Departamento de Lenguajes y Ciencias de la Computación Universidad de Málaga

Administración de Bases de Datos

(Ingeniería Técnica en Informática de Gestión)

Bases de Datos Relacionales:
SQL y el PL/SQL de Oracle

E.T.S.I. Informática ***J. Galindo Gómez***

SQL (Structured Query Language)

- **SQL está en continua evolución:** Es una evolución del lenguaje SEQUEL de D.D. Chamberlin y R.F. Boyce (1974) y fue implementado por primera vez por IBM en su BDR llamado SYSTEM R.
  - **ISO** (*International Standards Organization*) y **ANSI** (*American National Standards Institute*) desarrollaron una versión estándar en 1986, llamada **SQL86** o **SQL1**. Posteriormente, se desarrolló **SQL92** o **SQL2**. Actualmente se desarrolla **SQL3**, que incluye conceptos de BD orientadas a objetos.
- **SQL es un lenguaje estándar para GESTIÓN de BDR:**
  - **Está incluido en muchos SGBD (DBMS),** como DB2 (de IBM), Oracle, Ingres, Informix, Sybase, Access, SQL Server…
    - Las mismas sentencias sirven en distintos SGBD.
    - Si se usan sólo las características estándares facilita la tarea de migrar de SGBD ® Hay funciones no estándar en algunos SGBD.
  - **Fácil de usar y aprender:** Tiene similitudes con el Álgebra Relacional, aunque se parece más al Cálculo y es más fácil e intuitivo que ambos lenguajes formales.
  - Aquí se incluye una introducción a SQL estándar con algunos comentarios sobre el SGBD Oracle.

SQL (Structured Query Language)

- **SQL es un lenguaje COMPLETO:** Incluye sentencias para
  - **DDL y DML:** Permite definir esquemas, consultar, borrar, actualizar...
  - **Definición de vistas:** Para ver la BD de distintas formas.
  - **Seguridad:** Permisos de acceso distintos para cada usuario.
  - **Definir restricciones de integridad:** Integridad referencial...
  - **Especificar control de transacciones:** Para grandes empresas, recuperación de errores, archivos históricos...
  - **Puede incrustarse en lenguajes de alto nivel** (C, C++, Pascal, COBOL...).
  - Permite **operaciones tan complejas** que su total definición es complicada: Sólo veremos un subconjunto de las operaciones posibles.
- **ESQUEMA** (*schema*)**:** Conjunto de elementos (tablas, vistas, permisos...) que pertenecen a la misma BD. Cada esquema tiene un nombre y un usuario propietario del esquema**:**

  `CREATE SCHEMA <Nombre> AUTHORIZATION <Usuario>;`
- **CATÁLOGO** (*catalog*)**:** Conjunto de esquemas. Tiene un esquema especial llamado `INFORMATION_SCHEMA` que provee información sobre los demás esquemas, usuarios autorizados, definiciones de dominio, restricciones de integridad referencial (sólo entre tablas del mismo catálogo)...

SQL (Structured Query Language)

- **TIPOS de DATOS** de los atributos de las relaciones**:**
  - **Enteros** de distintos tamaños**: `INTEGER`** o **`INT`** y **`SMALLINT`**.
  - **Reales** de distinta precisión**: `FLOAT(`**p**`)`**, **`REAL`**, **`DOUBLE PRECISION`**, o el más genérico **`DECIMAL(`**precisión, escala**`)`** (o **`NUMBER`** en Oracle), donde precisión="número total de dígitos" (de 1 a 38 en Oracle) y escala="número de decimales" (de -84 a 127 en Oracle, con valor 0 por defecto). También admite **`DEC(`**p,e**`)`** o **`NUMERIC(`**p,e**`)`**.
  - **Caracteres: `CHAR(`**n**`)`** o **`CHARACTER(`**n**`)`**, n=longitud fija (por defecto n=1). También **`VARCHAR(`**n**`)`**, n=longitud máxima (Oracle aconseja usar **`VARCHAR2`** con 4000 de máximo). Para cadenas muy largas usar **`LONG`** (máximo 2GB) o **`CLOB`** (*Character Large OBject*, máx. 4GB).
    - **`NCHAR`** y **`NVARCHAR2`** usan el juego de caracteres **N**acional definido al crear la BD.
  - **Cadenas de bits** (para gráficos, sonidos, ficheros binarios...)**: `BIT(`**n**`)`**, n=longitud fija o **`BIT VARYING(`**n**`)`**, con n=longitud máxima. Por defecto n=1. En Oracle se prefiere usar **`RAW(`**tamaño_fijo_máx_2000bytes**`)`** o **`LONG RAW`** (sin argumento y con un tamaño máximo de 2GB). Últimamente Oracle aconseja usar **`LOB`** o **`BLOB`** (*Binary Large OBject*, máximo 4GB), o también **`BFILE`** para almacenar la localización de un fichero binario.
  - **Fecha y Hora: `DATE`** (año, mes, día: YYYY-MM-DD). **`TIME`** (horas, minutos, segundos: HH:MM:SS). **`TIMESTAMP`** incluye ambos (sinónimo a **`DATE`** en Oracle).
    - Se usan las funciones **`TO_CHAR`** y **`TO_DATE`** para convertir un dato de tipo fecha a texto y viceversa.
- **Definiciones de DOMINIOS:** Crear nuevos tipos de datos:

  `CREATE DOMAIN <Nombre> AS <Tipo>;`
  - **Ejemplo: `CREATE DOMAIN NIF_TYPE AS CHAR(12);`**
  - **Utilidades:** Hacer las definiciones más legibles y hacer más fáciles los cambios de dominio de ciertos atributos. En Oracle se traduce como **`CREATE TYPE`**.

DDL de SQL: CREATE TABLE

- **Comando CREATE TABLE:** Crea una nueva relación/tabla **base** con su nombre, atributos (nombres y dominios) y restricciones:

```
CREATE TABLE <NombreTabla> (
      <NombreA1> <TipoA1> <Valor por defecto>
                 <Restricciones para A1>,
      <NombreA2> <TipoA2> <Valor por defecto>
                 <Restricciones para A2>,
      ...
      <RestriccionesTabla>);
```

- **Valor por defecto:** Se asigna al atributo si no se especifica otro valor.
  - **DEFAULT** <Valor>
- **Restricciones para un Atributo:** Son restricciones que afectan sólo a un atributo particular.
- **Restricciones para la Tabla:** Pueden afectar a uno o más atributos.
  - Si una restricción involucra un único atributo, esta restricción puede escribirse como restricción para ese atributo o al final, como restricción para la tabla.
  - Si una restricción involucra varios atributos debe escribirse como restricción para la tabla.

DDL de SQL: CREATE TABLE

- **Restricciones:** Pueden tener un **Nombre**, que se escribe al principio, con el formato: **CONSTRAINT <nombreR>** y un **<Estado>** que se verá después.
  - El **nombre** es útil para referirse a la restricción (para borrarla, cambiar su estado...).
  - Cuando se incumple una restricción, el SGBD debe dar un error indicando el nombre de dicha restricción: Escoger un nombre adecuado es útil para saber el motivo del error.
- **Restricciones de Atributos** (puede haber varias por cada uno):
  - **NOT NULL**
    - Atributo que no admite valores NULL.
    - Es la única restricción que SÓLO puede ponerse como restricción de atributo. Las demás pueden también ponerse como restricción de tabla.
  - **PRIMARY KEY**
    - Clave o llave primaria (no admite NULL)
  - **UNIQUE**
    - Llave candidata.
  - **CHECK (**<Condición>**)**
    - Comprueba que se cumple esa condición (si el atributo es no nulo).
  - **REFERENCES...**
    - Llave externa (o foránea): Debe indicarse la tabla referenciada.

DDL de SQL: CREATE TABLE

- **Restricciones de Tabla:**
  - **PRIMARY KEY (**<Atributos de la Llave Primaria>**)**
    - La llave primaria puede estar formada por varios atributos.
  - **UNIQUE (**<Llave Candidata o Secundaria>**)**
  - **CHECK (**<Condición>**)**
    - La condición puede estar formada por varios atributos, pero se interpretan como los atributos de la misma fila.
  - **FOREIGN KEY (**<Llave Externa>**) REFERENCES** <Tabla>**(**<Atributos>**)**
    [**ON DELETE {CASCADE | SET NULL | SET DEFAULT}**]
    [**ON UPDATE {CASCADE | SET NULL | SET DEFAULT}**]

Tabla de Restricción

Si se **borra** la llave referenciada, se borran las tuplas que la referencian
Si se **actualiza** la llave referenciada, se actualizan las tuplas que la referencian

Si se **borra/actualiza** la llave referenciada, se ponen a NULL los valores que la referencian (llave externa).

Si se **borra/actualiza** la llave referenciada, se ponen los valores que la referencian a su valor por defecto.

Las Restricciones en Oracle

- **Propiamente, Oracle no distingue entre restricciones de tabla y restricciones de Atributo (o de Columna):**
  - Oracle las almacena todas en la vista **USER_CONSTRAINTS** del Diccionario de Datos, con atributos como:
    - **CONSTRAINT_NAME:** Nombre de la restricción. Si no se le ha dado uno, Oracle le asigna uno con un código.
    - **TABLE_NAME:** Nombre de la tabla con dicha restricción.
    - **CONSTRAINT_TYPE:** Es un carácter (**P** para **PRIMARY KEY**, **U** para **UNIQUE**, **R** para una restricción de integridad referencial, **C** para una restricción de tipo **CHECK** (o **NOT NULL**) con la condición almacenada en el atributo **SEARCH_CONDITION**...)
    - **STATUS:** Estado de la restricción (**ENABLE** o **DISABLE**).
  - La diferencia entre restricciones de tabla y de atributo es sólo a nivel sintáctico sobre dónde y cómo deben escribirse:
    - Las restricciones que involucren varios atributos deben ser consideradas forzosamente como restricciones de tabla.
    - Las restricciones **NOT NULL** y **DEFAULT** son forzosamente restricciones de atributo, aunque estrictamente hablando **DEFAULT** no es una restricción.
- **Oracle 8 no implementa las opciones de ON UPDATE ni la opción ON DELETE SET DEFAULT.**
  - Por defecto, Oracle no permite borrar una tupla si existe una o varias tuplas que estén haciendo referencia a algún valor de la tupla que se intenta borrar (ORA-2292).
- **Por defecto, las restricciones de tipo CHECK sólo se exigen si los atributos involucrados tienen valores distintos de NULL.**

DDL de SQL estándar: DROP *y* ALTER

- **Borrar Esquemas y Tablas: DROP**
  - **DROP SQUEMA <NombreEsquema [CASCADE | RESTRICT]**
    - **CASCADE** borra el esquema totalmente y **RESTRICT** sólo si está vacío.
  - **DROP TABLE <NombreTabla> [CASCADE | RESTRICT]**
    - **CASCADE:** Borra la tabla (y su contenido). Si hay referencias sobre ella (llaves externas en otras tablas), dichas restricciones son borradas.
    - **RESTRICT:** Borra la tabla si no hay referencias sobre ella.
      - En Oracle **RESTRICT** no existe y es la opción por defecto. Para borrar las restricciones que hacen referencia a la tabla se usa **CASCADE CONSTRAINTS**.
- **Modificar Tablas: ALTER TABLE <NombreTabla> <ACCIÓN>**
  - **Añadir columna: ADD (<NombreA, Tipo, Restric_de_Columna>);**
    - En Oracle, para modificar un atributo se usa **MODIFY** en vez de **ADD**.
  - **Borrar columna: DROP <NombreA> [CASCADE|RESTRICT];**
  - **Añadir restric. de Atributo: ALTER <NombreA> SET <RestricA>;**
  - **Borrar restric.: ALTER <NombreA> DROP <TipoRA: DEFAULT...>;**
  - **Borrar restric. de tabla** (debe tener un nombre)**:**
    DROP CONSTRAINT <NombreC> CASCADE;
  - **Añadir restric. de tabla: ADD (<Restric_de_Tabla>);**

ALTER TABLE *en Oracle 8*

- **Modificar Tablas: ALTER TABLE <NombreTabla> <ACCIÓN>**

Añadir Columna: ADD (<NombreA> <Tipo> [<Restric_Columna>]);

Añadir Restricción de Tabla: ADD (<Restric_de_Tabla>);

Modif. Col.: MODIFY (<NombreA> [<Tipo>] [DEFAULT <expr>][[NOT] NULL]);

Estado de una Restricción: MODIFY CONSTRAINT <NombreR> <Estado>;

donde **<Estado>** es: (el **<Estado>** puede seguir a toda restricción)

[NOT] DEFERRABLE: Indica si el chequeo de la restricción se aplaza al final de la transacción o no se aplaza (por defecto), comprobándola tras la sentencia DML.

ENABLE [VALIDATE|NOVALIDATE] : Activa la restricción para los nuevos datos (opción por defecto). **VALIDATE** (opción por defecto) exige que se compruebe si la restricción es *válida* en los datos antiguos (lo que tenía previamente la tabla).

DISABLE: Desactiva la restricción.

Si a una restricción no se le ha asignado ningún nombre, Oracle le asigna un nombre. Puede consultarse en la vista **USER_CONSTRAINTS** del diccionario.

Borrar R. por Tipo: DROP {PRIMARY KEY | UNIQUE(<Cols>)} [CASCADE];

No pueden borrarse esas dos restricciones si existe una llave externa que referencie sus atributos**:** Con **CASCADE** se borran todas esas llaves externas.

Borrar Restricción por Nombre: DROP CONSTRAINT <NombreR>;

Obligatorio para las restricciones de tipo **CHECK** y **REFERENCES**.

Borrar Columna: DROP COLUMN <NombreA> [CASCADE CONSTRAINTS];

Renombrar Tabla: RENAME TO <Nuevo_Nombre_Tabla>;

DDL de SQL: Ejemplos

```
– CREATE TABLE CLIENTE (
    NIF    CHAR(12) NOT NULL,
    Nombre CHAR(80) NOT NULL,
    Edad   NUMBER(2) CONSTRAINT Edad_Pos CHECK (Edad>0),
    CONSTRAINT PK_Cliente PRIMARY KEY(NIF));
– CREATE TABLE MASCOTA (
    NIF_Owner CONSTRAINT Sin_Propietario
              REFERENCES CLIENTE(NIF)
              ON DELETE CASCADE,
    Nombre CHAR(30) NOT NULL,
    Fecha_Nac DATE,
    Especie CHAR(30) DEFAULT 'Perro' NOT NULL,
    CONSTRAINT PK_Mascota PRIMARY KEY(NIF_Owner,Nombre));
– ALTER TABLE CLIENTE ADD Telefono CHAR(20);
– ALTER TABLE CLIENTE DROP COLUMN Edad CASCADE CONSTRAINTS;
– ALTER TABLE MASCOTA DROP CONSTRAINT Sin_Propietario;
– ALTER TABLE MASCOTA MODIFY Especie NULL;
– ALTER TABLE MASCOTA DROP PRIMARY KEY;
– ALTER TABLE MASCOTA ADD PRIMARY KEY (NIF_Owner,Nombre);
```

Observación: En una llave externa no es necesario poner el tipo. En ese caso, copia el tipo del atributo padre.

DML de SQL: Consultas con SELECT

- **SELECT <Lista_Atributos>**
  FROM <Lista_Tablas>
  WHERE <Condición>;

Esquema:
Suministrador (S#, NombreS, Dirección, Ciudad);
Pieza (P#, NombreP, Peso, Cantidad);
Suministros (S#,P#);

- **Ejemplos:**
  - "Nombres de Piezas que pesen más de 15 gramos y que su Cantidad sea menor o igual que 30":
    ```
    SELECT NombreP FROM Pieza
    WHERE Peso>15 AND Cantidad<=30;
    ```
  - "Piezas de las que se ignore el Peso":
    ```
    SELECT * FROM Pieza
    WHERE Peso IS NULL;
    ```
    No usar Peso=NULL. Cada NULL es distinto a otro.
  - "Números de Suministradores que Suministren una Pieza de 15 gramos":
    ```
    SELECT S# FROM Pieza, Suministros
    WHERE Pieza.P# = Suministros.P# AND Peso=15;
    ```
  - "Nombres de Suministradores que Suministren una Pieza de 15 gramos":
    ```
    SELECT NombreS FROM Suministrador, Pieza, Suministros
    WHERE Suministros.P# = Pieza.P# AND Peso=15
      AND Suministros.S# = Suministrador.S#;
    ```
  - "Nombres de Piezas suministradas desde Málaga": Usando **alias** de tablas.
    ```
    SELECT NombreP
    FROM Suministrador S, Pieza P, Suministros SP
    WHERE SP.P# = P.P# AND SP.S# = S.S# AND Ciudad='Málaga';
    ```

SELECT: *Pseudocolumnas en Oracle*

- **Pseudocolumnas:** Son valores que se comportan como si fueran columnas, pero no tienen valores almacenados en la BD. No se pueden insertar, actualizar o borrar sus valores.
  - **ROWID:** Es la dirección de una fila concreta en una tabla concreta. Permite averiguar cómo se almacenan las filas de una tabla. Además, es la forma más rápida de acceder a una fila particular.
    - No debe usarse como llave primaria, pues el SGBD puede reorganizarlas (al borrar o insertar una fila, por ejemplo).
    - Ejemplo: `SELECT ROWID,NIF FROM Empleado WHERE Dpto=20;`
  - **ROWNUM:** Es el número de orden de cada fila en una consulta particular.
    - Se asigna en el orden en el que las tuplas van siendo seleccionadas.
    - Ejemplo: "Selecciona los 10 empleados con mayor salario"
      `SELECT NIF FROM (SELECT * FROM Empleado ORDER BY Salario DESC) WHERE ROWNUM <= 10;`
      - La subconsulta es necesaria para que el **ROWNUM** se aplique después de ordenar.
    - Consulta que no devuelve nada: `SELECT * FROM Empleado WHERE ROWNUM>1;`
      - La primera fila recuperada es la 1 y la condición es falsa, por lo que NO se recupera. La segunda tupla es ahora la primera y también hace que la condición sea falsa.
    - Se puede utilizar para actualizar un atributo de cierta tabla, con valores únicos y correlativos: `UPDATE Tabla SET AtributoNum = ROWNUM;`
  - Secuencia.**CURRVAL** y Secuencia.**NEXTVAL:** Permiten obtener el valor actual y el siguiente de una Secuencia previamente creada (**CREATE SEQUENCE**).
    - Una secuencia permite generar una sucesión de números únicos (sin repetirse), ideal para generar llaves primarias artificiales.
  - **LEVEL:** Es el nivel en una consulta jerárquica (usando **START WITH** y **CONNECT BY**).

SELECT: *Funciones en Oracle*

- **Funciones:** Oracle permite utilizar funciones ya definidas, que simplifican ciertas operaciones. Además, también permite crear nuestras propias funciones. Algunos Ejemplos:
  - **Funciones Numéricas: ABS** (valor absoluto), **SIN** (seno), **SINH** (seno hiperbólico), **COS** (coseno), **TAN** (tangente), **SQRT** (raíz cuadrada), **POWER(base,exp)** (potencia), **EXP** (exponencial con e = 2.71828183), **LN** (logaritmo neperiano), **LOG(b,n)** (logaritmo en base **b** de **n**), **ROUND** (redondear números), **TRUNC** (truncar números)...
  - **Funciones de Caracteres: CHR** (carácter que corresponde a un número), **LOWER** (devuelve la cadena del argumento con todas las letras en minúsculas), **UPPER** (pasa a mayúsculas), **LPAD**/**RPAD** (ajusta a la izquierda/derecha una cadena), **LTRIM**/**RTRIM** (quita ciertos caracteres de la izquierda/derecha de una cadena), **SUBSTR** (extrae una subcadena de una cadena), **REPLACE** (reemplaza subcadenas),
  - **Funciones de Caracteres que devuelven números: ASCII** (código ASCII de un carácter), **LENGTH**/**LENGTHB** (longitud de una cadena en caracteres/bytes), **INSTR** (buscar caracteres en una cadena)...
  - **Funciones de Fecha: ADD_MONTHS** (suma meses a una fecha), **LAST_DAY** (devuelve el último día del mes de una fecha dada), **SYSDATE** (fecha y hora actual del sistema)...
  - **Funciones de Conversión: TO_CHAR** (convierte fechas y números a cadenas de caracteres), **TO_DATE** (convierte una cadena a un dato de tipo fecha **DATE**), **TO_LOB** (convierte datos de tipo **LONG**/**LONG RAW** a tipo **LOB**), **TO_NUMBER** (convierte una cadena con un número a un número)...
    - Ej.: `SELECT TO_CHAR(SYSDATE, 'MM-DD-YYYY HH24:MI:SS') "Fecha" FROM DUAL;` (**DUAL** es una tabla usada cuando se necesita poner algo en el **FROM**)
  - **Otras funciones: GREATEST/LEAST** (devuelve el mayor/menor valor de una lista de valores), **NVL**/**NVL2** (para cambiar valores **NULL** por otros valores), **USER**/**UID** (devuelven el nombre/número del usuario actual de la sesión), **USERENV** (para obtener diversa información sobre la sesión actual: Lenguaje, si es DBA, número de sesión...)...

Subconsultas o Consultas Anidadas

- **Subconsulta o Consulta anidada** (*nested query*)**:** Sentencia **SELECT** incluida dentro de otra **SELECT** (llamada consulta externa, *outer query*).
  - Una **subconsulta** genera un **multiconjunto** de valores: Podemos ver si un valor **pertenece** al multiconjunto (**IN**), si es mayor, igual... que **todos** (**ALL**) o **alguno** (**SOME**, **ANY**) de los valores del multiconjunto:

Las **subconsultas** suelen ser **ineficientes.**

```
SELECT * FROM Pieza, Suministrador
WHERE (S#,P#) IN (SELECT * FROM Suministros);
```

Sustituye a una doble Reunión Natural

```
SELECT * FROM Pieza
WHERE P# IN (SELECT P#
             FROM Suministros SP, Suministrador S
             WHERE SP.S#=S.S# AND Ciudad='Málaga')
   OR P# IN (SELECT P# FROM Pieza
             WHERE Peso IS NOT NULL AND Cantidad IN (15,20));
```

¡¡JOIN: No es la mejor manera de efectuar esta consulta.

```
SELECT Nombre FROM Empleado
WHERE Salario > ALL (SELECT Salario FROM Empleado
                     WHERE Dpto IN (1,3,5,7)
                        OR Dpto IN (SELECT Dpto FROM Empleado
                                    WHERE NIFSupervisor=999));
```

```
SELECT Nombre FROM Empleado E
WHERE  Dpto IN (SELECT Dpto FROM Empleado
                WHERE Salario > E.Salario);
```

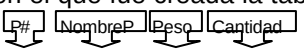
SUBCONSULTA correlacionada: Usa atributo de la consulta externa. **Se evalúa una vez por cada tupla de la consulta externa.**

Resumen del Comando SELECT

- **SELECT [DISTINCT]** <Select_list>
  - Expresiones a recuperar (columnas, funciones, operaciones, pseudocolumnas...), eliminando repetidos o no.
- **FROM** <Table_list>
  - Tablas necesarias para la consulta (incluyen tablas de reunión, subconsultas...).
- **[WHERE** <Condición> **]**
  - Condición de selección de tuplas, incluyendo condiciones de reunión.
- **[GROUP BY** <Atributos_para_Agrupar> **]**
  - Atributos por los que agrupar el resultado (cada grupo tendrá los mismos valores en estos atributos). Las funciones de grupo o de agregación (**COUNT**, **SUM**, **MAX**, **MIN**, **AVG**, **STDEV**, **VARIANCE**...), con **DISTINCT** o **ALL**, se aplicarán sobre cada uno de estos grupos. Se aplicarán sobre todas las tuplas si no existe esta cláusula.
- **[HAVING** <Condición_de_Grupo> **]**
  - Condición sobre los grupos (no sobre las tuplas).
- **[ORDER BY** <Atributos_para_Ordenar> **]**
  - Lista de atributos por los que ordenar o sus posiciones en la <Select_list>. Tras cada atributo puede ponerse **ASC** (por defecto) o **DESC**. El orden de estos atributos influye.

Insertar Tuplas: Comando **INSERT**

- **INSERTA** una o varias tuplas en una relación. Formato:
  INSERT INTO <Tabla> **VALUES** (a1, a2, ..., an);
  - **Lista de valores:** En el mismo orden en el que fue creada la tabla con **CREATE TABLE**.
  - **Ej.:** **INSERT INTO** pieza **VALUES** (4,'Teja',50,1000);



- Puede especificarse sólo un **subconjunto de atributos**. Formato:
  INSERT INTO <Tabla>**(**<Lista_Atribs>**) VALUES** (...);
  - **Lista de atributos:** Nombres de los atributos en los que se desea insertar algún valor. Los valores deben estar en ese orden.
    - Atributos ausentes: Se les asigna su valor por defecto o NULL.
    - Deben estar todos los atributos que no admiten NULL y que además no tienen valor por defecto (restr. **NOT NULL** y **DEFAULT**).
- **Insertar varias tuplas:** Separadas por comas y entre paréntesis.
  - Se puede sustituir la cláusula **VALUES**(. . .) por una subconsulta: Se insertarán TODAS las tuplas recuperadas por esa subconsulta.
- **SGBD:** Debe gestionar que se cumplan las restricciones (especialmente la de integridad referencial). Si no, lo tendrá que hacer el usuario.

Borrar Tuplas: Comando **DELETE**

- **BORRA** una o varias tuplas en una relación. Formato:
  DELETE [FROM] <Tabla> **[**<Alias>**] [WHERE** <Cond>**];**
  - **Borra** las tuplas que cumplan la condición**.**
  - **Puede implicar el borrado de otras tuplas en otras tablas,** para que se cumpla una restricción de integridad referencial.
  - **Si se omite la cláusula WHERE** (y su condición), se borran todas las tuplas de la tabla: La tabla quedará vacía (pero sigue existiendo).
  - **Ejemplos:**
    - **DELETE** Suministrador **WHERE** S# **IN** (2,4,8);
    - **DELETE** Suministros SP
      WHERE S# **IN** (**SELECT** S# **FROM** Suministrador
      WHERE Ciudad='Tegucigalpa');
    - **DELETE** Pieza
      WHERE Peso-250 > (**SELECT** AVG(Peso) **FROM** Pieza
      WHERE Peso > (**SELECT** AVG(Peso)
      FROM Pieza));

Actualizar Tuplas: Comando `UPDATE`

- **ACTUALIZA** o **MODIFICA** tuplas de una relación. Formato:

```
UPDATE <Tabla> [<Alias>] SET <Actualizaciones>
                         [ WHERE <Cond> ];
```

  - `<Actualizaciones>` puede ser:
    - **1.** Una lista de asignaciones separadas por coma del tipo:
      `<Atributo> = <Expresión>`
      (Una expresión puede ser una subconsulta que recupere sólo un valor).
    - **2.** Una lista de atributos entre paréntesis a los que se le asigna una subconsulta: `(<A1>, ..., <Am>) = (<Subconsulta>)`
  - La cláusula **`WHERE`** selecciona las tuplas a modificar.
  - Modificar la **llave primaria** puede acarrear la modificación de llaves externas en otras tablas.

- Ejs.:

```
• UPDATE Pieza SET Nombre='Eje', Peso=9 WHERE P#=5;
• UPDATE Pieza SET Cantidad=Cantidad+100
       WHERE P# IN (SELECT P# FROM Suministros
                    WHERE S# IN (3,7));
• UPDATE Pieza SET (Peso,Cantidad)=(SELECT Peso,
       Cantidad FROM Pieza WHERE P#=6) WHERE P#=5;
```


Creación de Vistas: `CREATE VIEW`

- **VISTA:** Es una **tabla virtual** cuyas tuplas derivan de otras tablas (que pueden ser tablas base o también otras vistas).
  - Sus tuplas no se almacenan sino que se calculan a partir de las tablas de las que dependa.
  - Son útiles para usar, como si fueran tablas, consultas que se efectúan frecuentemente, y también para cuestiones de **seguridad**.
  - **Formato:**

```
CREATE [OR REPLACE] [[NO] FORCE] VIEW
<NombreV> [(<Lista_Atrbs>)] AS (<Subconsulta>)
[WITH READ ONLY];
```

    - Crea la vista `<NombreV>`, asociada a la subconsulta especificada.
    - La **lista de atributos** es el nombre de los atributos de la vista: Por defecto toma los nombres de los atributos de la subconsulta. Son necesarios si los atributos son calculados (funciones de grupo...).
    - **`OR REPLACE`:** Permite modificar una vista ya existente sin borrarla.
    - **`WITH READ ONLY`:** Indica que no se permitirán borrados, inserciones o actualizaciones en la vista.
    - **`FORCE`:** Fuerza a que la vista se cree aunque no existan los objetos que se usan en ella (tablas, otras vistas...) o no se tengan privilegios suficientes. Esas condiciones serán necesarias para usar la vista. La opción contraria es **`NO FORCE`**, que es la opción por defecto.

Vistas: Ejemplos y Observaciones

– **Ejemplos:**

• **CREATE OR REPLACE VIEW** SumiNombres
AS (SELECT NombreS, NombreP
FROM Suministros SP,Suministrador S,Pieza P
WHERE SP.S#=S.S# **AND** SP.P#=P.P#**);**

• **CREATE OR REPLACE VIEW** Cantidad(NombreS,NumPiezas)
AS (SELECT NombreS, **COUNT(*)**
FROM Suministros SP, Suministrador S
WHERE SP.P#=P.P#
GROUP BY NombreS**);**

OJO: ¿Qué ocurre si varios Suministradores tienen el mismo nombre?

– **Observaciones:**

- Las vistas pueden **consultarse como si fueran tablas**.
- Una vista está **siempre actualizada** (*up to date*): Si se modifican las tablas de las que depende, la vista reflejará esos cambios.
- Para que la vista **NO se actualice,** no hay que crear una vista, sino una "instantánea", "foto" o vista materializada (*materialized view*) de la BD (con **CREATE SNAPSHOT**, o bien con **CREATE MATERIALIZED VIEW**).
- Para **borrar una vista** que ya no es útil: **DROP VIEW** <NombreV>;

Vistas: Implementación y Modificación

– **Implementación de vistas:** Dos técnicas principales:

- **Modificación a consulta:** Se modifica la vista, como si fuera una subconsulta, cada vez que se usa. Es muy **ineficiente**, especialmente si la vista es compleja y se usa frecuentemente.
- **Materialización de la vista:** Consiste en crear físicamente la vista la primera vez que se usa, suponiendo que se va a usar varias veces.
  - La vista **se borrará** si no se usa en cierto tiempo.
  - Técnicas para mantener la **vista actualizada:** Si se borran, modifican o insertan tuplas en las tablas base hay que hacerlo también en la vista.

– **Modificar directamente una vista puede ser complicado:**

- Es **fácil** si es una vista sobre una tabla, sin funciones de agregación e incluye la llave primaria (o una candidata). En general, se consideran vistas **NO actualizables** si están definidas sobre varias tablas (en una reunión) o si usan agrupamiento y/o funciones de agregación.
- A veces, una **modificación en una vista** puede traducirse de varias formas en las relaciones base, lo cual implica que tal modificación no debe admitirse por ser **ambigua**. A veces, se escoge la más probable.
  - **Ej.:** Si en la vista SumiNombres se modifica el nombre de una Pieza: ¿Queremos modificar el nombre de la pieza o queremos modificar la pieza que es suministrada por el correspondiente suministrador?

Control de Transacciones

- **SQL permite controlar la ejecución de una transacción:**
  - **Una transacción empieza con** la primera orden SQL después de conectarse a la base de datos o con la primera orden SQL después de terminar la transacción anterior.
  - **Una transacción termina con: `COMMIT`** o **`ROLLBACK`**.
- **Órdenes SQL de Control de Transacciónes:**
  - **`COMMIT`** (o **`COMMIT WORK`**)**:** Si la transacción terminó bien y se deben **guardar los cambios** efectuados a la base de datos.
    - Además, **`COMMIT`** hace que los cambios sean visibles por otras sesiones y que se liberen los bloqueos establecidos por la transacción.
    - Hasta que no se usa **`COMMIT`** los cambios no son permanentes, sino temporales y sólo pueden ser vistos por la sesión que los hizo.
  - **`ROLLBACK`:** Si la transacción no terminó bien y se deben **deshacer los cambios** efectuados a la base de datos.
    - **`ROLLBACK`** también libera los bloqueos establecidos.
    - Esta es la opción por defecto si una sesión se desconecta de la base de datos SIN terminar la transacción.
  - **`SAVEPOINT <Nombre>`:** Para deshacer sólo parte de la transacción, se pueden poner varios **puntos de salvaguarda** con distinto nombre cada uno.
    - Tras esto, la orden **`ROLLBACK TO SAVEPOINT <Nombre>`**, deshace todo lo hecho desde el punto **<Nombre>** y libera los bloqueos establecidos tras ese punto de salvaguarda.
    - La transacción no termina con este tipo de **`ROLLBACK`**.

Permisos: GRANT *y* REVOKE

- **Permisos: Concederlos** (**GRANT**) y **Revocarlos** (**REVOKE**).
  - Pueden usar estas órdenes los **propietarios** (*owner*) de los objetos (tablas, vistas...) o el **DBA** (administrador de la BD).
  - **Permisos sobre Objetos:** Hay que especificar el **Tipo de Permiso** y el **Objeto** al que se aplicará:
    - **Tipos de Permisos: `SELECT`**, **`INSERT`**, **`DELETE`**, **`UPDATE`**, **`ALTER`**, **`REFERENCES`**, **`INDEX`** (para crear índices) y **`EXECUTE`** (para programas).
      - **`ALL`:** Especifica TODOS los Permisos posibles sobre el Objeto.
        - » Por defecto, el propietario de un objeto tiene todos los permisos posibles sobre él.
    - **Tipos de Objetos:** Tablas, Vistas, Secuencias, Procedimientos, Funciones, *Snapshots*...
  - **Permisos del Sistema:** Son **permisos** para crear, modificar, borrar, consultar... **objetos** (tablas, vistas, usuarios, *triggers*, sesiones, roles, secuencias, *snapshots*...).
    - Ejemplos:
      - **`CREATE TABLE`:** Para crear tablas propias.
      - **`CREATE ANY TABLE`:** Para crear tablas en otros esquemas o usuarios.
      - **`ALTER ANY TABLE`:** Para modificar tablas. Sin **`ANY`** no existe porque es absurdo.
      - **`DROP ANY TABLE`:** Para borrar tablas de cualquier esquema.
      - **`DELETE ANY TABLE`:** Borrar filas de tablas/vistas de cualquier esquema.
      - **`INSERT ANY TABLE`:** Insertar filas en tablas/vistas de cualquier esquema.
      - **`UPDATE ANY TABLE`:** Actualizar tablas/vistas de cualquier esquema.
      - **`SELECT ANY TABLE`:** Consultar tablas/vistas de cualquier esquema.
      - **`EXECUTE ANY PROCEDURE`:** Para ejecutar procedimientos y funciones.
    - **`ALL PRIVILEGES`:** Especifica TODOS los Permisos del Sistema.

Permisos: GRANT *y* REVOKE

- **Formato:**
```
GRANT <Lista_Permisos> [ON <Objeto>]
TO <Usuarios> [WITH {GRANT|ADMIN} OPTION];
```
  - Con **Permisos del Sistema** se elimina la **cláusula ON**.
  - **<Usuarios>:** Pueden ser también un Rol (al que se asigna ese permiso). Si se asignan los permisos a **PUBLIC** se asignan a todos los usuarios.
  - **WITH GRANT OPTION:** Permite al **<Usuario>** conceder dicho privilegio a otro usuario. Con permisos del **sistema** usar **WITH ADMIN OPTION**.
  - Con **UPDATE** se puede especificar la columna**: GRANT UPDATE ON Tab(col)...**
  - **Existen unos Roles Predefinidos que Oracle crea automáticamente:**
    - **DBA:** Contiene todos los privilegios de la BD y debe concederse con cuidado.
    - **CONNECT:** Para conectarse, crear vistas y usar objetos a los que tenga acceso.
    - **RESOURCE:** Para crear objetos (tablas, índices...).
  - **Ejs.:**
```
• GRANT CONNECT, Rol_Programador TO Araujo;
• GRANT CREATE USER, ALTER USER, DROP USER
    TO Nous, Zeus  WITH ADMIN OPTION;
• GRANT CREATE ANY PROCEDURE, CREATE TRIGGERS TO Apolo;
• GRANT ALL ON Empleados TO Casandra WITH GRANT OPTION;
• GRANT SELECT ON Empleados TO PUBLIC;
• GRANT REFERENCES (DNI), UPDATE (Salario, Cta_Banco)
    ON Empleados TO Rol_Nominas;
```
- **Formato:**
```
REVOKE <Lista_Permisos> [ON <Objeto>]
FROM <Usuario>;
```
  - **Ej.:**
```
• REVOKE CREATE SESSION FROM Usuario1, Usuario2;
```


Introducción a PL/SQL

- **Características de PL/SQL (*Procedural Language*/SQL):**
  - Combina la potencia y flexibilidad de **SQL** con la de un lenguaje **3GL:**
    - **SQL es un Lenguaje de Cuarta Generación (4GL):** El lenguaje describe lo que debe hacerse pero no cómo hacerlo, dentro de una sintaxis relativamente simple.
    - **Lenguajes de Tercera Generación (3GL):** Tienen estructuras procedimentales para expresar cómo efectuar las operaciones.
      - **PL/SQL** permite utilizar o declarar: Variables y tipos de datos, estructuras de control (selección e iteración), procedimientos y funciones y, a partir de PL/SQL 8, también tipos de objetos y métodos.
  - Es un **lenguaje estructurado y potente**, con estructuras ideales para el trabajo con bases de datos.
  - **Integrado en el SGBD Oracle**, por lo que su ejecución es **eficiente**.
  - Permite empaquetar órdenes SQL, de forma que **minimice** la **comunicación** entre Cliente y Servidor de BD y los **accesos** a la BD.
  - PL/SQL se basa en el **Lenguaje Ada** (3GL) y tiene muchas de sus características (estructura de bloques, excepciones...)
  - Oracle y PL/SQL soportan el **estándar ANSI SQL92 (o SQL2)** a nivel básico (se pretende que sea a nivel completo en futuras versiones).

Introducción a PL/SQL

- **Caracteres:** PL/SQL utiliza los caracteres **ASCII**. PL/SQL no distingue entre mayúsculas y minúsculas, excepto en una cadena de caracteres entre comillas.
- **Palabras Reservadas:** PL/SQL tiene más de 200. Están reservadas para la definición del lenguaje y suelen ponerse siempre en mayúsculas.
- **Identificadores:** Empiezan con una letra, seguida opcionalmente de letras, números y tres caracteres especiales (**$**, _ y **#**). Máxima Longitud: 30.
  - **Identificadores entre comillas dobles:** Como norma general no deben usarse pues denotan un mal estilo de programación, pero son útiles para: a) Introducir otros caracteres imprimibles (incluyendo el espacio), b) Distinguir entre mayúsculas y minúsculas y c) Utilizar una palabra reservada como identificador.
    - Ejemplo: Si una tabla se llama **`EXCEPTION`**, usar **`"EXCEPTION"`** (en mayúsculas).
- **Delimitadores:** Símbolos con un significado especial, como son:
  - **Operadores Aritméticos:** **+**, **-**, *****, **/**, ****** (potenciación).
  - **Operadores Relacionales:** **=**, **>**, **<**, **>=**, **<=**, **<>**, **!=**, **^=**, **~=**.
  - **Otros Operadores:** **:=** (asignación), **||** (concatenar cadenas), **=>** (asociación), **..** (rango), **`LIKE`**, **`BETWEEN`**, **`IN`**, **`IS NULL`** y los operadores lógicos (**`NOT`**, **`AND`** y **`OR`**).
  - **Otros:** Paréntesis para alterar la precedencia (**(** y **)**), terminador de orden (**;**), comentarios (**--**, **/*** y ***/**), espacio, tabulador, retorno de carro, delimitadores de etiquetas (**<<** y **>>**), indicador de variable de asignación (**:**), delimitadores de cadenas (**'** , comilla simple), identificador entre comillas dobles (**"**), selector de componente de un registro, una tabla... (**.**), enlace a BD (**@**), indicador de atributo (**%**) y separador de elementos (**,**).

Bloques PL/SQL

- **Bloque PL/SQL:** Unidad básica en **PL/SQL.**
  - Todos los programas PL/SQL están compuestos por **bloques**, que pueden anidarse.
  - Normalmente cada **bloque** realiza una unidad lógica de trabajo
  - **Estructura de un bloque PL/SQL:**

```
DECLARE
  /* Declaración de variables, tipos, cursores y
     subprogramas locales, con VISIBILIDAD hasta END; */
BEGIN
  /* Programa: Órdenes (SQL y procedimentales) y bloques.
     Esta sección es la única obligatoria. */
EXCEPTION
  /* Manejo de excepciones (errores) */
END;
```

  - **Excepciones:** Permiten controlar los errores que se produzcan sin complicar el código del programa principal.
    - Cuando se produce un error, la ejecución del programa salta a la sección **`EXCEPTION`** y allí puede controlarse qué hacer dependiendo del tipo de error producido.

Bloques PL/SQL

- **Tipos de Bloques:**
  - **Anónimos (*anonymous blocks*):** Se construyen normalmente de manera dinámica para un objetivo muy concreto y se ejecutan, en general, una única vez.
  - **Nominados (*named blocks*):** Son similares a los bloques anónimos pero con una etiqueta que da nombre al bloque.
  - **Subprogramas:** Son procedimientos (*procedures*), funciones (*functions*) o grupos de ellos, llamados paquetes (*packages*).
    - Se construyen para efectuar algún tipo de operación más o menos frecuente y se almacenan para ejecutarlos cuantas veces se desee.
    - Se ejecutan con una llamada al procedimiento, función o paquete.
  - **Disparadores (*triggers*):** Son bloques nominados que se almacenan en la BD. Su ejecución está condicionada a cierta condición, como por ejemplo usar una orden concreta del DML.
- **Comentarios:** Pueden incluirse siempre que se desee.
  - **Monolínea:** Empiezan con 2 guiones **--** y terminan a final de línea.
  - **Multilínea:** Empiezan con **/*** y terminan con ***/** (como en C).

Bloques PL/SQL: Ejemplos

- **Bloque Anónimo:** Insertar 2 piezas Tornillo con distintos pesos.

```
DECLARE
  NumP NUMBER:=7; -- Número de Pieza, empezamos en 7
BEGIN
  INSERT INTO Pieza (P#,Nombre,Peso)
         VALUES (NumP,'Tornillo', 2);
  INSERT INTO Pieza (P#,Nombre,Peso)
         VALUES (NumP+1,'Tornillo', 4);
END;
```

- **Bloque Nominado:** Podemos poner una etiqueta a un bloque anónimo para facilitar referirnos a él.
  - Antes de la palabra **DECLARE** se pone el **nombre del bloque** entre ángulos dobles. **Ejemplo:** **<<Bloque_Insertar>>** **DECLARE ...**
  - Es buena costumbre, aunque es opcional, poner el nombre también después de la palabra **END**. **Ej.:** **END Bloque_Insertar;**
  - En **bloques anidados** puede ser útil nominarlos para referirse a una variable en particular de ese bloque, en caso de que existan varias con el mismo nombre: **Ejemplo:** **Bloque_Insertar.Var1**

Bloques PL/SQL: Ejemplos

- **Procedimientos:** Se usa la palabra reservada **PROCEDURE** junto con las de creación o reemplazo, sustituyendo a **DECLARE**.

```
CREATE OR REPLACE PROCEDURE Pieza7 AS
  NombrePieza7  VARCHAR2(50);
BEGIN
  SELECT NombreP INTO NombrePieza7 FROM Pieza
    WHERE P#=7;
  DBMS_OUTPUT.PUT_LINE ('Nombre de la Pieza 7: '
    ||NombrePieza7);
END;
```

Sobre SQL*Plus

- **PUT_LINE** es un procedimiento del paquete **DBMS_OUTPUT** que sirve para escribir un dato (**PUT**) seguido de un fin de línea (**NEW_LINE**). En **SQL*Plus** debe estar activada la salida por pantalla con: **SET SERVEROUTPUT ON [SIZE n]**, donde **n** Î **[2000,1000000]** es el máximo nº de bytes (**2000** por defecto).
- **SET LINESIZE n**, establece el tamaño de una línea en **SQL*Plus**, para todas sus salidas.
- **SHOW ERR**, muestra los errores de compilación de un bloque PL/SQL.
- **EXEC <sentencia>**, ejecuta una sentencia PL/SQL. Se puede usar para ejecutar un procedimiento sin tener que crear un bloque PL/SQL.

- **Disparadores o *Triggers*:** Hay que especificar **CUANDO** se disparará.

```
CREATE OR REPLACE TRIGGER PesoPositivo
  -- Se activará cada vez que se inserte o actualice
  BEFORE INSERT OR UPDATE OF Peso ON Pieza
  FOR EACH ROW
BEGIN
  IF :new.Peso < 0 THEN
     RAISE_APPLICATION_ERROR(-20100, 'Peso no válido');
  END IF;
END PesoPositivo;
```


Variables y Tipos de Datos

- **Variables PL/SQL:** Se declaran con el siguiente formato:
  <Nombre> <Tipo> [[CONSTANT|NOT NULL] :=<Valor_Inicial>];
  - **Tipos de datos:** Puede ser cualquier tipo válido en una columna de la BD (vistos anteriormente) y otros tipos adicionales. El valor por defecto es NULL, excepto que se especifique un valor como <Valor_Inicial> (puede sustituirse **:=** por **DEFAULT**). Con **NOT NULL** se requiere la inicialización.
- **Tipos Escalares:**
  - **Numéricos Reales: NUMBER(**p,e**)** y sus subtipos totalmente equivalentes definidos por cuestiones de compatibilidad: **DEC**, **DECIMAL**, **DOUBLE PRECISION**, **INT**, **INTEGER**, **NUMERIC**, **SMALLINT** y **REAL**.
    - Se almacenan en formato decimal: Para operaciones aritméticas deben traducirse a binario.
  - **Numéricos Enteros: BINARY_INTEGER**, que es un entero en binario (complemento a 2) con rango ±2147483647, ideal para variables sobre las que se efectuarán operaciones (contadores...).
    Tiene definidos subtipos restringidos en su rango: **NATURAL** [0, 2147483647], **NATURALN** (igual que **NATURAL** pero **NOT NULL**), **POSITIVE** [1, 2147483647], **POSITIVEN**, **SIGNTYPE** (–1, 0 y 1).
    - **PLS_INTEGER** es similar a **BINARY_INTEGER**, pero más rápido en las operaciones aritméticas y que genera un error si se produce un desbordamiento (ORA-1426) al asignarlo a un **NUMBER**.
  - **Carácter: VARCHAR2(** max_tama**)**, con max_tama<=32676 **bytes** (como columna de la BD admite 4000 → Cuidado con los errores). Si se usa un código distinto al código ASCII, el número total de caracteres puede ser menor.
    - **CHAR(** tama_fijo**)** con 1 por defecto y 32767 como máximo (como columna de la BD admite 255), *se rellena siempre con blancos*. **LONG** es una cadena de longitud variable con un máximo de 32760 (como columna de la BD admite 2GB). **NCHAR** y **NVARCHAR2** permiten almacenar cadenas en un conjunto de caracteres Nacional distinto al propio de PL/SQL.
  - **Binarios: RAW(**max_tama**)**, con max_tama<=32676 **bytes. LONG RAW** admite un máximo de 32760 **bytes. ROWID** es un tipo para almacenar identificadores de fila (pseudocolumna **ROWID**).
  - **Fecha: DATE**, como en SQL de Oracle almacena siglo, año, mes, día, hora, min. y segs. (7 bytes).
  - **Lógico: BOOLEAN**, con los siguientes valores posibles: **TRUE**, **FALSE** y **NULL**.
- **Tipos Compuestos: RECORD**, **TABLE** y **VARRAY**.
- **Tipos Referencias (punteros): REF CURSOR** y **REF** <TipoObjeto>.
- **Tipos LOB** (*Large OBject*)**: BFILE**, **LOB**, **CLOB** y **NLOB** (se usan con el paquete **DBMS_LOB**).

Variables y Registros

- **Especificación de Tipo con <Tabla>.<Columna>%TYPE:**
  - Sirve para dar a una variable el **tipo que tenga asignado una columna** de la BD, independientemente de cómo esté definido éste.
  - También puede usarse **sobre variables** anteriormente definidas.
  - Esto hace los **programas más robustos** frente a cambios de tipo.
  - **Ejemplos:**
```
DECLARE
   NumPieza Pieza.P#%TYPE;
   PiezaX   NumPieza%TYPE;
```
  - **La restricción `NOT NULL` no se hereda.**
- **Registros:** Son agrupaciones de datos relacionados similares a las estructuras (**`struct`**) del lenguaje C o los registros de Modula2.
  - **Es necesario definir un Tipo de Dato Registro, para declarar variables.**
  - **Formato:**
```
TYPE <Tipo_Registro> IS RECORD (
      <Campo1> <Tipo1> [[NOT NULL] :=<Expr1>],
                . . .
      <CampoN> <TipoN> [[NOT NULL] :=<ExprN>]);
```
  - **Notación Punto** para acceder a un campo**: <VariableRegistro>.<Campoi>**
  - **Asignar Registros:** Se permite si son del mismo tipo.
    - Si son de tipos distintos no se pueden asignar, aunque estén definidos igual. En ese caso, se pueden asignar campo a campo.
    - También se pueden asignar los campos de un **`SELECT`** en un registro compatible.
  - Declarar registros con los campos de una tabla**: <Tabla>%ROWTYPE.**

Órdenes SQL en un Programa PL/SQL

- **Órdenes SQL que pueden utilizarse:**
  - **DML: `SELECT`**, **`INSERT`**, **`UPDATE`** y **`DELETE`**.
    - Permiten utilizar variables como expresiones dentro de su sintaxis.
      - Consejo: No declarar variables con el mismo nombre que las columnas.
    - No pueden utilizarse variables para los nombres de tablas y columnas (para ello usar SQL dinámico).
  - **Control de Transacciones: `COMMIT`**, **`ROLLBACK`** y **`SAVEPOINT`**.
- **SQL Dinámico:** Permite **crear órdenes en Tiempo de Ejecución**, de forma que esas órdenes no son compiladas en **Tiempo de Compilación**. Se necesita usar el paquete **`DBMS_SQL`.**
  - **Ejemplos:** Para ejecutar órdenes de **DDL**, como crear una tabla (**`CREATE TABLE`**), pero también para insertar valores en ella (**`INSERT`**), ya que en tiempo de compilación no existirá esa tabla y, por tanto, no puede compilarse la orden de inserción.
- **Órdenes DML:**
  - **`INSERT`, `UPDATE`, `DELETE`:** No sufren modificaciones (excepto las órdenes **`UPDATE`** y **`DELETE`** para cursores usando **`CURRENT OF`**).
  - **`SELECT`:** Su sintaxis se ve modificada para indicar donde se almacenarán los datos seleccionados (cláusula **`INTO`**).

Orden SELECT-INTO *en PL/SQL*

- **SELECT:** Se requiere introducir los datos seleccionados en variables.
  - **Cláusula INTO:** Se coloca justo antes de **FROM** y después de la lista de elementos seleccionados. La palabra **INTO** va seguida de**:**
    - Una **lista de tantas variables** como elementos seleccionados y con sus tipos compatibles en el mismo orden, o bien,
    - Un **registro** con sus campos también compatibles.
  - **La consulta sólo debe recuperar una fila:** Si recupera varias se producirá un error. Para consultas múltiples, se deben usar **cursores**.
  - **Ejemplo:**

```
DECLARE
  Media   NUMBER;
  Hotel99 Hotel%ROWTYPE;
BEGIN
  -- Media de habitaciones de todos los hoteles:
  SELECT AVG(Nhabs) INTO Media FROM Hotel;
  -- Todos los datos del Hotel 99:
  SELECT * INTO Hotel99
     FROM Hotel WHERE HotelID = 99;
  -- Comparar con la Media:
  IF Media > Hotel99.Nhabs THEN
     UPDATE Hotel SET Nhabs=Nhabs+1 WHERE HotelID=99;
  ...
END;
```


Orden SELECT-INTO *en PL/SQL*

- La **orden SELECT..INTO** genera una **excepción o error,** cuando:
  - **Cuando NO se selecciona NINGUNA fila, o**
  - **Cuando se seleccionan VARIAS filas.**
- Esa **excepción** transfiere la ejecución automáticamente a la sección **EXCEPTION** del bloque PL/SQL actual.
  - **Ej.:**

```
DECLARE
  Hotel99 Hotel%ROWTYPE;
BEGIN
  SELECT * INTO Hotel99 FROM Hotel99
     WHERE Nombre='Costanera';
  -- Tratamiento del Hotel Costanera:
  ...
EXCEPTION
  -- Cuando no se recupera ninguna fila:
  WHEN NO_DATA_FOUND THEN
    INSERT INTO Hotel
      (HotelID,Nombre,PrecioHab,Nhabs,TienePiscina,Calif)
       VALUES (99, 'Costanera', 110, 60, 'S', 3);
  -- Cuando se recuperan varias filas:
  WHEN TOO_MANY_ROWS THEN
    DELETE Hotel WHERE Nombre='Costanera' AND HotelID<>99;
END;
```

Cursores Implícitos o Cursores SQL

- **Cursores SQL o Implícitos:** Se producen automáticamente cuando se ejecutan las órdenes SQL **INSERT**, **UPDATE**, **DELETE** y la orden **SELECT..INTO** (para recuperar una única fila).
  - Justo después de la orden SQL, pueden usarse las expresiones **SQL%FOUND**, **SQL%NOTFOUND** y **SQL%ROWCOUNT:**
  - **Ej.:**

```
UPDATE Hotel SET PrecioHab=PrecioHab + 10
  WHERE HotelID = 99;
IF SQL%NOTFOUND THEN
   INSERT INTO Hotel (HotelID,PrecioHab,Nhabs,TienePiscina)
         VALUES (99, 110, 60, 'S');
END IF;
```

  - **Ej.:**

```
UPDATE Hotel SET PrecioHab=PrecioHab + 10
  WHERE Nhabs > 50 AND TienePiscina = 'S';
DBMS_OUTPUT.PUT_LINE('Hoteles Procesados: '||SQL%ROWCOUNT);
IF SQL%ROWCOUNT=0 THEN
   INSERT INTO Hotel (HotelID,PrecioHab,Nhabs,TienePiscina)
         VALUES (99, 110, 60, 'S');
   DBMS_OUTPUT.PUT_LINE('Hotel 99 Insertado.');
END IF;
COMMIT;
```


Disparadores: TRIGGER

- Es un bloque PL/SQL que **se Ejecuta de forma Implícita** cuando se ejecuta cierta **Operación DML: INSERT, DELETE o UPDATE.**
  - Contrariamente, los procedimientos y las funciones se ejecutan haciendo una llamada **Explícita** a ellos.
  - **Un Disparador No admite Argumentos.**
- **Utilidad:** Sus aplicaciones son inmensas, como por ejemplo:
  - **Mantenimiento de Restricciones de Integridad complejas.**
    - Ej: Restricciones de Estado (como que el sueldo sólo puede aumentar).
  - **Auditoría de una Tabla,** registrando los cambios efectuados y la identidad del que los llevó a cabo.
  - **Lanzar cualquier acción** cuando una tabla es modificada.
- **Formato:**

```
CREATE [OR REPLACE] TRIGGER <NombreT>
   {BEFORE|AFTER} <Suceso_Disparo> ON <Tabla>
   [FOR EACH ROW [WHEN <Condición_Disparo>]]
<Bloque_del_TRIGGER>;
```

  - **<Suceso_Disparo>** es la operación **DML** que efectuada sobre **<Tabla>** disparará el *trigger*.
    - Puede haber varios sucesos separados por la palabra **OR.**

Elementos de un **TRIGGER**

- **Nombre:** Se recomienda que identifique su función y la tabla sobre la que se define.
- **Tipos de Disparadores:** Hay **12 Tipos** básicos según**:**
  - **Orden, Suceso u Operación DML: `INSERT`, `DELETE` o `UPDATE`.**
    - Si la orden **`UPDATE`** lleva una lista de atributos el Disparador sólo se ejecutará si se actualiza alguno de ellos**: `UPDATE OF <Lista_Atributos>`**
  - **Temporización:** Puede ser **`BEFORE` (anterior) o `AFTER` (posterior).**
    - Define si el disparador se activa **antes o después** de que se ejecute la operación **DML** causante del disparo.
  - **Nivel:** Puede ser **a Nivel de Orden** o **a Nivel de Fila (`FOR EACH ROW`).**
    - **Nivel de Orden** (*statement trigger*)**:** Se activan sólo una vez, antes o después de la **Orden** u operación **DML**.
    - **Nivel de Fila** (*row trigger*)**:** Se activan una vez por cada **Fila** afectada por la operación **DML** (una misma operación **DML** puede afectar a varias filas).
- **Ejemplo:** Guardar en una tabla de control la fecha y el usuario que modificó la tabla `Empleados`: (NOTA: Este trigger es **`AFTER`** y a nivel de orden)

```
CREATE OR REPLACE TRIGGER Control_Empleados
  AFTER INSERT OR DELETE OR UPDATE ON Empleados
BEGIN
  INSERT INTO Ctrl_Empleados (Tabla,Usuario,Fecha)
    VALUES ('Empleados', USER, SYSDATE);
END Control_Empleados;
```


Orden en la Ejecución de Disparadores

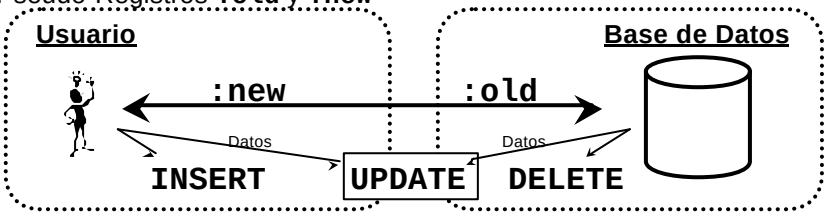
- Una tabla puede tener distintos Tipos de **Disparadores** asociados a una misma **orden DML.**
- En tal caso, el **Algoritmo de Ejecución** es**:**
  - **1. Ejecutar, si existe, el disparador tipo `BEFORE` a nivel de orden.**
  - **2. Para cada fila a la que afecte la orden:** (esto es como un bucle, para cada fila)
    - **a) Ejecutar, si existe, el disparador `BEFORE` a nivel de fila,** sólo si dicha fila cumple la condición de la cláusula **`WHEN`** (si existe).
    - **b) Ejecutar la propia orden.**
    - **c) Ejecutar, si existe, el disparador `AFTER` a nivel de fila,** sólo si dicha fila cumple la condición de la cláusula **`WHEN`** (si existe).
  - **3. Ejecutar, si existe, el disparador tipo `AFTER` a nivel de orden.**

Disparadores de Fila: `:old` *y* `:new`

- **Disparadores a Nivel de Fila:**
  - Se ejecutan **una vez por cada Fila** procesada por la orden **DML** disparadora.
  - Para acceder a la **Fila Procesada** en ese momento, se usan dos **Pseudo-Registros** de tipo **<TablaDisparo>%ROWTYPE: `:old` y `:new`**

Orden DML	`:old`	`:new`
INSERT	**No Definido: NULL** ∀ campo.	**Valores Nuevos** a Insertar.
UPDATE	**Valores Originales** (antes de la orden).	**Valores Actualizados** (después).
DELETE	**Valores Originales** (antes de la orden).	**No Definido: NULL** ∀ campo.

  - **Esquema** para la comprensión del significado y utilidad de los Pseudo-Registros `:old` y `:new`



Disparadores a Nivel de Fila: Ejemplo

  - **Ejemplo:** Programar un Disparador que calcule el campo código de Pieza (P#) cada vez que se inserte una nueva pieza:

OJO: No rellena huecos, si existen.

```
CREATE OR REPLACE TRIGGER NuevaPieza
  BEFORE INSERT ON Pieza FOR EACH ROW
BEGIN
  -- Establecer el nuevo número de pieza:
  SELECT MAX(P#)+1 INTO :new.P# FROM Pieza;
  IF :new.P# IS NULL THEN
     :new.P# := 1;
  END IF;
END NuevaPieza;
```

    - Cuando se ejecute la orden **DML** se emplean los valores del **Pseudo-Registro `:new`**. Una orden válida sería la siguiente, sin que produzca un error por no tener la llave**:**

```
INSERT INTO Pieza (Nombre, Peso)
       VALUES ('Alcayata', 0.5);
```

      - En este caso, si se suministra un valor para la llave primaria, tampoco producirá un error, pero ese valor será ignorado y sustituido por el nuevo valor calculado por el disparador.
  - **Modificar Pseudo-Registros:**
    - No puede modificarse el **Pseudo-Registro `:new`** en un disparador **AFTER** a nivel de fila.
    - El **Pseudo-Registro `:old`** nunca se modificará**:** Sólo se leerá.

Disparadores a Nivel de Fila : Ejemplos

– **Ejemplo:** Si se actualiza el código de una Pieza, actualizar el código de los Suministros en los que se usaba:

```
-- Actualizar Suministros.P# si cambia Pieza.P#:
CREATE OR REPLACE TRIGGER Actualiza_SuministrosP#
  BEFORE UPDATE OF P# ON Pieza FOR EACH ROW
BEGIN
  UPDATE Suministros SET P# = :new.P# WHERE P# = :old.P#;
END Actualiza_SuministrosP#;
```

– **Ejemplo:** Controlar que la tabla de empleados no sufre ninguna variación durante los fines de semana ni fuera del horario laboral:

```
-- Generar Error si es fuera del horario laboral
CREATE TRIGGER horario_de_cambios
  BEFORE  DELETE OR INSERT OR UPDATE  ON empleados
BEGIN /* Si es sábado o domingo devuelve un error.*/
  IF (TO_CHAR(SYSDATE,'DY')='SAT' OR TO_CHAR(SYSDATE,'DY')='SUN')
  THEN raise_application_error( -20501,
    'No cambiar datos de empleados durante el fin de semana');
  END IF;
  /* Si hora es anterior a las 8:00 o posterior a las 18:00 */
  IF (TO_CHAR(SYSDATE,'HH24')<8 OR TO_CHAR(SYSDATE,'HH24')>=18)
  THEN raise_application_error( -20502,
    'No modificar estos datos fuera del horario de la empresa');
  END IF;
END horario_de_cambios;
```

RAISE_APPLICATION_ERROR permite que un programa PL/SQL pueda Generar Errores tal y como lo hace Oracle: El primer argumento es un número entre –20.000 y –20.999. El segundo es el mensaje.

Disparadores a Nivel de Fila: **WHEN**

- **Formato: ... FOR EACH ROW WHEN <Condición_Disparo>**
  - Sólo es válida en Disparadores **a Nivel de Fila** y siempre es **opcional**.
  - Si existe, el cuerpo del disparador se ejecutará **sólo para las filas que cumplan la Condición de Disparo** especificada.
  - En la **Condición de Disparo** pueden usarse los Pseudo-Registros **:old** y **:new,** pero en ese caso **no se escribirán los dos puntos** (**:**), los cuales son obligatorios en el cuerpo del disparador.
  - **Ejemplo:** Deseamos que los precios grandes no tengan más de 1 decimal. Si tiene 2 ó más decimales, redondearemos ese precio:

```
-- Si el Precio>200, redondearlos a un decimal:
CREATE OR REPLACE TRIGGER Redondea_Precios_Grandes
  BEFORE INSERT OR UPDATE OF Precio ON Pieza
  FOR EACH ROW WHEN (new.Precio > 200)
BEGIN
  :new.Precio := ROUND(:new.Precio,1);
END Redondea_Precios_Grandes;
```

  - Se puede escribir ese disparador sin la cláusula **WHEN**, usando un **IF:**

```
... BEGIN
      IF :new.Precio > 200 THEN
         :new.Precio := ROUND(:new.Precio,1);
...   END IF;
```

Predicados INSERTING, DELETING, *y* UPDATING

- En los disparadores que se ejecutan cuando ocurren **diversas Operaciones DML (INSERT, DELETE o UPDATE),** pueden usarse 3 **Predicados Booleanos** para conocer la operación disparadora**:**
  - **INSERTING** Vale **TRUE** si la orden de disparo es **INSERT**.
  - **DELETING** Vale **TRUE** si la orden de disparo es **DELETE**.
  - **UPDATING** Vale **TRUE** si la orden de disparo es **UPDATE**.
- **Ejemplo:** Guardar en una tabla de control la fecha, el usuario que modificó la tabla `Empleados` y el **Tipo de Operación** con la que modificó la tabla. Usando los Pseudo-Registros **:old** y **:new** también se pueden registrar los valores antiguos y los nuevos (si procede)**:**

```
CREATE OR REPLACE TRIGGER Control_Empleados
  AFTER INSERT OR DELETE OR UPDATE ON Empleados
BEGIN
  IF INSERTING THEN
     INSERT INTO Ctrl_Empleados (Tabla,Usuario,Fecha,Oper)
       VALUES ('Empleados', USER, SYSDATE,'INSERT');
  ELSIF DELETING THEN
     INSERT INTO Ctrl_Empleados (Tabla,Usuario,Fecha,Oper)
       VALUES ('Empleados', USER, SYSDATE,'DELETE');
  ELSE
     INSERT INTO Ctrl_Empleados (Tabla,Usuario,Fecha,Oper)
       VALUES ('Empleados', USER, SYSDATE,'UPDATE');
END Control_Empleados;
```


Disparadores de Sustitución: INSTEAD OF

- **Disparadores de Sustitución:** Disparador que se ejecuta en lugar de la orden DML (ni antes ni después, sino sustituyéndola).
  - Cuando se intenta **modificar una vista** esta modificación puede no ser posible debido al formato de esa vista.
  - **Características:**
    - **Sólo pueden definirse sobre Vistas.**
    - **Se activan en lugar de la Orden DML** que provoca el disparo, o sea, la orden disparadora no se ejecutará nunca.
    - **Deben tener Nivel de Filas.**
    - Se declaran usando **INSTEAD OF** en vez de **BEFORE/AFTER**.
- **Ejemplo:** Supongamos la siguiente vista**:**

```
CREATE VIEW Totales_por_Suministrador AS
  SELECT S#, MAX(Precio) Mayor, MIN(Precio) Menor
  FROM Suministros SP, Pieza P
  WHERE SP.P# = P.P#  GROUP BY S#;
```

  - Disparador que borre un suministrador si se borra una tupla sobre esa vista:

```
CREATE OR REPLACE TRIGGER Borrar_en_Totales_por_S#
  INSTEAD OF DELETE ON Totales_por_Suministrador
  FOR EACH ROW
BEGIN
  DELETE FROM Suministrador WHERE S# = :old.S#;
END Borrar_en_Totales_por_S#;
```

Disparadores: Observaciones

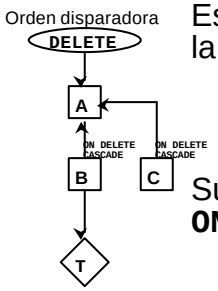
- Un *trigger* puede contener cualquier orden legal en un bloque PL/SQL, con las siguientes **Restricciones:**
  - **No puede emitir órdenes de Control de Transacciones: `COMMIT`, `ROLLBACK` o `SAVEPOINT`.**
    - El disparador se activa como parte de la orden que provocó el disparo y, por tanto, forma parte de la misma transacción. Cuando esa transacción es confirmada o cancelada, se confirma o cancela también el trabajo realizado por el disparador.
  - Cualquier **Subprograma** llamado por el disparador **tampoco puede emitir órdenes de control de transacciones.**
  - No puede ni declarar variables ni hacer referencia a columnas de tipo **`LONG`** o **`LONG RAW`** de la tabla sobre la que se define el disparador.
  - **Un disparador puede tener Restringido el acceso a ciertas tablas.**
    - Dependiendo del tipo de disparador y de las restricciones que afecten a las tablas, dichas tablas pueden ser **mutantes**.
- **Diccionario de Datos:** Todos los datos de un **`TRIGGER`** están almacenados en la vista **`USER_TRIGGERS`**, en columnas como **`OWNER`**, **`TRIGGER_NAME`**, **`TRIGGER_TYPE`**, **`TABLE_NAME`**, **`TRIGGER_BODY`**...
- **Borrar un Disparador: `DROP TRIGGER <NombreT>;`**
- **Habilitar/Deshabilitar un Disparador,** sin necesidad de borrarlo:
  `ALTER TRIGGER <NombreT> {ENABLE | DISABLE};`
  - Esto no puede hacerse con los subprogramas.

Disparadores: Tablas Mutantes

- **Tabla de Restricción o Tabla Padre** (*constraining table, Parent*)**:** Tabla a la que referencia una llave externa en una **Tabla Hijo** (*Child*), por una Restricción de Integridad Referencial.
- **Tabla Mutante** (*mutating table*)**:** Una tabla es mutante**:**
  - **1. Si está modificándose "actualmente" por una orden DML (`INSERT`, `DELETE` o `UPDATE`).**
  - **2. Si está siendo leída por Oracle para forzar el cumplimiento de una restricción de integridad referencial.**
  - **3. Si está siendo actualizada por Oracle para cumplir una restricción con `ON DELETE CASCADE`.**
    - **Ejemplo:** Si la cláusula **`ON DELETE CASCADE`** aparece en la tabla **Suministros**, borrar una **Pieza** implica borrar todos sus **Suministros.**
      - **Pieza** (y Suministrador) es una Tabla de Restricción de **Suministros**.
      - Al borrar una pieza ambas tablas serán mutantes, si es necesario borrar suministros. Si se borra una pieza sin suministros sólo será mutante la tabla **Pieza**.
- **Un Disparador de Fila se define sobre una Tabla Mutante** (casi siempre)**.**
  - En cambio, un **Disparador de Orden NO** (casi nunca)**:** El disparador de Orden se ejecuta antes o después de la orden, pero no a la vez.

Disparadores: Tablas Mutantes

- **Las <u>Órdenes SQL</u> en el cuerpo de un disparador tienen las siguientes 2 <u>Restricciones</u> por las que <u>NO PUEDEN</u>:**
  - **1. Leer o Modificar ninguna Tabla Mutante de la orden que provoca el disparo.**



Disparador de Fila T: No puede acceder a las tablas **A**, **B** y **C**, con esa orden disparadora.

Esto incluye lógicamente la tabla de dicha orden y la tabla del disparador, que pueden ser distintas.

Ej.: Borramos una tupla de una tabla **A**. Esto implica borrar tuplas de una tabla **B** (que tiene una restricción **`ON DELETE CASCADE`** sobre **A**). Si este segundo borrado sobre **B** dispara algún disparador **T**, entonces, ese disparador no podrá ni leer ni modificar las tablas **A** y **B** pues son **mutantes**. Si lo hace se producirá un error.

Sus tablas de restricción afectadas por la cláusula **`ON DELETE CASCADE`**, también son mutantes.

Ej.: Si borrar en **A** implica borrar en una tercera tabla **C**, el disparador sobre **B** no podrá tampoco acceder a **C**, si ese disparador se activa por borrar sobre **A** (aunque en **C** no se borre nada). Sí podrá acceder a **C** si se activa por borrar directamente sobre **B**. También podría acceder a **C** si ésta no tuviera el **`ON DELETE CASCADE`** pero hay que tener en cuenta que NO se podrán borrar valores en **A**, si están siendo referenciados en **C** (ORA-2292).

  - **2. Modificar las columnas de clave primaria, única o externa de una Tabla de Restricción a la que la tabla del disparador hacen referencia:** Sí pueden modificarse las otras columnas.

¿Qué Tablas son Mutantes?

- Al escribir disparadores es importante responder a **<u>Dos Preguntas:</u>**
  - **¿Qué Tablas son Mutantes?**
  - **¿Cuando esas Tablas Son mutantes?**

Para responder a esa pregunta es necesario conocer el tipo de orden **DML** que se está ejecutando.

- **<u>¿Qué Tablas son Mutantes</u>?**
  - **<u>1</u>.** Son mutantes las tablas afectadas por una operación **`INSERT`**, **`DELETE`** o **`UPDATE`**.
  - **<u>2</u>.** Si una **tabla Hijo** (p.e. `Empleados`) tiene un atributo llave externa (`Dpto`) a otra **tabla Padre** (`Departamentos`), **ambas** tablas serán mutantes si:
    - **Insertamos** (**`INSERT`**) en la tabla **Hijo:** Comprobar valores en la tabla padre.
    - **Borramos** (**`DELETE`**) de la tabla **Padre:** Impedir que tuplas hijas se queden sin padre.
    - **Actualizamos** (**`UPDATE`**) la tabla **Padre** o la tabla **Hijo:** Las 2 operaciones anteriores.
  - **<u>3</u>.** Si existe la restricción **`ON DELETE CASCADE`**, esta implica que si se borra de la **tabla Padre**, se borrarán las tuplas relacionadas en la **tabla Hijo** y, a su vez, pueden borrarse tuplas de tablas hijos de esa **tabla Hijo**, y así sucesivamente. En ese caso todas esas tablas serán mutantes.
    - En disparadores activados por un **`DELETE`**, es importante tener en cuenta si pueden ser activados por un borrado en cascada y, en ese caso no es posible acceder a todas esas tablas mutantes.

¿Cuándo son las Tablas Mutantes?

- **Las 2 Restricciones anteriores en las órdenes SQL de un Disparador se aplican a:**
  - **Todos los Disparadores con Nivel de Fila.**
    - **Excepción:** Cuando la orden disparadora es un **`INSERT`** que afecta a una única fila, entonces esa tabla disparadora no es considerada como mutante en el **Disparador de Fila-Anterior**.
      - Con las órdenes del tipo **`INSERT INTO Tabla SELECT...`** la tabla del disparador será mutante en ambos tipos de disparadores de Fila si se insertan varias filas.
      - Este es el único caso en el que un disparador de Fila puede leer o modificar la tabla del disparador.
  - **Los Disparadores a Nivel de Orden cuando la orden de disparo se activa como resultado de una operación `ON DELETE CASCADE` (al borrar tuplas en la tabla Padre).**
- **Los ERRORES por Tablas Mutantes se detectan y se generan en Tiempo de Ejecución y no de Compilación** (ORA-4091)**.**

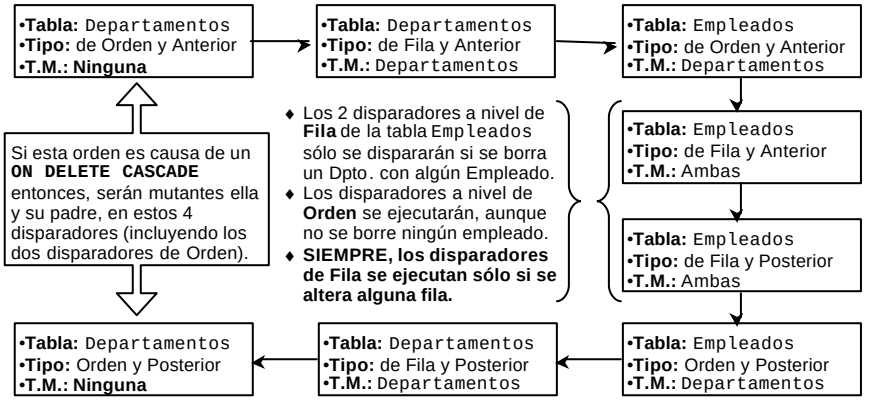
Tablas Mutantes: Disparador de Ejemplo

- **Disparador** que modifique el número de empleados de un departamento (columna `Departamentos.Num_Emp`) cada vez que sea necesario.
  - Ese número cambia al **INSERTAR** o **BORRAR** uno o más empleados, y al **MODIFICAR** la columna `Dpto` de la tabla `Empleados`, para uno o varios empleados.
  - La tabla `Departamentos` es una tabla de restricción de la tabla `Empleados`, pero el Disparador es correcto, porque modifica `Num_Emp`, que no es la llave primaria.
  - Este disparador no puede consultar la tabla `Empleados`, ya que esa tabla es mutante: `SELECT COUNT(*) INTO T FROM Empleados WHERE Dpto = :new.Dpto;`

```
CREATE OR REPLACE TRIGGER Cuenta_Empleados
  BEFORE DELETE OR INSERT OR UPDATE OF Dpto ON Empleados
  FOR EACH ROW
BEGIN
  IF INSERTING THEN
     UPDATE Departamentos SET Num_Emp = Num_Emp+1
       WHERE NumDpto=:new.Dpto;
  ELSIF UPDATING THEN
     UPDATE Departamentos SET Num_Emp = Num_Emp+1
       WHERE NumDpto=:new.Dpto;
     UPDATE Departamentos SET Num_Emp = Num_Emp-1
       WHERE NumDpto=:old.Dpto;
  ELSE
     UPDATE Departamentos SET Num_Emp = Num_Emp-1
       WHERE NumDpto=:old.Dpto;
  END IF;
END;
```

Tablas Mutantes: Ejemplo Esquemático

- Sea una **tabla Padre** `Departamentos`, y una **Hijo** `Empleados` cuyo atributo llave externa es `Dpto` con la restricción **`ON DELETE CASCADE`** que fuerza a borrar todos los empleados de un departamento si su departamento es borrado.
- Supongamos que para la orden **`DELETE`** están implementados los 4 tipos de disparadores posibles (de fila y de órden, anterior y posterior) para las dos tablas.
- Si se ejecuta una orden **`DELETE`** sobre la **tabla Padre** `Departamentos`, se ejecutarán los siguientes disparadores, con las tablas mutantes indicadas, en este orden:



Tablas Mutantes: Esquema Padre/Hijo

Órdenes sobre `Departamentos` (Tabla Padre):

`DELETE`:

Los 4 disparadores de la tabla Hijo `Empleados` sólo se dispararán si se borra un Departamento que tenga algún Empleado.

Si esta orden sobre el Padre es causa de un **`ON DELETE CASCADE`** sobre una tabla **X** (padre del padre) entonces, serán mutantes ella y su padre **X**, en sus 4 disparadores. Ambas tablas también serán mutantes en los 4 disparadores de la tabla Hijo `Empleados`.

`INSERT`: Su tabla Hijo no se ve afectada: No se disparan sus disparadores.

`UPDATE`: Su tabla Hijo no se ve afectada, porque sólo se permiten actualizar valores no referenciados en sus tablas Hijos (ORA-2292).

Órdenes sobre `Empleados` (Tabla Hijo): No se dispara ningún disparador del Padre.

`DELETE`: No afecta a la tabla Padre y sólo es mutante la Hijo.

`INSERT`:

- **Insertar una fila:**
  Las tabla Hijo **no** es mutante en el disparador de **Fila-Anterior** (a pesar de ser la tabla del disparador) y **sí** lo es en el de **Fila-Posterior**.
  La tabla Padre es mutante sólo si se intenta modificar el atributo al que hace referencia la tabla Hijo y sólo en el disparador de **Fila-Posterior**, ya que durante la ejecución de disparador de Fila-Anterior aún no está mutando ninguna tabla.
- **Insertar varias filas:** Las tablas Padre e Hijo son mutantes en los dos disparadores de **Fila**, pero la tabla Padre sólo si se modifica el atributo al que hace referencia la tabla Hijo.

`UPDATE`: Las tablas Padre e Hijo son mutantes en los dos disparadores de **Fila**, pero la tabla Padre sólo si se modifica el atributo al que hace referencia la tabla Hijo.

Tablas Mutantes: Solución

- El disparador anterior `Cuenta_Empleados` tiene dos inconvenientes:
  - Modifica el campo `Num_Emp` aunque este no tenga el valor correcto.
  - Si se modifica directamente ese campo, quedará incorrecto siempre.
- **Solución al Problema de la Tabla Mutante:**
  - Las tablas mutantes surgen básicamente en los disparadores a nivel de Fila. Como en estos no puede accederse a las tablas mutantes, la **Solución es Crear Dos Disparadores:**
    - **A Nivel de Fila:** En este guardamos los valores importantes en la operación, pero no accedemos a tablas mutantes.
      - Estos valores pueden guardarse en:
        - » **Tablas de la BD** especialmente creadas para esta operación.
        - » **Variables o tablas PL/SQL de un paquete:** Como cada sesión obtiene su propia instancia de estas variables no tendremos que preocuparnos de si hay actualizaciones simultáneas en distintas sesiones.
    - **A Nivel de Orden Posterior (`AFTER`):** Utiliza los valores guardados en el disparador a Nivel de Fila para acceder a las tablas que ya no son mutantes.
      - Se usará un bucle que examine uno a uno todos los valores guardados en el trigger a nivel de fila:
        - » Si se ha usado una tabla se deberá usar un cursor.

Tratamiento de Errores: Excepciones

- Todo **buen programa** debe ser capaz de **gestionar TODOS los posibles ERRORES** que puedan surgir durante su ejecución.
  - **PL/SQL** permite hacer una gestión de **errores** basada en **excepciones**, mecanismo heredado del lenguaje Ada.
  - Cuando se produce un **error**, se *genera* una **excepción** y el control del programa pasa al gestor de excepciones (sección **`EXCEPTION`**).
    - Esto permite tratar los errores independientemente, sin oscurecer la lógica del programa y hace que sea más fácil tratar todos los errores.
    - En los lenguajes sin excepciones (como C) es necesario controlarlos explícitamente poniendo condiciones de selección (`if`) antes o después de todas las operaciones que puedan producir errores.
    - **Sección `EXCEPTION`:** **`WHEN <Excepción> THEN <Gestor>`**
      - Pueden incluirse cuantas cláusulas **`WHEN`** se desee y enlazarlas con **`OR`**.
- **Tipos de Excepciones:** Dos tipos
  - **Predefinidas:** Son las más comunes y no es necesario definirlas.
  - **Definidas por el Usuario:** Permiten definir cualquier tipo de error, como por ejemplo un error relacionado con ciertos datos no válidos. Se declaran en la **sección declarativa**:
    - **Ejemplo declaración:**
      ```
      DECLARE
        MuchosEmpleados EXCEPTION;
      ```
    - **Ejemplo generación:** `RAISE MuchosEmpleados;`

Excepciones Predefinidas

- Hay **20 Excepciones Predefinidas** que controlan errores particulares (excepto **OTHERS** que controla cualquier tipo de error). Algunas son**:**
  - **INVALID_CURSOR:** Se genera al intentar efectuar una operación ilegal sobre un cursor, como cerrar o intentar extraer datos de un cursor no abierto.
  - **CURSOR_ALREADY_OPEN:** Surge al intentar abrir un cursor ya abierto.
  - **NO_DATA_FOUND:** Cuando una orden **SELECT..INTO** no devuelve ninguna fila o cuando se intenta referenciar un elemento de una tabla PL/SQL al que no se le ha asignado ningún valor previamente.
  - **TOO_MANY_ROWS:** Si una orden **SELECT..INTO** devuelve más de una fila.
  - **INVALID_NUMBER:** Si falla la conversión de cierto valor a un tipo **NUMBER** o cuando usamos un dato no numérico en lugar de un dato numérico.
  - **VALUE_ERROR:** Se genera cada vez que se produce un error aritmético, de conversión, de truncamiento o de restricciones en una orden procedimental (si es una orden SQL se produce la excepción **INVALID_NUMBER**). Ej.: Si asignamos una cadena o número de mayor longitud que el tipo de la variable receptora.
  - **STORAGE_ERROR y PROGRAM_ERROR:** Son errores internos que no deberían producirse. Ocurren respectivamente si PL/SQL se queda sin memoria o por un fallo en el motor PL/SQL de Oracle y debería avisarse del error al departamento de soporte técnico de Oracle.
  - **DUP_VAL_ON_INDEX:** Es el error ORA-1, que se genera cuando se intenta insertar una fila en una tabla con un atributo **UNIQUE** y el valor de ese campo en la fila que se intenta insertar ya existe.
  - **ZERO_DIVIDE:** Surge al intentar dividir por cero.

Generación de Excepciones

- Cuando se **Genera una Excepción** el control pasa a la sección **EXCEPTION** del bloque PL/SQL y resulta imposible devolver el control a la sección principal del bloque.
  - Las **Excepciones Predefinidas** se generan automáticamente cuando ocurre el error Oracle asociado a cada una.
  - Las **Excepciones Definidas por el Usuario** hay que generarlas explícitamente mediante la orden**: RAISE <Excepción>;**
- **Ej.:**

```
DECLARE
   Demasiados_Empleados EXCEPTION;
   Num_Empleados        NUMBER(3);
   Depart               Empleados.Dpto%TYPE;
BEGIN
   SELECT Dpto INTO Depart FROM Empleados WHERE DNI=99;
   SELECT COUNT(*) INTO Num_Empleados FROM Empleados
          WHERE Dpto = Depart;
   IF Num_Empleados > 20 THEN
      RAISE Demasiados_Empleados; -- Generar Excepción
   END IF;
EXCEPTION
   WHEN Demasiados_Empleados THEN
    INSERT INTO Tabla_Avisos(msg) VALUES ('Hay ' ||
      Num_Empleados||' empleados en el Dpto. '||Depart);
   WHEN NO_DATA_FOUND THEN
    INSERT INTO Tabla_Avisos(msg) VALUES
    ('No existe el empleado con DNI 99');
END;
```

El Gestor de Excepciones OTHERS

- **El Gestor OTHERS se ejecuta (si existe) para todas las excepciones que se produzcan:**
  - Debe ser siempre **el último gestor de excepciones**, ya que tras él no se ejecutará ninguno.
  - Es una buena práctica definir el gestor **OTHERS en el bloque más externo**, para asegurarse de que ningún error queda sin detectar.
  - **Funciones para Determinar el Error** (no pueden usarse en órdenes SQL)**:**
    - **SQLCODE:** Devuelve el **código del error** actual.
    - **SQLERRM:** Devuelve el **mensaje de error** actual (máxima longitud: 512 caracteres). Admite opcionalmente un código como argumento.
- **Ej.:**

```
DECLARE ...
  Codigo_Error NUMBER;
  Msg_Error    VARCHAR2(200);
BEGIN
  ...
EXCEPTION
  WHEN ... THEN
    ...
  WHEN OTHERS THEN
    Codigo_Error := SQLCODE;
    Msg_Error := SUBSTR(SQLERRM,1,200);
    INSERT INTO Tabla_Avisos (Cod_Error, Msg) VALUES
      (Codigo_Error, Msg_Error);
END;
```

- Si la asignación es superior al espacio reservado para la variable `Msg_Error` se generará la excepción **VALUE_ERROR**.
- También hay que asegurarse que el valor insertado en `Tabla_Avisos` no excede de la longitud máxima.

Localización de Errores

- Cuando en un mismo bloque PL/SQL puede generarse una misma excepción en distintas órdenes, ¿Cómo detectar cual ha sido?
  - **Ejemplo:**

```
BEGIN
    SELECT ...
    SELECT ...
    ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    -- ¿Qué orden SELECT generó el error?
END;
```

  - **Dos Soluciones:** Usar una variable o poner cada orden en su propio bloque.

```
DECLARE
  Contador NUMBER(1) := 1;
BEGIN
  SELECT ...
  Contador:=2;
  SELECT ...
...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    IF Contador=1 THEN
      ...
    ELSE
      ...
    END IF;
END;
```

```
BEGIN
  BEGIN
    SELECT ...
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      ...
  END;
  BEGIN
    SELECT ...
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      ...
  END;
  ...
END;
```

Métodos de Prueba y Depuración

- **Prueba:** Cualquier programa (incluidos los bloques PL/SQL) debe **Probarse** exhaustivamente para asegurar su correcto funcionamiento.
  - Las **Pruebas** se hacen en un entorno especial de **Desarrollo** y no de **Producción**, que debe ser suficientemente simple.
- **Depuración:** Si existen **Errores:**
  - **Localizar** dónde está el error y **Definir** qué está mal y porqué está fallando.
  - **Dividir** el programa en partes y **Probar** cada una de ellas, empezando por casos de prueba sencillos.
- **Técnicas de Depuración:**
  - **Insertar** los valores intermedios de las variables en una tabla temporal, que se consultará cuando el programa termine.
    - Existe un paquete llamado **DEBUG** que facilita esta tarea. Tiene dos procedimientos:
      - **`DEBUG.Debug(A,B)`** → **Inserta** en la tabla **`Debug_Table`** esos dos valores de tipo **`VARCHAR2`** en una fila (usualmente **`A`** es el nombre de una variable y/o algún comentario sobre el lugar donde se utiliza y **`B`** es el valor de esa variable).
      - **`DEBUG.Reset`** → Inicializa la depuración (borra las filas de **`Debug_Table`**).
  - **Visualizar** los valores intermedios de las variables, conforme el programa se va ejecutando: Paquete **`DBMS_OUTPUT`** y sus procedimientos **`PUT`**, **`PUT_LINE`**…
    - Esos procedimientos insertan sus argumentos en un buffer interno.
    - En SQL*Plus puede activarse para mostrar el buffer: **`set serveroutput on [size n]`**, donde **`n`** Î **[2000,1000000]** es el número de bytes máximo (**`2000`** por defecto).
    - PL/SQL es un lenguaje para gestionar BD, por lo que la E/S es limitada.
  - **Depuradores PL/SQL:** Existen varios programas *debuggers* para rastrear un programa PL/SQL, como son **Oracle Procedure Builder** y **SQL-Station**.