

Sesión 2 – SENTENCIA WITH , VALORES NULL, ALTER TABLE, IDENTITY, COMMIT, ROLLBACK y SAVEPOINT

WITH para simplificar consultas complejas

La sentencia WITH es un estándar (SQL-99) y se utiliza para simplificar queries complejos resolviéndolos mediante el uso de consultas intermedias que posteriormente son utilizadas por la sentencia principal. Pueden mejorar el rendimiento y sus ventajas son:

- Hace que la consulta resulte más sencilla de implementar e interpretar
- Evalúa una cláusula sólo una vez, incluso aunque aparezca varias veces en la consulta principal lo que mejora el rendimiento
- En la mayoría de los casos puede mejorar el rendimiento para las consultas grandes ya que facilita al sistema su plan de ejecución

```
WITH  QUERY_NAME_1 AS ( SELECT ..... ) ,  
      QUERY_NAME_2 AS (SELECT .....),  
      ... .  
      QUERY_NAME_N AS (SELECT .....)  
SELECT ...  
FROM  QUERY_NAME_1, QUERY_NAME_2 ..... .
```

Ejemplo:

Supongamos que tenemos dos tablas:

COMPRAS.(ID_COMPRA, COD_ART, DNI_CLIENTE, CANTIDAD, PRECIO_UNITARIO, FECHA)

CLIENTES (DNI, NOMBRE)

Supongamos que queremos por cada cliente las veces que ha comprado, la suma de artículos, la media de los precios pagados y el total comprado siempre que haya comprado más de 100 artículos.

Una posible query podría ser

```
Select DNI,  
       NOMBRE,  
       (SELECT COUNT(*) FROM COMPRAS WHERE DNI_CLIENTE = DNI) AS VECES,  
       (SELECT SUM(CANTIDAD) FROM COMPRAS WHERE DNI_CLIENTE = DNI) AS NUM_ART,  
       (SELECT AVG(PRECIO_UNITARIO) FROM COMPRAS WHERE DNI_CLIENTE = DNI) AS MEDIA_PRECIO,  
       (SELECT SUM(CANTIDAD*PRECIO_UNITARIO) FROM COMPRAS WHERE DNI_CLIENTE = DNI) AS TOTAL_COMPRADO  
FROM CLIENTES  
WHERE (SELECT SUM(CANTIDAD) FROM COMPRAS WHERE DNI_CLIENTE = DNI) > 100
```

Vemos como la sentencia (SELECT SUM(CANTIDAD) FROM COMPRAS WHERE DNI_CLIENTE = DNI) se va a ejecutar dos veces,

Con la clausula with sería

```
WITH C_COMPRAS as (  
    Select DNI_CLIENTE,  
           COUNT(*) as VECES,  
           SUM(CANTIDAD) as NUM_ART,  
           AVG(PRECIO_UNITARIO) as MEDIA_PRECIO,  
           SUM(CANTIDAD*PRECIO_UNITARIO) as TOTAL_COMPRADO  
    FROM COMPRAS  
    GROUP BY DNI_CLIENTE)  
SELECT DNI, NOMBRE, VECES, NUM_ART, MEDIA_PRECIO, TOTAL_COMPRADO  
FROM CLIENTES JOIN C_COMPRAS ON DNI = DNI_CLIENTE  
WHERE NUM_ART > 100
```

Uso de valores NULL

El comportamiento de los valores NULL en ciertas operaciones pueden ocasionar resultados inesperados si no se lleva cuidado con ellos ya que:

- NULL no es un valor
- NULL es un estado que indica que el valor es DESCONOCIDO (UNKNOWN)
- NULL no es cero (0) , ni blanco (' ') ni cadena vacía ('') y por tanto no se comporta como ellos.

En ORACLE tenemos la función NVL (campo, valor) para tratar los nulos:

```
Select nvl(importe, 0), nvl(descripcion,' Sin descripción') from ARTICULOS.
```

Si *importe* es NULL entonces devuelve 0, si *descripcion* es NULL entonces devuelve la cadena 'Sin descripción'.

El comportamiento es:

EXPRESIONES BOOLEANAS

Cualquier expresión booleana en que figure un NULL devolverá un "UNKNOWN" (nulo). Ese UNKNOWN tiene comportamientos diferentes en función de dónde se produzca:

En los where los UNKNOWN se evalúan como false

```
Select * from emp where sueldo < 1000    -> NO devolverá los que tienen sueldo a nulo  
Select * from emp where nvl(sueldo,0) < 1000    -> SI devolverá los que tienen sueldo a nulo
```

En las check constraints los UNKNOWN se evalúan como true -> No se disparan

Constraint Check sueldo < 1000 -> **SI permite** introducir sueldos a nulo
 Constraint Check nvl(sueldo,0) < 1000 -> **NO permite** introducir sueldos a nulo

FUNCIONES DE GRUPO

Mientras que en expresiones escalares devuelve un nulo, en funciones como SUM, AVG, etc. los ignora:

Supongamos la tabla VENTAS:

ID	PRECIO	DESCUENTO
1	100	20
2	200	
3	1000	100
4	500	

```

Select sum(descuento) from ventas;   devuelve 120
select sum(precio)-sum(descuento) from ventas;   devuelve 1680
select sum(precio-descuento) from ventas;   devuelve 980
  
```

Pero OJO!!! Si la tabla tuviese todos los descuentos a nulo:

ID	PRECIO	DESCUENTO
1	100	
2	200	
3	1000	
4	500	

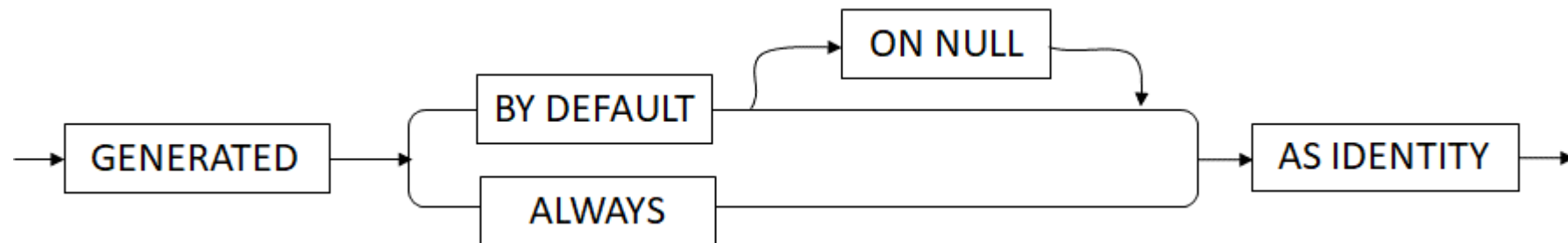
Tanto una sentencia como la otra devolverían NULL.

```
select sum(precio)-sum(descuento) from ventas;    devuelve  NULL
select sum(precio-descuento) from ventas;    devuelve NULL
```

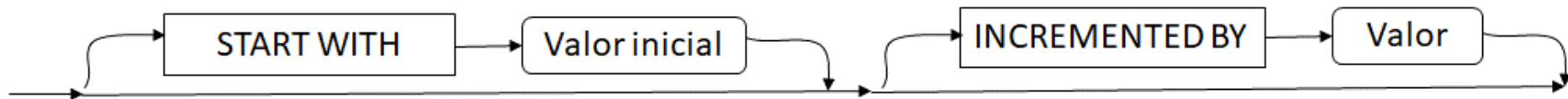
Por tanto **es siempre aconsejable el uso del NVL si el campo puede contener valores NULOS.**

IDENTITY para incluir columnas con valores que se pueden incrementar automáticamente en las tablas

Las columnas IDENTITY, también llamadas autonuméricas, las utilizaremos para crear valores que el gestor de la base de datos incremente y gestione de manera automática. Al realizar inserciones de nuevos datos, será el gestor de la base de datos quien incremente el nuevo valor para la fila que estamos incluyendo. Realmente IDENTITY es una característica especial de los tipos de datos numéricos. La estructura sintáctica es ésta:



- **BY DEFAULT:** Nos permite utilizar Identity si la columna no se hace referencia en una sentencia insert (igual que ALWAYS), pero si se hace referencia a la columna, el valor especificado se utilizará en lugar de la identidad. El intento de especificar el valor NULL en este caso se traduciría en un error, ya que las columnas de identidad son siempre NOT NULL.
- **ALWAYS:** Indica que no será necesario introducir un valor para esta columna en una sentencia insert. Por el contrario, indicar un valor, aunque sea Null, producirá un error de ORACLE.
- **[ON NULL]:** Si añadimos esta cláusula opcional a BY DEFAULT, cuando insertemos un valor NULL para la columna no fallará, y añadirá un valor automático nuevo. En cambio, si se especifica un valor nuevo, será ése el que se incluirá en la tabla.



Al definir una columna como IDENTITY podemos utilizar las siguientes opciones:

- **START WITH** initial_value: Controla el valor inicial que usará la columna tipo identity. El valor inicial por defecto es 1.
- **INCREMENT BY** interval_value: Define el intervalo que el gestor dejará entre dos valores que genere. Por defecto será 1. Por ejemplo, si definimos un intervalo de 10 y un valor inicial de 10 también, la serie generada será 10, 20, 30, etc.

Las columnas autonuméricas las utilizaremos principalmente en estos casos:

- 1) Cuando la clave primaria de una tabla pueda cambiar sus valores, y haya claves ajenas en otras tablas apuntando hacia ella. Salvo que el gestor de base de datos soporte **ON UPDATE CASCADE**, ese cambio provocaría errores de integridad referencial.
- 2) Para crear claves primarias automáticas en tablas de datos en las que es difícil (o imposible) encontrar una combinación de columnas que nos las produzca.
- 3) Para crear claves primarias en tablas que tienen claves primarias complejas (compuestas por varias columnas) o de mucho tamaño (cadenas largas). Eso producirá que las claves ajenas que apunten a esas tablas tengan menor coste computacional en los joins, y menos coste de almacenamiento.

A estas columnas se les suele denominar claves subrogadas (surrogate keys en inglés).

Ejemplo:

Tenemos la siguiente sentencia que crea una tabla que representa los jugadores de un juego ON – LINE.

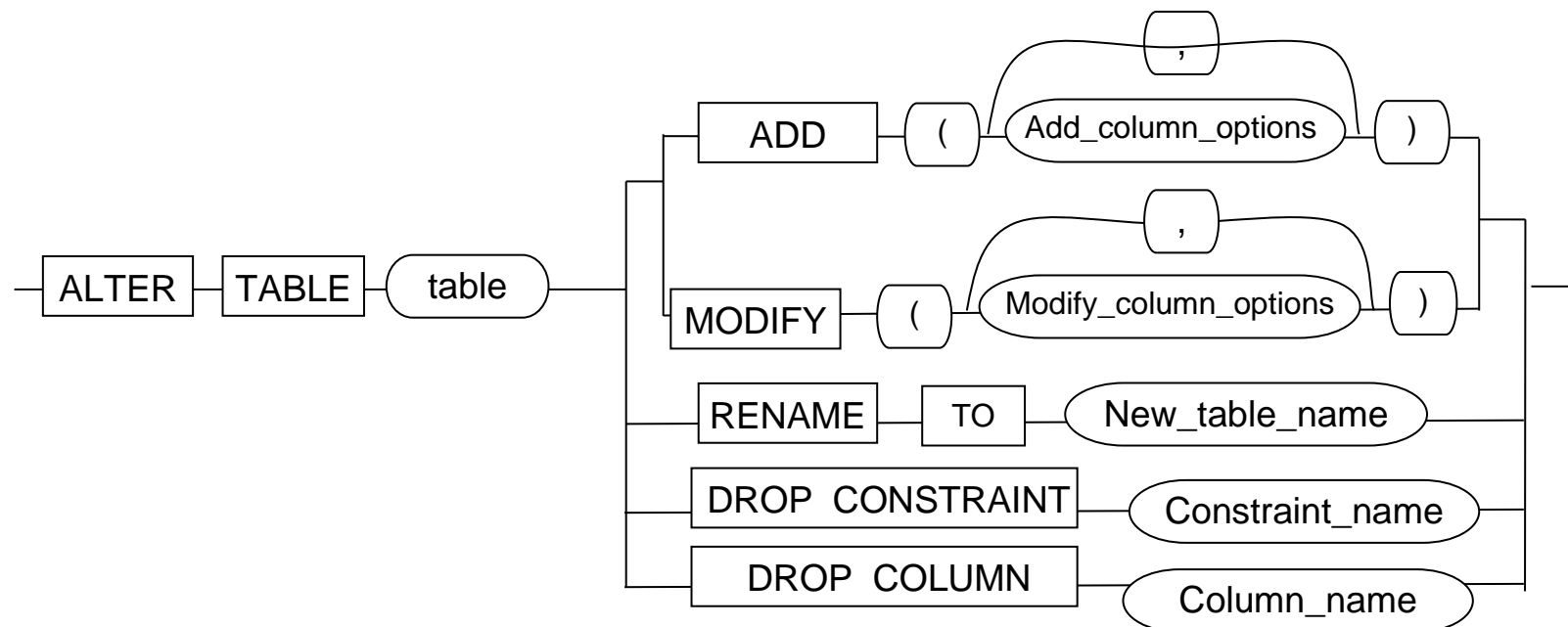
```
CREATE TABLE jugadores (  
    jugador_id      VARCHAR2(150) NOT NULL,  
    jugador_nombre  VARCHAR2(50) NOT NULL,  
    jugador_fecha_alta  DATE NOT NULL,  
    jugador_activo   INT DEFAULT 1 NOT NULL,  
    CONSTRAINT pk_jugadores PRIMARY KEY ( jugador_id ));
```

La columna JUGADOR_ID es la clave primaria, pero fijaos que es larga, y tiene 150 caracteres. Eso producirá que para cualquier tabla que tome claves ajenas desde ella, tengamos que incluir una columna de esa longitud. Modifiquémosla incluyendo una columna tipo IDENTITY:

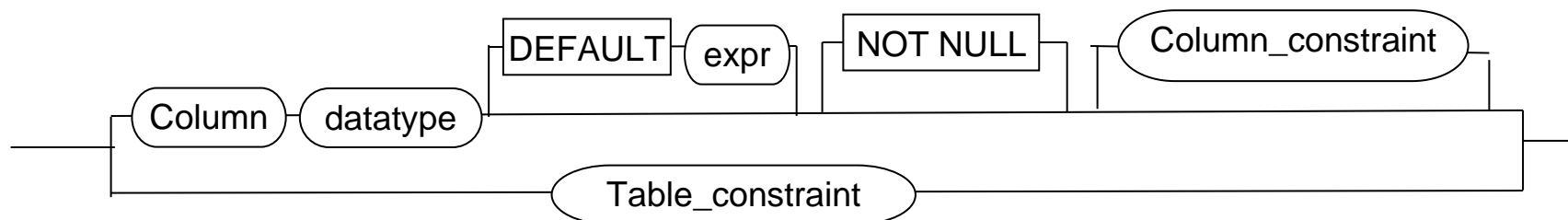
```
CREATE TABLE jugadores (  
    jugador_id      VARCHAR2(150) NOT NULL,  
    jugador_sk      INT NOT NULL GENERATED ALWAYS AS IDENTITY,  
    jugador_nombre  VARCHAR2(50) NOT NULL,  
    jugador_fecha_alta  DATE NOT NULL,  
    jugador_activo   INT DEFAULT 1 NOT NULL,  
    CONSTRAINT UNI_jugadores UNIQUE ( jugador_id ),  
    CONSTRAINT PK_jugadores PRIMARY KEY ( jugador_sk )  
);
```

ALTER TABLE para modificar la estructura y restricciones de una tabla

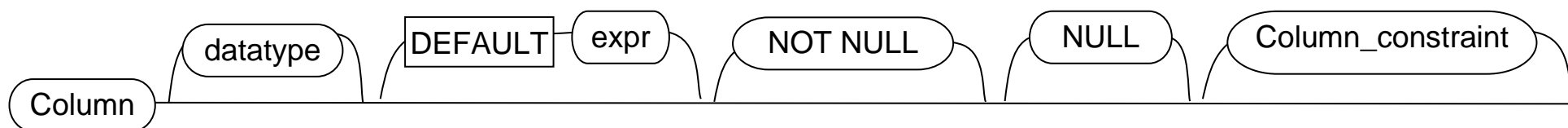
Después de crear las tablas se puede modificar su estructura con la sentencia ALTER TABLE. Nos permite añadir o borrar una nueva columna a una tabla, así como añadir o borrar restricciones a las columnas de la tabla.



Add_column_options



Modify_column_options



Ejemplos:

```
ALTER TABLE JUGADORES ADD Jugador_fnacimiento date;
```

```
ALTER TABLE JUGADORES MODIFY jugador_nombre null;
```

Suponiendo que existe una tabla NACIONALES con una columna llamada JUGADOR_ID. Para que sea FK hacia jugadores:

```
ALTER TABLE NACIONALES ADD constraint FK_jugadornacionales foreign key (JUGADOR_ID) references JUGADORES;
```

COMMIT, ROLLBACK y SAVEPOINT: concepto de transacción

TRANSACCIÓN:

- Conjunto de sentencias SQL que son tratadas por el SGDBR como unidad. **O todas son validadas o ninguna es validada.**
- El inicio lo marca la primera sentencia DML (Data Manipulation Language, es decir, inserts, updates, etc.) desde el último commit o rollback.
- *COMMIT*: hace permanentes los cambios producidos por la transacción.
- *ROLLBACK*: anula los cambios hechos por la transacción.
- Los cambios hechos por una transacción pendiente de COMMIT o ROLLBACK son visibles sólo por la sesión que ejecutó la transacción.
- Las sentencias DDL (Data Definition Language, es decir, CREATE, ALTER TABLE etc) llevan commit implícito.
- Si se produce una violación de constraint o cualquier otro error, el SGBDR hace un ROLLBACK automático.

COMMIT, ROLLBACK y SAVEPOINT: concepto de transacción

Update Clientes set domicilio = null
where codcli = '4536';

En este momento este usuario ve la dirección del cliente a nulo, pero el resto de sesiones todavía la ve sin cambios

Update facturas set Euros = pts*166.337
where pts is not null;

Delete from facturas
where Euros is null;

Si violasen alguna constraint o hubiese algún error la transacción Ejecutaría un rollback automático y terminaría

Commit;


La transacción se ha consolidado y ya todos ven la dirección a nulo. Ya no tiene posibilidad de rollback

COMMIT, ROLLBACK y SAVEPOINT: concepto de transacción

Update Clientes set domicilio = null
where codcli ='4536';

Update facturas set Euros = pts*166.337
where pts is not null;


Es el usuario el que provoca el rollback, con que se anulan todas las operaciones pendientes y comienza una nueva transacción



rollback;

Delete from facturas
where Euros is null;

La transacción se ha consolidado pero SÓLO se produce el borrado, los dos Updates anteriores ya estaban anulados



Commit;

COMMIT, ROLLBACK y SAVEPOINT: concepto de transacción

La sentencia SavePoint permite dividir una transacción grande en varios trozos. De esta forma si se produce un error no tenemos que deshacer toda la transacción, sino sólo hasta un save point

Update

insert;

Insert;

....

SavePoint P1

Definimos SavePoint



insert ...;

Delete

Rollback to P1

Deshacemos SÓLO las dos últimas sentencias



insert

COMMIT

NO incluye las sentencias entre SavePoint P1 y el ROLLBACK P1



Utilidad práctica de los conceptos vistos en la sesión

- Cláusula WITH, columnas IDENTITY, y ALTER TABLE:

WITH: Sirve para simplificar y mejorar el rendimiento de sentencias SQL complejas.

IDENTITY: Usaremos estas columnas para que el gestor de la base de datos autogestione su incremento.

ALTER TABLE: Nos permitirá modificar la estructura de las tablas.

- Commit, Rollback y Savepoint:

Nos dan un mecanismo para garantizar la integridad de la información en nuestra base de datos cuando efectuemos modificaciones de información delicadas o complejas.

Ejemplo:

Realización en un banco de una transferencia de una cuenta a otra.

- Sentencia 1: Obtener la cantidad a transferir.
- Sentencia 2: Modificar la cuenta origen, restándole el importe a transferir.
- Sentencia 3: Modificar la cuenta destino, sumándole el importe a transferir.

¿Qué ocurriría si la cuenta destino del dinero estuviese bloqueada, y no permitiese ingresos? La cuenta origen quedaría con el saldo mermado en la cantidad que queríamos transferir, y el dinero quedaría en el "limbo".

Usando la sentencia Rollback evitaríamos esa inconsistencia en los datos.