Diseño de Base de Datos

Conceptos prácticos

Universitat d'Alacant
Universidad de Alicante

Resumen

Veremos los tipos de datos y las funciones asociadas a estos, las distintas sentencias que se pueden usar con ejemplo generales de implementacion, creacion y manipulaciones de tablas, vistas y tablas temporales, ademas de la implementaciones de procedimientos y otro tipo de funciones.

La idea es tener acceso rapido a todos los conceptos mediante una tabla de conceptos que se mostrara a continuacion.

PD: habran fallos ortograficos debido a problemas con el interprete que utilizo para los apuntes.

Eduardo Espuch

Curso 2019-2020

Índice

1. Tipos de datos **2**
1.1. Datos simples . . . 2
1.2. Datos complejos . . . 2
1.2.1. Tipo de dato Date . . . 2

2. Sentencias básicas sobre una BD **3**
2.1. Sentencia SELECT . . . 3
2.1.1. Tipo join . . . 4

3. Creacion y manipulacion de tablas **6**
3.1. Estructura de una tabla . . . 7
3.1.1. Create . . . 7
3.1.2. Drop . . . 9
3.1.3. Alter . . . 9
3.2. Contenido de una tabla . . . 10
3.2.1. Insert . . . 10
3.2.2. Update . . . 10
3.2.3. Delete . . . 10

4. Introduccion a PL/SQL **11**
4.1. Uso del Bloque SQL . . . 11
4.2. Mensajes . . . 11
4.3. Parte declarativa . . . 12
4.3.1. Ampliacion sobre cursores . . . 12
4.3.2. Estructuras de control . . . 13

5. Procedimienos en PL/SQL **14**

6. Funciones en PL/SQL **14**

7. Disparadores en PL/SQL **15**

8. Excepciones en PL/SQL **16**
8.1. Predefinidas . . . 17
8.2. Definidas por el usuario . . . 17

9. Indices y transacciones en Oracle **18**
9.1. Sentencias para Indices . . . 18
9.2. Sentencias para Transacciones . . . 18

10. Seguridad en Oracle **19**
10.1. Vistas . . . 19
10.2. Privilegios . . . 19

11. De modelo relacional a EER, por Kenny **19**

1. Tipos de datos

Existen una gran cantidad de datos, veamos algunos:

1.1. Datos simples

- Char(n): alfanumerico que almacena cadenas de caracteres de longitud fija (n∈[1,2000] bytes), donde se reservara espacio para los n bytes.
- Varchar2(n): alfanumerico que almacena cadenas de carateres de longitud variable (n∈[1,4000] bytes), donde unicamente consumira los bytes que necesita para la cadena sin exceder el limite indicado (n)
- Number(p,s): numerico donde el parametro p indica el numero de digitos y el parametro s los decimales como maximo que se usaran para los valores introducidos. Si el parametro s es negativo, lo que hara es redondear los s primeros digitos, de tal manera que number(3,-2) redondearia a centenas.

Los alfanumericos deben de ir encerrados entre comillas simples.

1.2. Datos complejos

Lo mas complejo que veremos, o usaremos, es el tipo de dato DATE, que representa una fecha. Veremos que tiene diversas signaturas ademas de funciones definidas. Los datos DATE deben de ir encerrados entre comillas simples.

1.2.1. Tipo de dato Date

Los tipo DATE representa una fecha, y disponemos diversos formatos para representarlos:

textbf/'.;:'*texto*' Marcas de puntuacion y texto que se reproduce en el resultado			
Formatos para representar dias			
D	De la semana(1-7)	DAY	Nombre del dia
DD	Del mes(1-31)	DDD	Del año(1-366)
DY		Nombre reducido del dia	
Formatos para representar meses			
MM	Mes(1-12)	MON	Nombre abreviado del mes
MONTH		Nombre completo del mes	
Q		Trimestre del año(1-4)	
Formatos para representar años			
YYYY	Año con 4 digitos	Y,YYY	Año con punto millar
YY		Año con 2 digitos	

Estos formatos seran utilizadas en diversas funciones asociadas, siendo algunas de las que usaremos las siguientes:

- TO_CHAR(*fecha*[,formato]): convierte DATE a VARCHAR2 [en el formato especificado]
- TO_DATE(*cadena*[,formato]): convierte CHAR a DATE [en el formato especificado]
- SYSDATE: Devuelve la fecha actual del sistema
- ADD_MONTHS(*fecha*,*n*); devuelve la fecha con n meses mas

- MONTHS_BETWEEN(*fecha1*,*fecha2*): diferencia de meses entre las dos fechas.

Destacar que, para realizar comparaciones entre fechas directamente, se recomienda usar el formato en cadena de caracteres.

Las marcas de puntuacion definidas previamente, permiten concantenar distitnos formatos para obtener un formato de fecha especifico, por ejemplo, usando 'dd-mm-yy' obtendriamos una fecha del tipo '30-12-20'.

2. Sentencias básicas sobre una BD

Resumen de lo dado en FBD

2.1. Sentencia SELECT

Las sentencias SELECT vamos a partirlas en los distintos entornos o clausulas con las que poder definirla. Veamos la estructura principal:

1	**SELECT** [DISTINCT] listaColumnas [apodo]
2	**FROM** listaTablas [apodo]
3	[**WHERE** clausula condicional]
4	[**GROUP BY** listaColumnas por las que se quiere agrupar]
5	[**HAVING** condicion para los grupos]
6	[**ORDER BY** listaColumnas [ASC / DESC]]

1. Clausula SELECT: indica que columnas se muestran en la consulta. Pueden mostrarse columnas de tablas (accediendo a los datos de estas), cadenas de caracteres estaticas y funciones. Consideraremos las siguientes caracteristicas:

   - Si se da que dos columnas de tablas distintas tienen el mismo nombre, hay que indicar de que tabla proviene (se puede usar el nombre completo de la tabla o el apodo que se le de en la clausula FROM).
   - Se puede modificar el nombre de la columna añadiendo un apodo entre .
   - Algunas funciones (count, sum,...) que se muestran junto a otras columnas, se requerira que se use la clausula GROUP BY. De esta forma la funcion afectara a las columnas que se han decidido agrupar. Distinguiremos las siguientes funciones:

count(*)	cuenta las filas
NVL(listaColumna,'expresion')	Si se obtiene un NULL, se mostrara la expresion
Requieren de usar la clausula GROUP BY si hay varias columnas en el SELECT:	
count([DISTINCT] listaColumna)	cuenta las filas con valor en listaColumna [distinto]
sum(listaColumna)	suma los valores en listaColumna
avg(listaColumna)	promedio de los valores en listaColumna
min/max(listaColumna)	minimo/maximo de los valores en la listaColumna

*Las funciones por si solas, actuan con un GROUP BY implicito
**NVL puede ser usada en otras clausulas

Se pueden concatenar dos consultas que tengan el mismo formatos de salidas(mismas columnas) usando el operador UNION, de esta manera obtendremos el conjunto de elementos en las dos tablas. Es muy poco frecuente su uso.

2. Clausula FROM: indica sobre que tablas acceder a la informacion. Se puede considerar el uso de tipos **join** que actuara como una clausula condicional. Mas adelante se veran los distintos join y como usarlos adecuadamente.

3. Clausula WHERE: sirve para filtrar las listas segun diferentes operaciones comparativas. Veremos que usa tipos **join** tambien, pero fijemonos en las diferentes operaciones comparativas que podemos usar:

a=b a igual b	a<>b a distinto b	a is [NOT] null a es [no] nulo
a<b a menor que b	a>b a mayor que b	
exp1 AND axp2	exp1 y exp2 deben de cumplirse a la vez	
exp1 OR axp2	exp1 o exp2 deben de cumplirse	
a [NOT] IN (b∈B)	a que [no] pertenezcan al conjunto de B (a y b =Dato)	
[NOT] EXISTS (A⇔B)	similar al IN, pero interpreta el SUBQUERY como un bool	
a >= ALL(b∈B)	a >= que TODO b∈B	
a >= ANY(b∈B)	cualquier a >= que b∈B (muestra por orden)	
a [NOT] LIKE XXX	a que [no] contenga o sea como la cadena XXX*	
a [NOT] BETWEEN V1 AND V2	a [no] contenido entre los valores V1 y V2	

*XXX viene entre ' ', si se pone % se espera una subcadena opcional y _ para un caracter

4. Clausula **GROUP BY**: cuando se quiere mostrar columnas agrupadas por valores. Como ya se ha comentado, suele ser usado principalmente junto a las funciones definidas en la clausula SELECT, pero no quiere decir que sean necesarias.

   Sera necesario que las columnas en la clausula SELECT se indiquen en el GROUP BY, en especial si son indices. En ocasiones, sera necesarios poner en el GROUP BY los indices(identificadores) de la tabla aunque no se muestren en la consulta, forzando a agrupar las distintas identidades.

   Recordad que las funciones definidas previamente, se podrian considerar añadidas implicitamente en esta clausula.

5. Clausula **HAVING BY**: se usa sobre los elementos en la clausula GROUP BY, indicando condiciones que los grupos deben de cumplir.

6. Clausula **ORDER BY**: se usa para ordenar las filas en funcion a una o varias columnas (la prioridad va de izquierda a derecha en la listaColumnas.

   Si se indica que sea ASC(ascendente) (por defecto si no se indica nada), el valor mayor estara en la parte mas baja mientras que con DESC (descendentes), el mayor sera el primero en la lista. Esto se debe a que las filas se muestran de arriba a abajo.

2.1.1. Tipo join

Veamos los siquientes join que nos permiten establecer condiciones a la hora de combinar tablas:

- **Equijoin**: igualdades usuales para concatenar filas de varias tablas. Se podran considerar otro tipo de operaciones comparativas(?)

```
SELECT a.*,b.*
FROM A a, B b
WHERE a.key=b.key
```

- **Selfjoin**: permite relacionar elementos de una misma tabla.

```
SELECT a0.*,a1.*
FROM A a0, A a1
WHERE a0.key<>a1.key
```

- **Antijoin**: muestra las filas de una tabla que no se encuentran en una subconsulta.

```
SELECT a.*
FROM A a
WHERE a.key NOT IN (consulta de valores del tipo a.key)
```

- **Semijoin**: permite relacionar elementos de varias tablas pero unicamente mostrando los de una tabla.

```
SELECT a.*
FROM A a, B b
WHERE a.key=b.key
```

Todas estas pueden concatenarse con el uso de condicionales AND o OR, o ampliarse con el resto de operaciones comparativas previamente definidas.

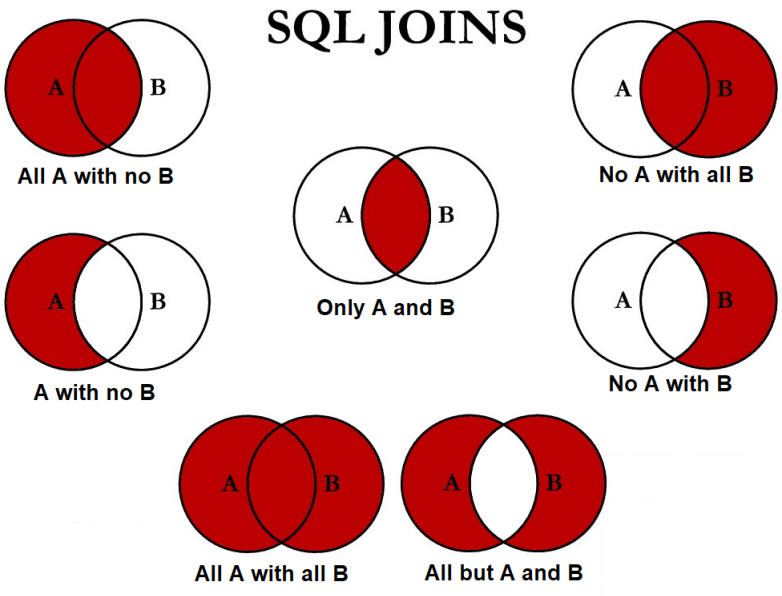
- **Inner join o Simple join**: actua como una forma de realizar una interseccion, los valores que se muestran son aquellos que se encuentran en ambas tablas relacionadas.

- **Outer join**: actua de tal manera que, segun el tipo de OUTER que sea, forzara que se muestren los valores de determinadas tablas aunque no coincida con la de la tabla relacionada.

  Si indicamos que RIGHT, la tabla a la derecha del from (y que interactua en el entorno del **ON**), se mostrara completa. Si es LEFT, entonces la tabla despues del **join** se mostrara entera. Si se indica FULL, entonces se mostraran ambas tablas (que estan relacionadas en el entorno **ON**

Recordad que estas estan definidas en la clausula **FROM**, como una condicion para las tablas que se definan en esta, y por lo tanto podemos indicar que haga la consulta sobre estas tablas.

El ejemplo practico a continuacion esta hecho de tal manera que se pueda copiar la sentencias teniendo como referencia grafica la imagen de conjuntos usada.



All A with no B	All B with no A	Only A and B
SELECT <select_list> FROM A LEFT JOIN B ON A.key=B.key	SELECT <select_list> FROM A RIGHT JOIN B ON A.key=B.key	SELECT <select_list> FROM A INNER JOIN B ON A.key=B.key

A with no B	B with no A	All A with All B
SELECT <select_list> FROM A LEFT JOIN B ON A.key=B.key WHERE B.key IS NULL	SELECT <select_list> FROM A RIGHT JOIN B ON A.key=B.key WHERE A.key IS NULL	SELECT <select_list> FROM A FULL OUTER JOIN B ON A.key=B.key

All but A and B
SELECT <select_list> FROM A FULL OUTER JOIN B ON A.key=B.key WHERE A.key IS NULL OR B.key IS NULL

*Esta ultima seria todo A y B salvo lo que sea A y B a la vez

3. Creacion y manipulacion de tablas

Hay que tener claro un par de cosas, entre ellas, la nomenclatura que se usara para especificar las restricciones o Constraints que se defininirar en la tabla. Veamoslo:

Nombre	Accion
PK_Tabla	Denota la clave primaria de la tabla indicada
UK_Tabla	Denota una clave unica de la tabla indicada
AK_Tabla	Denota una clave alternativa de la tabla indicada(UK+VNN)
FK_TActual_TReferencia	Denota la clave ajena en TActual que proviene de TReferencia
CK_Tabla_Columna	Denota a un Check respecto a una columna

*Si no las definimos con estos nombres, se les asignara uno automaticamente

**Las FK deben de referenciar a PK o AK de TReferencia

Ademas, SQLDeveloper nos permite hacer consultas sobre una BD interna que usa para controlar la informacion de las tablas que contiene, de esta forma, podemos acceder a informacion de estas sin tener que usar el entorno grafico, veamos algunos casos:

- Informacion sobre una tabla: usando **DESC Tabla** se obtiene informacion respecto a las columnas en esta.
- Obtener las tablas definidas por un usuario: considerando que en **user_tables** se encuentra un registro con informacion de todas las tablas, si hacemos una consulta sobre esta tabla, obtendremos informacion relevante sobre las tablas definidas.

  La columna **table_name** contiene el nombre dado a las tablas definidas. (Recomendable ya que hay muchas columnas en la tabla del usuario)

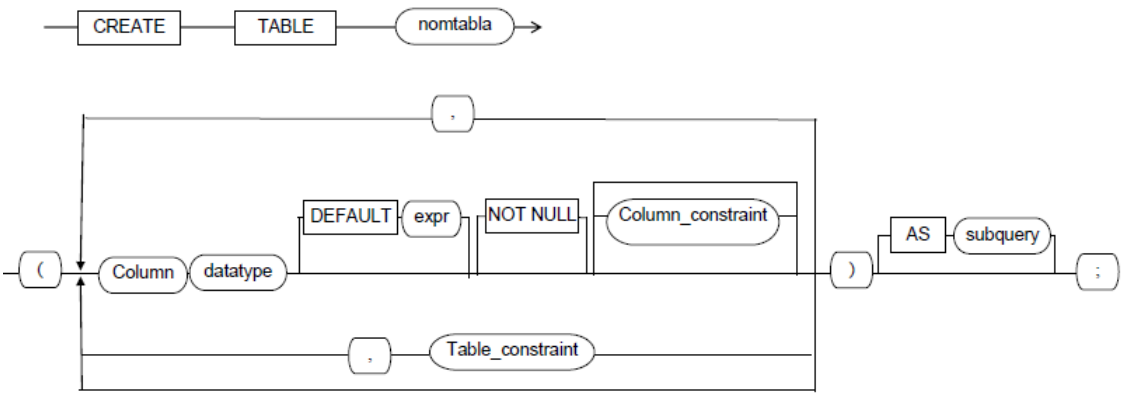
- Obtener informacion de las restricciones: de igual forma que con las tablas, existe otra tabla, **user_constraints**, que almacena las restricciones definidas en las tablas. Podremos buscar en esta por el nombre de la tabla (**table_name**), por el tipo de restriccion (**constraint_type**) o por el nombre de la restriccion que el usuario ha definido con la nomenclatura descrita arriba (**constraint_name**), buscando si contiene una determinada subcadena de caracteres.

3.1. Estructura de una tabla

El usuario puede crear y modificar una tabla, añadiendo columnas, restricciones y referencias a otras tablas, ademas de controladores sencillos (se veran que hay controladores mas complejos mas adelantes). Para poder definir la estructura de una tabla, veremos:

3.1.1. Create

La estructura principal que se usa para crear tablas es la siguiente:

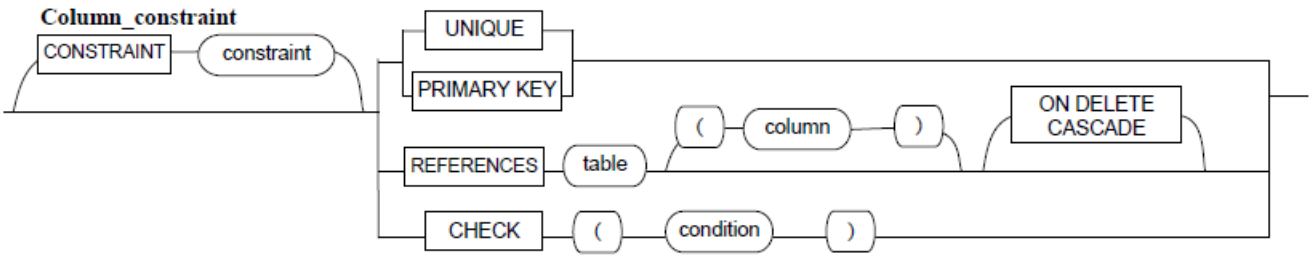


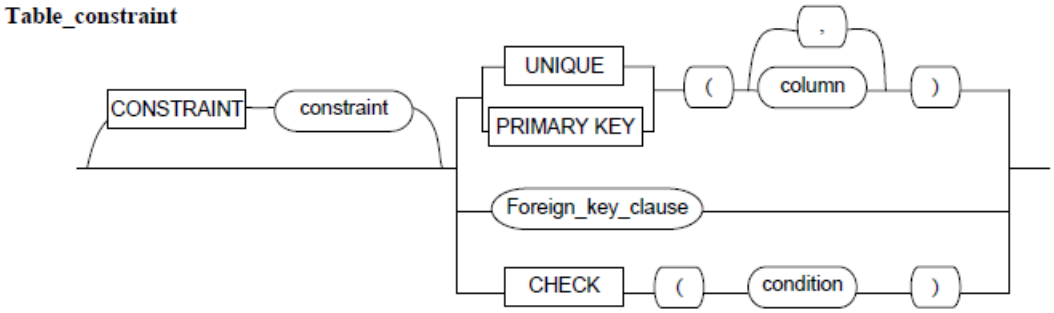
Donde tenemos que *nomTabla* denota el nombre de la tabla, *Column* al nombre de las tablas que sera del tipo especificado en *data_type*, *expr* sera un valor que se asignara por defecto (no usar en CCs), los parametros Contraints definirar restricciones sobre una columna o un conjunto de estas y, usando AS *subconsulta* podemos darle unos valores iniciales a la tabla (mientras coincida en formato con esta.

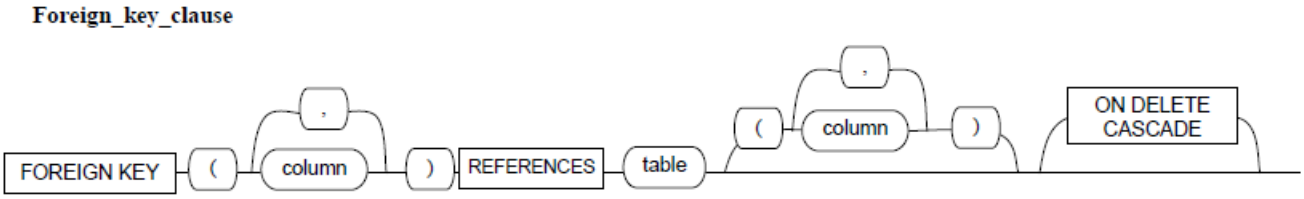
Se recomienda que, a la hora de realizar el script, se siga este esquema:

CREATE TABLE *nombreTabla*(
Columna1 y propiedades,
⋮
*Columna**n** y propiedades*,
Restriccion1 de la tabla,
⋮
*Restriccion**n** de la tabla*
);

Las constraint o restricciones las podemos ver a nivel de una columna (definiendos directamente como una propiedad de esta, o a nivel de la tabla, donde podra agrupar columnas. Veamos la estructura de estas y al acabar pongamos un ejemplo de cada rapidamente:







- Constraint del tipo Primary Key o Unique, sencillas se usar, solo es necesarios indicar el nombre de la restriccion en *constraint* y, en el caso de las de tabla, indicar con una lista de nombres las columnas que la componen.

  Las Unique junto con **NOT NULL** en las columnas que las componen, definen una Alternative Key.

- Constraint del tipo Foreing Key, descritas en la columna no requieren de usar la signatura del **Foreing Key**, pero descritas para la tabla, si. Hay que tener en cuenta que una tabla puede definir con otro nombre a una columna a pesar de referenciar a lo mismo.

  Las opciones de **On delete** y **Cascade** sirven para que, a la hora de que un valor se elimina en la tabla de referencia, interactue de cierta manera con los valores de la tabla actual. **On delete** pondra a NULL los valores a los que hacian referencia y, **Cascade**, eliminara las filas que contengan el valor que se ha eliminado en la tabla referenciada.

- Constraint del tipo Check, comunmente se usan sobre una columna, y por lo tanto no se vera usadas para la tabla en si. Permiten usar clausulas condicionales (vistas en la clausula Where) para filtar las insersiones que se realizan.

PK	**CONSTRAINT** PK_Tabla **PRIMARY KEY** [(columna1,...,columna$n$)
UK	**CONSTRAINT** UK_Tabla **UNIQUE** [(columna1,...,columna$n$)
AK*	**CONSTRAINT** AK_Tabla **UNIQUE** [(columna1,...,columna$n$)+columna$i$ NOT NULL
FK	[**FOREING KEY** (columna1,...,columna$n$)] **CONSTRAINT** FK_TAct_TRef **REFERENCES** *TRef* (TRef_columna1,...,TRef_columna$n$)
CK	**CONSTRAINT** CK_Tabla_Columna **CHECK (** *clausula condicion sobre Columna*)

3.1.2. Drop

Muy sencillo, elimina la tabla indicada, de tal manera que, si usamos:

DROP TABLE *nombreTabla*

La tabla *nombreTabla* se elimina de la base de datos. Es importante comprobar que otras tablas toman a esta como referencia ya que, al eliminarla, podemos afectar a la integridad referencial de la base de datos. Y de igual manera si eliminamos un conjuntos de tablas en serie.

3.1.3. Alter

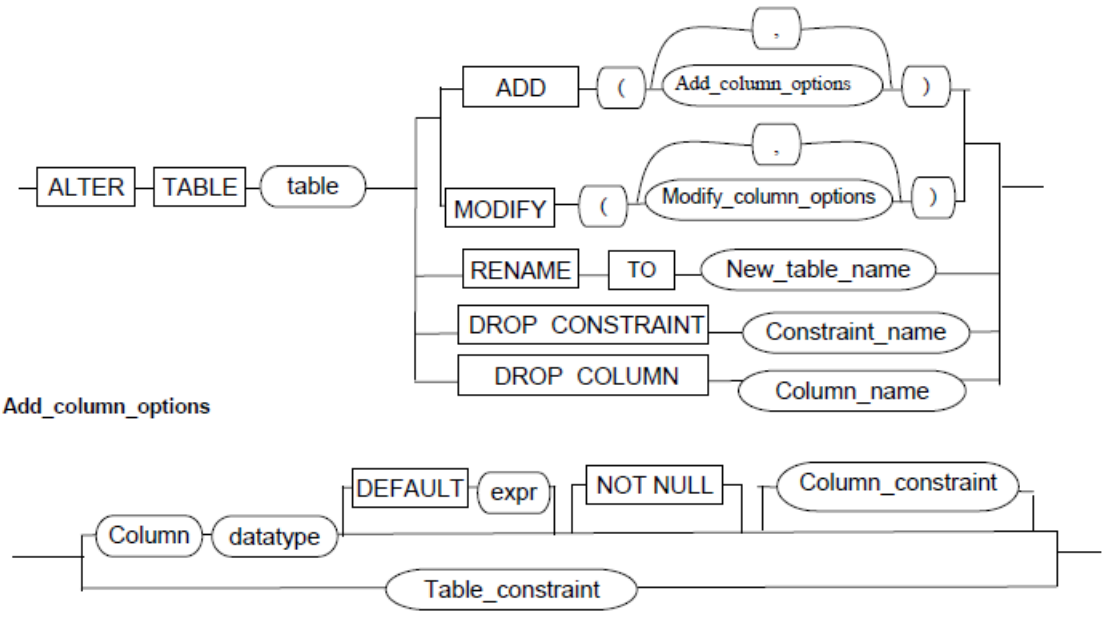
Usado para modificar tablas, veremos que es posible:

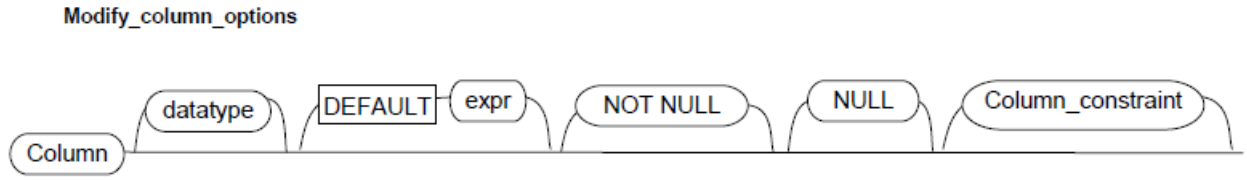
1. Añadir columnas o restricciones a la tablas usando **ADD**
2. Modificar columnas usando **MODIFY**
3. Renombrar la tabla usando **RENAME TO**
4. Eliminar una restriccion usando **DROP CONSTRAINT**
5. Eliminar una columna usando **DROP COLUMN**

Un ejemplo general del script seria:

ALTER TABLE *nombreTabla*	
1.a	**ADD** (*Columna1 y propiedades*,...,*Columna**n** y propiedades*);
1.b	**ADD** (*Restriccion1 de la tabla*,...,*Restriccion**n** de la tabla*);
2	**MODIFY** (*Columna1 y propiedades*,...,*Columna**n** y propiedades*);
3	**RENAME TO** *nuevoNombre*;
4	**DROP CONTRAINT** *nombreConstraint*;
5	**DROP COLUMN** *nombreColumna*;

La estructura usada para generar el script seria:





3.2. Contenido de una tabla

El contenido de una tabla veremos que se modificara haciendo uso se las siguientes instrucciones (considerar para un futuro que estas se podran usar tambien sobre vistas, que es un concepto que se vera al final)

3.2.1. Insert

Permite insertar un conjunto de valores en una tabla, este conjunto puede ser dado como resultado de una consulta o pueden darse de forma explicita usando **VALUES**.

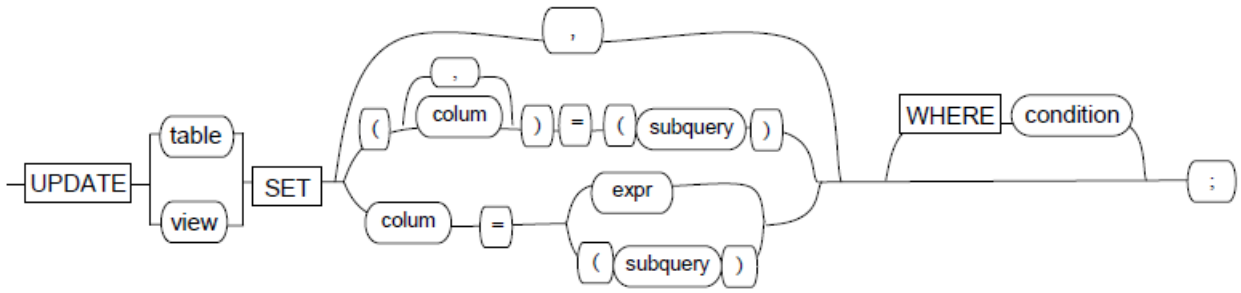
Puede darse el caso de que queramos introducir valores sobre un conjunto determinado de columnas, esto se puede hacer al denotarse tras indicar la tabla en la que realizar la insercion, de esta forma, veremos que el script generalizado sera:

INSERT INTO *nombreTabla[o vista]* (columna1,. . . ,columna$n$)]	
a	**VALUES** (valor1,. . . ,valor$n$[,. . . ,valor$m$]);
b	*lista resultado de una consulta*;

3.2.2. Update

Modifica las filas de una tablas, todas o aquellas que cumpla una condicion dada en la clausula WHERE. Unicamente se puede actualizar con un conjunto de valores una o varias filas a la vez, por lo tanto, aunque podamos usar subconsultas para asignar un valorm estas deben de devolver si o si una fila.

Con esto dado, vamos a ofrecer el esquema usado debido a que hacer un esquema general puede ser mas confuso de lo que ya se muestra en el esquema:



3.2.3. Delete

Elimina las filas de una tabla, todas o aquellas que cumplan una condicion dada en la clausula WHERE.

Tendremos que el script generalmente usado sera:

DELETE [FROM] *nombreTabla[o vista]* [**WHERE** clausula condicional];

4. Introduccion a PL/SQL

Sabemos hacer consultas y manipular tablas, vamos a tratar ahora con conceptos que nos permiten realizar funciones mas complejas e interactuar con el usuario de forma mas comoda.

4.1. Uso del Bloque SQL

Llamaremos Bloque SQL a una estructura formada por 3 partes:

- Declarativa: permite declarar y realizar una primera asignacion de valores a variables, contantes y cursores.
- Ejecutable: la unica parte obligatoria del bloque, con la que realizar diversas acciones. Podemos realizar ejecuciones en tiempo de ejecucion o compilarlas.

  Al compilarlas, veremos que definiremos procedimientos donde los parametros aportado pueden actuar como de entrada, salida o ambos, y funciones, donde al ejecutarlas se devolvera un valor en concreto. Es decir, usaremos procedimientos si no necesitamos que al realizar la llamada se devuelva un valor mientras que, con las funciones, la idea es que al usarse se obtendra algun resultado (de gran uso para las sentencias Select). Lo veremos mas adelante.
- Manejo de instrucciones: controlara las excepciones que se puedan generar y esten controladas.

```
[DECLARE
  –declaraciones]
BEGIN
  –sentencias a ejecutar
[EXCEPTION
  –manejo de excepciones]
END;
```

4.2. Mensajes

Lo mas importante, a la hora de trabajar con los dos tipos de mensajes que veremos, tendremos que ejecutar al principio de la sesion de trabajo el comando **SET SERVEROUPUT ON;** permitiendo la salida de informacion.

Distinguimos dos tipos de mensajes:

- **raise_application_error**(numero_error,mensaje), donde numero_error∈[-20999,-20000], interrumpira la ejecucion mostrando el mensaje indicado ya que genera un error.
- **dbms_output.put_line**(mensaje): deposita el mensaje en un buffer y prosigue la ejecucion.

Usaremos '——' para concatenar distintos tipos de datos y formar un mensaje que, aunque de formato String, se ha compuesto por deiversos tipos (recordad que alfanumericos entre comillas simples y podremos acceder a columnas de las tablas para obtener un valor determinado).

4.3. Parte declarativa

En la parte declaratica distinguiremos tres valores que podremos declarar:

- Constantes: valores inmutables y que se debe de asignar dicho valor una vez que sea declarado. La sintaxis es:

  nombreCons C**CONSTANT** *datatype***:=** valor;

- Variables: valores mutables que pueden inicializarse con un valor especifico en la parte declarativa o no, pudiendo ser modificado en la parte ejecutable.

  Por defecto, se asignara el valor a NULL pero podremos asignar un valor mediante una expresion o con una sentencia Select.

  La sintaxis es:

  nombreVar *datatype***:=** valor;
  o
  SELECT col1,...,col$n$ **INTO** var1,...,var$n$
  FROM ... [**WHERE**...]

- Cursores: actuan como una tabla ya que es el resultado de almacenar una consulta de forma temporal. Para trabajar sobre ella, usaremos las funciones **OPEN, FETCH y CLOSE** o estructuras de control, todo dentro de la parte ejectuable. Haremos mas hincapie mas adelante.

Podemos identificar el tipo de dato de una columna, para asegurarnos de la maxima compatibilidad entre variable/constante y columna usando **[tabla].*columna* %TYPE**, que indica el tipo de dato de la columna.

Ademas, podemos declarar valores que actuan como una estructura de datos o tupla a raiz de declararla en funcion al tipo de dato de una fila ya definida, tal que, haciendo *var* **tabla %ROWTYPE** tendremos que var contiene tantos atributos como columnas tienes tabla y cada atributo estara identificado de igual manera que la columna asociada.

4.3.1. Ampliacion sobre cursores

El cursor lo declaramos con:

CURSOR nombreCursor **IS**
sentencia_select;

El cursor tendra como atributos los que indicamos que mostrara el select.

Podemos trabajar sobre los cursores de dos formas:

- De registro a registro: en la parte ejecutable tenemos que abrir el cursor, asignar los valores del registro sobre el que esta el cursor en un set de variables y a continuacion cerrarlo (permite volver a abrirse). La sintaxis sera la siguiente:

```
DECLARE
  var1 tabla.col1 %type;
⋮
  varn tabla.coln %type;
  CURSOR nombreCursor IS sentencia_select;
BEGIN
  OPEN nombreCursor;
⋮
  FETCH nombreCursor INTO var1,...,varn;
⋮
  CLOSE nombreCursor;
END;
```

- De acceso iterativo: usando una estructura de control del tipo bucle FOR, el cursor se abre y se cierra implicitamente, ademas de realizar la instruccion FETCH iterativamente.

```
DECLARE
  CURSOR nombreCursor IS sentencia_select;
BEGIN
  FOR varComp IN nombreCursor LOOP
⋮
  accion que usa la tupla varComp
⋮
  END LOOP;
END;
```

Distinguiremos ademas los siguientes atributos para los cursores:

- **%FOUND**: al abrir un cursor sin haber hecho un FETCH, el valor sera NULL. Tras el primer FETCH, devolver **TRUE** si el FETCH anterior devolvio una fila y **FALSE** si no. El opuesto seria **%NOTFOUND**
- **%ISOPEN**: **TRUE** si esta el cursor abierto, **FALSE** si no.
- **%ROWCOUNT**:al abrir un cursor sin haber hecho un FETCH, el valor sera 0. Tras el primer FETCH, devolvera el numero de filas que se han devuelto al hacer el FETCH.
- Algo muy importante es que los apodos que les demos a las columnas en la sentencia select, estas seran los nombres que se almacenaran como el identificador de las columnas para el cursor.

4.3.2. Estructuras de control

Las siguientes estructuras de control que usaremos son:

- Condicional:

IF... **END IF;**	**IF... THEN ...** **ELSE...** **END IF;**	**IF... THEN ...** **ELSIF... THEN ...** ⋮ [**ELSE...**] **END IF;**

- Iterativo:

```
FOR var IN min..max LOOP
        ...
END LOOP;
```

```
WHILE condicion
        ...
END LOOP;
```

```
LOOP
        ...
EXIT WHEN ...;
END LOOP;
```

```
LOOP
        ...
  IF condicion THEN ...
   ...;
   EXIT;
  END IF;
        ...
END LOOP;
```

5. Procedimienos en PL/SQL

```
CREATE [OR REPLACE] PROCEDURE
nombreProc [(parametro1,..., parametron)]
IS [seccion declarativa]
BEGIN seccion ejecutable
[EXCEPTION seccion de manejo de excepciones]
END;
```

donde tenemos que descarar que:

- Para los parametros, tenemos que indicar *nombreParametro*, *tipoParametro* y *tipoDato*, este sin especificar la longitud. Ademas, los tipos de parametros podran ser *IN* (por defecto), *OUT* e *IN OUT*, que denotan si un parametro es de entrada, salida o ambos.
- Similar a un bloque SQL pero en lugar de usar DECLARE, declaramos el procedimiento junto con otros valores auxiliares a continuacion de la clausula **IS**.
- Para borrar un procedimiento, se usa **DROP PROCEDURE** *nombreProc*.
- Se debe de compilar para evitar posibles errores antes de ejecutar.
- Se puede ejecutar el procedimiento de dos formas: **exec** nombreProc o usando el procedimiento dentro de un bloque SQL o similar (recordad que los parametros sera una forma de obtener una salida de valores, o de no tener una respuesta directa de si ha funcionado).

6. Funciones en PL/SQL

```
CREATE [OR REPLACE] FUNCTION
nombreFunc [(parametro1,..., parametron)]
RETURN tipoDato (sin longitud)
IS [seccion declarativa]
BEGIN seccion ejecutable
RETURN ...; [EXCEPTION seccion de manejo de excepciones]
END;
```

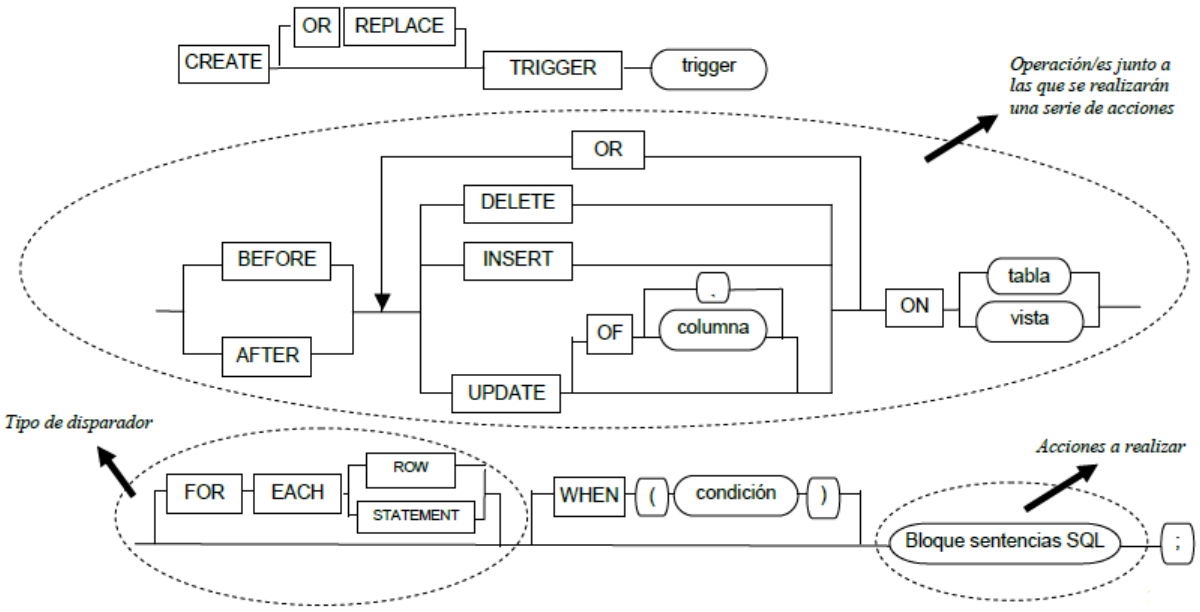
donde tenemos que descarar que:

- Para los parametros, tenemos que indicar *nombreParametro*, *tipoParametro* y *tipoDato*, este sin especificar la longitud. Ademas, los tipos de parametros podran ser *IN* (por defecto), *OUT* e *IN OUT*, que denotan si un parametro es de entrada, salida o ambos, pero no suelen ser definidos ya que el valor de salida se espera de normal en el return.
- Similar a un bloque SQL pero en lugar de usar DECLARE, declaramos la funcion junto con otros valores auxiliares a continuacion de la clausula **IS**.
- Para borrar un procedimiento, se usa **DROP FUNCTION** *nombreFunc*.
- Se debe de compilar para evitar posibles errores antes de ejecutar.
- Se puede ejecutar la funcion de diversas formas:
  - Ejecucion directa usando exec o una sentencia select sobre la tabla dual)
  - Llamada en un procedimiento.
  - En una sentencia **select** (en las clausulas select, where, group by y having)
  - En las sentencias **insert** dentro de los **values**
  - En las sentencias **update**, en la clausula **set**

7. Disparadores en PL/SQL

Un disparador es una secuencia de acciones que se asocian a las operaciones de manipulacion de una tabla o vista dada, sirviendo para completar secciones en funcion a otros valores o para actuar como una restriccion mas compleja que las que constraint CHECK permite realizar.

Cuando se realiza un insert, update o delete sobre una tabla o vista, se realizan las sentencias definidas en el disparador de forma automaticamente.



No podremos poner un script generalizado por lo complejo que puede ser y las diversas opciones, pero vamos a considerar los siguientes puntos:

- AL indicar que se realicen **BEFORE**, queremos que se hagan las sentencias del disparador **ANTES** de realizar la accion que ha provocado al disparador. Con **AFTER**, se espera realizar las sentencias **DESPUES** de la accion.
- Existen las variables *new* y *old* que se usan al hacer Insert y Update o Delete y Update, respectivamente. Si se usan dentro del bloque SQL, deben de venir indicada con :, es decir, *:new* y *:old*.
- Podemos comprobar que funcion ha provocado al disparador dentro del bloque SQL usando los condicionales *inserting*, *deleting*, *updating* y *updating(columna(s)*.
- El disparador del tipo ROW realiza el bloque SQL por cada fila afecatada con la opeacion desencadenadora.
- El disparador del tipo STATEMENT es el usado por defecto si no se indica que tipo de disparador es. Realiza el bloque SQL una unica vez sin importar cuantas filas se vean afectadas.
- El lanzamiento de un error durante la ejecucion impedira que la accion modifique la tabla.
- En muchos casos, el bloque de sentencias no podra acceder a la tabla sobre la que se esta realizando el cambio. Llamaremos tabla mutante a esta y veremos que, si intentamos hacerlo, se dara un error indicando que la tabla esta mutando.

  Se podra acceder a las tablas mutantes si tenemos que el disparadores es BEFORE unicamente en Insert.
- El orden de ejecucion de los disparadores sobre una tabla son: BEFORE sentencia, BEFORE fila, AFTER fila y finalmente AFTER sentencia.
- Activaremos o desactivaremos un disparador usando **ALTER TRIGGER** nombreDisp **ENABLE/DISABLE**;

8. Excepciones en PL/SQL

La estructura general para un bloque SQL con la parte de manejo de instrucciones desarrollada se veria tal que:

```
[DECLARE
  –declaraciones]
BEGIN
  –sentencias a ejecutar (se lanzan excepciones)
EXCEPTION
  WHEN nombreExc1 THEN
    <bloque de sentencias>;
  WHEN nombreExc1 THEN
    <bloque de sentencias>;
...
  [WHEN OTHERS THEN
    <bloque de sentencias>;]
END;
```

De donde tenemos que destacar:

- El bloque de sentencias puede ser una secuencia de sentencias atomicas que sabemos que no daran error o usar un bloque SQL (sin la parte declarativa) para anidar excepciones.

  Otra forma para anidar excepciones es concatenar dentro del BEGIN otro bloque SQL (sin la parte declarativa). Se verian de esta manera:

```
[DECLARE
  –declaraciones]
BEGIN
  –sentencias a ejecutar(Exc1 se lanza)
EXCEPTION
  WHEN nombreExc1 THEN
   BEGIN
     –sentencias a ejecutar(Exc2 se lanza)
   EXCEPTION
     WHEN nombreExc2 THEN
      <bloque de sentencias>;
   END;
END;
```

```
[DECLARE
  –declaraciones]
BEGIN
  –sentencias a ejecutar (Exc1 se lanza)
   BEGIN
     –sentencias a ejecutar(Exc2 se lanza)
   EXCEPTION
     WHEN nombreExc2 THEN
      <bloque de sentencias>;
   END;
EXCEPTION
  WHEN nombreExc1 THEN
   <bloque de sentencias>;
END;
```

8.1. Predefinidas

Excepciones predefinidas en Oracle que se disparan automaticamente con ciertos errores, siendo los mas frecuentes:

- **too_many_rows**: Se produce cuando en una asignacion con sentencia select se obtienen mas de una fila. (SELECT ... INTO)
- **no_data_found**: Se produce cuando en una asignacion con sentencia select no se obtiene ninguna fila. (SELECT ... INTO)
- **value_error**: error aritmetico o de conversion
- **zero_divide**:se produce una division entre cero
- **dupval_on_index**: se crea cuando se intenta almacenar un valor que crearia un duplicado
- **invalid_number**:se produce cuando se intenta convertir una cadena a un valor numerico.

8.2. Definidas por el usuario

Excepciones que el usuarios define mendieante los siguientes pasos:

1. Definir el nombre de la excepcion en la parte declarativa con las sintaxis:

```
Nombre_excepcion EXCEPTION;
```

2. Lanzar la excepcion, dada cierta condicion, usando **raise**, tal que:

```
dada una condicion
raise Nombre_excepcion;
```

3. Definir las acciones a seguir en la parte de manejo de instrucciones, tal que:

```
WHEN Nombre_excepcion THEN
   <bloque de sentencias>;
```

9. Indices y transacciones en Oracle

Un indice es una estructura de memoria secundaria que permite el acceso directo a las filas de una tabla, mejorando el tiempo de respuesta de una consulta y su rendimiento, ademas de optimizar el resultado. Por defectos Oracle interpreta las claves primarias y las unique como indices.

Una transaccion es un conjunto de sentencias SQL que se ejecutan en una BD como una unica opeacion, confirmandose o deshaciendose todo el conjunto de SQL.

9.1. Sentencias para Indices

- Creacion de un indice compuesto por un conjunto de columnas:

  CREATE INDEX nombre_indice
  ON nombre_tabla(columna1[,...,columna$n$]);

- Borrado de un indice:

  DROP INDEX nombre_indice;

- Prueba de uso de los indices: almacenando el plan de ejecucion de una sentencia, podremos comprobar si un indice se esta usando. Para ello, tenemos que usar primero la sentencia **EXPLAIN PLAN** que graba o busca los datos de un plan de ejecucion y a continuacion hacer una consulta a la tabla plan_table.

  Considerar que no se va a pedir mucho sobre esto, por lo tanto el conjunto de instrucciones a usar serian:

  EXPLAIN PLAN
  [**SET STATEMENT_ID**='nombre_sentencia'] por defecto sera NULL
  [**INTO** nom_tabla] por defecto sera PLAN_TABLE
  FOR *sentencia*;

  SELECT operation, options, object_name, position
  FROM nom_tabla
  [**WHERE** statement_id='nombre_sentencia']

9.2. Sentencias para Transacciones

- COMMIT: al hacer sentencias conta la BD, estas no se hacen efectivas hasta que no ejecutamos un COMMIT. Los Create Table y abandonar una sesion de trabajo implican un COMMIT implicito.

  COMMIT;

- SAVEPOINT: para marcar un punto dentro de un conjunto de sentencias que se estan ejecutando.

  SAVEPOINT *savepoint*;

- ROLLBACK: deshace acciones que se han ralizado, pudiendo volver hasta un savepoint si existe. En el caso de no indicar un savepoint, de deshace toda la transaccion.

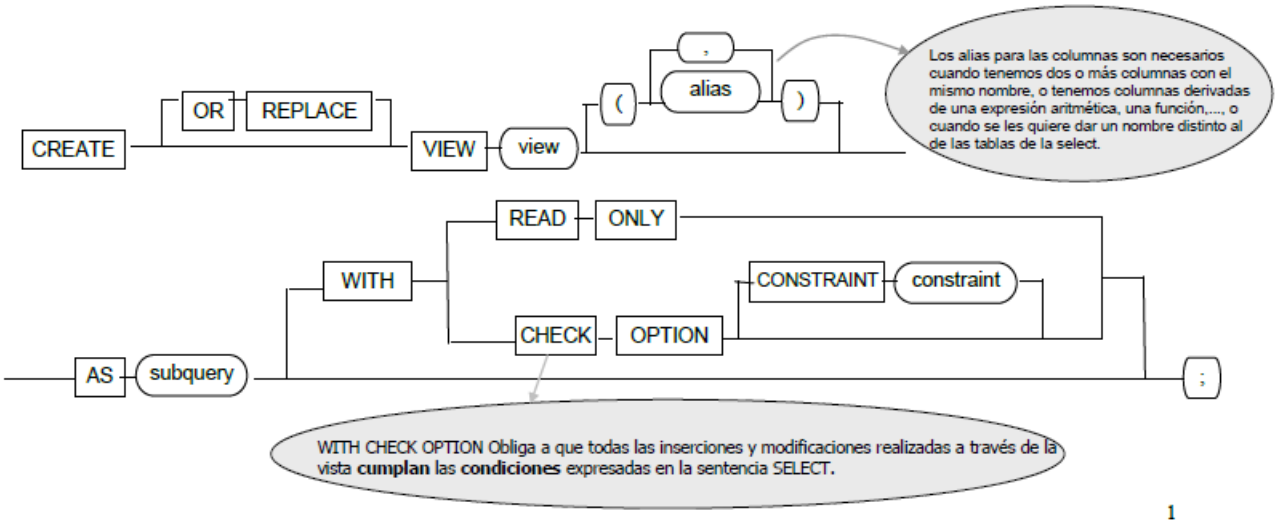
  ROLLBACK [**TO** [**SAVEPOINT**]*savepoint*];

10. Seguridad en Oracle

Una forma de garantizar la seguridad, ademas de simplificar la estructura para un usuario es el uso de vistas. Tambien podemos asignar determinados privilegios para los usuarios para permitir el uso de ciertas acciones.

10.1. Vistas

Las vistas actuan de forma similar a un select pero se asocian a una tabla. Es decir, podriamos interpretar que las vistas son un puntero a un select y, si modificamos algun parametro (añadiendo, eliminando o modificando columnas) las tablas asociadas al select se podrian ver afectadas.

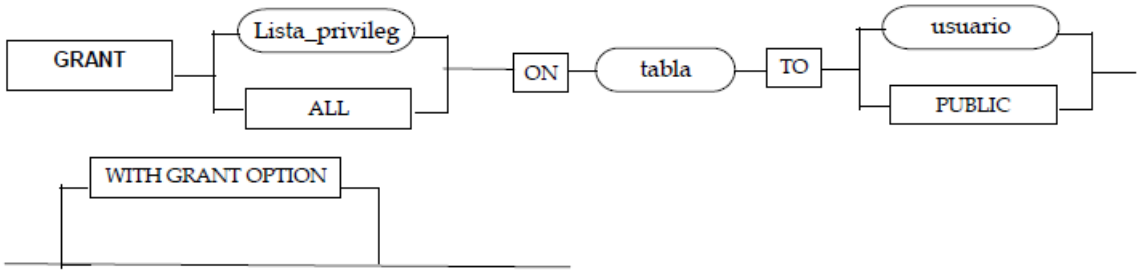


1

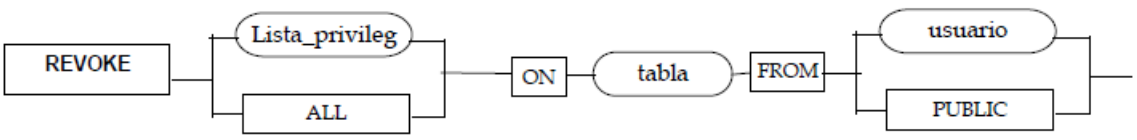
Para eliminarla, usaremos **DROP VIEW** nombre_view;

10.2. Privilegios

Podemos conceder privilegios con



y revocarlos con



11. De modelo relacional a EER, por Kenny

1:N

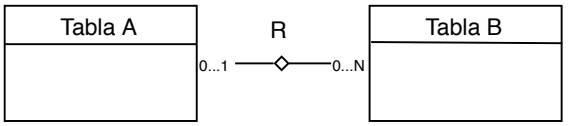


Tabla A (a, b)
CP(a)
C Aj(b) -> B

Tabla B (b)
CP(b)

1:N + RE

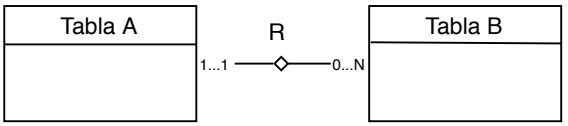


Tabla A (a, b)
CP(a)
C Aj(b) -> B
VNN(b)

Tabla B (b)
CP(b)

1:1

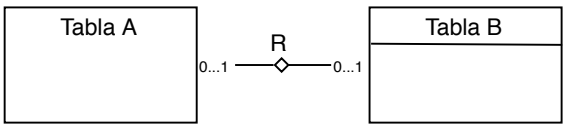


Tabla A (a, b)
CP(a)
C Aj(b) -> B

Tabla B (b)
CP(b)

Tabla R (a, b)
CP(a)
C Alt (b)
C Aj(a) -> A
C Aj -> B

1:1 + RE

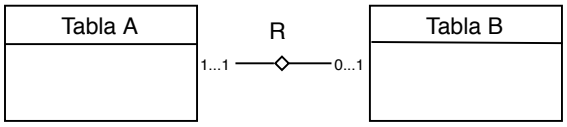


Tabla A (a, b)
CP(a)
C Aj(b) -> B
C Alt(b)

Tabla B (b)
CP(b)

1:1 + 2 RE

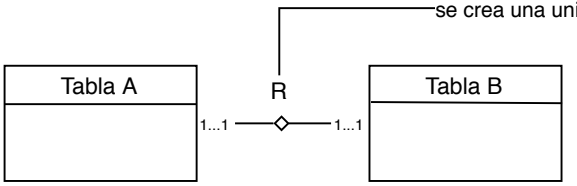


Tabla R (atributos de a, atributos de b)
CP (a o b)
C Alt(a o b)

N:M

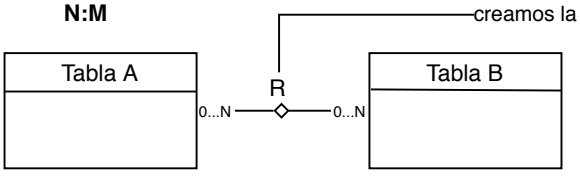


Tabla R(a, b)
CP(a, b)
C Aj(a) -> A
C Aj -> B

Generalización

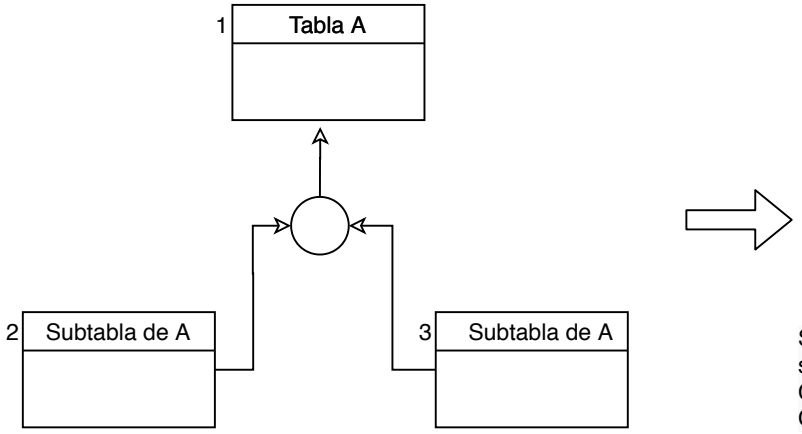


Tabla A (a)
CP(a)

Subtabla A.1 (a, atributos de subtabla)
CP (a)
C Aj(a) -> A

Subtabla A.2(a, atributos de subtabla)
CP (a)
C Aj(a) -> A

Dependencia de indentificador

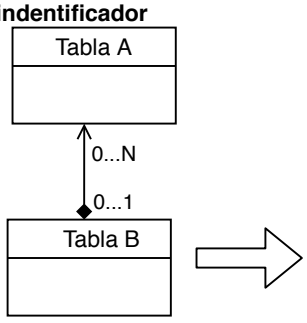
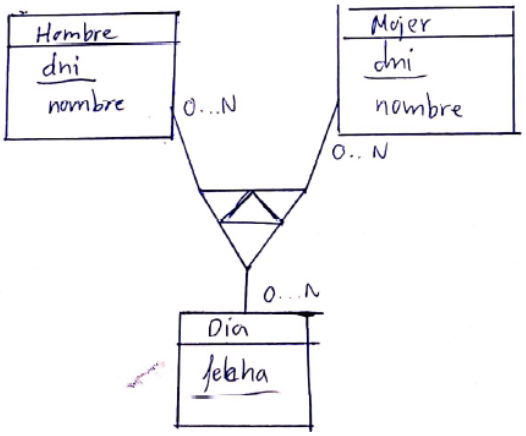


Tabla A (a)
CP(a)

Tabla B(a, b)
CP (a, b)
C Aj(a) -> A

¿Cuándo hay una ternaria?

- 3 entidades relacionadas.
- Por pares siempre son relaciones N:N.
- En la relación siempre participan todas (no nulos).



Definición:
Muchos hombres se pueden casar con muchas mujeres en muchos días. Sin embargo, podemos añadir restricciones coloreando el triángulo de la tabla correspondiente.

- Poder distinguir si se trata de una ternaria:
  1. Ver si hay al menos hay 3 CAj, no 3 tablas distintas.
  2. Ver si CP y CAlt apuntan a dos tablas, si solo hay CP y apunta a dos tablas se dará por válido este punto.
  3. Condición → Que todas participen → Que sus claves ajenas participen.