (F2 (F1 (col, arg1), arg2), arg3)

paso1

paso2

paso3

Ejemplos:

SELECT ename, NVL (TO\_CHAR (mgr), ‘ Sin jefe’)

FROM emp

WHERE mgr IS NULL;

SELECT TO\_CHAR (NEXT\_DAY (ADD\_MONTHS (hiredate, 6), ‘VIERNES’),

‘fmDay, Month ddth, YYYY’) “Prox revision de 6 meses”

FROM emp

ORDER BY hiredate;

## 4. Obtención de datos de múltiples tablas

En ocasiones necesitamos utilizar datos de más de una tabla al mismo tiempo. Por ejemplo, podríamos necesitar datos que estén en la tabla EMP y en la tabla DEPT. Para obtenerlos necesitamos unir ambas tablas o, lo que es lo mismo, realizar una JOIN.

La unión o JOIN de dos tablas se realiza a través de valores comunes, esto es, con las columnas de clave extranjera y la de clave primaria. Para ello, se escribe una condición en la cláusula WHERE de la sentencia SELECT.

Sintaxis:

SELECT tabla1.columna1, tabla2.columna2, …

FROM tabla1, tabla2

WHERE tabla1.columnaFK = tabla2.columnaPK;

Para combinar tablas se necesita como mínimo una cantidad de condiciones de JOIN equivalentes a la cantidad de tablas menos uno. Por lo tanto, para combinar 4 tablas, se necesitarían un mínimo de 3 condiciones. **Esta regla no se puede aplicar si la tabla tiene una clave primaria combinada**, en cuyo caso se pueden requerir más de una columna para identificar unívocamente a cada fila.

Ejemplo: Supongamos el siguiente esquema relacional:

CUSTOMER (name, custid)

ORD (custid, ordid)

ITEM (ordid, itemid)

SELECT customer.name, ord.ordid, item.itemid

FROM customer, ord, item

WHERE customer.custid = ord.custid

AND ord.ordid = i.ordid;

Cuando una condición de **JOIN no es válida o se omite completamente, el resultado es un producto cartesiano,** en el cual se muestran las combinaciones de todas las filas. Se combinan todas las filas de la primera tabla con todas las filas de la segunda.

Un producto cartesiano tiende a generar una gran cantidad de filas y este tipo de resultado es raramente útil. Se debería incluir una condición de JOIN válida en la cláusula WHERE, a menos que exista la necesidad específica de combinar todas las filas de todas las tablas.

Ejemplo:

SELECT emp.empno, emp.ename, emp.deptno, dept.deptno, dept.loc

FROM emp, dept

WHERE emp.deptno = dept.deptno;

Para evitar la ambigüedad, es necesario calificar los nombres de las columnas en la cláusula WHERE usando los nombres de las tablas. Sin los prefijos, la columna deptno podría ser tanto de la tabla dept como de la tabla emp.

Si no existen nombres repetidos en las tablas que se unen, no es necesario calificar las columnas. Sin embargo, se ganará una mejora del rendimiento con el uso de prefijos, ya que de esta manera le decimos exactamente a Oracle a donde ir para encontrar las columnas.

Los requerimientos para calificar a los nombres de columna ambiguos también son aplicables para las columnas que también podrían ser ambiguas en otras cláusulas como SELECT u ORDER BY.

Además del JOIN, se pueden añadir criterios adicionales para la cláusula WHERE. Por ejemplo, para visualizar el número de empleado de King, el nombre, número de departamento y localidad, necesitamos una condición adicional en la cláusula WHERE.

SELECT empno, ename, emp.deptno, loc

FROM emp, dept

WHERE emp.deptno = dept.deptno

AND ename = UPPER (‘King’);

**Alias de tabla**

Calificar las tablas con los nombres de tabla puede llevar mucho tiempo, especialmente si los nombres son largos. Para ello existen los alias de tabla. Así como los alias de columna dan a una columna otro nombre, los alias de tablas dan a una tabla otro nombre. Los alias de tablas ayudan a mantener reducido el código SQL, utilizando por tanto menos memoria.

Ejemplo:

SELECT e.empno, e.ename, e.deptno, d.deptno, d.loc

FROM emp e, dept d

WHERE e.deptno = d.deptno;

Las reglas de utilización de los alias de tabla son las siguientes:

* Pueden tener hasta 30 caracteres de longitud pero es preferible que sean cortos
* Si se usa un alias para un nombre de tabla en particular en la cláusula FROM, entonces debe ser sustituida a través de toda la sentencia SELECT.
* Los alias de tablas deben ser significativos
* El alias de tabla es válido sólo en la sentencia SELECT actual.

**Otros tipos de JOINS (Mirar el anexo 1 de tipos de JOINS)**

A las JOINS realizadas utilizando las claves primarias y extranjeras vistas hasta ahora se las denomina JOINS simples (EQUIJOINS o INNER JOINS), en las que los valores de las columnas comparadas son iguales.

En algunos casos las columnas a unir pueden estar en la misma tabla, correspondiéndose con relaciones reflexivas. Por ejemplo, el jefe de un empleado (MGR) puede combinarse con el número de empleado (ENAME) que supervisa.

SELECT trabajador.ename || ‘ trabaja para ’ || jefe.ename as currantes

FROM emp trabajador, emp jefe

WHERE trabajador.mgr = jefe.empno;

Existen otros tipos de JOINS:

* NON-EQUIJOINS
* OUTER JOINS

NON-EQUIJOINS:

Se dan **cuando no existe una correspondencia directa entre las tablas que se unen en el JOIN.** La relación se obtiene utilizando un operador distinto del igual (=).

Ejemplo:

SELECT e.ename, e.sal, s.grade

FROM emp e, salgrade s

WHERE e.sal BETWEEN s.losal AND s.hisal;

Imagen que contiene Texto

El contenido generado por IA puede ser incorrecto.

La relación entre las tablas EMP y SALGRADE es un NON-EQUIJOIN, lo cual significa que ninguna columna en la tabla EMP se corresponde directamente con una columna en la tabla SALGRADE. **La relación entre las dos tablas es que la columna SAL en la tabla EMP está entre las columnas LOSAL y HISAL de la tabla SALGRADE.**

Se produce la siguiente tabla-resultado:

Forma

El contenido generado por IA puede ser incorrecto.

Se está evaluando el grado de salario de un empleado, el cual debe estar entre cualquier par de los rangos de salario alto y bajo.

Es importante destacar que todos los empleados aparecen exactamente una vez cuando se ejecuta la consulta. Ningún empleado se repite en la lista. Hay dos razones para esto:

* Ninguna de las filas en la tabla del grado de salario contiene grados que se solapen, lo que se traduce en que el valor del salario de un empleado sólo puede estar entre el salario más bajo y el más alto de una de las filas en la tabla del grado del salario
* Todos los salarios de los empleados están dentro de los límites proporcionados por la tabla del grado del salario, lo que significa que ningún empleado gana menos del valor más bajo contenido en la columna LOSAL ni más del valor más alto contenido en la columna HISAL.

OUTER-JOINS:

Si una fila no satisface una condición de JOIN, no aparecerá en la tabla-resultado de la consulta. Por ejemplo, si relacionamos las tablas EMP y SALGRADE por la columna DEPTNO, no aparecerá el departamento OPERATIONS porque nadie trabaja en él. La(s) filas que faltan pueden ser recuperadas si en la condición de JOIN se usa un operador de OUTER-JOIN: **(+). Este operador se sitúa en el “lado” del JOIN que es deficiente en información. Tiene el efecto de crear una o más filas NULL para que aquellas filas de la tabla sin valores coincidentes en la otra puedan ser combinadas.**

Sintaxis:

SELECT tabla1.columna, tabla1.columna2, … tabla2.columnax

FROM tabla1, tabla2

WHERE tabla1.columnai (+) = tabla2.columnaj;

SELECT tabla1.columna, tabla1.columna2, … tabla2.columnax

FROM tabla1, tabla2

WHERE tabla1.columnai = tabla2.columnaj (+);

Ejemplo:

SELECT e.name, d.deptno, d.dname

FROM emp e, dept d

WHERE e.deptno (+) = d.deptno

ORDER BY e.deptno;

Diagrama

El contenido generado por IA puede ser incorrecto.

Utilizando una OUTER-JOIN se produce la siguiente tabla resultado:

Forma

El contenido generado por IA puede ser incorrecto.

Como se observa, el departamento OPERATIONS, que no tiene ningún empleado, también se visualiza.

A la hora de utilizar una OUTER-JOIN hay que tener en cuenta las siguientes restricciones:

* El operador (+) puede aparecer sólo en un lado de la expresión, aquel lado al que le falta la información. Éste recupera aquellas filas de una tabla que no tiene correspondencia directa en otra tabla.
* **Una condición que incluye un OUTER-JOIN no puede utilizar el operador IN o unirse a otra condición por el operador OR.**

## Datos Agregados por medio de Funciones de Grupo

A diferencia de las funciones a nivel de fila estudiadas en apartados anteriores, las ***las funciones de grupo*** operan sobre conjuntos de filas para dar un resultado por cada uno de ellos. Dichos grupos pueden estar constituidos por la tabla entera o por partes de la misma.

Forma

El contenido generado por IA puede ser incorrecto.

**Funciones de grupo:**

Estas funciones aceptan un argumento y se clasifican en:

* **AVG ([DISTINCT|ALL] n)**: calcula el valor promedio del argumento n, ignorando los valores nulos.
* **COUNT (\* | [DISTINCT|ALL] expr)**: devuelve la cantidad de filas cuando expr no resulta en un valor nulo. Para contar las filas se usa \*, el mismo incluye filas duplicadas o aquellas que contienen valores nulos.
* **MAX ([DISTINCT|ALL] expr)**: calcula el valor máximo de la expr que se le pase.
* **MIN ([DISTINCT|ALL] expr)**: calcula el valor mínimo del argumento expr pasado.
* **SUM ([DISTINCT|ALL] n)**: devuelve la suma de los valores del argumento pasado, ignorando los nulos.
* **STDDEV ([DISTINCT|ALL] n):** calcula la desviación estándar estadística de n, ignorando los valores nulos.
* **VARIANCE ([DISTINCT|ALL] n)**: calcula la varianza estadística de n, ignorando los nulos.

**Uso de las Funciones de grupo:**

**Sintaxis:**

**SELECT [columna, …] *función\_de\_grupo* (columna)**

**FROM tabla**

**[WHERE condición]**

**[GROUP BY columna1 [, columna2…]]**

**[ORDER BY columna1 [, columna2…]];**

* El uso de DISTINCT, hace que la función no considere los valores duplicados. ALL provoca que se consideren todos los valores, incluyendo los duplicados. Por defecto, si no se especifica nada, se sobreentiende ALL.
* Los tipos de datos de los parámetros en pueden ser CHAR, VARCHAR2, NUMBER o DATE.
* Las funciones AVG, SUM, VARIANCE y STDDEV sólo pueden ser utilizadas con valores numéricos.
* Todas las funciones de grupo excepto COUNT(\*) ignoran los valores nulos. Para sustituir NULL por un valor concreto, use la función NVL.
* Oracle, por defecto, ordena el resultado en orden ascendente cuando se usa una cláusula GROUP BY. Se puede realizar la ordenación descendente incluyendo la opción DESC en la cláusula ORDER BY.

Ejemplos:

SELECT AVG(SAL), MAX(SAL), MIN(SAL), SUM(SAL)

FROM EMP

WHERE JOB LIKE ‘SALES%’;

SELECT MIN(hiredate), MAX(hiredate)

FROM EMP;

SELECT COUNT(\*)

FROM EMP

WHERE deptno = 30;

SELECT COUNT(comm)

FROM EMP

WHERE deptno = 30;

SELECT COUNT(deptno)

FROM EMP;

SELECT COUNT(DISTINCT(deptno))

FROM EMP;

SELECT AVG(comm)

FROM EMP;

SELECT AVG(NVL(comm,0))

FROM EMP;

**La cláusula GROUP BY:**

Hasta ahora, todas las funciones de grupo trataban la tabla como un gran grupo de información. A veces, es necesario dividir la tabla de información en grupos más pequeños, asunto que se resuelve utilizando la cláusula GROUP BY.

Forma

El contenido generado por IA puede ser incorrecto.

**La cláusula GROUP BY se utiliza con el propósito de repartir las filas de una tabla en grupos más pequeños. Las funciones de grupo se pueden utilizar para devolver información resumida para cada grupo.**

Para utilizar esta cláusula hay que respetar unas reglas:

**Todas las columnas mendionadas en la SELECT que no son funciones de grupo, tienen que estar en la cláusula GROUP BY**

* Si se incluye una función de grupo en una cláusula SELECT, no se pueden seleccionar resultados individuales a menos que la columna aparezca en la cláusula GROUP BY. Si no se incluyen las columnas correspondientes en la lista, se recibirá un mensaje de error.
* Con el uso de la cláusula WHERE se pueden excluir filas antes de la división en grupos.
* No se pueden utilizar alias de columna en la cláusula GROUP BY.
* Por defecto, las filas se ordenan en forma ascendente de acuerdo a la lista GROUP BY. Puede modificarse con ORDER BY.

Ejemplo:

SELECT deptno, AVG (sal)

FROM emp

GROUP BY deptno;

SELECT AVG (sal)

FROM emp

GROUP BY deptno;

SELECT deptno, AVG (sal)

FROM emp

GROUP BY deptno

ORDER BY AVG(sal);

**Algunas veces, es necesario ver los resultados de grupos dentro de grupos. Esto se realiza incluyendo más de una columna en la cláusula GROUP BY.**

Ejemplo:

SELECT deptno, job, SUM (sal)

FROM emp

GROUP BY deptno, job;

Ejemplos de consultas no válidas que utilizan funciones de grupo:

SELECT deptno, COUNT (ename)

FROM emp;

ERROR >> columna no especificada en la cláusula GROUP BY

SELECT deptno, COUNT (ename)

FROM emp

GROUP BY deptno;

CORRECTO

SELECT deptno, AVG (sal)

FROM emp

WHERE AVG (sal) > 2000

GROUP BY deptno;

ERROR >> no puede usar la cláusula WHERE para restringir grupos

SELECT deptno, AVG (sal)

FROM emp

GROUP BY deptno

HAVING AVG (sal) > 2000;

CORRECTO

**Cláusula HAVING o Exclusión de resultados de un grupo:**

**De la misma manera que se utiliza la cláusula WHERE para restringir los registros que se selecciona, se utiliza la cláusula HAVING para restringir grupos.**

Por ejemplo, se desea localizar el máximo salario de cada departamento pero mostrar sólo aquellos cuyo máximo es superior a 2900. Para ello habrá que averiguar el máximo salario de cada departamento agrupando por nº de departamento y limitar los grupos a aquellos departamentos que superen 2900 su salario máximo.

Forma

El contenido generado por IA puede ser incorrecto.

Sintaxis:

SELECT [columna, …] *función\_de\_grupo* (columna)

FROM tabla

[WHERE condición]

[GROUP BY columna1 [, columna2…]]

**[HAVING condición\_de\_grupo]**

[ORDER BY columna1 [, columna2…]];

Ejemplos:

SELECT deptno, MAX (sal)

FROM emp

GROUP BY deptno

HAVING MAX (sal) > 2900;

SELECT job, SUM (sal)

FROM emp

WHERE job NOT LIKE ‘SALES%’

GROUP BY job

HAVING SUM (sal) > 5000

ORDER BY SUM (sal);

SELECT MAX (AVG (sal))

FROM emp

GROUP BY deptno;

## Subconsultas

Supongamos que se quiere escribir una consulta para localizar aquellos empleados que gana un salario superior al de Jones. Para realizar este problema, se necesitan 2 consultas:

* Una que averigüe el salario de Jones
* Otra que encuentre a los empleados que ganan más de esta cantidad

Forma

El contenido generado por IA puede ser incorrecto.

La consulta más interna o subconsulta devuelve una valor que será utilizado por la consulta principal (la consulta más externa). Usar una subconsulta es equivalente a realizar 2 consultas secuencialmente y utilizar el resultado de la primera que se ejecuta (subconsulta) como valor para la segunda consulta que se ejecuta o consulta principal.

Sintaxis:

SELECT columna1, ..., columnax

FROM tabla

WHERE expresión operador (SELECT columna1, ..., columnaz

FROM tabla);

Una subconsulta es una sentencia SELECT que está incluida en una cláusula de otra sentencia SQL. Se pueden construir comandos simples y potentes usando subconsultas. **Pueden ser útiles cuando se necesita seleccionar filas de una tabla con una condición que depende de los datos que están en la misma tabla.**

Se puede poner una subconsulta en cláusulas WHERE, HAVING y FROM de una SELECT o un DELETE.

El operador que precede a la subconsulta es un operador de comparación.

Ejemplo:

SELECT ename

FROM emp

WHERE sal > (SELECT sal

FROM emp

WHERE empno = 7566);

Para usar subconsultas se deben contemplar unas reglas:

* Encierre las subconsultas entre paréntesis
* Coloque la subconsulta a la derecha del operador
* No agregar una cláusula ORDER BY a una subconsulta. Se puede tener solamente una cláusula ORDER BY para una sentencia SELECT y específicamente debe ser la última cláusula en la sentencia SELECT principal.
* Para las subconsultas se utilizan 2 clases de operadores: a nivel de fila para subconsultas que devuelven sólo una fila (Operadores mono-registro) y a nivel de grupo para subconsultas que devuelven más de una fila (Operadores multi-registro).

|  |  |
| --- | --- |
| **OPERADORES MONO-REGISTRO** | **OPERADORES MULTI-REGISTRO** |
| =  >  >=  <  <=  <> | IN: igual a los valores de cierta lista  ANY: compara los valores con cada valor devuelto por la subconsulta  ALL: compara los valores con cada uno de los valores devueltos por la subconsulta |

Ejemplos de subconsultas mono-registro:

SELECT ename, job

FROM emp

WHERE job = (SELECT job

FROM emp

WHERE empno = 7369);

SELECT ename, job

FROM emp

WHERE job = (SELECT job

FROM emp

WHERE empno = 7369)

AND sal > (SELECT sal

FROM emp

WHERE empno = 7876);

SELECT ename, job, sal

FROM emp

WHERE sal = (SELECT MIN(sal)

FROM emp);

SELECT deptno, MIN (sal)

FROM emp

GROUP BY deptno

HAVING MIN (sal) > (SELECT MIN (sal)

FROM emp

WHERE deptno = 20);

Puesto de trabajo con la media más baja de salarios:

SELECT job, AVG (sal)

FROM emp

GROUP BY job

HAVING AVG (sal) = (SELECT MIN (AVG (sal))

FROM emp

GROUP BY job);

Consultas erróneas:

SELECT empno, ename

FROM emp

WHERE sal = (SELECT MIN(sal)

FROM emp

GROUP BY deptno);

>> ERROR: la subconsulta devuelve más de una fila (el operador = sólo

admite un valor). Se resolvería si se utilizara IN (operador multiregistro).

SELECT ename, job

FROM emp

WHERE job = (SELECT job

FROM emp

GROUP BY ename = ‘SMYTHE’);

>> No se devuelve ninguna fila pues SMYTHE no existe. La consulta principal recibe un NULL de la subconsulta y no existe ningún empleado

que tenga empleo nulo.

Ejemplos de subconsultas multi-registro:

SELECT ename, sal, deptno

FROM emp

WHERE sal IN (SELECT MIN (sal)

FROM emp

GROUP BY deptno);

SELECT empno, ename, job

FROM emp

WHERE sal < ANY (SELECT sal

FROM emp

WHERE job = ‘CLERK’);

Nota:

< ANY significa menos que el máximo

> ANY significa más que el mínimo

= ANY es equivalente a IN

SELECT empno, ename, job

FROM emp

WHERE sal > ALL (SELECT AVG (sal)

FROM emp

GROUP BY deptno);

Nota:

> ALL significa más que el máximo

> ALL significa menos que el mínimo

El operador NOT puede ser utilizado con los operadores IN, ANY y ALL.

## 5. OPERADORES DE UNION, INTERSECT Y MINUS

* Son operadores de conjuntos , por tanto se utilizaran entre dos select que serán las encargadas de seleccionar los dos conjuntos

Select… from …where

Operador\_de\_conjunto

Select …from..where

* UNION: Combina los resultados de las dos consultas. Elimina los valores repetidos

Select… from …where

UNION

Select …from..where

Select nombre from alumnos UNION select nombre from nuevos;

* UNION ALL : Visualiza también las filas duplicadas
* INTERSECT: Devuelve las filas que sean comunes en ambas tablas

Select… from …where

INTERSECT

Select …from..where

* MINUS: Devuelve las filas que están en la primera select y no están en la segunda

Select… from …where

MINUS

Select …from..where

A veces los programadores prefieren utilizar los operadores IN,AND, OR en lugar de los vistos anteriormente.

Select nombre from alumnos where nombre in (select nombre from nuevos)

En este caso anterior estoy haciendo una intersección

* Reglas de utilización de operadores de conjuntos
  1. Las columnas de las dos consultas se relacionan en orden , de izquierda a derecha
  2. Los nombres de columna de la primera select no tiene porque ser los mismos que los nombres de la segunda select
  3. La select necesitan tener el mismo número de columnas
  4. Los tipos de datos deben coincidir, aunque la longitud no tiene porque ser la misma.

## 6.Manipulación de datos

El lenguaje de manipulación de datos o DML (*Data Manipulation Language*) es una parte esencial de SQL. Cuando se quiere agregar, actualizar o eliminar datos, se ejecuta una sentencia DML. Un conjunto de sentencias DML que aún no se han hecho permanentes se denomina **transacción** o unidad lógica de trabajo.

Imagine una base de datos de un banco. Cuando un cliente del banco transfiere dinero de una cuenta a otra, la transacción consiste en tres operaciones separadas: disminuir el saldo, incrementar el de la otra cuenta y registrar la transacción. Oracle debe garantizar que las tres sentencias SQL se ejecutan para mantener las cuentas con el saldo adecuado. Cuando una de las sentencias no puede ejecutarse, el resto de las sentencias deben deshacerse.

## La sentencia INSERT

Permite añadir nuevos registros a una tabla.

Sintaxis:

INSERT INTO tabla [columna [, columna ...])]

VALUES (valor [, valor ...]);

Mediante esta sintaxis, sólo se inserta un registro o fila al mismo tiempo.

Ejemplo:

INSERT INTO DEPT (deptno, dname, loc)

VALUES (50, ‘DESARROLLO’, ‘MADRID’);

Es preciso listar los valores en el orden por defecto de las columnas de la tabla.

Opcionalmente se pueden listar las columnas en las cláusula INSERT.

Si se desean insertar valores nulos, se pueden utilizar dos métodos:

* Método implícito: omitir la columna en la lista.

INSERT INTO DEPT (deptno, dname)

VALUES (60, ‘MI DEPARTAMENTO’);

* Método explícito: especificar la palabra clave NULL.

INSERT INTO DEPT

VALUES (60, ‘MI DEPARTAMENTO’, NULL);

Antes hay que asegurarse que la columna permite valores nulos (comprobarlo con DESCRIBE).

Ejemplos:

INSERT INTO EMP (empno, ename, job, mgr, hiredate, sal, comm, deptno)

VALUES (7196, ‘HIGUERAS’, ‘SALESMAN’, 7782, SYSDATE, 2000, NULL, 10);

INSERT INTO EMP

VALUES (2296, ‘ORTEGA’, ‘SALESMAN’, 7782,

TO\_DATE (‘FEB 3, 97’, ‘MON DD, YY’), 1300, NULL, 10);

**Copia de registros de otra tabla**

Se pueden usar la sentencia INSERT para agregar filas a una tabla donde los valores se derivan de otras tablas ya existentes. En lugar de la cláusula VALUES, utilizamos una subconsulta.

INSERT INTO JEFES (id, nombre, salario, alta)

SELECT empno, ename, sal, hiredate FROM EMP WHERE job = ‘MANAGER’;

Cuidar la concordancia de columnas de la cláusula INSERT y las de la subconsulta.

## La sentencia UPDATE

Permite modificar los registros existentes. Puede afectar a más de un registro al mismo tiempo.

Sintaxis:

UPDATE tabla

SET columna = valor [, columna = valor]

[WHERE condición];

Los registros a modificar se especifican en la cláusula WHERE.

UPDATE EMP

SET deptno = 20

WHERE empno = 7782;

Si se omite la cláusula WHERE, se modificarán **todos** los registros de la tabla.

UPDATE EMPLEADOS

SET deptno = 20;

Ejemplos:

UPDATE EMP

SET (job, deptno) = (SELECT job, deptno

FROM EMP

WHERE empno =7499)

WHERE empno = 7698;

UPDATE EMP

SET deptno = 55

WHERE deptno = 10;

SQL>> ERROR: violación de restricción de integridad (el departamento 55 no existe en la tabla padre DEPT).

## La sentencia DELETE

Permite borrar registros de una tabla.

Sintaxis:

DELETE [FROM] tabla

[WHERE condición];

Los registros a eliminar se especifican en la cláusula WHERE.

DELETE FROM DEPARTAMENTOS

WHERE dname = ‘DEVELOPMENT’;

Si se omite la cláusula WHERE se borrarán **todos** los registros de la tabla.

DELETE FROM DEPARTAMENTOS;

Ejemplos:

DELETE FROM EMPLEADOS

WHERE deptno = (SELECT deptno

FROM DEPT

WHERE dname = ‘SALES’);

DELETE FROM DEPT

WHERE deptno = 10;

SQL>> ERROR: violación de restricción de integridad (no se puede eliminar un registro que contiene una clave primaria, usada como clave extranjera en otra tabla).

## 7.Transacciones en la base de datos

El servidor Oracle asegura la consistencia de los datos basándose en transacciones. Las transacciones otorgan más flexibilidad y control cuando cambian los datos y aseguran en los datos ante un fallo eventual en el proceso del usuario o un fallo del sistema.

Las transacciones consisten de comandos DML que llevan a cabo un cambio en los datos de manera consistente. Por ejemplo, una transferencia de fondos entre dos cuentas, debiera incluir un crédito en una cuenta y un débito en la otra por la misma cantidad. Ambas acciones debieran tener éxito o fallar como una sola operación. El crédito no debería confirmarse sin el débito correspondientes.

Tipos de transacciones:

* Un conjunto de sentencias DML que Oracle tratará como una unidad simple
* Una única sentencia DDL
* Una única sentencia DCL

Las transacciones comienzan cuando se ejecuta el primer comando SQL.

Las transacciones finalizan con uno de los siguientes eventos:

* COMMIT o ROLLBACK
* Ejecución de un comando DCL o DCL (COMMIT automático)
* Salida del usuario
* Caída del sistema

Las sentencias COMMIT y ROLLBACK aseguran la consistencia de los datos, permiten visualizar los cambios sobre los datos antes de hacerlos permanentes y agrupan lógicamente tareas relacionadas entre sí.

Forma

El contenido generado por IA puede ser incorrecto.

* *COMMIT*: finaliza la transacción actual haciendo permanente todos los cambios
* *SAVEPOINT nombre*: Marca un Savepoint en la transacción actual
* *ROLLBACK [TO SAVEPOINT nombre]*: ROLLBACK finaliza la transacción actual deshaciendo todos los cambios pendientes. ROLLBACK TO SAVEPOINT deshace la transacción actual hasta el Savepoint especificado, deshaciendo por lo tanto el Savepoint y todos los cambios posteriores. Si se omite esta cláusula, la sentencia ROLLBACK deshace la transacción entera.

Se produce un COMMIT automático cuando se aplica un comando DDL o DCL o una salida normal desde SQL\*Plus (sin realizar un COMMIT o ROLLBACK explícitamente) con el comando quit.

SQL\*Plus dispone de un tercer comando: AUTOCOMMIT, que puede tomar valores ON u OFF. Si es ON, cada sentencia DML individual es confirmada (se le hace COMMIT) tan rápido como es ejecutada. Los cambios no son reversibles (sin ROLLBACK).

Se produce un ROLLBACK automático cuando ocurre una terminación anormal de SQL\*Plus o un fallo del sistema. Esto previene que el error cause cambios no deseados a los datos y devuelve a las tablas su estado inicial desde el último COMMIT. De esta manera, SQL\*Plus protege la integridad de las tablas.

**Estado de los datos antes de COMMIT o ROLLBACK**

* El estado previo de los datos puede ser recuperado.
* El usuario actual puede revisar los resultados de sus operaciones DML usando la sentencia SELECT.
* Otros usuarios no pueden ver los resultados de las sentencias DML ejecutadas por el usuario actual.
* Las filas afectadas son bloqueadas; otros usuarios no pueden cambiar los datos pertenecientes a esas filas.

**Estado de los datos después de COMMIT**

* Los cambios se hacen permanentes en la base de datos.
* Los datos anteriores se pierden definitivamente.
* Todos los usuarios pueden ver los resultados.
* Se liberan los bloqueos aplicados a las filas afectadas; esas filas están ahora disponibles para que otros usuarios las manipulen.
* Se borran todos los SAVEPOINTS.

Ejemplos:

UPDATE EMP SET deptno = 10 WHERE empno = 7782;

INSERT INTO DEPT VALUES (60, ‘MARKETING’, ‘BARCELONA’);

COMMIT;

DELETE FROM TEST;

ROLLBACK;

DELETE FROM TEST WHERE ID = 100;

SELECT \* FROM TEST WHERE ID = 100;

COMMIT;

**Hacer ROLLBACK hasta una marca**

Se puede crear una marca en una transacción usando el SAVEPOINT. En consecuencia, la transacción puede dividirse en dos secciones más pequeñas. Esto permite que se descarten los cambios hechos hasta la marca usando la sentencia ROLLBACK TO SAVEPOINT.

Si se crea un segundo SAVEPOINT con el mismo nombre que un SAVEPOINT anterior, este último es eliminado.

Ejemplo:

UPDATE DEPARTAMENTOS SET loc = ‘MIAMI’ WHERE deptno = 40;

SAVEPOINT update\_hecho;

INSERT INTO DEPARTAMENTOS VALUES (80, ‘PUBLICIDAD’, ‘PARIS’);

ROLLBACK TO update\_hecho;

* Si una única sentencia DML falla durante su ejecución, entonces se hace ROLLBACK sobre esa sentencia solamente.
* El usuario debería terminar explícitamente las transacciones usando una sentencia COMMIT o ROLLBACK.

**Consistencia en Lectura**

Los usuarios de la base de datos realizan dos tipos de acceso a la base de datos:

* Operaciones de lectura (sentencia SELECT)
* Operaciones de escritura (sentencias INSERT, UPDATE, DELETE)

La consistencia en lectura es necesaria para que:

* Las lecturas y escrituras a la base de datos aseguren una vista consistente de los datos
* Las lecturas no vean datos que están en proceso de cambio
* Las escrituras aseguren que los cambios a la base de datos se hacen de forma consistente
* Los cambios realizados por una escritura no crean conflictos con los cambios que otra escritura está realizando

El propósito de la consistencia en lectura es asegurar que cada usuario ve los datos tal como están desde el último COMMIT, antes de que comience una operación DML.

**Implementación de la consistencia en lectura**

La consistencia en lectura es una implementación automática. Mantiene una copia parcial de la base de datos en segmentos de rollback. Cuando una operación DML se realiza contra la base de datos, el servidor Oracle hace una copia de los datos antes de su cambio y la escribe en un segmento de rollback.

Todas las lecturas, excepto aquella que realiza el cambio, ven la base de datos tal como estaba; ven la imagen de los segmentos de rollback, que es como una foto de los datos.

Antes de hacer el COMMIT de los cambios en la base de datos, sólo el usuario que modifica los datos ve los cambios; el resto lee la imagen del segmento de rollback. Esto garantiza que las lecturas son consistentes. Al hacer COMMIT, el cambio realizado a la base de datos se convierte en visible a cualquiera que ejecute una sentencia SELECT.

Si la transacción experimenta un rollback, los cambios “se deshacen”:

* La versión original, antigua de los datos, en el segmento de rollback se escriben de nuevo a la tabla
* Todos los usuarios ven la base de datos tal como estaba antes de comenzar la transacción.

**Bloqueos**

Los bloqueos son mecanismos que previenen conflictos entre transacciones, que acceden a los mismos recursos, bien sea un objeto de usuario (como tablas o registros), o un objeto del sistema no visible a los usuarios (como estructuras de datos compartidas y registros del diccionario de datos).

El bloqueo en Oracle es completamente automático y no requiere acción por parte del usuario. El bloqueo implícito ocurre para todas las sentencias SQL. El bloqueo por defecto de Oracle es un mecanismo automático que utiliza el nivel más bajo aplicable de restricción, portano, ofrece el grado más alto de concurrencia y máxima integridad de datos. Oracle también permite que el usuario bloquee los datos manualmente.

Oracle utiliza dos modelos de bloqueo:

* Exclusivo: previene la compartición de un recurso. La primera transacción bloquea un recurso exclusivamente, es la única que puede alterarlo, hasta liberar el recurso.
* Compartido: permite la compartición de un recurso. Múltiples usuarios leyendo datos pueden compartir los datos, manteniendo bloqueos para prevenir acceso concurrente por una escritura (que necesita un bloqueo exclusivo). Varias transacciones pueden adquirir bloqueos compartidos sobre el mismo recurso.

### PARTE 6 INTRODUCCION

* PL/SQL es la extensión estructurada y procedimental (permite crear funciones y procedimientos) del lenguaje SQL implementada por Oracle junto a la versión 6.
* Con los scripts de SQL se tienen limitaciones como uso de variables, modularidad,etc.
* Con PL/SQL se pueden usar sentencias SQL para acceder a bases de datos Oracle y sentencias de control de flujo para procesar los datos, se pueden declarar variables y constantes, definir procedimientos, funciones, subprogramas, capturar y tratar errores en tiempo de ejecución, etc...
* En un programa escrito en PL/SQL se pueden utilizar sentencias SQL de tipo LMD (Lenguaje de Manipulación de Datos) directamente y sentencias de tipo LDD (Lenguaje de Definición de Datos) mediante la utilización de paquetes.
* Oracle incorpora un gestor PL/SQL en el servidor de bbdd y en las principales herramientas, Forms , Reports, Graphics, etc.
* Basado en Ada incorpora todas las características propias de los lenguajes de tercera generación; manejo de variables, estructura modular (procedimientos y funciones) , estructuras de control, control de excepciones,etc
* El código PL/SQL puede estar almacenado en la base de datos (procedimientos, funciones, disparadores y paquetes) facilitando el acceso a todos los usuarios autorizados.
* La ejecución de los bloques PL/SQL puede realizarse interactivamente desde herramientas como SQL\*Plus, Oracle Forms, etc..., o bien, cuando el S.G.B.D. detecte determinados eventos, también llamados disparadores.
* Nosotros vamos a utilizarlos en SQL Worksheet de Oracle Live

SQL para poder probar nuestro Pl/SQL

* Los programas se ejecutan en el servidor, con el consiguiente ahorro de recursos en los clientes y disminución de tráfico en la red.

### ARQUITECTURA DE PL/SQL

La arquitectura de PL/SQL consiste principalmente en los tres componentes que se citan a continuación:

1. El bloque PL/SQL
2. El motor PL/SQL
3. El servidor de la base de datos

El bloque PL/SQL:

* + Es el componente que contiene el código de pl/sql
  + Consiste en distintas secciones que forman la lógica del bloque
  + También contiene instrucciones de sql que interactuar con el servidor de base de datos
  + Hay diferentes tipos de bloques o Units de Pl/sql y son :

1. Bloques anonimos
2. Procedimientos
3. Paquetes
4. Trigger

#### El motor PL/SQL

Es el componente donde se procesa el código

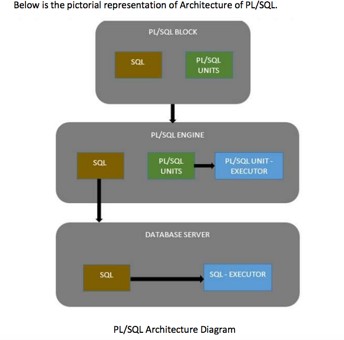
El motor PL/SQL separa las unidades de PL/SQL y las partes de SQL y este motor será el encargado de manejar los bloques PL/SQL

La parte de SQL será enviada al servidor donde interactua con la base de datos

#### El servidor de base de datos

Es la parte más importante pues es la que contiene los datos

El motor de The PL/SQL usa el SQL de los bloques PL/SQL para interactuar con el servidor de base de datos



Diferencias entre SQL y PL/SQL

|  |  |
| --- | --- |
| sql | pl/sql |
| SQL es una consulta sencilla usada para desarrollar operaciones DML and DDL. | PL/SQL es un bloque de codigo usado para escribir programas enteros con procedimientos, bloques y funciones |
| Es declarativo, es decir, se dice que cosas queremos más que como se hacen | Es prodedimental, por tanto, decimos como se hacen las cosas. |
| Ejecuta una sentencia sencilla | Ejecuta todo un bloque |
| Interactua con el servidor de la base de datos | No interactua con les hervor de base de datos |
| No puede contener código PL/SQL | Contiene SQL en sus bloques |

### CARACTERÍSTICAS DEL LENGUAJE. BLOQUES

En PL/SQL, tel código no es ejecutado como una linea, siempre es ejecutado como un grupo de sentencia denominado BLOQUE.

Los bloques contienen instrucciones tanto de PL/SQL como de SQL. La estructura básica del lenguaje es el bloque.

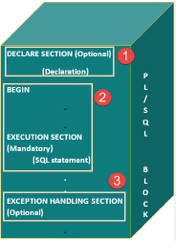
Todos los programas se escriben en bloques de código que se dividen en diferentes secciones independientes.

Se pueden anidar bloques formando estructuras de programación más complejas

**BLOQUES ANÓNIMOS**: La estructura de un bloque sin nombre es la siguiente:

[DECLARE

Declaraciones;] BEGIN

Sentencias; [EXCEPTION

Excepciones;]

END;

* El bloque consta de tres secciones:

**DECLARE**: Sección de declaración. En ella se declaran todos los objetos que vayamos a usar en el programa

Es opcional, salvo cuando se deba realizar algún tipo de declaraciones de

elementos, tales como variables, constantes, cursores, etc… Si se pone esta sección debe ser la primera.

Solo se utiliza en Bloques anónimos y en triggers pero los procedimientos no tienen esta parte Declare.

##### BEGIN:

Sección que contiene las sentencias que se van a ejecutar.

Es la única parte obligatoria , ya que es la parte ejecutable. Comienza con la palabra BEGIN y finaliza con END;

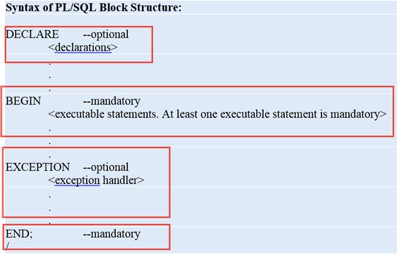
En esta sección es donde especificamos el código que se va a ejecutar

##### EXCEPTION:

Sección que contiene las sentencias para el manejo de errores. También contiene sentencias PL/SQL .

Es opcional, salvo cuando tengamos la necesidad de tratar los errores. Normalmente se utiliza para capturar errores ya predefinidos y hay que especificar las acciones a realizar en caso de que se produzca un error.

Por último se especifica END que indica final del bloque. Especifica final de la zona ejecutable aunque vaya después de la zona de excepciones cuando la incluya.



* Los bloques pueden contener sub-bloques, es decir, podemos tener bloques anidados. Las anidaciones se pueden realizar en la parte ejecutable y en la de manejo de excepciones, pero no en la declarativa.

[DECLARE

Declaraciones;] BEGIN

[DECLARE

Declaraciones;] BEGIN

Sentencias: [EXCEPTION

Excepciones;]

END; [EXCEPTION

Excepciones;]

END;

* A continuación tenemos un ejemplo de bloque PL/SQL:

DECLARE

**sueldo NUMBER(8); BEGIN**

**SELECT salario INTO sueldo FROM plantilla**

**WHERE apellido LIKE 'MORENO';**

**IF sueldo < 100000 THEN sueldo := sueldo + 20000;**

##### END IF;

**UPDATE plantilla**

**SET salario = sueldo WHERE apellido LIKE 'MORENO'; COMMIT;**

##### END;

**/**

**EJEMPLO UPDATE CON NULL**

**UPDATE EMP**

**SET SAL = NVL(SAL, 0) \* (1 + X\_PORCENTAJE / 100)**

**WHERE EMPNO = N\_EMPLEADO;**

**CONTROLAR NULLCON VL**

* Con este programa actualizamos el sueldo del empleado con apellido ‘MORENO’, incrementándolo en 20.000 en caso de que dicho sueldo no supere 100.000 pesetas.
* El programa tiene parte declarativa, en la que se define una variable, y parte ejecutable.
* Podemos observar que hay sentencias SQL, como SELECT y UPDATE, sentencias de control como IF... THEN y sentencias de asignación. El símbolo ‘:=’ representa el operador de asignación.
* El bloque de código se ejecuta al encontrar el operador de ejecución ‘/’.

### TIPOS DE BLOQUES

Se reconocen dos tipos de bloques en PL/SQL:

* + Bloques anónimos: Bloques sin nombre
  + Bloques nombrados: Que a su vez se subdividen en procedimientos y funciones.

Bloques anonimos:

Son bloques que no tienen nombre y necesitan construirse en una sesión para probar algún código.

Al no tener nombre no serán almacenados en la base de datos.

Son escritos y ejecutados directamente y se compilan y ejecutan en el mismo proceso. Bloques nombrados:

Los bloques con nombre son almacenados como objetos en la base de datos.

Como son guardados en el servidor , podremos invocarlos y ejecutarlos cuando lo necesitemos.

La compilación se produce cuando los creamos y después se ejecutaran en caso de ser necesario.

Características de estos bloques:

* Pueden ser invocados por otros bloques
* Pueden tener otros bloques anidados como dijimos antes
* Se dividen en Procedimientos y Funciones, que se verán mas adelante.

**TIPOS DE DATOS**

###### ¿Qué son los tipos de datos PL/SQL?

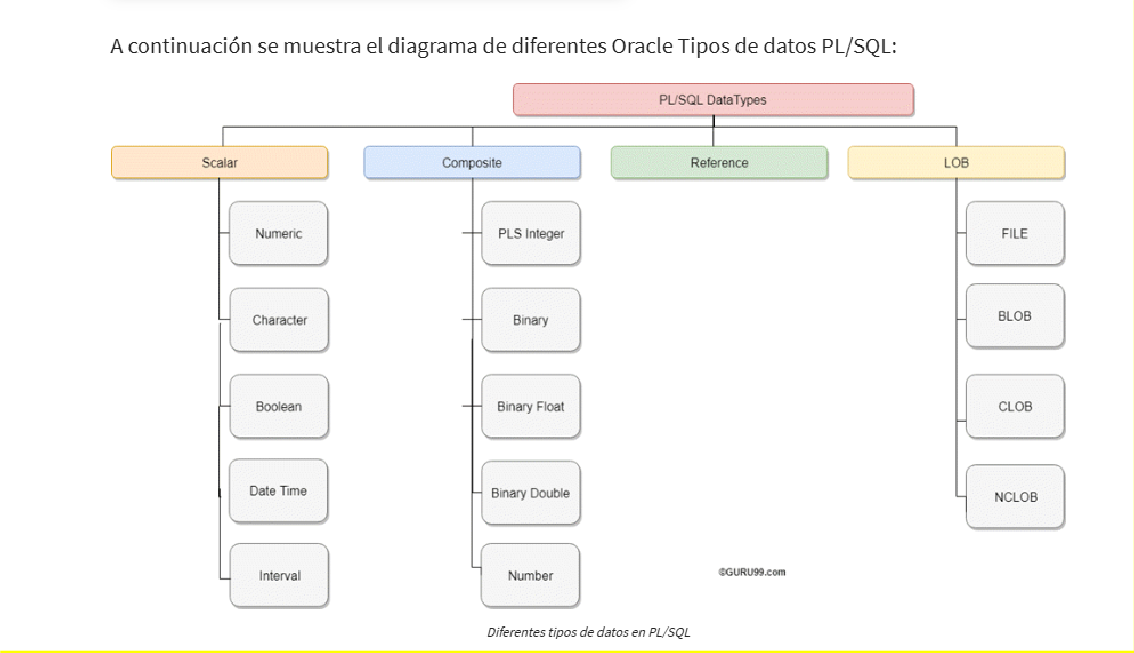
**Tipos de datos** en PL/SQL se utilizan para definir cómo los datos serán

almacenados, manejados y tratados por Oracle durante el almacenamiento y procesamiento de datos. Los tipos de datos están asociados con el formato de

almacenamiento específico y las restricciones de rango. En Oracle, a cada valor o constante se le asigna un tipo de datos.

La principal diferencia entre PL/SQL y [SQL](https://www.guru99.com/es/sql.html) tipos de datos es, el tipo de datos SQL se limita a la columna de la tabla, mientras que los tipos de datos PL/SQL se

utilizan en la [bloques PL/SQL](https://www.guru99.com/es/blocks-pl-sql.html)



###### Tipo de datos de carácter PL/SQL

Este tipo de datos básicamente almacena caracteres alfanuméricos en formato de cadena.

Los valores literales siempre deben estar entre comillas simples al asignarlos al tipo de datos CHARACTER.

###### Este tipo de datos de carácter se clasifica además de la siguiente manera:

* + CHAR Tipo de datos (tamaño de cadena fijo)
  + VARCHAR2 Tipo de datos (tamaño de cadena variable)
  + VARCHAR Tipo de datos
  + NCHAR (tamaño de cadena fijo nativo)

|  |  |  |
| --- | --- | --- |
| **Tipo de datos** | **Descripción** | **Sintaxis** |
| **CHAR** | Este tipo de datos almacena el valor de la cadena y el tamaño de la cadena se fija en el momento de  declarar el [variable](https://www.guru99.com/es/pl-sql-identifiers.html).   * Oracle La variable se rellenaría en blanco si la variable no ocupara todo el tamaño que se ha declarado para ella, por lo tanto Oracle   asignará la memoria para el tamaño  declarado incluso si la variable no la ocupó por completo.   * La restricción de tamaño para este tipo de datos es de 1 a 2000 bytes. * El tipo de datos CHAR es más apropiado para usar siempre que se maneje el tamaño de datos fijo. | grade CHAR;  manager CHAR (10):= 'guru99';  **Explicación de sintaxis:**   * La primera declaración declaró la   variable "grado" del tipo de datos CHAR con el tamaño máximo de 1 byte (valor predeterminado).   * La segunda declaración declaró la variable 'administrador' del tipo de   datos CHAR con el tamaño máximo de 10 y asignó el valor 'guru99' que es de 6 bytes. Oracle asignará la memoria de 10 bytes en lugar de 6 bytes en este caso. |
| **VARCHAR2** | Este tipo de datos almacena la cadena, pero la longitud de la cadena no es fija.   * La restricción de tamaño para este tipo de datos es de 1 a 4000 bytes para el tamaño de la columna de la tabla y de 1 a 32767 bytes para las variables. * El tamaño se define para cada variable en el momento de la declaración de la variable. * Pero Oracle asignará memoria sólo después de que se defina la variable, es decir, Oracle considerará solo la longitud real de la cadena que está almacenada en una variable para la asignación de memoria en lugar del tamaño que se ha dado para una variable en la parte de declaración. * Siempre es bueno utilizar VARCHAR2 en lugar del tipo de datos CHAR para optimizar el uso de la memoria. | manager VARCHAR2(10) := ‘guru99';  **Explicación de sintaxis:**   * La declaración anterior declaró la variable 'administrador' del tipo de datos VARCHAR2 con el tamaño máximo de 10 y le asignó el valor 'guru99' que es de 6 bytes. Oracle   asignará memoria de sólo 6 bytes en este caso. |
| **VARCHAR** | Esto es sinónimo del tipo de datos VARCHAR2. | manager VARCHAR(10) := ‘guru99';  **Explicación de sintaxis:** |

|  |  |  |
| --- | --- | --- |
| **Tipo de datos** | **Descripción** | **Sintaxis** |
|  | * Siempre es una buena práctica utilizar   VARCHAR2 en lugar de VARCHAR para evitar cambios de comportamiento. | * La declaración anterior declaró la variable 'administrador' del tipo de   datos VARCHAR con el tamaño máximo de 10 y asignó el valor 'guru99' que es de 6 bytes. Oracle asignará memoria de  sólo 6 bytes en este caso. (Similar a VARCHAR2) |

* + NVARCHAR2 (tamaño de cadena variable nativa)
  + LARGO y LARGO CRUDO

|  |  |  |
| --- | --- | --- |
| **NCHAR** | Este tipo de datos es el mismo que el tipo de datos CHAR, pero el juego de caracteres será el juego de caracteres nacional.   * Este juego de caracteres se puede definir para la sesión utilizando NLS\_PARAMETERS. * El juego de caracteres puede ser UTF16 o UTF8. * La restricción de tamaño es de 1 a 2000 bytes. | native NCHAR(10);  **Explicación de sintaxis:**   * La declaración anterior declara la variable 'nativa' del tipo de datos NCHAR con un tamaño máximo de 10. * La longitud de esta variable depende del (número de longitudes) por byte tal como se define en el juego de caracteres. |
| **NVARCHAR2** | Este tipo de datos es el mismo que el tipo de datos VARCHAR2, pero el juego de caracteres será el juego de caracteres nacional.   * Este juego de caracteres se puede definir para la sesión utilizando NLS\_PARAMETERS. * El juego de caracteres puede ser UTF16 o UTF8. * La restricción de tamaño es de 1 a 4000 bytes. | Native var NVARCHAR2(10):='guru99';  **Explicación de sintaxis:**   * La declaración anterior declara la variable 'Native\_var' del tipo de datos NVARCHAR2 con un tamaño máximo de 10. |
| **LARGO y LARGO** | Este tipo de datos se utiliza para almacenar texto grande o datos sin procesar hasta un tamaño máximo de 2 GB. | Large\_text LONG; Large\_raw LONG RAW;  **Explicación de sintaxis:** |

|  |  |  |
| --- | --- | --- |
|  | * Se utilizan principalmente en el diccionario de datos. * El tipo de datos LONG se utiliza para almacenar datos del juego de caracteres, mientras que LONG RAW se utiliza para almacenar datos en formato binario. * El tipo de datos LONG RAW acepta objetos multimedia, imágenes, etc., mientras que LONG solo funciona con datos que se pueden almacenar utilizando un juego de caracteres. | * La declaración anterior declara la variable 'Large\_text' del tipo de datos LONG y 'Large\_raw' del tipo de datos LONG RAW.   **Nota:** No se recomienda el uso del tipo de datos LARGO Oracle. En su lugar, se debe preferir |

###### NÚMERO PL/SQL Tipo de datos

Este tipo de datos almacena números de punto fijo o flotante de hasta 38 dígitos de precisión. Este tipo de datos se utiliza para trabajar con campos que contendrán solo datos numéricos. La variable se puede declarar con precisión y detalles de dígitos decimales o sin esta información. Los valores no necesitan estar entre comillas al asignarse para este tipo de datos.

A NUMBER(8,2);

B NUMBER(8);

C NUMBER;

###### Explicación de sintaxis:

* + En lo anterior, la primera declaración declara que la variable 'A' es de tipo de datos numéricos con precisión total 8 y dígitos decimales 2.
  + La segunda declaración declara que la variable 'B' es de tipo de datos numéricos con precisión total 8 y sin dígitos decimales.
  + La tercera declaración es la más genérica, declara que la variable 'C' es de tipo numérico sin restricción en precisión o decimales. Puede tener hasta un máximo de 38 dígitos.

###### Tipo de datos booleano PL/SQL

Este tipo de datos almacena los valores lógicos. Oracle El tipo de datos booleano representa VERDADERO o FALSO y se utiliza principalmente en declaraciones

condicionales. No es necesario que los valores estén entre comillas al asignarlos para este tipo de datos.

Var1 BOOLEAN;

###### Explicación de sintaxis:

* + En lo anterior, la variable 'Var1' se declara como tipo de datos BOOLEANO. La salida del código será verdadera o falsa según la condición establecida.

###### Tipo de datos de fecha PL/SQL

Este tipo de datos almacena los valores en formato de fecha, como fecha, mes y año. Siempre que una variable se define con el tipo de datos FECHA junto con la fecha, puede contener información de hora y, de forma predeterminada, la información de hora se establece en 12:00:00 si no se especifica. Los valores

deben estar entre comillas al asignarse para este tipo de datos.

El Oracle El formato de hora para entrada y salida es 'DD-MON-AA' y nuevamente se establece en NLS\_PARAMETERS (NLS\_DATE\_FORMAT) en el nivel de sesión.

newyear DATE:='01-JAN-2015';

current\_date DATE:=SYSDATE;

###### Explicación de sintaxis:

* + En lo anterior, la variable "año nuevo" se declara como tipo de datos FECHA y se le asigna el valor del 1 de enero.st, fecha 2015.
  + La segunda declaración declara la variable current\_date como tipo de datos DATE y le asigna el valor con la fecha actual del sistema.
  + Ambas variables contienen la información de tiempo.

###### Tipo de datos LOB PL/SQL

Este tipo de datos se utiliza principalmente para almacenar y manipular grandes bloques de datos no estructurados como imágenes, archivos multimedia, etc.

Oracle prefiere LOB en lugar del tipo de datos LONG ya que es más flexible que el tipo de datos LONG. Las siguientes son las principales ventajas del tipo de datos LOB sobre el tipo de datos LARGO.

* + El número de columnas en una tabla con tipo de datos LONG está limitado a 1, mientras que una tabla no tiene restricción en el número de columnas con tipo de datos LOB.
  + La herramienta de interfaz de datos acepta el tipo de datos LOB de la tabla durante la replicación de datos, pero omite la columna LARGA de la tabla. Estas columnas LARGAS deben replicarse manualmente.
  + El tamaño de la columna LARGA es de 2 GB, mientras que LOB puede almacenar hasta 128 TB.
  + Oracle está mejorando constantemente el tipo de datos LOB en cada una de sus versiones de acuerdo con los requisitos modernos, mientras que el tipo de datos LONG es constante y no recibe muchas actualizaciones.

Por lo tanto, siempre es mejor utilizar el tipo de datos LOB en lugar del tipo de datos LONG. A continuación, se muestran los diferentes tipos de datos LOB. Pueden almacenar hasta un tamaño de 128 terabytes.

1. BLOB
2. CLOB y NCLOB
3. ARCHIVOB

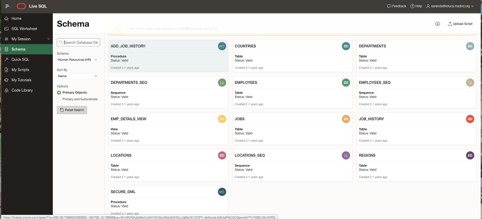
|  |  |  |
| --- | --- | --- |
| **Tipo de datos** | **Descripción** | **Sintaxis** |
| **BLOB** | Este tipo de datos almacena los datos LOB en formato de archivo binario hasta un tamaño máximo de 128 TB. No almacena datos basados en los detalles del conjunto de caracteres, por lo que  puede almacenar datos no estructurados, como objetos multimedia, imágenes, etc. | Binary\_data BLOB;  **Explicación de sintaxis:**   * En lo anterior, la   variable 'Binary\_data' se declara como BLOB. |
| **CLOB y NCLOB** | El tipo de datos CLOB almacena los datos LOB en el juego de  caracteres, mientras que NCLOB almacena los datos en el juego de caracteres nativo. Dado que estos tipos de datos utilizan  almacenamiento basado en juegos de caracteres, no pueden almacenar datos como multimedia, imágenes, etc. que no se  pueden poner en una cadena de caracteres. El tamaño máximo de estos tipos de datos es 128 TB. | Charac\_data CLOB;  **Explicación de sintaxis:**   * En lo anterior, la   variable 'Charac\_data' se declara como tipo de datos CLOB. |
| **ARCHIVOB** | * BFILE son los tipos de datos que almacenaron los datos en formato binario no estructurado fuera de la base de datos como un archivo del sistema operativo. * El tamaño de BFILE es para un sistema operativo limitado, son archivos de solo lectura y no se pueden modificar. |  |

### ENTORNO DE DESARROLLO

* Existen diferentes entornos de desarrollo para PL/SQL. Los principales son:
  + SQL\*PLUS(ISQLPLUS)
  + Oracle Developer Procedure Builder (herramienta de desarrollo)
  + El primero utiliza el motor en el servidor Oracle y el segundo cuenta con las dos opciones disponibles.
  + Dispondremos de una opciones u otras en relación a la construcción de programas con bloques:
    - Un bloque anónimo se puede utilizar en cualquier entorno PL/SQl. Son un conjunto de instrucciones que se ejecutan en modo local. Normalmente se utilizan para hacer pruebas
    - Bloque nominado, es igual pero tiene una etiqueta. También se ejecuta en modo local

### ORACLE LIVE SQL

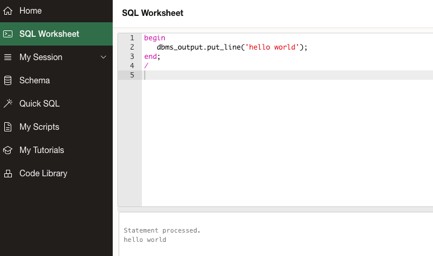
Dada la situación actual vamos a utilizar Live SQL de orca para probar nuestro código. En este entorno entras a SQL



Como veis en sql Worksheet podemos introducir nuestro código sql y PL/sql.

Os he hecho una captura del las tablas del usuario o esquema HR que viene como ejemplo en Oracle

Como vemos a continuación , acabamos de ejecutar un bloque anónimos que visualiza un “Hello World” de prueba.



Ejemplo:

declare

l\_today date := sysdate; begin

dbms\_output.put\_line(

'today is '||to\_char(l\_today,'Day')); exception when others then

dbms\_output.put\_line(sqlerrm); end;

/

El código anterior nos visualiza que hoy es miércoles (wednesday) y saltaría una excepción que nos visualiza un código de error , si se produjese algún error.

### ASIGNACION DE VALORES A VARIABLES.

* La asignación de valores a una variable en la parte ejecutable de un bloque se puede realizar a través del operador := pero también con la cláusula INTO de las sentencias SELECT y con FETCH (Se estudiará posteriormente en el tema).

Ejemplos:

total := unidades \* pts\_unidad;

SELECT salario INTO sueldo FROM plantilla WHERE apellido LIKE 'MORENO';

FECHT cursor1 INTO nombre;

Para empezar a familiarizarnos con el SQL dentro de un bloque PL/SQL vemos que las Select ya no hacen la salida en un terminal, sino que lo que recuperan lo almacenan en la variable que pongamos detrás del Into.

En este caso recuerda de la base de datos el sueldo de “MORENO2 y lo que recupere lo guarda en la variable sueldo.

* + También es posible realizar asignaciones de valor a las variables en el momento de su declaración, para ello se utilizará la palabra clave DEFAULT, o bien, el operador de asignación.

Ejemplos:

importe NUMBER (9) DEFAULT 3000;

nombre VARCHAR2(20) := ‘Marta Parrado’;

### EXPRESIONES.

* Una expresión está formada por un conjunto de operando unidos por operadores, los cuales ya fueron estudiados en el Tema 3.
* Cabe mencionar algún operador que incorpora PL/SQL y la prioridad de los operadores:

|  |  |
| --- | --- |
| Operador | Operación |
| \*\*,NOT | Exponenciacion , negación |
| +,- | Operadores unario ( posi t i vo, negativo) |
| \*./ | Multiplicacion,división |
| +,-, || | Suma, resta, concatenación |
| = , ! = , < > , > , < , > = , < = , I S NULL,LIKE,BETWEEN, IN | Operadores de comparación |
| AND | Conjunción |
| OR | Inclusión |

### VARIABLES Y CONSTANTES.

* PL/SQL permite declarar e inicializar variables y/o constantes.
* Las variables pueden ser de cualquiera de los tipos vistos en SQL o de otros tipos que veremos.
* Todas las variables han de ser declaradas antes de ser utilizadas
* Las variables se pueden utilizar:
  + Para almacenar temporalmente datos de entrada hasta que ya no se necesiten
  + Para realizar operaciones sobre los datos
  + Facilidad de mantenimiento: %ROWTYPE,%TYPE
* Para declarar una variable utilizaremos el siguiente formato: Nombre\_variable [CONSTANT] tipo [NOT NULL]

[{:=| DEFAULT } expresión ];

* Las palabras clave NOT NULL especifican que la variable no podrá tomar valor nulo.
* La palabra clave DEFAULT permite inicializar una variable a la vez que se define.
* Expresión podrá ser un literal, otra variable o el resultado de una expresión que puede incluir operadores y funciones.
* Para los identificadores de las variables se deben utilizar las mismas reglas que para los objetos de SQL
  + Longitud máxima: 30 caracteres
  + Letras, números y caracteres solo $, \_, #
  + El nombre debe empezar por una letra
  + No puede ser una palabra reservada de Oracle
* Para asignar valores por defecto podemos utilizar indistintamente := o DEFAULT
* Una variable no inicializada queda con valor NULL hasta asignarle valor en la ejecución
* Las constantes y variables declaradas como NOT NULL deben ser inicializadas
* Dos objetos pueden tener el mismo nombre siempre y cuando estén declarados en distintos bloques
* No se debe poner el mismo nombre a un identificador de variable que a la columna de la que vaya a recibir los datos. Esto evita confusiones y facilita el mantenimiento del software

Ejemplos:

importe NUMBER (9);

nombre VARCHAR2(20) NOT NULL;

descuento NUMBER(4,2) NOT NULL DEFAULT 10.27;

nombre char(20) NOT NULL := `MIGUEL’;

casado BOOLEAN DEFAULT FALSE;

* Podemos utilizar el atributo %TYPE para dar a una variable el tipo de dato de otra variable o de una columna de la base de datos.

#### Nom\_variable tabla.columna%type;

Ejemplo:

importe NUMBER (9);

total importe %TYPE;  Declarada total como NUMBER(9)

nuevonombre emp.nombre%TYPE -> Declara nuevonombre del mismo tipo que la columna nombre de emp

* La principal ventaja del uso de este atributo es la facilidad de mantenimiento de los programas, ya que ante cualquier cambio de tipos en la base de datos no hay que hacer ningún cambio en el código Pl/SQL
* %TYPE también puede utilizarse para declara una variable del mismo tipo que otra declarada previamente.

###### Si una columna tiene la restricción NOT NULL, la variable definida de ese tipo mediante el atributo %TYPE, no asume dicha restricción, pudiendo, por lo tanto, ponerla a NULL, a no ser que se especifique lo contrario en la declaración.

Ejemplo: planta vecinos.piso %TYPE NOT NULL; Declarada como NUMBER(2) y no admitiendo valores nulos.

* También podemos utilizar %ROWTYPE que crea una variable tipo registro para cargar los valores de toda una fila.

Mifila emp%ROWTYPE

#### Variable tabla%ROWTYPE;

* Se pueden declarar constantes utilizando la palabra reservada CONSTANT, pero teniendo en cuenta que la constante se debe inicializar en el momento de la declaración.
* El valor de una constante, una vez determinada no modificarse. La declaración de constantes facilita la modificación de programas que contienen datos constantes.
* La declaración de una constante es igual a la de una variable, salvo que hay que utilizar CONSTANT e inicializarla.

Ejemplo: iva CONSTANT NUMBER(2) DEFAULT 16;

* En el ejemplo anterior, si no declaramos la constante ‘iva’, un cambio en el porcentaje del impuesto obligaría a cambiarlo en todas las expresiones donde apareciese.
* Cualquier variable o constante debe ser declarada antes de ser referenciada en otra declaración o sentencia, ya que en caso contrario se produciría un error.

Ejemplo: total importe %TYPE;  Error importe NUMBER(9,2);

Procedimientos y Funciones PL/SQL

# INTRODUCCION

* Los procedimientos y funciones son subprogramas que pueden ser invocados por los usuarios.
* En PL/SQL el desarrollador puede definir sus propios subprogramas o bien utilizar

funciones predefinidas en PL/SQL.

* Los subprogramas pueden ser de dos tipos:
  + Funciones: subprogramas que devuelven un valor
  + Procedimientos: subprogramas que ejecutan una secuencia de instrucciones pero que el nombre del subprograma en si mismo no devuelve un valor.

# FUNCIONES PREDEFINIDAS PL/SQL

PL/SQL proporciona un gran número de funciones muy útiles para ayudar a manipular la información y permite incorporar en sus expresiones casi todas las funciones disponibles en SQL. Se pueden agrupar en categorías:

* caracteres
* numéricas
* fechas
* conversión de tipo de datos
* manejo de nulos
* misceláneas
* error-reporting

Las funciones de agrupación de SQL como por ejemplo AVG, MIN, MAX, COUNT, SUM, STDDEV, y VARIANCE, no están implementadas en PL/SQL, sin embargo se pueden usar en sentencias SQL. Tampoco se pueden usar algunas otras como DECODE, DUMP, y VSIZE.

A continuación vamos a ver algunas de las más utilizadas.

# FUNCIONES PREDEFINIDAS CARACTERES

* LENGTH: Devuelve la longitud de un tipo CHAR.

resultado := LENGTH('HOLA MUNDO’); -- Devuelve 10

* INSTR

Busca una cadena de caracteres (la que se indica en el segundo parámetro pasado) dentro de otra (la que se indica en el primer parámetro) y devuelve la posición de la ocurrencia de la cadena buscada dentro de la cadena. En el tercer parámetro se indica la posición desde la que se comienza a buscar (opcional) y en el cuarto el número de ocurrencia que se busca (opcional).

Su sintaxis es la siguiente: INSTR(<char>, <search\_string>, <startpos>, <occurrence> )

resultado := INSTR('AQUI ES DONDE SE BUSCA', 'BUSCA', 1, 1); -- Devuelve 18

# FUNCIONES PREDEFINIDAS CARACTERES

* REPLACE: Reemplaza un texto por otro en una cadena de caracteres. REPLACE(<expresion>, <busqueda>, <reemplazo>)

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

MUNDO'.

resultado := REPLACE ('HOLA MUNDO','HOLA', 'VAYA'); -- devuelve VAYA MUNDO

* SUBSTR: Obtiene una parte de una cadena de caracteres, desde una posición de inicio hasta una determinada longitud.

SUBSTR(<expresion>, <posicion\_ini>, <longitud> )

resultado := SUBSTR('HOLA MUNDO', 6, 5); -- Devuelve MUNDO

# FUNCIONES PREDEFINIDAS CARACTERES

* UPPER: Convierte una cadena alfanumérica a mayúsculas.

resultado := UPPER('hola mundo'); -- Devuelve HOLA MUNDO

* LOWER: Convierte una cadena alfanumérica a minúsculas.

resultado := LOWER('HOLA MUNDO'); -- Devuelve hola mundo

* RTRIM: Elimina los espacios en blanco a la derecha de una cadena de caracteres.

resultado := RTRIM ('Hola Mundo ');

* LTRIM: Elimina los espacios en blanco a la izquierda de una cadena de caracteres.

resultado := LTRIM (' Hola Mundo');

* TRIM: Elimina los espacios en blanco a la izquierda y derecha de una cadena de caracteres.

resultado := TRIM (' Hola Mundo ');

# FUNCIONES PREDEFINIDAS NUMERICAS

* MOD: Devuelve el resto de la división entera entre dos números. MOD(<dividendo>, <divisor> )

resultado := MOD(20,15); -- Devuelve el modulo de dividir 20/15

* TRUNC: Trunca un número y devuelve la parte entera.

resultado := TRUNC(9.99); -- Devuelve 9

* ROUND: Devuelve el entero más próximo.

resultado := ROUND(9.99); -- Devuelve 10

# FUNCIONES PREDEFINIDAS FECHAS

* SYSDATE: Devuelve la fecha del sistema.

resultado := SYSDATE;

* TRUNC: Trunca una fecha, elimina las horas, minutos y segundos de la misma.

resultado := TRUNC(SYSDATE);

# FUNCIONES PREDEFINIDAS CONVERSION TIPO DATOS

* TO\_DATE: Convierte una expresión al tipo fecha. TO\_DATE(<expresion>, [<formato>])

El parámetro opcional formato indica el formato de entrada de la expresión no el

de salida.

En este ejemplo se convierte la cadena de caracteres '01/12/2006' a una fecha (tipo DATE). El formato indica que la fecha está escrita como día/mes/año, de forma que la fecha sea el uno de diciembre y no el doce de enero.

resultado := TO\_DATE('01/12/2006', 'DD/MM/YYYY');

El siguiente ejemplo muestra la conversión con formato de día y hora.

resultado := TO\_DATE('31/12/2006 23:59:59', 'DD/MM/YYYY HH24:MI:SS');

# FUNCIONES PREDEFINIDAS CONVERSION TIPO DATOS

* TO\_CHAR: Convierte una expresión al tipo CHAR. TO\_CHAR(<expresion>, [<formato>])

El parámetro opcional formato indica el formato de salida de la expresión.

resultado := TO\_CHAR(SYSDATE, 'DD/MM/YYYYY HH24:MI:SS');

* TO\_NUMBER: Convierte una expresión alfanumérica en numérica, se puede especificar el formato de salida (opcional).

TO\_NUMBER(<expresion>, [<formato>])

resultado := TO\_NUMBER ('10.21', ’99.99’); -- resultado: 10,21 (el separador decimal es ,)

# FUNCIONES PREDEFINIDAS MANEJO DE NULOS

* NVL: Devuelve el valor recibido como parámetro en el caso de que expresión sea NULL o el valor de la 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;

# FUNCIONES PREDEFINIDAS MISCELANEAS

* 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 (el último parámetro).

Es muy común escribir la función DECODE indentada como si se tratase de un

bloque IF.

SELECT DECODE (co\_pais, /\* Expresion a evaluar \*/

FROM PAISES;

|  |  |  |
| --- | --- | --- |
| 'ESP', 'ESPAÑA', /\*  'MEX', 'MEXICO', /\* | Si co\_pais = 'ESP'  Si co\_pais = 'MEX' | ==> 'ESPAÑA' \*/  ==> 'MEXICO' \*/ |
| 'PAIS '||co\_pais)/\* | ELSE ==> concatena | \*/ |

# FUNCIONES PREDEFINIDAS

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Carácter** | **Numéricas** | **Fecha** | **Conversión** | **Manejo Nulos** | **Misceláneas** | **Error** |
| ASCII | ABS | ADD\_MONTHS | CHARTOROWID | NVL | DECODE | SQLCODE |
| CHR | ACOS | CURRENT\_DATE | CONVERT |  | DUMP | SQLERRM |
| CONCAT | ASIN | CURRENT\_TIMESTAM | HEXTORAW |  | GREATEST |  |
| INITCAP | ATAN | LAST\_DAY | NLS\_CHARSET\_ID |  | GREATEST\_LB |  |
| INSTR | ATAN2 | LOCALTIMESTAMP | NLS\_CHARSET\_NAME |  | LEAST |  |
| INSTRB | CEIL | MONTHS\_BETWEEN | RAWTOHEX |  | LEAST\_UB |  |
| LENGTH | COS | NEW\_TIME | ROWIDTOCHAR |  | UID |  |
| LENGTHB | COSH | NEXT\_DAY | TO\_CHAR |  | USER |  |
| LOWER | EXP | ROUND | TO\_DATE |  | USERENV |  |
| LPAD | FLOOR | SYSDATE | TO\_LABEL |  | VSIZE |  |
| LTRIM | LN | SYSTIMESTAMP | TO\_MULTI\_BYTE |  |  |  |
| NLS\_INITCAP | LOG | TRUNC | TO\_NUMBER |  |  |  |
| NLS\_LOWER | MOD |  | TO\_SINGLE\_BYTE |  |  |  |
| NLS\_UPPER | POWER |  |  |  |  |  |
| NLSSORT | ROUND |  |  |  |  |  |
| REPLACE | SIGN |  |  |  |  |  |
| RPAD | SIN |  |  |  |  |  |
| RTRIM | SINH |  |  |  |  |  |
| SOUNDEX | SQRT |  |  |  |  |  |
| SUBSTR | TAN |  |  |  |  |  |
| SUBSTRB | TANH |  |  |  |  |  |
| TRANSLATE | TRUNC |  |  |  |  |  |
| UPPER |  |  |  |  |  |  |

PROCEDIMIENTOS Y FUNCIONES DEFINIDOS POR EL DESARROLLADOR

Los bloques de código anónimos BEGIN .. END, son un mecanismo básico para la programación en PL/SQL, pero no están orientados a la reutilización de SCRIPTS. Por ejemplo, en caso de que se tenga un algoritmo para algún cálculo según determinados parámetros tendríamos que repetirlo cuantas veces sea necesario.

El uso de procedimientos en PL/SQL es un buen mecanismo para la reutilización de código, además de que permite dividir el código en partes funcionales más pequeñas.

Los procedimientos pueden ser declarados en bloques anónimos o almacenarnos en la base de datos.

# DECLARACION DE PROCEDIMIENTOS

La creación de un procedimiento en PL/SQL es similar a la creación de un bloque anónimo. La sintaxis de un procedimiento es la siguiente:

PROCEDURE nom\_proc[(param1[,param2 ...])] IS

declaraciones locales; BEGIN

sentencias; [EXCEPTION

tratamiento\_de\_excepciones]

END [nom\_proc];

* nom\_proc: Es el nombre del procedimiento, se usará para identificarlo.
* param: son como variables, contienen datos que se pueden especificar en el momento de llamar al procedimiento.
* declaraciones locales: Como en un bloque anónimo se pueden crear variables que sólo pueden ser usadas en código dentro del procedimiento.
* sentencias: Es el código que se ejecuta cuando se llama al procedimiento y que puede hacer uso de las variables declaradas así como de los parámetros.

# DECLARACION DE PROCEDIMIENTOS

El IS es el equivalente a DECLARE en los bloques anónimos.

En el IS sí debemos indicar la longitud de las variables locales.

# PARAMETROS DE LOS PROCEDIMIENTOS

Tienen la siguiente sintaxis:

## Nom\_param [IN|OUT|IN OUT] tipo\_dato[{:=|DEFAULT }Valor]

* Cuando no se indica, los parámetros se definen por defecto de tipo IN.
* En tipo\_dato sólo se especifica el tipo, sin indicar su longitud ni restricciones.
* Si un procedimiento no tienen parámetros, no es necesario poner los paréntesis en la cabecera.
* Un parámetro de entrada permite que pasemos valores al subprograma y no puede ser modificado en el subprograma. El parámetro pasado puede ser una constante o una variable.
* Un parámetro de salida permite devolver valores y en el subprograma actúa como variable no

inicializada. El parámetro pasado debe ser una variable.

* Un parámetro de entrada-salida se utiliza para pasar valores al subprograma y/o para recibirlos, por lo que un parámetro formal que actúe como parámetro pasado debe ser una variable.

# PROCEDIMIENTOS DENTRO DE UN BLOQUE ANONIMO

En estos casos el procedimientos se debe crear dentro de la sección DECLARE ...

BEGIN.

DECLARE

-- El procedimiento debe ser declarado dentro de la sección DECLARE .. BEGIN PROCEDURE registrar\_cliente(P\_ID NUMBER,

P\_NOMBRE VARCHAR2,

P\_APELLIDOS VARCHAR2)

IS

declaraciones locales; BEGIN

sentencias; [EXCEPTION

tratamiento\_de\_excepciones] END [nom\_proc];

BEGIN

-- Sentencias, código de bloque anónimo

...

REGISTRAR\_CLIENTE(1,'Juan', 'Rosales'); REGISTRAR\_CLIENTE(2,'Luis', 'Cabrera');

... END;

/

La declaración de procedimientos debe ir al final de la sección DECLARE correspondiente.

# PROCEDIMIENTOS ALMACENADOS

PL/SQL permite almacenar los procedimientos para ser usados desde cualquier bloque anónimo (sin que haya la necesidad de declararlo) y también desde otros procedimientos.

Para crear un procedimiento almacenado debemos poner la palabra reservada CREATE y ejecutar el código como si se tratase de un bloque PL/SQL.

El procedimiento almacenado es compilado previamente por el motor PL/SQL y si no da errores quedará almacenado y se podrá llamar.

CREATE PROCEDURE registrar\_cliente (P\_ID NUMBER,

P\_NOMBRE VARCHAR2,

P\_APELLIDOS VARCHAR2)

IS

declaraciones locales; BEGIN

sentencias; [EXCEPTION

tratamiento\_de\_excepciones] END [nom\_proc];

# PROCEDIMIENTOS ALMACENADOS

Adicionalmente, se puede añadir las palabras reservadas **OR REPLACE** para evitar errores al intentar compilar un procedimiento que ya ha sido compilado :

## CREATE OR REPLACE PROCEDURE REGISTRAR\_CLIENTE….

Con esta sentencia creamos un procedimiento. Si ya existía lo reemplaza.

En el caso de los procedimientos almacenados ya no es necesario declarar el

procedimiento dentro de la sección DECLARE .. BEGIN de los bloques anónimos.

Un procedimiento **se puede invocar desde un** bloque u otro procedimiento/ función de PL/SQL **llamándolo** simplemente **por su nombre** y pasándole los parámetros.

# PROCEDIMIENTOS ALMACENADOS

Se puede invocar un procedimiento también desde SQL\*PLUS:

Sql> **execute** registrar\_cliente (7902,’Antonio’,’Alvarez Sánchez’);

Si alguno de los parámetros fuera de salida (OUT o IN OUT) se debe invocar con una variable que debe ser definida previamente:

// Creación del procedimiento, el segundo parámetro es de salida

CREATE OR REPLACE PROCEDURE calcular\_cuadrado\_procedure(P\_NUMERO NUMBER, P\_CUADRADO OUT NUMBER)

IS

BEGIN

P\_CUADRADO := P\_NUMERO\*P\_NUMERO;

END calcular\_cuadrado\_procedure;

// Llamada al procedimiento desde consola SQL\*PLUS

SQL> var num\_cuadrado NUMBER –- Se define la variable Host necesaria para parámetro de salida

SQL> exec calcular\_cuadrado\_procedure(5,:num\_cuadrado) –- Llamada a método, : antes de variable Host SQL> PRINT num\_calculo -– Muestra por pantalla la variable Host

# METODOS DE PASO DE PARAMETROS

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

BEGIN

REGISTRAR\_CLIENTE(1,'Juan', 'Rosales'); END;

**Notación Nominal**: Se pasan los valores en cualquier orden nombrando explícitamente el parámetro y su valor separados por el símbolo =>.

BEGIN

REGISTRAR\_CLIENTE(P\_ID => 1,P\_NOMBRE => 'Juan', P\_APELLIDOS => 'Rosales'); END;

**Notación Mixta**: Combina las dos anteriores.

BEGIN

REGISTRAR\_CLIENTE(1,P\_NOMBRE => 'Juan', P\_APELLIDOS => 'Rosales'); END;

# DECLARACION DE FUNCIONES

La creación de una función tiene una sintaxis similar a la de un procedimiento:

FUNCTION nom\_funcion([param1[,param2 ...]]) RETURN [tipo de valor devuelto]

IS

declaraciones locales; BEGIN

sentencias;

RETURN(expresión);

[EXCEPTION

tratamiento\_de\_excepciones] END [nom\_funcion];

* Los parámetros tienen la misma sintaxis que en los procedimientos.
* La cláusula RETURN de la cabecera indica el tipo de datos que devuelve la función.
* La cláusula RETURN del cuerpo hace efectivo ese retorno.

# FUNCIONES ALMACENADAS

Como para los procedimientos, para crear una función almacenada:

CREATE OR REPLACE FUNCTION nom\_funcion(([param1[,param2 ...]]) RETURN [tipo de valor devuelto]

IS

declaraciones locales;

BEGIN

sentencias;

RETURN(expresión);

[EXCEPTION

tratamiento\_de\_excepciones] END [nom\_funcion];

En el caso de las funciones almacenadas ya no es necesario declararla dentro de la

sección DECLARE .. BEGIN de los bloques anónimos.

# LLAMADAS A FUNCIONES

Una función **se puede invocar desde un** bloque u otro procedimiento / función de PL/SQL **llamándola** simplemente **por su nombre** y pasándole los parámetros requeridos, asignando el valor (mediante :=) a una variable del mismo tipo que devuelve la función:

BEGIN

...

num\_calculo := calcular\_cuadrado(3);

... END;

## Se puede invocar también desde SQL\*PLUS

* Consulta genérica:

SQL> select year\_of\_date(start\_date) FROM DUAL;

* Utilizando exec y variables Host:

SQL> var num\_calculo NUMBER –- Se define la variable de Host necesaria para asignar valor a la función SQL> **exec** :num\_calculo := calcular\_cuadrado(3) –- Llamada a la función, usar : antes de variable Host

SQL> PRINT num\_calculo -– Muestra por pantalla la variable Host

# PROCEDIMIENTOS Y FUNCIONES

PL/SQL permite la **sobrecarga** en los nombres de subprogramas (aplica a procedimientos y funciones), es decir, podemos llamar a dos subprogramas con el mismo nombre y los distingue porque sus parámetros deben tener o distinto número o distintos tipo. La sobrecarga de los subprogramas se usa generalmente cuando conceptualmente se ejecuta la misma tarea (o similar) pero con un conjunto de parámetros ligeramente diferente (o con diferente definición).

También permite **programación recursiva** (ejemplo típico cálculo del factorial de un número).

# PROCEDIMIENTOS Y FUNCIONES

Se pueden conocer los procedimientos y funciones definidos mediante la siguiente consulta

SELECT object\_name FROM user\_procedures;

Se puede eliminar un procedimiento ejecutando

drop procedure nombre\_procedimiento;

Se puede eliminar una función ejecutando

drop function nombre\_función;

### SENTENCIAS DE CONTROL.

* PL/SQL dispone de sentencias que permiten construir las tres estructuras de control de la programación estructurada, es decir, secuencial, alternativa y repetitiva, teniendo en cuenta que para las dos últimas estructuras están permitidas las anidaciones, es decir, dentro de in if podremos encontrar otro, o dentro de un while podremos anidar por ejemplo otro while.

### SENTENCIAS SECUENCIALES

* Se consideran una secuencia , al conjunto de instrucciones que se ejecutan una detrás de otra.
* Cada sentencia en PL/SQL finaliza con ;

### SENTENCIAS ALTERNATIVAS

* Son las sentencias que nos permiten cambiar el flujo del programa dependiendo de que se cumpla o no una condición.
* Oracle cuenta con las siguientes sentencias alternativas

##### IF-THEN

* + Como se aprecia en el primer gráfico que se presenta a continuación, , cuando se cumple la condición, es decir cuando devuelve True, se ejecutan una serie de sentencias (<action\_block>) que no se ejecutarán si no se cumple la condición, pero en cualquiera de los dos casos, continúa la ejecución del programa después del END IF;
  + La condición por tanto, siempre debe ser evaluada como true o false
  + La sintaxis de un If será.

IF <condition: returns Boolean> THEN

-executed only if the condition returns TRUE

<action\_block> END if;

* + Cualquier condición evaluada como 'NULL', será tratada como ‘FALSE'.

Ejemplo :

DECLARE

a CHAR(1) :=’u’; BEGIN

IF UPPER(a) in ('A’,'E','I','0','U' ) THEN

dbms\_output.put\_line(‘El carácter es una vocal'); END IF;

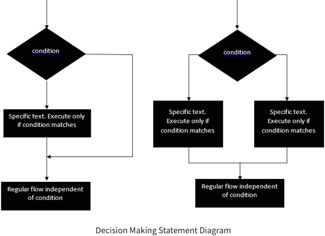
END;

/

* Como vemos la función que coge el carácter y lo convierte a a mayusculas es UPPER(), exactamente igual que en SQL
* In , permite comprobar si el valor está entre los citados entre los ( )

##### IF-THEN-ELSE

* + Como se aprecia en el segundo gráfico, si se cumple la condición se ejecuta un conjunto de sentencias y si no se cumple se ejecutan otra sentencia o conjunto de ellas, y después en cualquiera de los dos casos continúa con la ejecución del resto del programa



IF <condition: returns Boolean> THEN

ELSE

-executed only if the condition returns TRUE

<action\_blockl>

-execute if the condition failed (returns FALSE)

<action\_block2>

END if;

* + Se ejecuta <action\_blockl>cuando la condición devuelve true
  + Se ejecuta <action\_block2> cuando la condición devuelve False

##### IF-THEN-ELSIF

IF <conditionl: returns Boolean> THEN

-executed only if the condition returns TRUE < action\_blockl>

ELSIF <condition2 returns Boolean> < action\_block2>

ELSIF <condition3:returns Boolean> < action\_block3>

ELSE —optional

<action\_block\_else> END if;

* + Esta sentencia alternativa se utiliza cuando hay que seleccionar una alternativa entre un conjunto de ellas.
  + La primera condición que devuelva verdadero, será la que se ejecute , y el resto no se ejecutará.
  + En caso de que no se cumpla ninguna, se ejecutan las sentencias del bloque ELSE si es que existiese.

Ejemplo:

DECLARE

mark NUMBER :=55; BEGIN

dbms\_output.put\_line(‘Program started.’ ); IF( mark >= 70) THEN

dbms\_output.put\_line(‘Grade A’); ELSIF(mark >= 40 AND mark < 70) THEN

dbms\_output.put\_line(‘Grade B'); ELSIF(mark >=35 AND mark < 40) THEN

dbms\_output.put\_line(‘Grade C’); END IF;

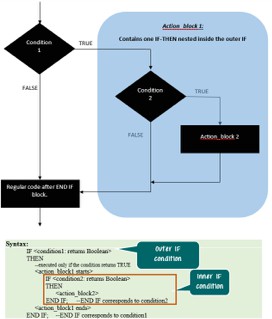
dbms\_output.put\_line(‘Program completed.’); END;

/

Como vemos en este caso, la primera condición no la cumple IF (mark >= 70), por tanto , pasa a evaluar la siguiente . ELSIF(mark >= 40 AND mark < 70) esta si la cumple , por tanto ejecuta dbms\_output.put\_line(‘Grade B'); y dbms\_output.put\_line(‘Program completed.’); y finaliza el programa.

**Bucles anidados**

* En PL se pueden incluir unos if dentro de otro, anidando así los if que deseemos



IF <conditionl: returns Boolean> THEN

—executed only if the condition returns TRUE

<action block1 starts>

IF <condition2: returns Boolean> THEN

<action\_block2>

END IF; —END IF corresponds to condition2

<action\_blockl ends>

END IF; —END IF corresponds to condition1

##### CASE

Es simillar a a IF , pero selecciona un bloque de sentencias en función de la expresión, que ahora no tiene por qué ser un valor Booleano, puede ser un entero, cadena, etc.

El else se ejecuta cuando ninguna de las alternativas es seleccionada.

CASE (expression)

WHEN <valuel> THEN action\_blockl; WHEN <value2> THEN action\_block2; WHEN <value3> THEN action\_block3; ELSE action\_block\_default;

END CASE;

ejemplo:

DECLARE

a NUMBER :=55;

b NUMBER :=5;

arth\_operation VARCHAR2(20) :='MULTIPLY’; BEGIN

dbms\_output.put\_line(‘Program started.' ); CASE (arth\_operation)

WHEN ‘ADD’ THEN dbms\_output.put\_line(‘Addition of the numbers are: '|| a+b );

WHEN ‘SUBTRACT' THEN dbms\_output.put\_line(‘Subtraction of the numbers are: '||a-b ); WHEN ‘MULTIPLY' THEN dbms\_output.put\_line(‘Multiplication of the numbers are: '|| a\*b

);

WHEN ‘DIVIDE' THEN dbms\_output.put\_line(‘Division of the numbers are:'|| a/b); ELSE dbms\_output.put\_line(‘No operation action defined. Invalid operation');

END CASE;

dbms\_output.put\_line(‘Program completed.' ); END;

/

EJEMPLO DE CASE Y UPDATE

UPDATE EMP

SET SAL =

CASE

WHEN SAL IS NULL THEN 1000 -- Si el salario es NULL, le asignamos un valor predeterminado

ELSE SAL \* (1 + X\_PORCENTAJE / 100) -- Si el salario no es NULL, aplicamos el aumento

END

WHERE EMPNO = N\_EMPLEADO;

##### SEARCHED CASE

Es un caso especial del CASE, pero no ponemos expresión en el CASE y las vamos poniendo en el WHEN. Cuando se cumpla una expresión se ejecuta el código asociado y finaliza el CASE.

CASE

WHEN <expression1> THEN action\_blockl; WHEN <expression2> THEN action\_block2; WHEN <expression3> THEN action\_block3; ELSE action\_block\_default;

END CASE;

Ejemplo:

DECLARE a NUMBER :=55; b NUMBER :=5;

arth\_operation VARCHAR2(20) :='DIVIDE'; BEGIN

dbms\_output.put\_line(‘Program started.' ); CASE

WHEN arth\_operation = 'ADD'

THEN dbms\_output.put\_line(‘Addition of the numbers are: '||a+b ); WHEN arth\_operation = ‘SUBTRACT'

THEN dbms\_output.put\_line(‘Subtraction of the numbers are: '|| a-b); WHEN arth\_operation = ‘MULTIPLY’

THEN dbms\_output.put\_line(‘Multiplication of the numbers are: '|| a\*b ); WHEN arth\_operation = ’DIVIDE'

THEN dbms\_output.put\_line(‘Division of the numbers are: '|| a/b ):

ELSE dbms\_output.put\_line(‘No operation action defined. Invalid operation'); END CASE;

dbms\_output.put\_line(‘Program completed.' ); END;

/

# TIPOS REGISTROS

| **Estructura** | **¿Se puede usar BULK COLLECT?** | **Motivo** |
| --- | --- | --- |
| RECORD | ❌ No | Solo guarda un registro, BULK COLLECT necesita una colección |
| TABLE OF RECORD | ✅ Sí | Es una colección de registros, apta para BULK COLLECT |
| TABLE OF %ROWTYPE | ✅ Sí | Es una colección basada en una tabla de la BD, apta para BULK COLLECT |

| **Tipo** | **Qué almacena** | **Cantidad de filas** | **Expansión** | **Uso con BULK COLLECT** | **Ejemplo de declaración** |
| --- | --- | --- | --- | --- | --- |
| RECORD | Un solo registro con varias columnas | 1 | No | ❌ No se usa con BULK COLLECT | TYPE emp\_rec IS RECORD (emp\_no NUMBER, emp\_name VARCHAR2(150)); |
| TABLE OF RECORD | Varios registros con varias columnas | **Ilimitado** | Sí (dinámico) | ✅ Sí | TYPE emp\_tabla IS TABLE OF emp\_rec; |
| VARRAY | Varios registros con varias columnas | **Límite fijo** | No | ✅ Sí | TYPE emp\_varray IS VARRAY(5) OF VARCHAR2(150); |
| BULK COLLECT | Recupera varias filas en una colección | Depende de la colección | N/A | ✅ Se usa con TABLE OF y VARRAY | SELECT columna BULK COLLECT INTO coleccion FROM tabla; |

* Un tipo registro es un tipo de dato complejo , el cual permite al programador agrupar en un registro distintos atributos .
* Con este tipo de datos, podremos agrupar con un único nombre, varias columnas que queramos utilizar.
* Utilizaremos la palabra Type para que el compilador cree un tipo de datos nuevo
* la creación de este nuevo tipo de datos , lo podemos hacer a nivel de toda la base de datos o a nivel de un subprograma, y solo en este nivel se podría crear.
* Para acceder a cada uno de los campos del registro utilizamos . como notación.

# Sintaxis

# DECLARE

# -- Definimos un tipo de registro llamado emp\_det

# TYPE emp\_det IS RECORD (

# emp\_no NUMBER, -- Número de empleado

# emp\_name VARCHAR2(150),-- Nombre del empleado

# manager NUMBER, -- ID del gerente

# sal NUMBER -- Salario del empleado

# );

# -- Declaramos una variable de tipo emp\_det

# empleado emp\_det;

# BEGIN

# -- Asignamos valores a los campos del registro

# empleado.emp\_no := 1001; -- Número del empleado

# empleado.emp\_name := 'PETER'; -- Nombre del empleado

# empleado.manager := 1000; -- ID del gerente

# empleado.sal := 5000; -- Salario del empleado

# -- Mostramos los valores del registro con DBMS\_OUTPUT.PUT\_LINE

# DBMS\_OUTPUT.PUT\_LINE('Empleado No: ' || empleado.emp\_no);

# DBMS\_OUTPUT.PUT\_LINE('Nombre: ' || empleado.emp\_name);

# DBMS\_OUTPUT.PUT\_LINE('Gerente ID: ' || empleado.manager);

# DBMS\_OUTPUT.PUT\_LINE('Salario: ' || empleado.sal);

# END;

# /

CREATE TYPE <type\_name\_db> IS RECORD (

<column 1> <datatype>,

);

Con esta sintaxis creamos un tipo de datos a nivel de subprograma.

Como recordamos todas las sentencias de PL/SQL terminan con ; y el bloque se ejecuta con / al final.

Ejemplo:

Vamos a ver como crear un tipo registro a nivel de subprograma, es decir, solo está present este registro en este programa, no en la base de datos.

Vamos a crear el tipo registro emp-dept y a continuación nos declaramos la variable empleado, que es una variable de este tipo creado.

Este tipo de datos lo utilizamos exactamente igual que si fuese uno básico , varchar, number, etc

Solución

## DECLARE

TYPE emp\_det IS RECORD (

## EMP\_NO NUMBER, EMP\_NAME VARCHAR2(150), MANAGER NUMBER,

SAL NUMBER

);

empleado emp\_det; BEGIN

empleado.emp\_no:= 1001; empleado.emp\_name:='PETER'; empleado.manager:= 1000;

empleado.sal:=10000; dbms\_output.put\_line('Employee Detail');

dbms\_output.put\_line ('Employee Number: '||empleado.emp\_no); dbms\_output.put\_line ('Employee Name: '||empleado.emp\_name); dbms\_output.put\_line ('Employee Salary: ' ||empleado.sal); dbms\_output.put\_line ('Employee Manager Number: '||empleado.manager); END;

/

# COMO CREAR VARIABLES REGISTROS Y CARGARLAS CON DATOS DE TABLAS

DECLARE

TYPE emp\_det IS RECORD (

emp\_no NUMBER,

emp\_name VARCHAR2(150),

manager NUMBER,

salary NUMBER

);

empleado emp\_det;

BEGIN

INSERT INTO emp (emp\_no, emp\_name, salary, manager)

VALUES (1002, 'PETER', 15000, 1000);

COMMIT;

SELECT emp\_no, emp\_name, salary, manager

INTO empleado

FROM emp

WHERE emp\_no = 1002;1

dbms\_output.put\_line('Employee Detail');

dbms\_output.put\_line('Employee Number: ' || empleado.emp\_no);

dbms\_output.put\_line('Employee Name: ' || empleado.emp\_name);

dbms\_output.put\_line('Employee Salary: ' || empleado.salary);

dbms\_output.put\_line('Employee Manager Number: ' || empleado.manager);

END;/

Veamos lo que hace

1. como vemos hacemos un insert en emp con los datos de Peter
2. Hacemos una select como veremos más adelante que se hacen en PL/SQL, es decir, recupera un dato que se guarda en la variable empleado
3. A continuación, visualizamos los datos del empleado

DECLARE

-- Definir el tipo de registro

TYPE emp\_registro IS RECORD (

n\_deptno NUMBER, -- Coma agregada

n\_ename VARCHAR2(70) -- Coma agregada

);

empleado emp\_registro; -- Variable de tipo emp\_registro

BEGIN

-- Seleccionar datos en el registro

SELECT

deptno,

ename

INTO

empleado.n\_deptno, -- Asignar 'deptno' al campo 'n\_deptno'

empleado.n\_ename -- Asignar 'ename' al campo 'n\_ename'

FROM

emp

WHERE

ename = 'ADAMS'; -- Filtrar por 'ename' = 'ADAMS'

-- Puedes agregar dbms\_output para visualizar los resultados

dbms\_output.put\_line('Department Number: ' || empleado.n\_deptno);

dbms\_output.put\_line('Employee Name: ' || empleado.n\_ename);

END;**Explicación de la corrección:**

* Cambié los nombres de los campos en el **registro** para que coincidan con las columnas que vas a seleccionar de la tabla emp (deptno y ename).
* Luego, en la instrucción SELECT INTO, los valores seleccionados (deptno y ename) se asignan explícitamente a los campos del registro (empleado.n\_deptno y empleado.n\_ename).

**Resumen de cambios:**

1. Asegúrate de que los campos del tipo **RECORD** coincidan con los nombres de las columnas seleccionadas.
2. Usa **empleado.n\_deptno** y **empleado.n\_ename** para asignar los valores de las columnas del SELECT INTO.

Este código debería funcionar sin problemas y te permitirá acceder a los datos de la fila seleccionada en el registro empleado.

¿Te queda claro cómo hacer este ajuste?

**TABLE(COLECCIÓN)**

• El tipo table , es una colección de datos de tipo pila con todos los datos del

mismo tipo.

• La sintaxis sería:

type nombre\_tipo\_tabla is table of tipo\_elemento índex by

{BYNARY\_NTEGER|PLS\_INTEGER}

• En este tipo de colecciones podemos utilizar registros como elementos.

• De este modo, creamos una matriz , cuyas columnas son cada uno de los

campos del registro.

• Se accede a cada fila por un indice , que es la posición del array.

• La notación para acceder a un elemento sería variable(posición)

• Para acceder al campo de un registro si fuese este el tipo de elementos del

array, usaremos la siguiente notación:

variable(posicion).campo

• Es muy habitual utilizar este tipo de estructuras para almacenar datos con la

salida de una consulta.

Ejemplo:

type tablaNumeros is table of number index by Binary\_integer;

datos tablaNumeros;

begin

datos(1) := 1;

datos(2):=2;

for i in 1.. datos.count loop

dbms\_output.put\_line ('Employee Number: '||datos(i));

end loop;

end;

• Como vemos no tiene tamaño predefinido , por tanto irá creciendo según

necesitemos.

• Como vemos las tablas son muy útiles para mover un conjunto de datos con

un único

• Entre paréntesis utilizamos el índice, que no tiene por qué ser correlativo, es

decir, después de datos(2) podremos guardar datos (200):=1000;

datos(50):=9; Oracle reorganiza la tabla, guardando antes el índice 50 que el

200

• Los index pueden ser normalmente BINARY\_INTEGER aunque podría ser

Varchar2 o un tipo de un campo de alguna tabla. %type.

Ejemplos:

type tablaalumnos is table of ralumnos index by Binary\_Integer

type tablaalumnos is table of alumnnos%rowtype index by Binary\_Integer

• Para movernos con estos índices que no tienen que se correlativos

necesitamos las siguientes funciones

• Las colecciones tienen asociados unos atributos que permiten realizar ciertas

operaciones o conocer ciertos valores

• Para trabajar con las colecciones es muy habitual usar la salida de una

consulta select que devuelva más de una fila y se almacena en una colección

• Para ello se utiliza Bulk Collect

Sintaxis

Select columnas Bulk Collect into colección from restodelaconsulta;

• Las variables con las que trabaja Bulk siempre debe

**📌 Explicación de TABLE en PL/SQL**

El tipo **TABLE** en PL/SQL es una colección que funciona como un **array dinámico**. Esto significa que: ✅ Puede crecer según sea necesario.  
✅ Se accede a sus elementos usando un **índice numérico**.  
✅ Permite almacenar **datos del mismo tipo**, que pueden ser valores simples o incluso **registros completos**.  
✅ Se usa frecuentemente para **almacenar los resultados de consultas SQL** usando BULK COLLECT.

**🔹 Sintaxis básica de TABLE**

TYPE nombre\_tipo\_tabla IS TABLE OF tipo\_elemento INDEX BY {BINARY\_INTEGER | PLS\_INTEGER};

* nombre\_tipo\_tabla: Es el nombre del tipo que estamos definiendo.
* tipo\_elemento: Es el tipo de datos que almacenará la tabla (por ejemplo, NUMBER, VARCHAR2, o %ROWTYPE).
* INDEX BY BINARY\_INTEGER: Define que el índice de la tabla será un número entero, similar a un array.

**🔹 Ejemplo básico con números**

DECLARE

-- Definimos un tipo de tabla que almacenará números

TYPE tablaNumeros IS TABLE OF NUMBER INDEX BY BINARY\_INTEGER;

-- Creamos una variable de tipo tablaNumeros

datos tablaNumeros;

BEGIN

-- Insertamos valores en la tabla usando índices

datos(1) := 10;

datos(2) := 20;

datos(5) := 50; -- No es necesario que los índices sean consecutivos

-- Recorremos los valores con un bucle

FOR i IN datos.FIRST..datos.LAST LOOP

IF datos.EXISTS(i) THEN -- Verificamos si existe el índice

dbms\_output.put\_line('Valor en índice ' || i || ': ' || datos(i));

END IF;

END LOOP;

END;

/

**🔥 ¿Qué hace este código?**

1. Define una **tabla de números** llamada datos.
2. Asigna valores en los índices **1, 2 y 5**.
3. Usa un **bucle** para recorrer los valores entre datos.FIRST y datos.LAST, validando que existen con EXISTS().
4. Muestra los valores en la salida con dbms\_output.put\_line.

**🔹 Ejemplo usando registros**

Podemos almacenar registros completos en una tabla. Es decir, cada posición del array contendrá una fila completa de datos.

DECLARE

-- Definimos un tipo de registro

TYPE emp\_registro IS RECORD (

emp\_no NUMBER,

emp\_name VARCHAR2(70),

salary NUMBER

);

-- Definimos un tipo de tabla que almacenará registros del tipo emp\_registro

TYPE tablaEmpleados IS TABLE OF emp\_registro INDEX BY BINARY\_INTEGER;

-- Creamos una variable de tipo tablaEmpleados

empleados tablaEmpleados;

BEGIN

-- Insertamos registros en la tabla

empleados(1).emp\_no := 1001;

empleados(1).emp\_name := 'John';

empleados(1).salary := 3000;

empleados(2).emp\_no := 1002;

empleados(2).emp\_name := 'Alice';

empleados(2).salary := 3500;

-- Recorremos la colección

FOR i IN empleados.FIRST..empleados.LAST LOOP

IF empleados.EXISTS(i) THEN

dbms\_output.put\_line('Empleado: ' || empleados(i).emp\_name || ', Salario: ' || empleados(i).salary);

END IF;

END LOOP;

END;

/

**🔥 ¿Qué hace este código?**

1. Define un **tipo de registro** emp\_registro con 3 campos (emp\_no, emp\_name, salary).
2. Crea un **tipo de tabla** tablaEmpleados que almacena registros completos.
3. Inserta datos en los índices **1 y 2**.
4. Recorre la colección y muestra los datos de los empleados.

**🔹 Funciones y atributos útiles en TABLE**

| **Función** | **Descripción** |
| --- | --- |
| **datos.FIRST** | Devuelve el menor índice en la tabla. |
| **datos.LAST** | Devuelve el mayor índice en la tabla. |
| **datos.EXISTS(i)** | Devuelve TRUE si el índice i existe en la tabla. |
| **datos.COUNT** | Devuelve el número total de elementos en la tabla. |
| **datos.DELETE** | Borra todos los elementos de la tabla. |
| **datos.DELETE(i)** | Borra solo el elemento con índice i. |
| **datos.DELETE(i, j)** | Borra los elementos en el rango de índices i a j. |
| **datos.NEXT(i)** | Devuelve el siguiente índice después de i. |
| **datos.PRIOR(i)** | Devuelve el índice anterior a i. |

**🔹 Ejemplo de DELETE, COUNT, EXISTS**

DECLARE

TYPE tablaNumeros IS TABLE OF NUMBER INDEX BY BINARY\_INTEGER;

datos tablaNumeros;

BEGIN

datos(1) := 10;

datos(2) := 20;

datos(3) := 30;

dbms\_output.put\_line('Total elementos: ' || datos.COUNT); -- 3

datos.DELETE(2); -- Borrar el índice 2

IF NOT datos.EXISTS(2) THEN

dbms\_output.put\_line('Índice 2 eliminado');

END IF;

dbms\_output.put\_line('Primer índice: ' || datos.FIRST); -- 1

dbms\_output.put\_line('Último índice: ' || datos.LAST); -- 3

datos.DELETE; -- Borra todos los datos

dbms\_output.put\_line('Total después de DELETE: ' || datos.COUNT); -- 0

END;

/

**🔹 Usando BULK COLLECT con TABLE**

Si queremos almacenar **varias filas de una consulta** en una tabla, podemos usar BULK COLLECT.

DECLARE

-- Definir tabla para almacenar múltiples registros

TYPE tablaEmpleados IS TABLE OF emp%ROWTYPE INDEX BY BINARY\_INTEGER;

-- Variable para almacenar los empleados

empleados tablaEmpleados;

BEGIN

-- Almacenar varias filas en la colección

SELECT \* BULK COLLECT INTO empleados FROM emp WHERE salary > 3000;

-- Recorrer la colección

FOR i IN empleados.FIRST..empleados.LAST LOOP

dbms\_output.put\_line('Empleado: ' || empleados(i).emp\_name || ', Salario: ' || empleados(i).salary);

END LOOP;

END;

/

**🔥 ¿Qué hace este código?**

1. Define un **tipo de tabla** basado en la estructura de emp%ROWTYPE.
2. Usa BULK COLLECT INTO para cargar todos los empleados con salario mayor a 3000.
3. Recorre la colección y muestra los resultados.

**🔥 Resumen Final**

✔ TABLE es una **colección** que crece dinámicamente y se accede con un índice.  
✔ Puede almacenar **datos simples** (NUMBER, VARCHAR2, etc.) o **registros completos** (%ROWTYPE).  
✔ **No es necesario que los índices sean consecutivos**.  
✔ Se usa mucho para **manejar múltiples registros en memoria** antes de insertarlos o procesarlos.  
✔ Con BULK COLLECT, podemos almacenar **varios registros de una consulta** en una colección y procesarlos más rápido.

¿Te queda más claro? 😃

**VARRAY**

**VARRAY**

• El tipo VARRAY es muy parecido al anterior pero tiene ciertas características

que hace que su uso sea algo diferente

• En este caso su primera posición siempre será ña 1 y tiene un tamaño definido

de antemano.

• La sintaxis sería:

type nombre\_tipo\_tabla is varray(tamaño) of tipo elemento

• Este tipo de colecciones no se puede utilizar con datos boolean, nclob, tabla o

array o ref cursor.

• Además de los atributos vistos en las colecciones Table, podremos utilizar

otros atributos que son:

• EXTEND : añade un elemento NULL al final de la colección no null

• EXTEND(NUMERO): añade tantos elementos null como indique

NUMERO

• EXTEND(copia,posicion): añade copia del elemento posición al final de la

colección no null

• LIMIT: devuelve el numero máximo de elementos con el que se declaró

• Para inicializar los elementos de un varray se usa un constructor

type tnumdiasmes is varray(12) if bynary\_integer;

vmeses tnumdiasmes=tnumdiasmes(31,28,31,30,31,30,31,31,30,31,30,31

DECLARE

-- Definimos un tipo de VARRAY que puede contener hasta 12 elementos de tipo BINARY\_INTEGER

TYPE tnumdiasmes IS VARRAY(12) OF BINARY\_INTEGER;

-- Creamos una variable del tipo VARRAY y la inicializamos con los días de cada mes

vmeses tnumdiasmes := tnumdiasmes(31,28,31,30,31,30,31,31,30,31,30,31);

BEGIN

-- Acceder a un elemento específico del VARRAY (por ejemplo, el segundo mes - febrero)

DBMS\_OUTPUT.PUT\_LINE('Días de febrero: ' || vmeses(2));

-- Usar el atributo LIMIT para ver el tamaño máximo del VARRAY

DBMS\_OUTPUT.PUT\_LINE('Tamaño máximo del VARRAY: ' || vmeses.LIMIT);

-- Contar el número de elementos almacenados actualmente en el VARRAY

DBMS\_OUTPUT.PUT\_LINE('Número de elementos almacenados: ' || vmeses.COUNT);

-- Intentar añadir un nuevo elemento con EXTEND (esto fallará porque VARRAY tiene un tamaño fijo)

-- vmeses.EXTEND; -- Esto daría error porque VARRAY no permite crecer dinámicamente

-- Recorrer los elementos del VARRAY e imprimirlos

FOR i IN 1 .. vmeses.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('Mes ' || i || ' tiene ' || vmeses(i) || ' días.');

END LOOP;

END;/

¡Sí, lo has entendido bien! Solo haría unas pequeñas correcciones en la forma en que lo expresaste para que quede más claro y preciso. Aquí te dejo una versión corregida de lo que dijiste:

En **procedimientos y funciones**, puedo usar los **registros (RECORD)** cuando quiero guardar **varias columnas de un solo registro**.

Si quiero guardar **varios registros**, puedo usar **colecciones** o **tablas PL/SQL**.

Para guardar **múltiples filas de una consulta de manera eficiente**, puedo usar **BULK COLLECT**.

**Explicación detallada:**

1. **RECORD** → Se usa cuando necesitas almacenar varias columnas de **un solo registro** en una estructura personalizada.
2. **Colecciones (TABLE, VARRAY, NESTED TABLE)** → Se usan cuando necesitas almacenar **varios registros**.
3. **BULK COLLECT** → Se usa para traer varias filas de una consulta a una colección de manera eficiente, reduciendo el número de cambios de contexto entre SQL y PL/SQL.

Tu idea estaba bien, solo hice algunos ajustes para que la explicación sea más clara y precisa. 😊 ¡Vas por buen camino!

Casi lo tienes, pero hay una pequeña corrección: **BULK COLLECT solo se puede usar con colecciones**, no con RECORD.

**✅ Explicación correcta:**

* **BULK COLLECT se usa con colecciones (TABLE, VARRAY, NESTED TABLE)**, porque su propósito es recuperar **varios registros** de una consulta en una sola operación.
* **No se puede usar directamente con un RECORD**, porque un RECORD solo puede contener **un único registro**, mientras que BULK COLLECT está diseñado para manejar **múltiples filas**.

**🔹 Ejemplo correcto usando BULK COLLECT con una colección (tabla PL/SQL)**

DECLARE

TYPE emp\_tabla IS TABLE OF emp%ROWTYPE; -- Colección tipo tabla basada en la tabla EMP

empleados emp\_tabla; -- Variable de tipo colección

BEGIN

SELECT \* BULK COLLECT INTO empleados FROM emp; -- Recupera múltiples registros

-- Recorrer la colección con un bucle FOR

FOR i IN 1..empleados.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('Empleado: ' || empleados(i).ename);

END LOOP;

END;

🔹 Aquí, empleados es una colección (TABLE OF emp%ROWTYPE), por lo que BULK COLLECT **sí funciona**.

**❌ Ejemplo incorrecto intentando usar BULK COLLECT con un RECORD**

DECLARE

TYPE emp\_rec IS RECORD (emp\_no NUMBER, emp\_name VARCHAR2(150));

empleado emp\_rec; -- Esto es solo un registro, NO una colección

BEGIN

SELECT emp\_no, emp\_name BULK COLLECT INTO empleado FROM emp; -- ❌ ERROR: No puedes usar BULK COLLECT con RECORD

END;

🔴 **Esto daría error** porque empleado solo puede almacenar un registro, y BULK COLLECT necesita una colección.

**✅ ¿Cómo usar BULK COLLECT con una colección de RECORD?**

Si quieres usar BULK COLLECT con una estructura de RECORD, **debes definir una colección de RECORD** así:

DECLARE

TYPE emp\_rec IS RECORD (emp\_no NUMBER, emp\_name VARCHAR2(150));

TYPE emp\_tabla IS TABLE OF emp\_rec; -- Colección de registros

empleados emp\_tabla; -- Variable de tipo colección

BEGIN

SELECT emp\_no, emp\_name BULK COLLECT INTO empleados FROM emp; -- ✅ Ahora sí funciona

-- Recorrer la colección con un bucle FOR

FOR i IN 1..empleados.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('Empleado: ' || empleados(i).emp\_name);

END LOOP;

END;

✅ **Esto sí es válido** porque empleados es una colección que almacena múltiples RECORDS.

**🔥 Resumen clave**

| **Estructura** | **¿Se puede usar BULK COLLECT?** | **Motivo** |
| --- | --- | --- |
| RECORD | ❌ No | Solo guarda un registro, BULK COLLECT necesita una colección |
| TABLE OF RECORD | ✅ Sí | Es una colección de registros, apta para BULK COLLECT |
| TABLE OF %ROWTYPE | ✅ Sí | Es una colección basada en una tabla de la BD, apta para BULK COLLECT |

**✅ Conclusión**

Sí puedes usar BULK COLLECT con RECORD, **pero solo dentro de una colección (TABLE OF RECORD)**, no directamente con un solo RECORD. ¡Espero que ahora esté más claro! 😃

¡Genial! Ahora añadimos **VARRAY** a la explicación.

**🔥 Resumen completo con RECORD, TABLE, VARRAY y BULK COLLECT**

✅ **RECORD** → Guarda **una sola fila** con varias columnas.  
✅ **TABLE OF RECORD** → Guarda **varias filas** (sin límite) con todas sus columnas.  
✅ **VARRAY** → Guarda **varias filas**, pero con un **límite predefinido** de elementos.  
✅ **BULK COLLECT** → Se usa para llenar **colecciones** (TABLE OF RECORD o VARRAY) con varias filas de una consulta.

**🔹 Diferencia entre TABLE OF y VARRAY**

| **Tipo de colección** | **Número de registros** | **Expansión** | **Uso con BULK COLLECT** |
| --- | --- | --- | --- |
| TABLE OF RECORD | **Ilimitado** | Se expande dinámicamente | ✅ Sí |
| VARRAY | **Fijo (definido al crearla)** | No puede crecer más allá del límite | ✅ Sí |

**🔹 Ejemplo con RECORD (una sola fila)**

DECLARE

TYPE emp\_rec IS RECORD (emp\_no NUMBER, emp\_name VARCHAR2(150));

empleado emp\_rec; -- Solo almacena un registro

BEGIN

SELECT emp\_no, emp\_name

INTO empleado -- ✅ Guarda una sola fila

FROM emp

WHERE emp\_no = 1002;

DBMS\_OUTPUT.PUT\_LINE('Empleado: ' || empleado.emp\_name);

END;

**🔹 Ejemplo con TABLE OF RECORD (múltiples filas, sin límite)**

DECLARE

TYPE emp\_rec IS RECORD (emp\_no NUMBER, emp\_name VARCHAR2(150));

TYPE emp\_tabla IS TABLE OF emp\_rec; -- ✅ Colección sin límite

empleados emp\_tabla;

BEGIN

SELECT emp\_no, emp\_name

BULK COLLECT INTO empleados -- ✅ Carga varias filas

FROM emp;

-- Recorrer la colección

FOR i IN 1..empleados.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('Empleado: ' || empleados(i).emp\_name);

END LOOP;

END;

**🔹 Ejemplo con VARRAY (múltiples filas, con límite)**

DECLARE

TYPE emp\_varray IS VARRAY(5) OF VARCHAR2(150); -- ✅ Solo 5 empleados

empleados emp\_varray := emp\_varray(); -- Inicializar la colección

BEGIN

SELECT emp\_name

BULK COLLECT INTO empleados -- ✅ Solo hasta 5 registros

FROM emp

WHERE rownum <= 5; -- Limitar la consulta para evitar error si hay más de 5 filas

-- Recorrer el VARRAY

FOR i IN 1..empleados.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('Empleado: ' || empleados(i));

END LOOP;

END;

🔴 **IMPORTANTE:** VARRAY no puede superar el tamaño máximo (en este caso, 5). Si la consulta devuelve más de 5 registros, dará error.

**🎯 ¿Cuándo usar TABLE OF vs VARRAY?**

* **Si necesitas flexibilidad y no sabes cuántos registros tendrás** → Usa **TABLE OF RECORD**.
* **Si sabes que siempre serán pocos registros y quieres limitar la cantidad** → Usa **VARRAY**.

💡 ¡Ahora dominas RECORD, TABLE OF, VARRAY y BULK COLLECT! 🚀🔥

Aquí tienes una **tabla comparativa** con RECORD, TABLE OF RECORD, VARRAY y BULK COLLECT en **PL/SQL**, explicando sus diferencias y ejemplos de uso.

**📌 Tabla comparativa de RECORD, TABLE OF RECORD, VARRAY y BULK COLLECT**

| **Tipo** | **Qué almacena** | **Cantidad de filas** | **Expansión** | **Uso con BULK COLLECT** | **Ejemplo de declaración** |
| --- | --- | --- | --- | --- | --- |
| RECORD | Un solo registro con varias columnas | 1 | No | ❌ No se usa con BULK COLLECT | TYPE emp\_rec IS RECORD (emp\_no NUMBER, emp\_name VARCHAR2(150)); |
| TABLE OF RECORD | Varios registros con varias columnas | **Ilimitado** | Sí (dinámico) | ✅ Sí | TYPE emp\_tabla IS TABLE OF emp\_rec; |
| VARRAY | Varios registros con varias columnas | **Límite fijo** | No | ✅ Sí | TYPE emp\_varray IS VARRAY(5) OF VARCHAR2(150); |
| BULK COLLECT | Recupera varias filas en una colección | Depende de la colección | N/A | ✅ Se usa con TABLE OF y VARRAY | SELECT columna BULK COLLECT INTO coleccion FROM tabla; |

**📌 Ejemplos con cada uno en PL/SQL**

Aquí tienes un ejemplo para cada caso.

**🔹 Ejemplo con RECORD (una sola fila)**

DECLARE

TYPE emp\_rec IS RECORD (emp\_no NUMBER, emp\_name VARCHAR2(150));

empleado emp\_rec; -- Solo almacena 1 fila

BEGIN

SELECT emp\_no, emp\_name

INTO empleado -- ✅ Guarda una sola fila

FROM emp

WHERE emp\_no = 1002;

DBMS\_OUTPUT.PUT\_LINE('Empleado: ' || empleado.emp\_name);

END;

🔴 **NO se puede usar BULK COLLECT aquí** porque empleado solo almacena **una fila**.

**🔹 Ejemplo con TABLE OF RECORD (varias filas, sin límite)**

DECLARE

TYPE emp\_rec IS RECORD (emp\_no NUMBER, emp\_name VARCHAR2(150));

TYPE emp\_tabla IS TABLE OF emp\_rec; -- ✅ Colección sin límite

empleados emp\_tabla;

BEGIN

SELECT emp\_no, emp\_name

BULK COLLECT INTO empleados -- ✅ Se usa BULK COLLECT

FROM emp;

-- Recorrer la colección

FOR i IN 1..empleados.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('Empleado: ' || empleados(i).emp\_name);

END LOOP;

END;

✅ **Se usa BULK COLLECT con TABLE OF RECORD** porque almacena **varias filas**.

**🔹 Ejemplo con VARRAY (varias filas, con límite)**

DECLARE

TYPE emp\_varray IS VARRAY(5) OF VARCHAR2(150); -- ✅ Solo 5 empleados

empleados emp\_varray := emp\_varray(); -- Inicializar la colección

BEGIN

SELECT emp\_name

BULK COLLECT INTO empleados -- ✅ Se usa BULK COLLECT pero con límite

FROM emp

WHERE rownum <= 5; -- Asegurar que no haya más de 5 registros

-- Recorrer el VARRAY

FOR i IN 1..empleados.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('Empleado: ' || empleados(i));

END LOOP;

END;

🔴 **Si la consulta devuelve más de 5 filas, dará error**, porque VARRAY(5) **tiene un límite fijo**.

**🎯 ¿Cuándo usar cada uno?**

| **Caso** | **Usar** |
| --- | --- |
| Solo necesitas **una fila con varias columnas** | RECORD |
| Necesitas **varias filas** sin límite | TABLE OF RECORD |
| Necesitas **varias filas pero con un límite fijo** | VARRAY |
| Quieres recuperar **varias filas de una consulta** de manera eficiente | BULK COLLECT con TABLE OF o VARRAY |

🔥 **Resumen Final**

* RECORD = **1 fila con varias columnas**.
* TABLE OF RECORD = **Varias filas, sin límite**.
* VARRAY(N) = **Varias filas, pero con límite**.
* BULK COLLECT = **Para recuperar varias filas en una colección** (TABLE OF o VARRAY).

💡 ¡Ahora dominas todas las opciones de almacenamiento en PL/SQL! 🚀🔥

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*parte de CURSORES PAR ABAJO\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CURSORES

Introducción:

• Definimos un cursor como un área de memoria que se reserva para ejecutar sentencias

SQL y para almacenar información de procesamiento de las mismas

• Con los cursores se puede acceder al área de trabajo o contexto para procesar la

información recuperada en una consulta

• Existen dos tipos de cursores:

Explícitos.- Son definidos por el programador para acceder a las filas recuperadas

en una consulta.

Implícitos.- Generados automáticamente por Oracle al ejecutar una instrucción no

asociada a un cursor por el programador. Es decir, cuando se lanza una sentencia

SQL tiene un cursor asociado

• Para una mejor comprensión, podríamos decir que utilizar cursores explícitos es como

trabajar con ficheros virtuales en memoria, donde el fichero sería el cursor y cada una

de las filas recuperadas en la consulta sería un registro.

• Cuando lanzamos una sentencia SELECT esta debe devolver una única fila , de lo

contrario obtendremos la excepción TOO\_MANY\_ROWS. Pero en muchas ocasiones

existe la necesidad de trabajar con más de una fila, para ello utilizamos los cursores

explícitos

Declaración:

• Para declarar un cursor utilizaremos el siguiente formato:

DECLARE

tipo :=

CURSOR nom\_cur [(param1 { } [{ }valor], ...)]

var %TYPE DEFAULT

IS instrucción\_select;

nom\_cur.- Nombre del cursor a declarar.

param1.- Variable usada como parámetro en la instrucción SELECT. Puede aparecer

más de un parámetro. Se debe especificar su tipo de forma obligatoria, siendo opcional

la posibilidad de darle un valor por defecto.

Instrucción\_select.- Instrucción SELECT que especifica las distintas condiciones y

procedencias que deben cumplir las filas recuperadas para el cursor. En caso de que el

cursor incorpore algún parámetro, éste aparecerá en cualquier parte de dicha

instrucción.

Ejemplo:

CURSOR c\_vecinos IS SELECT \* FROM vecinos;

CURSOR c\_libros IS

SELECT nom\_autor, titulo, editorial FROM libros;

CURSOR c\_facturas IS

SELECT \* FROM facturas WHERE unidades > 70;

CURSOR c\_alumnos (nota notas.ev1 %TYPE) IS

SELECT nom\_alu, ape1\_alu, ape2\_alu, ev1

FROM alumnos a, notas n

WHERE a.num\_alu = n.num\_alu AND ev1>=nota;

• Una vez declarado el cursor, para poder acceder a sus datos debemos abrirlo. El

formato es:

BEGIN

OPEN nom\_cur [(valor1[, ...])];

END;

nom\_cur.- Nombre del cursor a abrir.

valor1.- Valor que tomará el parámetro. Es de obligada especificación únicamente en

el caso de que, al declarar el cursor, se especificará al menos un parámetro y no se le

hubiera asignado valor por defecto.

Ejemplo: OPEN c\_vecinos;

OPEN c\_libros;

OPEN c\_facturas;

OPEN c\_alumnos(7); • OPEN asigna de forma dinámica una area de memoria para que sea utilizada por el

cursor

• Obtiene lo valores de la sentencia select

• Posiciona el puntero del cursor en la primera fila de las filas recuperadas

• Una vez abierto el cursor, podemos acceder a las filas en él almacenadas, para ello

utilizaremos la instrucción FECHT, cuyo formato es:

BEGIN

.....

FETCH nom\_cur INTO {variable1[, variable2 ...]| registro};

....

END;

/

• La sentencia FECHT lee la fila marcada por el puntero, poniendo el valor de sus

columnas en las variables que siguen a la palabra clave INTO.

• La lectura de una fila mediante FECHT hace que el puntero pase a señalar la fila

siguiente, con lo que el acceso se realizará secuencialmente en el orden en que se

recuperaron las filas.

• Debe existir una correspondencia en cuanto a número y tipo entre las columnas de la

consulta y las variables de FECHT.

• A continuación ponemos un ejemplo de un cursor al que hay que pasarle un

parámetro. No debemos olvidar que para abrir un cursor con valores diferentes para

los parámetros, debemos cerrarlo antes de abrirlo si ya ha sido utilizado con

anterioridad.

Ejemplo:

CREATE OR REPLACE PROCEDURE ver\_emple\_apell(cadena VARCHAR2)

AS

cad varchar2(70);

CURSOR c\_emple (cad varchar2)IS SELECT ename, empno FROM emp

WHERE ename LIKE cad;

vr\_emple c\_emple%ROWTYPE;

BEGIN

cad:=upper(cadena);

cad:='%'||cad||'%';

OPEN c\_emple(cad);

FETCH c\_emple INTO vr\_emple;

WHILE (c\_emple%FOUND) LOOP

DBMS\_OUTPUT.PUT\_LINE(vr\_emple.empno||'\*'||vr\_emple.ename);

FETCH c\_emple INTO vr\_emple;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('NUMERO DE EMPLEADOS: '|| c\_emple

%ROWCOUNT);

CLOSE c\_emple;

END ver\_emple\_apell;

/

Ejemplo:

DECLARE

CURSOR c\_notas IS SELECT \* FROM notas;

num NUMBER(2);

nota1 NUMBER(4,2);

nota2 NUMBER(4,2);

BEGIN

OPEN c\_notas;

FETCH c\_notas INTO num, nota1, nota2;

...

END;

/

• Una vez finalizada el acceso a las filas el cursor debe cerrarse, para ello utilizaremos la

instrucción CLOSE , cuyo formato es:

CLOSE nom\_cur;

nom\_cur.- Nombre del cursor a cerrar.

Ejemplo: CLOSE c\_notas;

• Cada cursor tiene asociados cuatro atributos predefinidos por el sistema. Cuando se

trabaja con el cursor, estos atributos devuelven información sobre el resultado de la

sentencia ejecutada sobre el cursor. Los atributos son:

%ISOPEN

Retorna TRUE si el cursor está abierto y FALSE si está cerrado.

%FOUND

Devuelve FALSE si el último FECHT ejecutado no encontró más filas y en caso de

encontrar una fila devuelve TRUE. Después de abrir el cursor y antes del primer

FECHT retorna NULL.

EJEMPLO

BEGIN

OPEN cur1;

FETCH cur1 INTO v\_nom,v\_loc;

WHILE cur1%FOUND LOOP

Dbms\_output.put\_line(v\_nom|| v\_loc);

FETCH cur1 INTO v\_nom,v\_loc;

END LOOP;

CLOSE cur1;

END;

%NOTFOUND

Devuelve FALSE si FECHT encontró una fila y TRUE en caso de no encontrarla. Es

exactamente lo contrario del atributo anterior. Después de abrir el cursor y antes del

primer FECHT retorna NULL.

EJEMPLO:

….

EXIT WHEN CUR1%notfound; -- Se suele utilizar como salida de los bucles

%ROWCOUNT

Devuelve el número de filas accedidas por FECHT hasta el momento.

• Para utilizar estos atributos nombre\_cursor%vble

• El atributo %ISOPEN es realmente útil con cursores explícitos, ya que el control de

apertura y cierre de los cursores explícitos lo tiene el programador, mientras que los

cursores implícitos los controla el servidor de Oracle

BUCLE FOR PARA TRATAR CURSORES MEDIANTE REGISTROS.

• Cuando se trabaja con cursores y registros, existe una versión del bucle FOR específica

para el tratamiento de la información. Su formato es:

FOR nom\_registro IN nom\_cursor [(valor1[,...])] LOOP

sentencias;

END LOOP;

nom\_registro.- Nombre del registro que generará automáticamente el sistema para

el control del cursor asociado.

nom\_cursor.- Nombre del cursor que el bucle controlará su tratamiento.

valor1,....- Valores de los parámetros del cursor.

• La sentencia FOR se encarga de abrir el cursor y recupera la primera fila en su

ejecución. En sucesivas ejecuciones, va recuperando cada una de las filas del cursor.

Cuando ya no hay más filas, se cierra automáticamente el cursor y se termina la

ejecución del bucle.

• Los valores de las filas del cursor son accesibles a través de los campos del registro, el

cual, lo declara el sistema automáticamente y es del tipo nom\_cursor %ROWTYPE.

• Al registro no se le puede hacer referencia fuera del bucle FOR.

Ejemplo:

DECLARE

CURSOR c\_notas (nota notas.ev1 %TYPE) IS

SELECT num\_alu, ev1 FROM notas

WHERE ev1 >= nota

FOR UPDATE;

BEGIN

FOR var\_notas IN c\_notas(7) LOOP

IF var\_notas.ev1>7 AND var\_notas.ev1<10 THEN

var\_notas.ev1 := var\_notas.ev1 + 1;

UPDATE notas SET ev1 = var\_notas.ev1

WHERE CURRENT OF c\_notas;

END IF;

END LOOP;

COMMIT;

END;

/! El bucle FOR se encarga generar el cursor, de abrirlo, posicionar el puntero en

todas y cada una de las filas del cursor y de cerrarlo después de procesar la última

fila.

Bucles For para Cursores

• Procesan filas de un cursor explícito.

• Cuando usamos un bucke For de estas características no tenemos que preocuparnos

de abrir,cerrar y realizar la instrucción fetch.

• Lo único que hacemos es declararlo

FOR id\_reg IN id\_cursor LOOP

Sentencias;

END LOOP;

Id\_reg: es una variable de tipo registro donde se almacenan los datos abiertos por el

cursor. No hace falta declararlo y su ambito es unicamente el bucle

Id\_cursor: Es el identificador del cursor que previamente hemos declarado

• El bucle se termina automáticamente cuando se recupera la última fila

EJEMPLO:

DECLARE

CURSOR MICUR IS SELECT \* FROM scott.salgrade WHERE hisal>=2000;

BEGIN

FOR REG\_EMP IN MICUR LOOP

DBMS\_OUTPUT.PUT\_LINE(to\_char(micur%rowcount,’99.’)||REG\_EMP.GRADE||':'||

reg\_emp.grade);

end loop;

end;

/

%FOUND %ISOPEN %NOTFOUND %ROWCOUNT

open Antes Invalid\_cursor F Invalid\_cursor Invalid

Después NULL T NULL 0

Primer

fetch

Antes NULL T NULL 0

Después T T F 1

Ultimo

fetch

Antes T T F N

Después T T T N

close Antes T T T N

Después Invalid\_cursor F invalid invalid

Atributos en cursores implícitos

• Se abre implícitamente un cursor cuando se procesa un comando SQL que no está

asociado a un cursor explícito

• El cursor implícito se llama SQL y dispone de los 4 atributos vistos anteriormente

SQL%NOTFOUND: TRUE si el último INSERT,UPDATE,DELETE o SELECT han fallado o

no hay filas

SQL%FOUND: TRUE si el último INSERT,UPDATE,DELETE o SELECT han afectado a una

o mas filas

SQL%ROWCOUNT devuelve el número de filas afectadas en el último

INSERT,UPDATE,DELETE o SELECT

SQL%ISOPEN siempre devuelve falso ya que oracle cierra automáticamente el cursor

después de hacer la sentencia SQL asociada

Diferencias del cursor implicito con un cursor explicito

• En el caso de SELECT INTO debe devolver una y solo una fila de lo contrario se

producen errores y se levantan automáticamente las excepciones:

NO\_DATA\_FOUND: Ninguna fila

TOO\_MANY\_ROWS: Más de una fila

Se detiene la ejecución del programa y se bifurca a la sección de EXCEPTION

• Esto no es aplicable a INSERT,UPDATE,DELETE, por eso tiene sentido SQL

%ROWCOUNT

Utilización de una subconsulta en un bucle for

• Si utilizamos una subconsulta en un bucle FOR de cursor no hace

falta declararse el cursor en la zona declarativa

EJEMPLO

BEGIN

FOR REG\_EMP IN (SELECT \* FROM scott.salgrade WHERE hisal>=2000) LOOP

DBMS\_OUTPUT.PUT\_LINE(REG\_EMP.GRADE||':'|| reg\_emp.grade);

end loop;

end;

/

alias en las columnas

• Cuando utilizamos variables registro declaradas del mismo tipo que el cursor o que la

tabla, los campos tiene el mismo nombre que las columnas correpondientes.

Ejemplo:

CURSOR C1 Select dept\_no, count(\*) from emple group by dept\_no;

OPEN C1;

FETCH c1 INTO v\_reg;

• Para visualizar una columna hacemos:

V\_reg.dept\_no

• Para visualizar lo que hemos contado no podríamos por eso debemos ponerles un alias

Ejemplo:

CURSOR C1 Select dept\_no, count(\*) total from emple group by dept\_no;

OPEN C1;

FETCH c1 INTO v\_reg;

• Para visualizar el total hacemos:

V\_reg.total

EJERCICIO:

Escribir un bloque PL/SQL que visualice el ename y hiredate(fecha

de alta) de la empresa, ordenados por fecha:

a) Mediante for de cursor

b) Mediante while

SOLUCIONES:

a) DECLARE

CURSOR c\_emple IS SELECT ename,hiredate\_from emple order by 2;

V\_reg\_emp c\_emp%rowtype;

Begin

For v\_reg\_emp IN c\_emple LOOP

DBMS\_OUTPUT.PUT\_LINE(v\_reg\_emp.ename||v\_reg\_emp.hiredate)

END LOOP;

END;

/

b) DECLARE

CURSOR c\_emple IS SELECT ename,hiredate\_from emple order by 2;

V\_reg\_emp c\_emp%rowtype;

Begin

OPEN c\_emple;

FETCH c\_emple INTOv\_reg\_emp;

WHILE c\_emple%FOUND LOOP

DBMS\_OUTPUT.PUT\_LINE(v\_reg\_emp.ename||v\_reg\_emp.hiredate)

FETCH c\_emple INTOv\_reg\_emp;

END LOOP;

Close c\_emple;

END;

/

FOR UPDATE :

• A veces podemos requerir bloquear las filas devueltas por el cursor para realizar alguna

modificación en los datos.

• Para mantener bloqueadas las filas durante el proceso se utiliza FOR UPDATE en la

sentencia SELECT asociada al cursor

DECLARE

CURSOR nombre IS SELECT \* FROM notas FOR UPDATE;

• La clausula FOR UPDATE es la última de la sentencia SELECT incluso después de

ORDER BY si lo hubiese

• Los bloqueos de las filas se realizan en la apertura del cursor, no en las recuperaciones

• No se debe lanzar una sentencia Commit antes de cerrar un cursor que lleve la

cláusula FOR UPDATE ya que en ese caso se liberan los bloqueos realizados por el

cursor

Ejemplo:

DECLARE

CURSOR c\_dept IS SELECT \* FROM emp where deptno=10 FOR UPDATE;

BEGIN

OPEN c\_dept;

FETCH c\_dept INTO v\_reg;

WHILE c\_dept %FOUND LOOP

IF v\_reg.comm IS NULL THEN

UPDATE emp SET comm=300

WHERE empno=v\_reg.empno;

END IF;

FETCH c\_dept INTO v\_reg;

END LOOP;

CLOSE c\_notas;

COMMIT;

END;

/

WHERE CURRENT OF nom\_cursor;

nom\_cursor.- Nombre del cursor que recuperó las filas a modificar o borrar.

• Se utiliza WHERE CURRENT OF para hacer referencia a la fila que se está utilizando

actualmente sin necesidad de hacer referencia explícitamente con ROWID o algún

identificativo de la fila

• Para utilizar esta claúsula , la sentencia select debe incluir la cláusula FOR UPDATE

• No se puede utilizar WHERE CURRENT OF cuando la select accede a más de una tabla

• Sintaxis:

WHERE CURRENT OF id\_cursor;

Ejemplo:

DECLARE

CURSOR c\_notas (nota notas.ev1 %TYPE) IS

SELECT nom\_alu, ape1\_alu, ape2\_alu, ev1

FROM alumnos a, notas n

WHERE a.num\_alu = n.num\_alu

AND ev1 >= nota FOR UPDATE;

nombre VARCHAR2(9);

ape1 nombre %TYPE;

ape2 nombre %TYPE;

nota1 NUMBER(4,2);

BEGIN

OPEN c\_notas(7);

FETCH c\_notas INTO nombre, ape1, ape2, nota1;

WHILE c\_notas %FOUND LOOP

IF nota1 > 7 AND nota1 < 10 THEN

nota1 := nota1 + 1;

UPDATE notas SET ev1 = nota1

WHERE CURRENT OF c\_notas;

END IF;

FETCH c\_notas INTO nombre, ape1, ape2, nota1;

END LOOP;

CLOSE c\_notas;

COMMIT;

END;

/

! ERROR, ya que el cursor está formado por más de una tabla y por tanto no

puede coincidir con la tabla notas que intentamos actualizar.

• La solución al caso anterior pasa por no utilizar la opción WHERE CURRENT OF

nom\_cursor y utilizar otra condición en la cláusula WHERE.

Ejemplo:

DECLARE

CURSOR c\_notas (nota notas.ev1 %TYPE) IS

SELECT n.num\_alu,nom\_alu,ape1\_alu,ape2\_alu,ev1

FROM alumnos a, notas n

WHERE a.num\_alu=n.num\_alu AND ev1>=nota;

num NUMBER(2);

nombre VARCHAR2(9);

ape1 nombre %TYPE;

ape2 nombre %TYPE;

nota1 NUMBER(4,2);

BEGIN

OPEN c\_notas(7);

FETCH c\_notas INTO num,nombre,ape1,ape2,nota1;

WHILE c\_notas %FOUND LOOP

IF nota1 > 7 AND nota1 < 10 THEN

nota1 := nota1 + 1;

UPDATE notas SET ev1 = nota1

WHERE num\_alu = num;

END IF;

FETCH c\_notas INTO num, nombre, ape1, ape2, nota1;

END LOOP;

CLOSE c\_notas;

COMMIT;

END;

/

Ejemplo correcto:

DECLARE

CURSOR c1 IS

SELECT \*

FROM emp

WHERE deptno=10 FOR UPDATE;

BEGIN

FOR REG IN C1 LOOP

IF REG.COMM IS NULL THEN

UPDATE EMP SET COMM=300 WHERE CURRENT OF C1;

END IF;

END LOOP;

END;

/

AMBITO DE LAS VARIABLES PL/SQL

• Cuando Existen bloques anidados es muy importante

determinar el ámbito de las variables, es decir, determinar la

región del programa

• Un identificador estará visible en el bloque en el que está

declarado y en todos los subbloques anidados, dentro de él.

• Un bloque no podrá referenciar identificadores que sean

declarados en bloques anidados en él.

• Cuando un bloque encuentra un identificador en alguna de

sus sentencias, busca en su zona declarativa la declaración

de este identificador. En caso de no encontrarlo pasará a

revisar la zona declarativa de su bloque inmediatamente

exterior hasta localizarlo.

DECLARE

A NUMBER;

BEGIN

AMBITO DE A

DECLARE

B NUMBER;

BEGIN

AMBITO DE B

END;

END;

CONTROL DE ERRORES. EXCEPCIONES

• En PL/SQl se denomina excepción a cualquier identificador que nos permita manejra un

error que se produzca durante la ejecución del programa, las excepciones pueden ser

definidas internamente por el sistema o definidas por el usuario.

• El tratamiento de excepciones de PL/SQL clarifica el código del programa al separarlo

del resto del proceso, definiendo las acciones a realizar en caso de producirse algún

error.

• Cuando se produce un error, se activa una excepción y la ejecución normal del

programa se detiene y finaliza el bloque donde se produzca el error, pasándose

el control a la sección de excepciones del bloque.

• En la parte de excepciones del programa existirán distintas rutinas, cada una dedicada

a procesar una o varias excepciones.

• Cada rutina comienza con una cláusula WHEN donde se especifica la excepción o

excepciones a las que se aplicará dicha rutina y después de ejecutarse la rutina

correspondiente a un error, el proceso continúa con la siguiente sentencia

del bloque contenedor.

• Si queremos que una misma rutina trate varias excepciones, detrás de la palabra

WHEN pondremos los nombres de esas excepciones, separados por el operador

relacional OR.

Sintaxis:

EXCEPTION

WHEN exception1 [OR exception2...]THEN

Instrucciones;

WHEN exception2 [OR exception2...]THEN

Instrucciones2;

WHEN OTHERS

Instrucciones3;

• La palabra EXCEPTION es obligatoria.

• Las excepciones no pueden aparecer en asignaciones de variables ni en sentencias SQL

• Como mucho podremos tener una claúsula OTHERS y en ella se filtran todos los

errores que no tengan su correspondiente manejador

• Podríamos tener en una zona de excepciones como único manejador la clausula

OTHERS de tal modo que cualquier error en ejecución se controla dentro de esta zona.

• Un error producido en la ejecución de la sección de excepciones no puede ser tratado

en ella. En este caso, se termina la ejecución del bloque con un error que se intentará

tratar en algunos de los bloques externos.

• En la declaración de variables pueden producirse excepciones al realizar inicializaciones

incorrectas. Las excepciones producidas en la parte de declaraciones no pueden ser

tratadas en el bloque, dando lugar a la finalización de la ejecución del bloque con un

error no tratado.

Ejemplo:

DECLARE

maximo CONSTANT NUMBER(4) := 10000;

! Finalización del programa con error.

Ejemplo:

DECLARE

comunidad NUMBER(7);

BEGIN

BEGIN

SELECT pts\_comunidad INTO comunidad

FROM vecinos

WHERE apellido1 LIKE 'SALVADOR';

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

SELECT pts\_comunidad INTO comunidad

FROM vecinos

WHERE apellido1 LIKE 'CASTRILLO';

END;

IF comunidad < 7000 THEN

comunidad := comunidad + 1000;

END IF;

INSERT INTO vecinos VALUES (8, 'A', 'ROBERTO', 'LOSADA',

'22-JUN-99', comunidad, comunidad/166.386);

COMMIT;

END;

/

• Si en la parte de excepciones de un bloque no existe rutina para el tratamiento de un

determinado error, o si no existe sección de excepciones, dicho error pasa sin tratar al

bloque contenedor, que intentará tratarlo en su sección de excepciones. De esta forma,

un error no tratado va pasando de bloque en bloque hasta que es tratado o hasta que

termina el programa con error.

• Existen dos tipos de excepciones:

Excepciones predefinidas.- Son activadas por el Servidor Oracle de forma

automática cuando se produce el error correspondiente.

Excepciones definidas por el usuario.- Deben de activarse explícitamente con la

sentencia RAISE.

Excepciones predefinidas

CURSOR\_ALREADY\_OPEN.- Se intenta abrir un cursor ya abierto.

DUP\_VAL\_ON\_INDEX.- Intento de insertar o actualizar, violando la condición de

unicidad de índice.

INVALID\_CURSOR.- Operación ilegal sobre un cursor.

INVALID\_NUMBER.- En una sentencia SQL se intentó realizar una conversión de

cadena de caracteres a número y la cadena contenía caracteres no numéricos.

LOGIN\_DENIED.- Se intentó iniciar una sesión de base de datos con nombre de

usuario o password no válidos.

NO\_DATA\_FOUND.- Una sentencia SELECT ... INTO no devuelve ninguna fila.

NOT\_LOGGED\_ON.- PL/SQL realiza una llamada a ORACLE y no existe conexión.

PROGRAM\_ERROR.- Error interno de PL/SQL.

STORAGE\_ERROR.- Se produce un error de memoria.

TIMEOUT\_ON\_RESOURCE.- Se termina el tiempo de espera por un recurso.

TOO\_MANY\_ROWS.- Una sentencia SELECT ... INTO devuelve más de una fila.

VALUE\_ERROR .- Se produjo un error aritmético o de conversión, un

truncamiento o la violación de una restricción.

ZERO\_DIVIDE.- Se intentó dividir por cero.

OTHERS.- En esta rutina se tratará cualquier excepción para la que no exista

tratamiento específico.

Hay alguna más pero estas son las más utilizadas y tenemos que tener en cuenta

que no es necesario declararlas en la sección DECLARE.

Excepciones definidas por el usuario

El programador puede definir sus propias excepciones, las cuales deben ser declaradas,

llamadas y activadas. Los formatos a utilizar son:

DECLARE

...

nom\_excep\_usu EXCEPTION; ! Declaración de excepción

...

BEGIN

...

RAISE nom\_excep\_usu; ! Llamada a la excepción

...

EXCEPTION

...

WHEN nom\_excep\_usu THEN ! Activar sentencias ejecutar

sentencias;

...

END;

/

Pasos a realizar:

Se requieren tres pasos para trabajar con excepciones de usuario:

Definición: se realiza en la zona de DECLARE con el siguiente formato:

nombre\_excepción EXCEPTION

Disparar o levantar la excepción mediante la orden raise: RAISE ;

Tratar la excepción en el apartado EXCEPTION: WHEN THEN ;

Ejemplo: De la tabla VECINOS incrementar en 1000 pesetas su columna de

PTS\_COMUNIDAD y la cantidad correspondiente a la columna de valor en euros,

salvo para las filas de fecha llegada nula, que se modificará únicamente la columna

FECHA\_LLEGADA con el valor de la fecha de sistema.

DECLARE

fecha\_nula EXCEPTION;

fecha\_no\_nula EXCEPTION;

CURSOR c\_vecinos IS

SELECT \* FROM vecinos FOR UPDATE;

BEGIN

FOR i IN c\_vecinos LOOP

BEGIN

IF i.fecha\_llegada IS NULL THEN

RAISE fecha\_nula;

END IF;

RAISE fecha\_no\_nula;

EXCEPTION

WHEN fecha\_nula THEN

UPDATE vecinos SET fecha\_llegada = SYSDATE

WHERE CURRENT OF c\_vecinos;

WHEN fecha\_no\_nula THEN

i.pts\_comunidad := i.pts\_comunidad + 1000;

UPDATE vecinos SET

pts\_comunidad = i.pts\_comunidad,

e\_comunidad = i.pts\_comunidad / 166.386

WHERE CURRENT OF c\_vecinos;

END;

END LOOP;

COMMIT;

END;

/

Otras excepciones

Existen otros errores internos de Oracle que no tienen asignada una excepción, sino

un código de error y un mensaje, a los que se accede mediante funciones SQLCODE

y SQLERRM.

Cuando se produce un error de estos se trasfiere directamente el control a la

sección EXCEPTION donde se tratara el error en la clausula WHEN OTHERS de la

siguiente forma:

WHEN OTHERS THEN DBMS\_OUTPUT.PUT\_LINE('Error'||SQLCODE||SQLERRM.)

RAISE\_APPLICATION\_ERROR

En el paquete DBMS\_STANDARD se incluye un procedimiento llamado

RAISE\_APPLICATION\_ERROR que nos sirve para levantar errores y definir mensajes

de error. Su formato es el siguiente:

RAISE\_APPLICATION\_ERROR(numero\_error,mensaje\_error);

Es importante saber que el numero de error esta comprendido entre -20000 y

-20999 y el mensaje es una cadena de caracteres de hasta 512 bytes.

Este procedimiento crea una excepción que solo puede ser tratada en WHEN

OTHERS.

Ejemplo:

CREATE or REPLACE PROCEDURE subir\_horas (emple NUMBER, horas\_subir

NUMBER)

IS

horas\_actuales NUMBER;

BEGIN

Select horas into horas\_actuales from empleados where id\_empleado=emple;

if horas\_actuales is NULL then

RAISE\_APPLICATION\_ERROR(-20010,'No tiene horas');

else

update empleados set horas=horas\_actuales + horas\_subir where

id\_empleado=emple;

end if;

End subir\_horas;

**Explicación del Documento sobre Paquetes en PL/SQL**

Los **paquetes en PL/SQL** permiten **agrupar** de manera lógica **procedimientos, funciones, variables, tipos de datos y cursores** en una sola unidad dentro de la base de datos. Esto mejora la organización y modularización del código, permitiendo ocultar la implementación interna y optimizando el rendimiento.

**¿Por qué el PACKAGE BODY no tiene BEGIN?**

* **Un paquete (PACKAGE) en PL/SQL es una colección de procedimientos, funciones, cursores y variables.**
* **El PACKAGE BODY solo implementa lo que se ha declarado en el PACKAGE.**
* **Como el PACKAGE BODY no ejecuta código directamente, no necesita un BEGIN.**
* **Solo los procedimientos y funciones dentro del PACKAGE BODY necesitan un BEGIN para ejecutar su lógica.**

**Ejemplo de estructura correcta de un paquete en PL/SQL**

**1️⃣ Especificación del paquete (PACKAGE)**

**Aquí solo declaramos los procedimientos y funciones, sin definir su lógica.**

**CREATE OR REPLACE PACKAGE pk AS**

**PROCEDURE datos\_cliente(p\_cod\_cliente IN clientes.cod\_cliente%TYPE);**

**END pk;**

**/**

**2️⃣ Implementación del paquete (PACKAGE BODY)**

**Aquí escribimos el código de los procedimientos.**

**CREATE OR REPLACE PACKAGE BODY pk IS -- No necesita BEGIN aquí**

**PROCEDURE datos\_cliente(p\_cod\_cliente IN clientes.cod\_cliente%TYPE) IS**

**-- Declaración de variables y cursor**

**TYPE t\_datos\_cliente IS RECORD (**

**cod\_cliente clientes.cod\_cliente%TYPE,**

**nombres clientes.nombres%TYPE,**

**distrito clientes.distrito%TYPE,**

**telefono clientes.telefono%TYPE**

**);**

**v\_aux t\_datos\_cliente;**

**CURSOR c\_datos\_cliente IS**

**SELECT cod\_cliente, nombres, distrito, telefono**

**FROM clientes**

**WHERE cod\_cliente = p\_cod\_cliente;**

**BEGIN -- Aquí empieza la ejecución del procedimiento**

**OPEN c\_datos\_cliente;**

**FETCH c\_datos\_cliente INTO v\_aux;**

**WHILE c\_datos\_cliente%FOUND LOOP**

**dbms\_output.put\_line(lpad('-', 70, '-'));**

**dbms\_output.put\_line('DATOS EMPLEADO' );**

**dbms\_output.put\_line(lpad('-', 70, '-'));**

**dbms\_output.put\_line(v\_aux.cod\_cliente||' ' || v\_aux.nombres);**

**dbms\_output.put\_line(v\_aux.distrito||' ' || v\_aux.telefono);**

**FETCH c\_datos\_cliente INTO v\_aux;**

**END LOOP;**

**CLOSE c\_datos\_cliente;**

**END datos\_cliente; -- Fin del procedimiento**

**END pk; -- Fin del paquete**

**/**

**✔ El PACKAGE BODY no tiene BEGIN porque solo define procedimientos y funciones.  
✔ Cada procedimiento y función dentro del PACKAGE BODY sí tiene su propio BEGIN y END.  
✔ El END pk; cierra el PACKAGE BODY.**

**💡 Si el PACKAGE BODY tuviera un BEGIN, significaría que queremos ejecutar código automáticamente al compilarlo, lo cual no es el propósito de un paquete. 🚀**

**📌 Conceptos Clave**

1. **División de un Paquete**
   * **Especificación (Obligatoria)**: Define los elementos públicos (funciones, procedimientos, variables, cursores) accesibles desde fuera del paquete.
   * **Cuerpo del Paquete (Opcional)**: Implementa las funciones y procedimientos declarados en la especificación. Contiene detalles internos ocultos a los usuarios externos.
2. **Beneficios de Usar Paquetes**
   * Modulariza el código, facilitando su mantenimiento.
   * Oculta detalles de implementación, protegiendo la integridad del código.
   * Mejora el rendimiento al cargar el paquete en memoria en la primera ejecución.
   * Permite la **sobrecarga de funciones** (varias funciones con el mismo nombre pero con diferentes parámetros).
3. **Cómo Crear un Paquete**
   * Se crean en **dos pasos**:
     1. **Cabecera del paquete** (PACKAGE).
     2. **Cuerpo del paquete** (PACKAGE BODY).

**📜 Reglas de Oro**

1. **Toda función o procedimiento debe estar declarado en la cabecera si se desea usar fuera del paquete.**
2. **Las variables y cursores pueden ser públicas (cabecera) o privadas (solo en el cuerpo).**
3. **Para llamar un procedimiento o función de un paquete, se usa:**

nombre\_paquete.nombre\_procedimiento(parámetros);

1. **Un paquete se ejecuta solo una vez por sesión cuando se usa por primera vez.**
2. **La sobrecarga de funciones permite reutilizar nombres de funciones con distintos parámetros.**
3. **Se puede usar una sección de inicialización en el paquete, que se ejecuta solo una vez.**
4. **Los paquetes de Oracle ofrecen funciones útiles (ej. DBMS\_OUTPUT.PUT\_LINE).**

**📌 Ejemplo de Creación de un Paquete en PL/SQL**

-- CABECERA DEL PAQUETE

CREATE OR REPLACE PACKAGE PK1 IS

-- Declaración de un procedimiento accesible desde fuera

PROCEDURE listar\_articulos(familia IN Articulos.cArtFml%TYPE);

END PK1;

/

-- CUERPO DEL PAQUETE

CREATE OR REPLACE PACKAGE BODY PK1 IS

PROCEDURE listar\_articulos(familia IN Articulos.cArtFml%TYPE) IS

codigo Articulos.cArtCdg%TYPE;

descripcion Articulos.cArtDsc%TYPE;

-- Cursor para seleccionar los artículos de una familia específica

CURSOR cArticulos IS

SELECT cArtCdg, cArtDsc FROM Articulos WHERE cArtFml = familia;

BEGIN

OPEN cArticulos;

LOOP

FETCH cArticulos INTO codigo, descripcion;

EXIT WHEN cArticulos%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(descripcion);

END LOOP;

CLOSE cArticulos;

END listar\_articulos;

END PK1;

/

Para ejecutar el procedimiento desde fuera del paquete:

BEGIN

PK1.listar\_articulos('ELEC'); -- Lista artículos de la familia "ELEC"

END;

/

**📌 Declaración de Cursores en Paquetes**

Podemos declarar cursores dentro de los paquetes de esta forma:

CREATE OR REPLACE PACKAGE empleados AS

CURSOR a RETURN empleado%ROWTYPE;

END empleados;

/

CREATE OR REPLACE PACKAGE BODY empleados AS

CURSOR a RETURN empleado%ROWTYPE IS

SELECT \* FROM empleado WHERE salario < 10000;

END empleados;

/

**📌 Paquetes de Oracle Útiles**

Algunos paquetes predefinidos en Oracle:

* **DBMS\_OUTPUT** → Imprimir mensajes en la consola (PUT\_LINE).
* **DBMS\_SQL** → Ejecutar SQL dinámicamente.
* **STANDARD** → Contiene funciones como TO\_CHAR y ABS.
* **DBMS\_STANDARD** → Manejo de errores con RAISE\_APPLICATION\_ERROR.

Con esta guía, puedes empezar a practicar la creación y uso de **paquetes en PL/SQL** con ejemplos claros y reglas esenciales. 🚀

✅ Cursor Público (en PACKAGE) → Lo pueden usar todos los procedimientos del paquete y se puede llamar desde fuera.

✅ Cursor Privado (en PACKAGE BODY) → Solo lo pueden usar los procedimientos dentro del PACKAGE BODY.

✅ Cursor Local (dentro de un PROCEDURE) → Solo lo puede usar ese procedimiento específico.

🔹 Si necesitas que varios procedimientos compartan un cursor, usa un cursor público o privado. Si el cursor es solo para un procedimiento, decláralo dentro de él. 🎯🚀

🔍 Ejemplo para Verlo Claramente

-- CABECERA DEL PAQUETE: DECLARAMOS UN CURSOR PÚBLICO

CREATE OR REPLACE PACKAGE pk\_empleados AS

-- Cursor público: Puede ser usado en cualquier procedimiento dentro del paquete

CURSOR c\_publico IS

SELECT id\_empleado, nombre, salario

FROM empleados;

-- Procedimiento público

PROCEDURE listar\_empleados\_publico;

PROCEDURE listar\_empleados\_privado;

END pk\_empleados;

/

-- CUERPO DEL PAQUETE: IMPLEMENTAMOS LOS PROCEDIMIENTOS

CREATE OR REPLACE PACKAGE BODY pk\_empleados AS

-- Cursor privado: Puede ser usado por cualquier procedimiento dentro del PACKAGE BODY

CURSOR c\_privado IS

SELECT id\_empleado, nombre, salario

FROM empleados WHERE salario > 3000;

-- Procedimiento que usa el cursor público

PROCEDURE listar\_empleados\_publico IS

v\_id empleados.id\_empleado%TYPE;

v\_nombre empleados.nombre%TYPE;

v\_salario empleados.salario%TYPE;

BEGIN

OPEN c\_publico;

LOOP

FETCH c\_publico INTO v\_id, v\_nombre, v\_salario;

EXIT WHEN c\_publico%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('PUBLICO - ID: ' || v\_id || ' - Nombre: ' || v\_nombre || ' - Salario: ' || v\_salario);

END LOOP;

CLOSE c\_publico;

END listar\_empleados\_publico;

-- Procedimiento que usa el cursor privado

PROCEDURE listar\_empleados\_privado IS

v\_id empleados.id\_empleado%TYPE;

v\_nombre empleados.nombre%TYPE;

v\_salario empleados.salario%TYPE;

BEGIN

OPEN c\_privado;

LOOP

FETCH c\_privado INTO v\_id, v\_nombre, v\_salario;

EXIT WHEN c\_privado%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('PRIVADO - ID: ' || v\_id || ' - Nombre: ' || v\_nombre || ' - Salario: ' || v\_salario);

END LOOP;

CLOSE c\_privado;

END listar\_empleados\_privado;

-- Procedimiento con un cursor local (dentro del procedimiento)

PROCEDURE listar\_empleados\_local IS

CURSOR c\_local IS

SELECT id\_empleado, nombre, salario FROM empleados WHERE salario < 2000;

v\_id empleados.id\_empleado%TYPE;

v\_nombre empleados.nombre%TYPE;

v\_salario empleados.salario%TYPE;

BEGIN

OPEN c\_local;

LOOP

FETCH c\_local INTO v\_id, v\_nombre, v\_salario;

EXIT WHEN c\_local%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('LOCAL - ID: ' || v\_id || ' - Nombre: ' || v\_nombre || ' - Salario: ' || v\_salario);

END LOOP;

CLOSE c\_local;

END listar\_empleados\_local;

END pk\_empleados;

/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////////////////

Aquí cursor con return

**🎯 Explicación del RETURN en Cursores de PL/SQL**

En PL/SQL, cuando declaramos un cursor **en la cabecera del paquete**, usamos RETURN para definir **qué tipo de datos devuelve el cursor**.

**📌 Sintaxis del RETURN en Cursores**

CURSOR nombre\_cursor RETURN tipo\_de\_registro IS

SELECT columna1, columna2 FROM tabla;

✅ RETURN **define la estructura de los datos** que devolverá el cursor.  
✅ Es útil cuando queremos **devolver un conjunto de filas** desde un procedimiento.

**✍ Ejemplo Completo con RETURN en un Cursor Público**

🔹 **Un procedimiento usa un cursor público declarado en la cabecera del paquete.**  
🔹 **Otro procedimiento usa un cursor privado dentro del PACKAGE BODY.**

**🟢 Paso 1: Crear el Paquete**

-- CABECERA DEL PAQUETE (PACKAGE)

CREATE OR REPLACE PACKAGE pk\_empleados AS

-- 1️⃣ Definimos un tipo de registro basado en la tabla empleados

TYPE t\_empleado IS RECORD (

id empleados.id\_empleado%TYPE,

nombre empleados.nombre%TYPE,

salario empleados.salario%TYPE

);

-- 2️⃣ Definimos un cursor público con RETURN usando el tipo de registro

CURSOR c\_empleados RETURN t\_empleado IS

SELECT id\_empleado, nombre, salario FROM empleados;

-- Procedimientos

PROCEDURE listar\_empleados\_publico;

PROCEDURE buscar\_empleado(p\_id IN empleados.id\_empleado%TYPE);

END pk\_empleados;

/

🔍 **Explicación**  
✅ Creamos el **tipo de registro t\_empleado**, que representa una fila de la tabla empleados.  
✅ El **cursor c\_empleados usa RETURN t\_empleado**, indicando que devuelve registros con esa estructura.  
✅ Declaramos dos procedimientos que lo usarán.

**🟠 Paso 2: Crear el Cuerpo del Paquete**

-- CUERPO DEL PAQUETE (PACKAGE BODY)

CREATE OR REPLACE PACKAGE BODY pk\_empleados AS

-- Cursor privado: Solo se usa dentro del PACKAGE BODY

CURSOR c\_empleado\_unico(p\_id empleados.id\_empleado%TYPE) IS

SELECT id\_empleado, nombre, salario

FROM empleados WHERE id\_empleado = p\_id;

-- Procedimiento que usa el cursor público

PROCEDURE listar\_empleados\_publico IS

v\_empleado t\_empleado; -- Variable del tipo de registro del cursor

BEGIN

OPEN c\_empleados;

LOOP

FETCH c\_empleados INTO v\_empleado;

EXIT WHEN c\_empleados%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('ID: ' || v\_empleado.id ||

' - Nombre: ' || v\_empleado.nombre ||

' - Salario: ' || v\_empleado.salario);

END LOOP;

CLOSE c\_empleados;

END listar\_empleados\_publico;

-- Procedimiento que usa el cursor privado

PROCEDURE buscar\_empleado(p\_id IN empleados.id\_empleado%TYPE) IS

v\_id empleados.id\_empleado%TYPE;

v\_nombre empleados.nombre%TYPE;

v\_salario empleados.salario%TYPE;

BEGIN

OPEN c\_empleado\_unico(p\_id);

FETCH c\_empleado\_unico INTO v\_id, v\_nombre, v\_salario;

IF c\_empleado\_unico%FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Empleado encontrado: ' || v\_id || ' - ' || v\_nombre || ' - ' || v\_salario);

ELSE

DBMS\_OUTPUT.PUT\_LINE('No se encontró ningún empleado con el ID: ' || p\_id);

END IF;

CLOSE c\_empleado\_unico;

END buscar\_empleado;

END pk\_empleados;

/

**🚀 Ejecutar los Procedimientos**

-- Llamar al procedimiento que usa el cursor público

BEGIN

pk\_empleados.listar\_empleados\_publico;

END;

/

-- Llamar al procedimiento que usa el cursor privado pasando un ID de empleado

BEGIN

pk\_empleados.buscar\_empleado(101);

END;

/

**📜 Explicación del Uso del RETURN en el Cursor Público**

| **Parte del Código** | **Explicación** |
| --- | --- |
| TYPE t\_empleado IS RECORD (...) | Crea un tipo de registro que almacena id, nombre y salario. |
| CURSOR c\_empleados RETURN t\_empleado IS ... | Declara un cursor que devuelve registros con la estructura de t\_empleado. |
| v\_empleado t\_empleado; | Declara una variable para almacenar registros del cursor. |
| FETCH c\_empleados INTO v\_empleado; | Extrae una fila y la guarda en la variable v\_empleado. |

**🎯 Resumen Final**

✅ **El RETURN en un cursor se usa para definir la estructura de los datos que devuelve.**  
✅ **Los cursores con RETURN permiten trabajar con registros de forma más clara y estructurada.**  
✅ **El cursor público (c\_empleados) se puede usar en cualquier procedimiento del paquete.**  
✅ **El cursor privado (c\_empleado\_unico) solo se puede usar dentro del PACKAGE BODY.**

🔹 **Con esto ya puedes usar cursores públicos con RETURN de forma correcta en tus paquetes PL/SQL. 🚀**

¡Buena pregunta, Bryan! 🔥

Si quieres almacenar **muchos registros** en un paquete, necesitas usar **colecciones** en lugar de una sola variable de tipo RECORD o %ROWTYPE.

**📌 ¿Cómo Almacenar Múltiples Registros con un Cursor en un Paquete?**

Tanto RECORD como %ROWTYPE **solo almacenan un registro a la vez**.  
Si necesitas manejar **varios registros**, debes usar una **tabla PL/SQL (colección)**.

**💡 Opciones para Almacenar Múltiples Registros**

| **Método** | **Descripción** | **Ejemplo** |
| --- | --- | --- |
| **Colecciones con TABLE y RECORD** | Se usa un tipo de tabla para almacenar múltiples registros personalizados. | TYPE t\_empleado\_tab IS TABLE OF t\_empleado; |
| **Colecciones con TABLE y %ROWTYPE** | Se usa una tabla PL/SQL con %ROWTYPE para guardar filas completas. | TYPE t\_empleado\_tab IS TABLE OF empleados%ROWTYPE; |

**🔥 Ejemplo 1: Cursor Público con RETURN y Colección en la Cabecera**

CREATE OR REPLACE PACKAGE pk\_empleados AS

-- 1️⃣ Definimos un tipo de registro con solo algunas columnas

TYPE t\_empleado IS RECORD (

id empleados.id\_empleado%TYPE,

nombre empleados.nombre%TYPE

);

-- 2️⃣ Definimos una tabla (colección) de empleados

TYPE t\_empleado\_tab IS TABLE OF t\_empleado;

-- 3️⃣ Declaramos un cursor público con `RETURN`

CURSOR c\_empleados RETURN t\_empleado IS

SELECT id\_empleado, nombre FROM empleados;

-- 4️⃣ Procedimiento que llena la colección con empleados

PROCEDURE listar\_empleados (p\_lista OUT t\_empleado\_tab);

END pk\_empleados;

/

**🎯 Ejemplo 2: Implementación en el PACKAGE BODY**

CREATE OR REPLACE PACKAGE BODY pk\_empleados AS

PROCEDURE listar\_empleados (p\_lista OUT t\_empleado\_tab) IS

v\_empleado t\_empleado;

v\_lista t\_empleado\_tab := t\_empleado\_tab(); -- Inicializa la colección

i PLS\_INTEGER := 1; -- Índice para la tabla

BEGIN

OPEN c\_empleados;

LOOP

FETCH c\_empleados INTO v\_empleado;

EXIT WHEN c\_empleados%NOTFOUND;

v\_lista.EXTEND; -- Aumenta el tamaño de la colección

v\_lista(i) := v\_empleado; -- Guarda el registro

i := i + 1;

END LOOP;

CLOSE c\_empleados;

-- Devuelve la lista completa

p\_lista := v\_lista;

END listar\_empleados;

END pk\_empleados;

/

**🛠 Ejemplo 3: Llamando al Procedimiento y Mostrando Múltiples Registros**

DECLARE

v\_empleados pk\_empleados.t\_empleado\_tab; -- Variable para almacenar la lista de empleados

BEGIN

pk\_empleados.listar\_empleados(v\_empleados);

-- Recorrer la colección e imprimir cada registro

FOR i IN 1 .. v\_empleados.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('ID: ' || v\_empleados(i).id || ' - Nombre: ' || v\_empleados(i).nombre);

END LOOP;

END;

/

**🚀 Resumen**

✅ **Si quieres almacenar muchos registros**, usa **colecciones** (TABLE OF RECORD).  
✅ **Puedes devolver la colección desde un procedimiento** para usarla en PL/SQL.  
✅ **Las colecciones permiten recorrer y manejar múltiples registros fácilmente.**

📌 **Cuando tienes un cursor que devuelve muchas filas, la mejor práctica es almacenarlas en una colección para poder manipularlas.** 🔥

No puedes declarar el cursor en la **cabecera** del paquete y luego **no redeclararlo en el body**. Esto se debe a que **PL/SQL requiere que el cursor tenga su implementación en el cuerpo del paquete**.

Sin embargo, sí puedes **declararlo solo en la cabecera** si lo conviertes en un **cursor explícito con retorno de un registro (RETURN RECORD)**. Pero **no puedes usar RETURN T\_AUX (colección de registros) en la cabecera del paquete** porque los cursores solo pueden devolver **una fila a la vez**.

**🔥 SOLUCIÓN: Declarar solo en la cabecera y usar RETURN RECORD**

Si quieres que el cursor **solo esté en la cabecera del paquete** y que en el body solo lo abras, entonces usa **un cursor con RETURN RECORD** en la cabecera.

**📌 CABECERA DEL PAQUETE (Declarar solo una vez el cursor)**

CREATE OR REPLACE PACKAGE pk\_facturas AS

-- 📌 1️⃣ Tipo RECORD para una fila de datos

TYPE TD\_C\_VER\_FACTURAS IS RECORD (

COD\_FACTURA FACTURAS.COD\_FACTURA%TYPE,

FECHA\_EMISION FACTURAS.FECHA\_EMISION%TYPE,

COD\_CLIENTE FACTURAS.COD\_CLIENTE%TYPE,

COD\_PRODUCTO DETALLE\_FACTURAS.COD\_PRODUCTO%TYPE,

CANTIDAD DETALLE\_FACTURAS.CANTIDAD%TYPE,

SUBTOTAL DETALLE\_FACTURAS.SUBTOTAL%TYPE

);

-- 📌 2️⃣ Cursor que devuelve una sola fila (pero se puede recorrer con WHILE)

CURSOR c\_ver\_facturas(p\_cod\_factura FACTURAS.COD\_FACTURA%TYPE)

RETURN TD\_C\_VER\_FACTURAS;

-- 📌 3️⃣ Procedimientos

PROCEDURE ver\_facturas (p\_cod\_factura IN FACTURAS.COD\_FACTURA%TYPE);

PROCEDURE ver\_facturas;

PROCEDURE ver\_facturas (p\_cod\_cliente IN FACTURAS.COD\_CLIENTE%TYPE, p\_mostrar\_detalles IN BOOLEAN);

END pk\_facturas;

/

✅ **El cursor está solo en la cabecera**  
✅ **El cursor devuelve una fila (RETURN RECORD)**

**📌 IMPLEMENTACIÓN DEL PAQUETE (Solo se usa el cursor, no se redeclara)**

CREATE OR REPLACE PACKAGE BODY pk\_facturas AS

-- 📌 1️⃣ Procedimiento para ver una factura con WHILE

PROCEDURE ver\_facturas (p\_cod\_factura IN FACTURAS.COD\_FACTURA%TYPE) IS

v\_factura TD\_C\_VER\_FACTURAS; -- 📌 Variable para almacenar la fila del cursor

BEGIN

OPEN c\_ver\_facturas(p\_cod\_factura);

FETCH c\_ver\_facturas INTO v\_factura;

WHILE c\_ver\_facturas%FOUND LOOP

dbms\_output.put\_line('Factura: ' || v\_factura.cod\_factura);

dbms\_output.put\_line('Fecha: ' || v\_factura.fecha\_emision);

dbms\_output.put\_line('Cliente: ' || v\_factura.cod\_cliente);

dbms\_output.put\_line('Producto: ' || v\_factura.cod\_producto ||

', Cantidad: ' || v\_factura.cantidad ||

', Subtotal: ' || v\_factura.subtotal);

dbms\_output.put\_line('------------------------------------------------');

FETCH c\_ver\_facturas INTO v\_factura; -- Siguiente fila

END LOOP;

CLOSE c\_ver\_facturas;

END ver\_facturas;

-- 📌 2️⃣ Procedimiento para ver todas las facturas

PROCEDURE ver\_facturas IS

CURSOR c\_todas\_facturas IS

SELECT cod\_factura FROM facturas;

v\_cod\_factura FACTURAS.COD\_FACTURA%TYPE;

BEGIN

OPEN c\_todas\_facturas;

FETCH c\_todas\_facturas INTO v\_cod\_factura;

WHILE c\_todas\_facturas%FOUND LOOP

pk\_facturas.ver\_facturas(v\_cod\_factura);

FETCH c\_todas\_facturas INTO v\_cod\_factura;

END LOOP;

CLOSE c\_todas\_facturas;

END ver\_facturas;

-- 📌 3️⃣ Procedimiento para ver facturas de un cliente

PROCEDURE ver\_facturas (p\_cod\_cliente IN FACTURAS.COD\_CLIENTE%TYPE, p\_mostrar\_detalles IN BOOLEAN) IS

CURSOR c\_facturas\_cliente IS

SELECT cod\_factura FROM facturas WHERE cod\_cliente = p\_cod\_cliente;

v\_cod\_factura FACTURAS.COD\_FACTURA%TYPE;

BEGIN

OPEN c\_facturas\_cliente;

FETCH c\_facturas\_cliente INTO v\_cod\_factura;

WHILE c\_facturas\_cliente%FOUND LOOP

IF p\_mostrar\_detalles THEN

pk\_facturas.ver\_facturas(v\_cod\_factura);

ELSE

dbms\_output.put\_line('Factura: ' || v\_cod\_factura);

END IF;

FETCH c\_facturas\_cliente INTO v\_cod\_factura;

END LOOP;

CLOSE c\_facturas\_cliente;

END ver\_facturas;

END pk\_facturas;

/

**🔥 Beneficios de esta solución**

✅ **El cursor c\_ver\_facturas solo se declara en la cabecera.**  
✅ **No necesitas redeclararlo en el body, solo lo usas con OPEN, FETCH y CLOSE.**  
✅ **Puedes recorrerlo con WHILE o LOOP sin problemas.**  
✅ **No necesitas colecciones (TABLE OF RECORD), solo un RECORD normal.**

**⛔ Lo que NO se puede hacer**

❌ **No puedes hacer RETURN T\_AUX (una colección) en la cabecera del paquete.**  
❌ **No puedes declarar un cursor sin RETURN RECORD en la cabecera y evitar redeclararlo en el body.**  
❌ **No puedes devolver una colección directamente desde un cursor en un paquete.**

**🚀 Conclusión**

Si **quieres que el cursor solo esté en la cabecera**, usa **RETURN RECORD** y **un WHILE para recorrerlo** en el body.  
Este es el método **correcto y permitido en PL/SQL**. 🎯

Aquí tienes una tabla que resume las reglas sobre el uso de cursores en un paquete PL/SQL:

| **Regla** | **¿Se puede hacer?** | **Explicación** |
| --- | --- | --- |
| **Declarar un cursor en la cabecera y no redeclararlo en el body** | ✅ Sí, pero con RETURN RECORD | Si el cursor devuelve una sola fila a la vez (RETURN RECORD), puedes declararlo en la cabecera y usarlo en el body sin redeclararlo. |
| **Declarar un cursor en la cabecera con RETURN TABLE OF RECORD** | ❌ No | Los cursores en la cabecera solo pueden devolver **una fila a la vez** (RECORD), no una colección (TABLE OF RECORD). |
| **Usar OPEN cursor\_name, FETCH, CLOSE en los procedimientos del body** | ✅ Sí | Puedes recorrer los resultados usando un WHILE con %FOUND. |
| **Recorrer los resultados de un cursor sin usar un FETCH dentro de un loop** | ❌ No | Para leer cada fila del cursor, **debes hacer FETCH manualmente** dentro de un loop (WHILE o LOOP). |
| **Hacer BULK COLLECT INTO desde un cursor declarado en la cabecera** | ❌ No | BULK COLLECT INTO solo funciona con colecciones (TABLE OF RECORD), pero los cursores en la cabecera no pueden devolver colecciones. |
| **Devolver una colección (TABLE OF RECORD) desde un cursor en un paquete** | ❌ No | Solo se pueden devolver **una fila a la vez** (RETURN RECORD). Para colecciones, hay que definirlas y usarlas dentro del body. |
| **Llamar a un procedimiento que usa un cursor sin abrirlo manualmente** | ❌ No | Si defines un cursor en la cabecera, **debes abrirlo (OPEN) y recorrerlo (FETCH) manualmente** en el body. |
| **Usar cursores implícitos en procedimientos dentro de un paquete** | ✅ Sí | Puedes hacer SELECT INTO sin declarar un cursor, pero solo si esperas **una sola fila**. |

Con esto ya sabes exactamente qué se puede hacer y qué no dentro de un paquete PL/SQL con cursores. 🚀

* **Un cursor es un conjunto de filas** que se puede procesar **una fila a la vez**.
* **No puede almacenar colecciones**. El cursor solo maneja una fila en su memoria a la vez mientras se va recorriendo.
* **Sí puede recorrer colecciones**. Puedes usar un cursor para recorrer una colección, pero el cursor en sí solo procesa una fila a la vez, no almacena todas las filas de una colección de forma simultánea.

Esto es cierto tanto si usas el cursor dentro de un **paquete**, **procedimiento** o **función**, o si lo usas directamente en un bloque anónimo. El comportamiento es el mismo: el cursor recorre las filas de la base de datos una a una, y no puede almacenar todas esas filas en su memoria a la vez.

**Ejemplo más claro:**

Un paquete consta de los siguientes elementos:

• Especificación o cabecera: contiene las

declaraciones públicas (es decir, accesibles

desde cualquier parte de la aplicación) de sus

programas, tipos, constantes, variables, cursores,

excepciones, etc.

• Cuerpo: contiene los detalles de implementación

y declaraciones privadas, es decir, accesibles

solamente desde los objetos del paquete.

La sintaxis de la cabecera es la siguiente:

create [or replace] package nombre\_paquete as

<declaraciones públicas>

<especificaciones de subprogramas>

end nombre\_paquete;

La sintaxis del cuerpo sería la siguiente:

create [or replace] package body nombre\_paquete as

<declaraciones privadas>

<código de subprogramas>

[begin

<instrucciones iniciales>]

end nombre\_paquete;

Como podéis observar la cabecera se compila

independientemente del cuerpo. Os dejo un ejemplo

de paquete para que lo veáis más claro.

/\* Cabecera \*/

create or replace package busar\_emple as

TYPE t\_reg\_emple is RECORD

(num\_empleado emple.emp\_no%TYPE,

apellido emple.apellido%TYPE,

salario emple.salario%TYPE,

departamento emple.dept\_no%TYPE);

procedure ver\_por\_numero(v\_cod

emple.emp\_no%TYPE);

procedure ver\_por\_apellido(v\_ape

emple.apellido%TYPE);

function datos (v\_cod emple.emp\_no%TYPE)

return t\_reg\_emple;

end buscar\_emple;

/\* Cuerpo \*/

create or replace package body buscar\_emple as

vg\_emple t\_reg\_emple;

procedure ver\_emple; /\* procedimiento privado\*/

procedure ver\_por\_numero (v\_cod

emple.emp\_no%TYPE)

is

begin

select emp\_no, apellido, salario, dept\_no into

vg\_emple from emple where emp\_no=v\_cod;

ver\_emple;

end ver\_por\_numero;

procedure ver\_por\_apellido (v\_ape

emple.apellido%TYPE)

is

begin

select emp\_no,apellido,salario,dept\_no into

vg\_emple from emple where apellido=v\_apellido;

ver\_emple;

end ver\_por\_apellido;

function datos (v\_cod emple.emp\_no%TYPE)

return t\_reg\_emple

is

begin

select emp\_no,apellido,salario,dept\_no into

vg\_emple from emple where emp\_no=v\_cod;

procedure ver\_emple

is

begin

DBMS\_OUTPUT.PUT\_LINE(vg\_emple.num\_emple

ado || '\*' || vg\_emple.apellido || '\*' || vg\_emple.salario ||

'\*'|| vg\_emple.departamento);

end ver\_emple;

end buscar\_emple;

Como podéis ver este paquete nos permite buscar un

empleado de tres formas distintas y visualizar sus

datos.

Utilización de los objetos definidos en los

paquetes

Podemos utilizar los objetos definidos en los

paquetes básicamente de dos maneras distintas:

• Desde el mismo paquete: esto quiere decir que

cualquier objeto puede ser utilizado dentro del

paquete por otro objeto declarado en el mismo.

Para utilizar un objeto dentro del paquete tan sólo

tendríamos que llamarlo. La llamada sería algo

así: v\_emple :=buscar\_emple.datos(v\_n\_ape);

(como veis no utilizamos el execute ya que nos

encontramos dentro del paquete).

• Desde fuera del paquete: Podemos utilizar los

objetos de un paquete siempre y cuando haya

sido declarado en la especificación del mismo.

Para llamar a un objeto o procedimiento desde

fuera del paquete utilizaríamos la siguiente

notación: execute

nombre\_paquete.nombre\_procedimiento(lista de

parametros);

Declaración de cursores en paquetes

En los paquetes también podemos introducir cursores,

para ello debemos declararlo en la cabecera del

paquete indicando su nombre, los parámetros y tipo

devuelto. Para que lo veáis más claro os dejo un

ejemplo a continuación:

CREATE or REPLACE PACKAGE empleados AS

.....

CURSOR a RETURN empleado%ROWTYPE;

...

END empleados;

CREATE or REPLACE PACKAGE BODY empleados AS

....

CURSOR a RETURN empleado%ROWTYPE

SELECT \* FROM empleado WHERE salario < 10000;

....

END empleados;

DISPARADORES DE LA BASE DE DATOS (TRIGGER).

• Un disparador o trigger es un procedimiento PL/SQL asociado

a una tabla que Oracle ejecuta automáticamente cuando se

realiza una determinada operación sobre la tabla.

• Los trigger se asemejan a procedimientos y funciones en que

son bloque pl/sql nominados con sección declarativa,

ejecutable y de control de errores

• Sin embargo, un trigger es un conjunto de acciones que se

ejecutan implícitamente cuando ocurre un evento.

• Con un disparador, por ejemplo, se puede controlar que a

determinadas tablas no se acceda en determinados días, o

bien que no se realicen determinadas operaciones por

determinados usuarios.

• El acto de ejecutar un trigger se conoce como disparar y el

evento que lo provoca generalmente son DML sobre tablas.

• Con los disparadores en general podemos:

o Implementar restricciones complejas de integridad y

seguridad

o Prevenir transacciones erróneas

o Implementar reglas administrativas complejas

o Generar automáticamente valores derivados

• No se deben definir trigger para duplicar o reemplazar la

funcionalidad ya construida dentro de la bbdd. Por ejemplo, no

definir triggers para implementar las reglas de integridad

• El uso excesivo de triggers puede dar lugar a

interdependencias complejas o podría dificultar el

mantenimiento en aplicaciones largas.

• Cuidado con los efectos recursivos y en cascada

• Son tres los eventos que hacen que se dispare un trigger:

insert, delete, update, pudiendo elegir si se dispara antes o

después de producirse uno o varios de los eventos anteriores.

• Para poder crear, modificar o borrar disparadores es necesario

tener los privilegios necesarios:

CREATE TRIGGER  Crear un disparador en el esquema propio.

CREATE ANY TRIGGER  Crear un disparador en cualquier

esquema.

ALTER ANY TRIGGER  Modificar un disparador en cualquier

esquema.

DROP ANY TRIGGER  Eliminar cualquier disparador en

cualquier esquema.

• La sintaxis para crear disparadores de la base de datos es la

siguiente:

CREATE [OR REPLACE] TRIGGER

nombre {BEFORE| AFTER }

{DELETE| INSERT|UPDATE [OF

columnas...]} [OR {DELETE|

INSERT|UPDATE [OF columnas...]} ON

tabla

[REFERENCING OLD as alias NEW as alias]

[FOR EACH {STATEMENT|ROW[WHEN

CONDICIONN]}] /\*Aqui comienza el bloque

pl/sql\*/ [DECLARE

Declaraciones; Es opcional]

BEGIN

Sentencias;

END;

• Con las opciones BEFORE o AFTER se le indica si se dispara

antes o después de producirse el evento.

• Los eventos pueden ser: la inserción de nuevas filas, la

actualización de todas o algunas de las columnas y el borrado de

todas o algunas filas. Podemos especificar más de un evento a la

vez, separándolos con operadores de relación OR. En este caso

se lanza un mismo trigger por distintos eventos.

• Se puede especificar la columna o columnas que deben

modificarse para que se lance el disparador

• El nivel del disparador puede ser:

• A nivel de orden (por defecto): Se activa una sola vez por cada

orden, independientemente del número de filas afectadas. Se

puede incluir FOR EACH STATEMENT aunque no hace falta

• A nivel de fila: Se activa un vez por cada fila afectada. Se

debe incluir FOR EACH ROW

• La cláusula WHEN sirve para especificar la restricción del trigger,

es decir, se dispara solo cuando se cumple la condición. Esta

condición tiene las siguientes restricciones:

- Solo se puede utilizar con triggers a nivel de filas

- Debe ser una condición SQL no PL/SQL

- No puede incluir una consulta a ninguna tabla

NEW y OLD

• Con ellos podemos referenciar a los valores antes y después

de la actualización a nivel de fila

• Solo se puede utilizar a nivel de fila

• Se utiliza :new.columna :old.columna

• Ejemplo: if :new.columna<:old.columna

• Cuando hacemos DELETE :new.columna es NULL

• Cuando hacemos INSERT :old.columna es NULL

• Cuando hacemos UPDATE ambas tienen valor

• Cuando utilicemos new y old con la clausuala WHEN del

trigger no hace falta :

Ejemplo: when new.salario<old.salario

• Cuando queramos poner un alias a old o new

podremos hacerlo con REFERENCING

Ejemplo: Vamos a controlar qué usuarios modifican la

información de la tabla emp (insertan, borran o actualizan) y

cuándo lo hacen. El ejemplo ejecutarlo a nivel de fila y de

orden y ver las diferencias

1º. Creamos una tabla donde almacenaremos el usuario y el

momento de utilización de la tabla y la operación realizada

sobre ella.

CREATE TABLE controla\_usuarios (

usuario VARCHAR2(17),

operacion VARCHAR2(6),

fecha DATE);

2º. Creamos los disparadores, para lo que es necesario tener

el privilegio CREATE TRIGGER, si lo creamos en nuestro

esquema, o el de CREATE ANY TRIGGER, si se crea en

cualquier otro esquema.

CREATE TRIGGER actualiza\_emple

BEFORE UPDATE ON EMP

DECLARE

nom\_usuario VARCHAR2(30);

BEGIN

nom\_usuario := USER;

INSERT INTO controla\_vecinos

VALUES (nom\_usuario, ‘UPDATE’, sysdate);

END;

/

CREATE TRIGGER inserta\_emp

BEFORE INSERT ON emp

DECLARE

nom\_usuario VARCHAR2(30);

BEGIN

nom\_usuario := USER;

INSERT INTO controla\_emp

VALUES (nom\_usuario, ‘INSERT’, sysdate);

END;

/

CREATE OR REPLACE TRIGGER

borra\_emp BEFORE DELETE ON emp

DECLARE

nom\_usuario VARCHAR2(30);

BEGIN

nom\_usuario := USER;

INSERT INTO controla\_emp

VALUES (nom\_usuario, 'delete',

sysdate); END;

/

Una vez creados los disparadores, cada vez que un usuario

realiza cualquier tipo de modificación del contenido de la tabla

EMP, se genera una fila en la tabla CONTROLA\_EMP, que nos

permite conocer que usuario realizó una determinada

operación y en que momento.

TIPOS DE DISPARADORES Y ORDEN DE EJECUCIÓN

• Cuando existen varios disparadores se lanzaran en el

siguiente orden:

1º . Los asociados a BEFORE ... FOR EACH STATEMENT

2º Para cada fila los BEFORE .. FOR EACH ROW

3º Se realiza INSERT, DELETE o UPDATE bloqueando las filas

hasta que la transacción se confirme

4º Se ejecutan los AFTER .. FOR EACH ROW

7º AFTER FOR EACH STATEMENT

• Cuando se dispara un trigger este forma parte de la operación

de actualización que lo lanzó. Si el trigger falla, lo hace toda la

actualización y Oracle hará un Rollback de toda la actualización

•A veces en lugar de varios trigger se hace uno solo que se

puede lanzar por múltiples eventos

Multiples eventos de disparo

• Cuando un mismo trigger es disparado por múltiples eventos

esto se hace: Before delete or update ON emp

• Para distinguir posteriormente utilizamos los siguientes

predicados:

• INSERTING: Devuelve TRUE Si el evento que disparó el

trigger fue INSERT

• DELETING: Devuelve TRUE Si el evento que disparó el

trigger fue DELETE

• UPDATING: Devuelve TRUE Si el evento que disparó el

trigger fue UPDATE

• UPDATING(nombrecolumna): Devuelve TRUE Si el evento

que disparó el trigger fue update y la columna

especificada ha sido modificada

Begin

If INSERTING then

ELSEIF DELETING then

ELSEIF UPDATING(‘SALARIO’) THEN

ENDIF

Restricciones sobre los trigger

 En el bloque del trigger debemos tener las siguientes

consideraciones:

o No podemos escribir ninguna orden de control de

transacciones(COMMIT,ROLLBACK). Tampoco podrán ser

utilizadas estas sentencias en funciones o procedimientos

que sean llamados desde el trigger

Triggers Instead of

 Son un tipo especial de trigger que solo se puede definir sobre

vistas de la bbdd

 Se activan en lugar de la orden DML funcionando de manera

invisible

 No deja especificar los of en el update, se debe hacer con codigo

updating(‘columna’);

CREATE OR REPLACE TRIGGER nombre

INSTEAD OF {DELETE| INSERT|UPDATE

} [OR {DELETE| INSERT|UPDATE }

ON nombrevista

[referencing old as alias new as alias]

[for each row]

bloque pl/sql

 Todos los trigger INSTEAD OF son triggers a nivel de fila por lo

que esta cláusula no es obligatoria, ya que está implícita

 En estos trigger no se puede usar When

Ejemplo: Crear una vista con los empleados del departamento 20.

(EMP20)

A continuación, si intentamos actualizar o borrar algo en esta vista

no da problemas. Sin embargo, cuando intentamos insertar si da

problemas, ya que no introduzco el número de departamento y es

NOT NULL en la tabla (EMP)

Solución: Crear un disparador INSTEAD OF sobre DEPT20

…….

INSTEAD OF INSERT ON emp\_20

BEGIN

Insert into emp (empno,ename,job,mgr,hiredate,sal,comm.,deptno)

v a l u e s ( : n e w . e m p n o , : n e w . e n a m e , : n e w . j o b ,

7766,sysdate,:new.sal,null,20);

End;

/

sql> Insert into emp\_20 values (7000,’PETER’, ‘CLERK’,2000);

Otros triggers en la bbdd

 Estos trigger están disponibles a partir de la versión

8i o Triggers sobre sentencias DDL

o Triggers sobre eventos del sistema

 Estos trigger se definen a nivel de bbdd o a nivel de esquema.

Los definidos a nivel de bbdd se lanzan para todos los usuarios y

los triggers a nivel de esquema se lanzan para el usuario del

esquema del trigger.

Trigger sobre sentencias DDL

CREATE OR REPLACE TRIGGER nombre {BEFORE|AFTER}

{CREATE|ALTER|DROP}[OR {CREATE|ALTER|DROP] ON

{DATABASE|nombreusuario.SCHEMA} Pl/sql block;

 Cuando no se especifica nombredeusuario de toma por defecto el

esquema del usuario que está creando el trigger

Trigger sobre eventos del sistema

Estos trigger se disparan cuando se arranca o para la base de

datos, entra o sale un usuario, cuando se crea, modifica o

elimina un objeto, etc.

En Oracle para crear este tipo de trigger tenemos que tener

privilegios de Administer database trigger.

La sintaxis de este trigger seria la siguiente:

create [or replace] trigger nombre\_trigger

{ before | after } { <lista eventos de definición> | <lista

eventos del sistema>}

on { database | schema} [when (condición)]

<cuerpo del trigger (bloque PL/SQL)>

Donde la lista de eventos de definición puede tener uno o más

eventos DDL separados por or y la lista de eventos del

sistema igualmente separados por or.

Al asociar un disparador a un evento debemos indicar el

momento en que se dispare. A continuación os dejo una tabla

de evento, momento y cuando se dispararía para dejarlo todo

mas o menos claro.

db\_event1. Eventos de la Descripción

bbdd

AFTER SERVERERROR Se dispar después de que se produzca

un error

Información sobre triggers

DESC USER\_TRIGGERS

TRIGGER\_TYPE: Puede tener Before,After o Instead of

TRIGGERING\_EVENT: DML que dispara el Trigger

REFERENCING\_NAMES: Alias para old y New

TRIGGER\_BODY: El cuerpo del trigger

Operaciones sobre triggers

DROP TRIGGER nombre

Cuando eliminamos una tabla se eliminan todos los trigger

asociados a ella

AFTER LOGON

Se dispara después de que el usuario

se conecte a la bbdd

BEFORE LOGOFF Se dispara antes de que un usuario se

desconecte

AFTER STARTUP Se disparan después del arranque de

la bbdd. Este evento solo se puede

aplicar a nivel de Database

BEFORE SHUTDOWN Se dispara antes de la parada de la

bbdd. Este evento solo se puede

aplicar a nivel de Database

ALTER TRIGGER nombre\_trigger ENABLE|DISABLE

Para activar o desactivar el trigger

ALTER TABLE nombre\_trigger ENABLE|DISABLE ALL

TRIGGERS Activa o desactiva todos los triggers de la tabla

[Diccionario de datos para triggers](https://aulavirtual32.educa.madrid.org/ies.mariadezayas.majadahonda/mod/url/view.php?id=60070)

<https://www.jpromero.com/2011/07/informacion-de-los-disparadores-oracle.html>

DCL. CREACION DE USUARIOS

Desde la versión 12c debemos usar

alter session set "\_ORACLE\_SCRIPT"=true;

cada vez que vayamos a crear usuarios.

CREATE USER statement

The CREATE USER statement allows you to create a new database user which you can

use to log in to the Oracle database.

The basic syntax of the CREATE USER statement is as follows:

CREATE USER username

IDENTIFIED BY password

[DEFAULT TABLESPACE tablespace]

[QUOTA {size | UNLIMITED} ON tablespace]

[PROFILE profile]

[PASSWORD EXPIRE]

[ACCOUNT {LOCK | UNLOCK}];Code language: SQL (Structured Query Language) (sql)

In this syntax:

CREATE USER username

Specify the name of the user to be created.

IDENTIFIED BY password

Specify a password for the local user to use to log on to the database. Note that you can

create an external or global user, which is not covered in this tutorial.

DEFAULT TABLESPACE

Specify the tablespace of the objects such as tables and views that the user will create.

If you skip this clause, the user’s objects will be stored in the database default

tablespace if available, typically it is USERS tablespace; or the SYSTEM tablespace in case

there is no database default tablespace.

QUOTA

Specify the maximum space in the tablespace that the user can use. You can have

multiple QUOTA clauses, each for a tablespace.

Use UNLIMITED if you don’t want to restrict the size of the tablespace that the user can

use.

PROFILE profile

A user profile limits the database resources or password that the user cannot exceed.

You can assign a profile to a newly created user. If you skip this clause, Oracle will

assign the DEFAULT profile to the user.

PASSWORD EXPIRE

Use the PASSWORD EXPIRE if you want to force the user to change the password for the

first time the user logs in to the database.

ACCOUNT {LOCK | UNLOCK}

Use ACCOUNT LOCK if you want to lock the user and disable access. On the other hand,

specify ACCOUNT UNLOCK to unlock user and enable access.

To execute the CREATE USER statement, you must have the CREATE USER system

privilege. Once you create the new user, the privilege domain of the user will be empty.

Therefore, if you want the user to be able to login to the database, you should grant the

CREATE SESSION system privilege to the user.

Oracle CREATE USER examples

Let’s practice with the CREATE USER statement.

1) Using Oracle CREATE USER statement to create a new local user example

This example uses the CREATE USER statement to create a new local user named john

with the password abcd1234:

CREATE USER john IDENTIFIED BY abcd1234;

Code language: SQL (Structured Query Language) (sql)

Oracle issues the following output indicating that the user john has been created

successfully.

User JOHN created.

Code language: SQL (Structured Query Language) (sql)

To find a list of users with the OPEN status, you query the information from the

dba\_users:

SELECT

username,

default\_tablespace,

profile,

authentication\_type

FROM

dba\_users

WHERE

account\_status = 'OPEN';

Code language: SQL (Structured Query Language) (sql)

As you can see from the output, the user john has a default tablespace as USERS, profile

as DEFAULT, and log in to the database using a PASSWORD.

Tambien podremos modificar y borrar usuarios de la siguiente manera

ALTER USER Antonio QUOTA UNLIMITED ON usuarios

DROP USER usuario [CASCADE]

La opción CASCADE elimina los objetos del esquema del usuario antes de eliminar al

propio usuario. Es obligatorio si el esquema contiene objetos.´

GESTION DE PRIVILEGIOS

Los privilegios son permisos que damos a los usuarios para que puedan realizar ciertas

operaciones con la base de datos. En Oracle hay más de cien posibles privilegios. Se

dividen en:

• Privilegios de sistema. Son permisos para modificar el funcionamiento de la

base de datos. Son cambios, en definitiva, que afectan a todos los usuarios y

usuarias.

• Privilegios de objeto. Son permisos que se aplican a un objeto concreto de la

base de datos.

1. PRIVILEGIOS DEL SISTEMA

Se comentan algunos de los privilegios de sistema más importantes

Privilegio Significado

CREATE SESSION Permite al usuario conectar con la base de datos

RESTRICTED

SESSION

Permite al usuario establecer sesión con la base de datos en

caso de que la base de datos esté en modo restringido

mediante la instrucción:

ALTER SYSTEM ENABLE RESTRICTED

SESSION

Sólo los usuarios con este privilegio puede conectar con la

base de datos si ésta se encuentra en este modo.

Privilegio Significado

ALTER DATABASE Permite modificar la estructura de la base de datos

ALTER SYSTEM Permite modificar los parámetros y variables del sistema

CREATE TABLE Permite crear tablas. Incluye la posibilidad de borrarlas.

GRANT ANY

OBJECT PRIVILEGE

Permite conceder privilegios sobre objetos que no son del

usuario (pertenecen a otros usuarios) a terceros usuarios.

CREATE ANY

TABLE Permite crear tablas en otros esquemas de usuario

DROP ANY TABLE Permite borrar tablas de otros usuarios

SELECT ANY TABLE Permite seleccionar datos en tablas de otros usuarios

INSERT ANY TABLE Permite añadir datos en tablas de otros usuarios

UPDATE ANY

TABLE Permite eliminar datos en tablas de otros usuarios

DELETE ANY

TABLE Permite eliminar datos en tablas de otros usuarios

En la tabla anterior se ha hecho hincapié en los privilegios referidos a las tablas, para

otros objetos el funcionamiento es similar: igual que hay CREATE TABLE, se puede

usar CREATE VIEW para las vistas o INDEX, TRIGGER, PROCEDURE,

SEQUENCE, SYNONYM, TYPE,… y de esa forma podemos conceder privilegio de

creación de otros objetos. Lo mismo con el resto de operaciones

CONCEDER y REVOCAR PRIVILEGIOS

GRANT CREATE SESSION , ALTER SESSION, CREATE VIEW, CREATE VIEW TO ANTONIO;

2. GESTION DE PRIVILEGIOS SOBRE OBJETOS

Las instrucciones vistas anteriormente otorgan o quitan permisos generales, es decir

dictan qué operaciones, en general, puede realizar un usuario.

Los privilegios de objeto marcan qué operaciones le están permitidas a un usuario

realizar sobre el objeto de otros usuarios.

Los privilegios permiten definir el modo en que otros usuarios van a tratar los

objetos de otros usuarios.

GRANT {privilegio [(listaColumnas)] [,privilegio

[(listaColumnas)] [,…]] |

ALL [PRIVILEGES]}

ON [esquema.]objeto

TO {usuario | rol | PUBLIC} [,{usuario | rol | PUBLIC} [,…]]

[WITH GRANT OPTION]

La opción ALL concede todos los privilegios posibles sobre el objeto. Se pueden

asignar varios privilegios a la vez y también varios posibles usuarios. La opción WITH

GRANT OPTION permite al usuario al que se le conceden los privilegios, que pueda,

a su vez, conceder esos mismos privilegios a otro usuario.

Ejemplo de uso de GRANT con privilegios de objeto:

GRANT UPDATE, INSERT ON jsanchez.personas TO anozal;

En la siguiente tabla se enumeran los posibles privilegios que se pueden aplicar a un

determinado objeto:

REVOKE {privilegio1 [,privilegio2] [,…]] |

ALL [PRIVILEGES]}

ON [esquema.]objeto

FROM {usuario | rol | PUBLIC} [,{usuario | rol | PUBLIC} [,…]]

[CASCADE CONSTRAINTS]

CASCADE CONSTRAINTS elimina cualquier restricción que impida el

borrado del privilegio.

Sólo puede revocar los privilegios de objeto concedidos, el usuario que concedió

dichos privilegios.

3. OTORGA R PRIVILEGIOS A UN ROL

Los roles son privilegios aglutinados sobre un mismo nombre, bajo la idea

de que ese conjunto denote un uso habitual sobre la base de datos.

Gracias a los roles se facilita la asignación de privilegios a los usuarios. Un

usuario puede tener asignados varios roles y viceversa.

Se puede crear un rol con el comando:

CREATE ROLE rol

Nos permite asignar permisos a un usuario o a un rol.