Continuación SQL

Índice

1. Continuación del SQL como DML (Lenguaje de Manipulación de Datos)

1.1. Inserción de datos

Para insertar datos en una tabla, usaremos la sentencia `INSERT`, que tiene la siguiente sintaxis:

```
INSERT INTO <nombre_tabla> [(<lista campos>)] VALUES (<lista valores>)
```

La lista de campos (atributos de la tabla) es opcional. Si se omite, es lo mismo que si se incluyese la lista de todos los atributos de la tabla, siguiendo el orden definido en la sentencia de creación de la tabla. Si se incluye, pueden indicarse sólo una parte de los campos (o todos) en el mismo o distinto orden de la definida en la creación de la tabla. Así, las dos sentencias siguientes son equivalentes:

```
INSERT INTO EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO)
VALUES (8555, 'Jose', 'ANALISTA', 7369, '12-FEB-02', 3000, 300,10);

INSERT INTO EMP VALUES (8555, 'Jose', 'ANALISTA', 7369, '12-FEB-02', 3000, 300,10);
```

La lista de valores, por supuesto, debe contener tantos valores como atributos hay en la lista de campos, y corresponderse un al tipo del atributo correspondiente. Por supuesto, el valor NULL puede ser usado, si el campo correspondiente fue definido aceptando valores nulos.

Si se usa la lista de campos, podemos variar el orden de los campos, o omitir alguno de ellos. En los campos omitidos se insertarán nulos, o el valor por defecto (se así se definió para el campo). Por ejemplo, las siguientes tres sentencias también serían equivalentes.

```
INSERT INTO DEPT(DEPTNO,DNAME) VALUES(50,'COMPRAS');
INSERT INTO DEPT (DNAME,DEPTNO) VALUES ('COMPRAS',50);
INSERT INTO DEPT VALUES (50,'COMPRAS', NULL);
```

Además de esta forma de insertar datos, que se realiza siempre de tupla en tupla, también se pueden insertar varias tuplas con una sentencia, obteniendo los datos mediante una sentencia `SELECT` (una consulta) sobre una o más tablas. La sintaxis general es la siguiente:

```
INSERT INTO <nombre_tabla> [(<lista de campos>)]
    <Sentencia SELECT>
```

La única restricción es que la sentencia `select` debe seleccionar el mismo número de atributos que campos hay en la lista de campos, y ser unión-compatibles. Como en el método anterior, si en la lista de campos se omite, se asume que el select devuelve una lista con todos los atributos de la tabla en el mismo orden que el establecido en la creación de la tabla. Por ejemplo, supongamos que tenemos una tabla `COPIA_DEPT` que tiene exactamente la misma estructura que a tabla `DEPT`. La siguiente sentencia serviría para insertar varias tuplas en la tabla `DEPT` a partir de una consulta.

```
INSERT INTO DEPT (DEPTNO, DNAME, LOC)
    SELECT C.DEPTNO, DNAME, LOC
        FROM COPIA_DEPT C, EMP E
        WHERE C.DEPTNO=E.DEPTNO
```

1.2. Borrado de datos

El borrado de datos se realiza mediante la sentencia `DELETE`, que borra tuplas de una tabla. La sintaxis general es:

```
DELETE FROM <nombre_tabla>
    [WHERE <Condicion>]
```

Donde la `Condicion` es cualquier condición válida en SQL, es decir, cualquier condición que podríamos poner en la cláusula `WHERE` de una sentencia `SELECT`. Por ejemplo,

```
DELETE FROM EMP
    WHERE SAL > 1500
      AND JOB <> 'MANAGER'
```

Es importante indicar que la sentencia `DELETE` borra siempre tuplas completas, nunca determinados atributos de una tupla. La cláusula `WHERE` es opcional, y se no se utiliza, se borrarán todas las tuplas de la tabla.

1.3. Modificación de Datos

La sentencia `UPDATE` permite modificar los valores almacenados en una tabla, es decir, cambiar el valor que hay en un atributo de una tupla por otro valor. Su sintaxis es:

```
UPDATE <nombre_tabla>
    SET <atributo_1> = <expre1>
        [,<atributo_2>=<expre2>...]
    WHERE <Condicion>
```

Un ejemplo,

```
UPDATE EMP
    SET  SAL = SAL*1.06,
    JOB = 'ANALYST'
    WHERE EMPNO = 7369
```

La sentencia de actualización examina todas las tuplas que satisfagan a condición impuesta en la cláusula `WHERE` (si no se indica, los cambios afectarán a todas las tuplas de la tabla). Para cada una de estas filas, el atributo tomará el nuevo valor, que puede ser una constante (como en `JOB = 'ANALYST'`) o una expresión. Las expresiones pueden contener atributos de la tabla que se está actualizando, incluso el mismo nombre de atributo que está a la izquierda del igual. En este caso, el valor del atributo a la derecha del igual es el valor de dicho atributo antes de realizarse la actualización. Es decir, la asignación `SAL=SAL*1.06` indica que el salario actual se va incrementar en un 6 % respecto al salario anterior.

2. SQL como DDL (Lenguaje de Definición de Datos)

Hasta ahora hemos utilizado el lenguaje SQL como un **DML** (Data Manipulation Language) o Lenguaje de Manipulación de Datos, es decir, para añadir, borrar, modificar o consultar datos. Pero además el SQL puede utilizarse como **DDL** (Data Definición Language) o Lenguaje de Definición de Datos; esto es, podemos utilizar SQL para crear, modificar o borrar diferentes objetos de bases de datos como tablas, vistas o índices.

2.1. Creación de tablas

La sentencia SQL de creación de tablas es `CREATE TABLE`, y en ella se definen por un lado los atributos que componen la tabla, y por otro las restricciones que afectan a la tabla, tales como claves primarias o externas. La sintaxis general de la sentencia de creación se compone de una lista (separada por comas) de definiciones de atributos seguida opcionalmente de una lista de definiciones de restricciones:

```
CREATE TABLE <nombre_tabla>(
    <definicion atributo 1>,
    <definicion atributo 2>,
    ...
    <definicion atributo n>,
    <definicion restriccion 1>,
    <definicion restriccion 2>,
    ...
    <definicion restriccion m>
    );
```

A continuación veremos cómo realizar cada una de estas definiciones.

2.1.1. Definición de atributos

Una definición de atributo describe el nombre y tipo de datos asociados al atributo, así como si admite o no valores nulos, valores por defecto y restricciones asociadas al atributo. La sintaxis general es la siguiente:

```
<nombre_atributo> <tipo> [DEFAULT <valor>] [NOT [NULL]] [<restriccion>]
```

Donde

`<nombre_atributo>` es el nombre del atributo.

`<tipo>` es el tipo del atributo, que será uno de los tipos de datos vistos anteriormente.

`DEFAULT <valor>` indica el valor por defecto que almacenará el campo si se inserta una tupla sin indicar un valor para ese atributo (si no se indica un valor por defecto, se insertará un valor nulo). `<valor>` será, evidentemente, una constante del tipo definido para el atributo. Existen algunos valores predefinidos para algunos tipos de datos. Así, tenemos valores como `SYSDATE`, que nos da la fecha actual, o `USER` que es un string que nos da el nombre del usuario (en el caso de Oracle). También se puede usar el valor `NULL` para cualquier tipo de datos, siempre que el atributo lo admita. Sin embargo, hacer esto no tiene demasiado sentido, ya que es el comportamiento por defecto si no se indica un valor `DEFAULT`.

`[NOT] NULL` indica si el atributo acepta valores nulos (`NULL`) o no los acepta (`NOT NULL`). Si no se indica nada se supone que sí los acepta.

`<restriccion>` es una restricción sobre el atributo. Veremos principalmente las restricciones que indican que un atributo es una clave primaria o una clave externa que

referencia otra tabla. Si el atributo es una clave primaria, se indicará con `[CONSTRAINT <nombre_constr>] PRIMARY KEY`, y si es una clave externa, mediante `[CONSTRAINT <nombre_constr>] REFERENCES <tabla>[(<atributos>)]`. Veremos más adelante cómo indicar las restricciones de forma independiente de la definición de los atributos.

Otros tipos de restricción usados son `CHECK`, para realizar comprobaciones en los valores que se introducen en la tabla, o `UNIQUE`, que establece que un atributo (o lista) no puede tener duplicados en la tabla.

Ejemplo 2.1 Una creación de la tabla de departamentos podría ser:

```
CREATE TABLE DEPT(
DEPTNO NUMBER(4) CONSTRAINT rest1 PRIMARY KEY,
DNAME VARCHAR2(14),
LOC VARCHAR2(13));
```

2.1.2. Definición de restricciones

Una definición de restricción tiene el formato general siguiente:

```
[CONSTRAINT <nombre_constraint>] <tipo> <definicion>
```

donde la palabra `CONSTRAINT` y el nombre son opcionales, `<tipo>` indica el tipo de restricción. El nombre de la restricción, en caso de usarse, debe ser único para toda la base de datos (aunque esté en distinta tabla no puede tener el mismo nombre). Finalmente, `<definición>` describe completamente la restricción. Dado que esto último depende de cada restricción, veremos los distintos tipos por separado.

Definición de claves primarias Una clave primaria puede estar compuesta de un atributo o de varios, como ya hemos visto. En cualquier caso, todos los atributos que formen parte de la clave primaria, ya que van a identificar unívocamente una tupla, deben estar definidos como `NOT NULL` (si no se indica explícitamente, el SGBD lo hace de forma automática).

Si la restricción de clave primaria afecta a un solo atributo (como `DEPTNO` en la tabla `DEPT`), puede indicarse directamente en la definición del atributo, pero también como una restricción independiente, utilizando la sintaxis siguiente.

```
[CONSTRAINT <nombre>] PRIMARY KEY (<lista de atributos separados por comas>)
```

como por ejemplo, `CONSTRAINT pk_dept PRIMARY KEY(DEPTNO)`. Por convención, se suele preceder el nombre de una restricción de clave primaria por el prefijo `pk_` (por Primary Key).

Definición de claves externa Una clave externa establece una relación de integridad referencial entre dos tablas.

Es obligatorio que la tabla referenciada tenga una clave primaria (o, en su defecto, que exista una restricción única), y será precisamente esa clave primaria (o la lista de atributos con restricción única) la que se referencia en la clave externa. De la misma forma, nótese que para poder crear una tabla que referencia a otra, la tabla referenciada debe existir con anterioridad, es decir, debe ser creada antes.

Al igual que para las clave primarias, para establecer una clave externa tenemos dos opciones: incluirla en la definición del atributo, como ya se ha indicado:

```
<atributo> <tipo> REFERENCES <tabla_referenciada>[(<columnas_referenciadas>)]
```

Por ejemplo,

```
DEPTNO NUMBER(4) REFERENCES DEPT
```

Una segunda opción, la única utilizable cuando la clave externa está formada por varios atributos (pero válida también si es uno solo) es definir la clave externa como una restricción aparte:

```
[CONSTRAINT <nombre_constraint>] FOREIGN KEY (<lista atributos separados por comas>)
            REFERENCES <tabla_referenciada> [(<columnas_referenciadas>)]
            [ON DELETE CASCADE]
```

Por convención, normalmente se usa el prefijo `fk_` (Foreign Key) para nombrar las restricciones de clave externa. La cláusula opcional `ON DELETE CASCADE` indica cómo reacciona la tabla para mantener la integridad en caso de borrar una tupla de la tabla referenciada. Si se indica `ON DELETE CASCADE`, cuando se borra la tupla "padre" (la referenciada), se borran también los "hijos" (es decir, si se borra un departamento, se borrarán automáticamente los empleados que trabajan para él, para mantener la integridad referencial). Si no se usa esta cláusula, el gestor no dejará borrar la tupla "padre".

```
CREATE TABLE EMP (
EMPNO NUMBER(4)
ENAME VARCHAR2(10),
JOB VARCHAR2(9),
MGR NUMBER(4) REFERENCES EMP,
HIREDATE DATE,
SAL DECIMAL(7,2),
COMM DECIMAL(7,2),
DEPTNO NUMBER(4),
PRIMARY KEY (EMPNO),
CONSTRAINT fk_deptno FOREIGN KEY (DEPTNO)
            REFERENCES DEPT(DEPTNO)
            ON DELETE CASCADE)
```

El poner nombre a las restricciones es importante por si más adelante se desean modificarlas o eliminarlas.

Por otro lado, aunque Oracle no lo soporta, en SQL2 se permitían otras opciones a la hora de borrar o actualizar una tupla "padre":

```
[ON {DELETE|UPDATE} SET {NULL|CASCADE}]:
```

En caso de borrado o actualización:

- SET NULL: se coloca el valor nulo en la clave externa.
- CASCADE: se borran las tuplas .

Restricciones de unicidad (`UNIQUE`) Una lista de atributos (o uno solo) puede establecerse como único, es decir, que no admita duplicados. Esto es útil, entre otras cosas, para garantizar que se cumplan las claves candidatas, ya que sólo es posible indicar una única clave primaria por tabla. La sintaxis es la siguiente:

```
<descripción del atributo> UNIQUE
```

o

```
[CONSTRAINT <nombre>] UNIQUE(<lista de atributos>)
```

Por ejemplo,

```
DNAME VARCHAR2(14) UNIQUE
```

o

```
CONSTRAINT u_dept_loc UNIQUE(DNAME, LOC)
```

Como se ha indicado anteriormente, una restricción de clave externa puede referenciar una lista de atributos que estén afectados por una restricción `UNIQUE`.

Restricciones de comprobación (`CHECK`) Se usa estas restricciones para realizar comprobaciones sobre los valores que se introducen en la tabla. Dependiendo de si se indica la restricción en la declaración del atributo o de forma aislada, la sintaxis general es la siguiente:

```
<descripción del atributo> CHECK (<condicion>)
```

o

```
[CONSTRAINT <nombre>] CHECK (<condicion>)
```

El nombre de la restricción es similar a las anteriores. La condición será simple (del tipo de las que vimos en las sentencias `SELECT`, aunque sin consultas subordinadas). Las restricciones tipo `CHECK` pueden afectar a más de un atributo. En el caso de que afecten a uno solo, se pueden indicar al final de la descripción del mismo. Veamos dos ejemplos.

```
... SAL NUMBER(7,2) CHECK (SAL > 0)
```

o

```
CONSTRAINT C_comm_job CHECK( (COMM < 2000) AND (JOB <> 'No lo sé'))
```

Como se ve, si la restricción se indica de forma independiente, como la segunda del ejemplo anterior, puede comprobar varias columnas. Si, como en el primer caso, se indica en la declaración de un atributo, sólo puede referenciarlo a él.

2.2. Borrado de Tablas

La sentencia SQL para borrar una tabla es `DROP TABLE <nome_tabla> [CASCADE CONSTRAINTS]`. Nótese que por "borrar" se entiende destruir totalmente el objeto tabla, no sólo los datos que contenga, sino también la definición de su estructura. Por ejemplo,

```
DROP TABLE EMP; DROP  TABLE DEPT;
```

borraría las tablas EMP y DEPT. Cuando se borra una tabla, también se borrarán otras estructuras asociadas a ella, tales como índices, pero para que se borren algunas restricciones (como las claves foráneas que hacen referencia a esa tabla) es necesario especificar [CASCADE CONSTRAINTS], sino el SGBD no permite el borrado de la tabla.

Al igual que en la creación de tablas, existe una restricción en el proceso de borrado de tablas cuando hay claves externas: se debe borrar antes la tabla referenciadora que la referenciada. Tal como se ve en el ejemplo previo, se borra la tabla `EMP` (que tiene una clave externa que referencia a la tabla `DEPT`) antes de borrar la tabla `DEPT`.

2.3. Modifición (Alteración) de Tablas

La estructura de una tabla (creada con la sentencia `CREATE TABLE`) puede ser modificada, añadiendo y/o modificando o borrando atributos o restricciones. Para ello se utiliza la sentencia SQL `ALTER TABLE`. Veamos a continuación las opciones disponibles para esta sentencia.

2.3.1. Alteración de atributos

El estándar SQL2 permite añadir, borrar o modificar un atributo de una tabla. Sin embargo, Oracle, hasta la versión 8.0 inclusive, sólo permite añadirlos y modificarlos, pero no borrarlos. Para añadir un atributo (que se añadirá al final de los ya existentes) se utiliza la siguiente sintaxis:

```
ALTER TABLE <nombre_tabla>
      ADD <Definicion de atributo>;
```

Por ejemplo, si consideramos que la tabla de proveedores debe incluir la dirección del proveedor, podemos añadir ese atributo:

```
ALTER TABLE EMP
      ADD DIRECCION VARCHAR2(50);
```

El nuevo campo se incluye en la estructura de la tabla, evidentemente para todas sus tuplas. Las tuplas ya existentes incluirán un valor nulo en ese atributo, o el valor por defecto si se utiliza la cláusula `DEFAULT`. No sería posible, por lo tanto, añadir un atributo definido como `NOT NULL` y sin un valor por defecto, si la tabla ya contiene datos.

```
ALTER TABLE EMP                              -- Incorrecto!!
      ADD DIRECCION VARCHAR2(50) NOT NULL;

ALTER TABLE EMP                              -- Correcto
      ADD DIRECCION VARCHAR2(50) DEFAULT 'Desconocida' NOT NULL;
```

Para modificar un atributo existente (sólo se puede modificar su tipo, no el nombre), se utilizará

```
ALTER TABLE <nombre_tabla>
      MODIFY <Definicion de atributo existente>;
```

Por ejemplo, para indicar que necesitamos agrandar el atributo dirección hasta 120 caracteres, utilizaremos la sentencia siguiente:

```
ALTER TABLE EMP
      MODIFY DIRECCION VARCHAR2(120);
```

Cuando modificamos el tipo de una columna o atributo, debemos tener en cuenta los siguientes aspectos, que se explicarán tomando como base que la tabla EMP tiene el atributo dirección definido como `DIRECCION VARCHAR2(50) DEFAULT 'Desconocida' NOT NULL`:

- Si la columna tiene valores, no se puede reducir el tamaño de la columna, pero sí aumentarlo.

  ```
  ALTER TABLE EMP                          -- Incorrecto!!
        MODIFY DIRECCION VARCHAR2(10);

  ALTER TABLE EMP                          -- Correcto
        MODIFY DIRECCION VARCHAR2(150);
  ```

- Si la columna tiene valores nulos, no puede especificarse ahora que no se admiten nulos.
- Si se desea indicar que ahora se admiten valores nulos y/o no hay valores por defecto, debe indicarse explícitamente. Para ello, si queremos que ahora acepte nulos debemos indicarlo con `NULL`, y si queremos que no exista un valor por defecto, indicar `DEFAULT NULL` (es decir, el valor por defecto es `NULL`, que es el que toma si no se indica otro).

  ```
  ALTER TABLE EMP     -- Sigue sin aceptar nulos y tiene el mismo
        MODIFY DIRECCION VARCHAR2(120);  -- valor por defecto

  ALTER TABLE EMP     -- Correcto. Ahora admite nulos
        MODIFY DIRECCION VARCHAR2(120) NULL;

  ALTER TABLE EMP     -- Correcto. Ahora no hay valor por defecto (es NULL)
        MODIFY DIRECCION VARCHAR2(120) DEFAULT NULL
  ```

Oracle, a partir de su versión 8*i* (en concreto, 8.1.6), también permite borrar campos de una tabla. Para ello se utiliza la sentencia

```
ALTER TABLE <tabla>
    DROP (<columna>) [CASCADE CONSTRAINTS]
```

Por ejemplo,

```
ALTER TABLE EMP
    DROP (DIRECCION) CASCADE CONSTRAINTS
```

La opción CASCADE CONSTRAINTS es necesaria, por ejemplo, si queremos borrar una columna que es referenciada por una clave externa:

```
SQL> ALTER TABLE EMP DROP (EMPNO);

ALTER TABLE EMP DROP (EMPNO)
                      *
ERROR en línea 1: ORA-12992: no se puede borrar la columna de
claves principales


SQL> ALTER TABLE EMP DROP (EMPNO) CASCADE CONSTRAINTS;

Tabla modificada.
```

En el segundo caso, en el que la sentencia funciona correctamente, la restricción es eliminada. Debemos hacer notar que, sin embargo, para otras restricciones como UNIQUE no es necesaria la cláusula CASCADE CONSTRAINTS, y la restricción asociada se borra igualmente.

Se permite añadir una restricción nueva a una tabla, utilizando la sintaxis siguiente:

```
ALTER TABLE <nombre_tabla>
      ADD <Definicion de restriccion>;
```

Por ejemplo, si hubiésemos creado la tabla EMP sin clave primaria y quisiésemos añadir esa restricción, utilizaríamos la sentencia

```
ALTER TABLE EMP
      ADD CONSTRAINT pk_empleado PRIMARY KEY(EMPNO);
```

Para borrar una restricción, usaremos:

```
ALTER TABLE <nombre_tabla>
      DROP CONSTRAINT <nombre de restriccion>;
```

Es evidente que para poder borrar una restricción debemos conocer su nombre, por lo que es recomendable darles un nombre cuando se crean. Si no se le ha dado nombre, es posible, examinando el catálogo (de Oracle), saber su nombre para poder borrarla, pero se recomienda en todo caso darle un nombre para facilitar estas operaciones. Así, para eliminar la restricción de clave primaria que acabamos de crear, usaríamos lo siguiente:

```
ALTER TABLE PROVEEDOR
      DROP CONSTRAINT pk_empleado;
```

2.4. Creación de índices con SQL

Utilizando SQL, la sentencia básica de creación de un índice es la siguiente:

```
CREATE INDEX <nombre índice> ON <tabla>(<lista de atributos>);
```

Por ejemplo, si planeamos realizar muchas consultas sobre el nombre y el puesto de trabajo, el siguiente índice sería útil.

```
SQL> CREATE INDEX idx_nom_puesto ON EMP(ENAME,JOB);

Índice creado.
```

Este tipo de índices permite valores duplicados, puesto que no se ha indicado lo contrario. Si deseamos crear un índice que no los admita (por ejemplo, si el índice se crea sobre el campo que es la clave primaria de una tabla y que por lo tanto no admite duplicados), utilizaremos la sintaxis siguiente:

```
CREATE UNIQUE INDEX <nombre índice> ON <tabla>(<lista de atributos>);
```

Por ejemplo, podríamos indexar crear un índice sobre la clave primaria de `EMP`, usando .

```
CREATE UNIQUE INDEX idx_pk_EMP ON EMP(EMPNO);
```

Originalmente se utilizaban los índices únicos para controlar la integridad de entidad, es decir, para controlar que no existiesen claves primarias duplicadas. De hecho, los gestores siguen creando un índice de este tipo sobre la clave primaria. Por ello, veamos lo que ocurre si intentamos crear ese índice sobre la tabla EMP donde se definió como clave primaria el atributo EMPNO:

```
SQL> CREATE UNIQUE INDEX idx_pk_EMP ON EMP(EMPNO);

CREATE UNIQUE INDEX idx_pk_dni ON EMP(EMPNO)
                                      *
ERROR en línea 1: ORA-01408: esta lista de columnas ya está indexada
```

Como se ve, esto produce un error. Esto es debido a que existe una restricción (muy lógica), que es que una `<lista de atributos>` sólo puede estar indexada por un único índice, y la creación de una clave primaria implica que el SGBD crea automáticamente un índice único sobre dicha clave. En otro caso estaríamos creando varios índices exactamente iguales, con lo que lo único que conseguiríamos sería desperdiciar espacio.

En el caso de Oracle, hay más tipos de índices soportados, que se crean de forma similar:

```
CREATE <tipo_indice> INDEX <nombre> ON....
```

Entre los tipos soportados se encuentran el tipo `BITMAP` (que no usa árboles B^+ sino un bitmap para almacenar el índice), o `UNSORTED` (que no ordena los datos en base a los campos indexados como hace si no se indica nada).

2.5. Vistas

La definición más simple de una vista es la que nos dice que *una vista es una consulta a la que se da un nombre*. Una vez que la vista se ha creado la definición de la vista, se puede acceder a ella exactamente igual que si fuese una tabla para ver su estructura o para seleccionar los datos que contiene. El mayor problema asociado a las vistas es determinar cuándo los datos de una vista se pueden actualizar (insertar, borrar o modificar filas).

Las tablas indicadas en la consulta que define una vista se denominan *tablas base*, y pueden a su vez ser vistas.

2.5.1. Creación de Vistas

La sentencia utilizada para crear una vista es `CREATE VIEW`, cuya sintaxis general se muestra a continuación.

```
CREATE [OR REPLACE] VIEW <nombre_vista> [( <esquema_vista> )]
   AS <sentencia select>
```

Si indicamos `CREATE VIEW...` y ya existe una vista con ese nombre, el SGBD dará un error. Si indicamos `CREATE OR  REPLACE VIEW...` y la vista existe, se sustituye la declaración de la vista existente por la nueva.

La `<sentencia select>` es una consulta realizada utilizando la sentencia `SELECT`.

El nombre `<nombre_vista>` especifica el nombre que se da a la vista que se crea. Así, por ejemplo, la siguiente sentencia define una vista sobre la tabla `EMP`, extrayendo solamente el EMPNO y nombre.

```
SQL> CREATE VIEW EMPLEADOS
       AS SELECT EMPNO, ENAME
            FROM EMP;

Vista creada.
```

Una vez definida, la vista se puede consultar como si fuese una tabla. Los nombres de los atributos de la vista serán los mismos atributos que se seleccionan de la(s) tabla(s). Así, veamos la vista anterior:

```
SQL> describe EMPLEADOS
 Nombre                                    +Nulo?   Tipo
 ----------------------------------------- -------- ------------
 EMPNO                                     NOT NULL NUMBER(4)
 ENAME                                              VARCHAR2(10)
```

Para renombrar los atributos (o expresiones de la select) se pueden tomar dos alternativas:

1. Especificar un alias para las expresiones que no corresponden a nombres de atributos:

   ```
   SQL> CREATE VIEW V1
          AS SELECT ENAME AS NOMBRE, SAL+COMM AS SALARIO_TOTAL
             FROM EMP;

   Vista creada.
   ```

   En este caso, la vista tendrá un atributo `NOMBRE` y otro `SALARIO_TOTAL`.

2. Utilizar la parte opcional `<esquema_vista>`, indicando (en una lista entre paréntesis) los nombres que se darán a los atributos de la vista que representan las expresiones seleccionadas:

```
SQL> CREATE OR REPLACE VIEW V1 (NOMBRE, SALARIO_TOTAL)
       AS SELECT ENAME, SAL+COMM
          FROM EMP;

Vista creada.
```

El esquema de la vista sería el mismo que en el caso anterior.

En cualquier caso, en la primera de las opciones el uso de alias sólo es necesario para aquellas expresiones de nombre inválido. Es decir, si alguna expresión es válida, no es necesario especificar un alias (aunque sí se puede hacer).

Las consultas pueden ser todo lo complejas que se quieran, incluyendo las cláusulas `WHERE`, `GROUP BY` o `HAVING`. Así, la siguiente consulta nos ofrece una relación en la que se ven los nombres de los departamento, el número de empleados de cada departamento, y el montante total de los salarios que pagan:

```
CREATE OR REPLACE VIEW UNA_VISTA (NOMBRE, NUM_EMP, SALARIO_TOTAL)
       AS SELECT DNAME, COUNT(*), SUM(SAL)
         FROM EMP E INNER JOIN DEPT D ON E.DEPTNO=D.DEPTNO
         GROUP BY DNAME
```

Una consulta sobre esta vista nos daría el resultado siguiente:

```
SQL> SELECT * FROM UNA_VISTA;

NOMBRE            NUM_EMP SALARIO_TOTAL
-------------- ---------- -------------
ACCOUNTING              3          8750
RESEARCH                5         10875
SALES                   6          9400

3 filas seleccionadas.
```

2.5.2. Borrado de vistas

Para borrar una vista (sólo la definición, no los datos) se utiliza la sentencia siguiente:

```
DROP VIEW <nombre vista>
```

Por ejemplo,

```
SQL> DROP VIEW UNA_VISTA;

Vista borrada.
```

2.5.3. Ventajas del uso de vistas

Existen múltiples ventajas derivadas del uso de vistas en la definición de un esquema de bases de datos. Entre ellas destacan las siguientes, de las que se da algún ejemplo:

- Ocultación de información: Un empleado puede gestionar datos de los empleados, pero quizás no debería ver cierta información, como por ejemplo el salario. Para esto se puede definir una vista que obtenga los datos de los empleados, pero no el atributo salario ni comisión. Así, la persona que gestiona los empleados no accede a información a la que no debe.

```
SQL> CREATE OR REPLACE VIEW EMPLEADOS
  2     AS SELECT EMPNO, ENAME, JOB, HIREDATE,
  3               MGR, DEPTNO
  4     FROM EMP;

Vista creada.
```

- Independencia con respecto al diseño de tablas: Si se cambia la estructura de alguna tabla, como por ejemplo de algún atributo de la misma (que non está incluido en la definición de la vista), eso no afecta a los programas que acceden a la vista. Por ejemplo, si añadimos un atributo "fecha de nacimiento" para los empleados, los programas que acceden a la vista `EMPLEADOS` no se verán afectados.

- Facilitar la realización de consultas: Una vista puede definir una consulta relativamente compleja, y se pueden hacer a su vez consultas sobre esta vista. Así, si en algún caso un usuario debe realizar consultas complejas y no es experto en SQL, el uso de una vista que haga parte de la consulta de de gran ayuda. Por ejemplo, en una base de datos de contabilidad, existe una tabla en la que se almacenan los apuntes de los asientos contables. Podríamos definir una vista para calcular los saldos del debe y el haber de las cuentas, y un usuario normal tendría que hacer una consulta realmente simple para obtener el saldo de una determinada cuenta (del tipo `SELECT * FROM VISTA WHERE CUENTA=<cuenta>`) en vez de tener que definir la consulta completa, con sumas, agrupamientos, etc.

2.5.4. Consultas sobre vistas

Como ya se ha indicado, realizar una consulta sobre una vista es exactamente igual (sintácticamente) que realizar una consulta sobre una tabla real.

La ejecución (teórica) de una consulta que usa una vista seguiría 2 pasos:

1. Ejecutar la consulta que define la vista, obteniendo una relación temporal que contiene los datos de la vista.

2. Ejecutar la consulta pedida sobre esa relación temporal.

Lógicamente, la realización de una consulta sobre una visa se traslada a una consulta a las tablas base sobre las que se define la vista. Sin embargo, normalmente, los gestores de

bases de datos implementan algoritmos de optimización sobre las consultas, por lo que las condiciones establecidas en la consulta se suelen añadir a las establecidas en la definición de la vista (automáticamente y de forma transparente para el usuario que tecleó la consulta) de modo que es esta última consulta la ejecutada.

2.5.5. Actualización de vistas

Al igual que cuando se hace una consulta, cuando se lanza una sentencia de actualización de datos (`INSERT`, `UPDATE`, `DELETE`) sobre una vista, el gestor de bases de datos tratará de trasladar esa actualización a la tabla o tablas base sobre las que está definida la vista. Al intentar una actualización, en algunos casos es posible realizarla, pero en otros no es posible y el gestor dará un error.

De forma genérica, podemos dividir las vistas en las siguientes categorías:

Todas las vistas: Todas las vistas que pueden ser definidas.

Vistas teóricamente actualizables: Es un subconjunto del anterior, formado por las vistas que en teoría se podrían actualizar. Es decir, la traducción de actualizar los datos en la vista a actualizar los datos en las tablas base es teóricamente posible.

Vistas realmente actualizables: Es un subconjunto de las vistas teóricamente actualizables, formado por aquellas vistas en realidad son consideradas actualizables por lo gestores de bases de datos. Probablemente cada gestor de bases de datos permitirá la actualización de datos en un subconjunto (reducido) de este conjunto de vistas. Veremos concretamente la reacción de Oracle ante el intento de actualización de varios tipos de vistas.

A continuación veremos varios tipos de consultas que se podrían usar para definir una vista, y qué es lo que pasaría, utilizando Oracle, si tratamos de actualizar datos en la vista. Como regla general, una vista es actualizable si las tuplas de las tablas base son "back-traceable", esto es, cuando es posible identificar la tupla de la tabla base que corresponde a cada tupla de la vista. Sin embargo, veremos por separado si se pueden insertar, borrar o modificar datos de las vistas creadas.

Proyecciones de atributos: Veamos una vista definida proyectando atributos de una sola tabla, y las posibles actualizaciones sobre ella.

Consideremos el siguiente ejemplo de vista, que obtiene el nombre y salario de todos los empleados (es decir, se hace solamente una proyección de datos):

```
SQL> CREATE VIEW V1
  2     AS SELECT ENAME, SAL
  3           FROM EMP;
```

Consideremos las posibles actualizaciones sobre esta vista. Evidentemente, se tratará de traducir todas las actualizaciones sobre la vista a actualizaciones sobre la tabla `EMP`.

- Inserción: Se pueden dar varios casos. Si alguno de los atributos de la tabla base que no están en la vista no admite valores nulos ni tiene valores por defecto, la inserción no es posible (ya que la traducción intentaría insertar valores nulos). Este es el caso de `V1`, ya que el atributo `EMPNO`, que no está en la definición de la vista, no admite valores nulos, por ser la clave primaria.

```
SQL> INSERT INTO V1
  2     VALUES('NOMBRE',0);
INSERT INTO V1
*
ERROR en línea 1:
    ORA-01400: no se puede realizar una inserción NULL en
    ("SCOTT"."EMP"."EMPNO")
```

Como regla general, en el resto de los casos sí es posible la inserción, es decir, cuando los atributos de la tabla base no incluidos en la definición de la vista o bien admiten valores nulos o tienen definidos valores por defecto. Sin embargo, hay algunos casos en los que no es posible.

- Borrado: Siempre es posible, ya que las tuplas de la vista en este caso se corresponden una a una con las de la tabla base. Por lo tanto, las tuplas que se borren de la vista se borran realmente de la tabla base.

  En este sentido, debemos matizar algo. Veamos la siguiente vista y una consulta sobre ella:

```
SQL> CREATE VIEW VSALARIO
  2     AS SELECT SAL
  3          FROM EMP;

Vista creada.

SQL> SELECT * FROM VSALARIO
  2     ORDER BY SAL;

       SAL
----------
       800
       950
      1100
      1250
      1250
      1300
      1500
      1600
      2450
      2850
      2975
      3000
      3000
      5000

14 filas seleccionadas.
```

Como se ve, hay dos tuplas con salario 1250. Si intentamos borrar de la vista `VSALARIO`, no podemos distinguir entre esas dos tuplas, por lo que si ejecutamos la sentencia

```
SQL> DELETE FROM VSALARIO
  2     WHERE SALARIO=1250;

2 filas suprimidas.

SQL> SELECT COUNT(*) FROM EMP;

  COUNT(*)
----------
         12

1 fila seleccionada.
```

Vemos que se borran las dos tuplas correspondientes en la tabla base. Que *nosotros* no podamos diferenciar las tuplas no indica que Oracle no pueda, internamente, hacerlo. Por ello, si la consulta es **sólo** una proyección de atributos, siempre se podrán borrar los datos de ella.

- Modificación: Al igual que el borrado, siempre es posible modificar los datos de una vista de este tipo. Evidentemente, sólo es posible modificar aquellos atributos que aparecen en la definición de la vista.

Sólo selecciones de datos: Consideremos el siguiente ejemplo de vista, que obtiene (todos los atributos) de aquellos empleados que ganan más de 1.250 €. Es importante hacer notar que no se eliminan los duplicados de las filas obtenidas, es decir, no se usa la palabra clave `DISTINCT` en la definición de la vista.

```
SQL> CREATE VIEW VISTA2
  2    AS SELECT *
  3       FROM EMP
  4       WHERE SAL > 1250;
```

Veamos las posibles actualizaciones sobre este tipo de vista:

- Inserción: Siempre es posible insertar tuplas, ya que cada una de las tuplas de la vista se corresponde directamente a una tupla de la tabla base, y se conocen todos los atributos. Sin embargo, las inserciones pueden dar lugar a situaciones extrañas. Por ejemplo, si insertamos una fila en la anterior vista, con un salario menor o igual a 1250, la fila se inserta en la tabla base `EMP`, pero no aparece en la vista ya que no satisface la condición.
- Borrado: También es siempre posible, por el mismo motivo. Las tuplas que se borren de la vista se borran realmente de la tabla base. En este caso las situaciones extrañas del caso de la inserción no suceden, ya que a la condición del borrado, si existe, se añade la condición de la vista antes de realizar el borrado.

```
SQL> DELETE FROM VISTA2;

9 filas suprimidas.

SQL> SELECT COUNT(*) FROM EMP;

  COUNT(*)
----------
         5

1 fila seleccionada.
```

Como se ve, la sentencia borra 9 tuplas, las que aparecen en la vista, pero no las demás tuplas de la tabla `EMP`. De todas formas, se sugiere actuar con cuidado en estos casos y comprobar el comportamiento de cada versión de SGBD.

- Modificación: También es posible siempre modificar los datos de una vista de este tipo, actuando con las mismas consideraciones que con el borrado de filas.

  Además, hay otro tipo de consideraciones. Veamos el siguiente ejemplo:

```
 UPDATE VISTA2 SET SAL=800
     WHERE EMPNO=7499;

1 fila actualizada.

 select ENAME, EMPNO, SAL
  from VISTA2;


ENAME           EMPNO        SAL
---------- ---------- ----------
JONES            7566       2975
BLAKE            7698       2850
CLARK            7782       2450
SCOTT            7788       3000
KING             7839       5000
TURNER           7844       1500
FORD             7902       3000
MILLER           7934       1300

8 filas seleccionadas.
```

El resultado es que la fila indicada se actualiza en la tabla base `EMP`, pero como resultado de ello, esa fila desaparece de la vista (como si fuese borrada) porque ya no satisface la condición de ganar más de 1250 €.

Combinación de proyecciones y selecciones de datos: Combinando selecciones y proyecciones sobre una tabla se pueden obtener vistas con una gran potencia. Los problemas de este tipo de vistas combinarían los aspectos indicados individualmente para las selecciones o las proyecciones.

Consultas con `DISTINCT` o agrupamiento sobre una tabla Consideremos la siguiente definición de vista, que utiliza `DISTINCT` para eliminar tuplas duplicadas:

```
SQL> CREATE VIEW VISTA3
  2    AS SELECT DISTINCT JOB, DEPTNO
  3      FROM EMP;
```

A diferencia de las vistas definidas anteriormente, para esta no es posible saber la tupla original de la tabla base sobre la que se ha definido la vista, ya que se eliminan los duplicados.

- Inserción: No es posible, ya que al intentar trasladar la inserción a la tabla base, no se sabría cómo realizar la inserción. Por ejemplo, no se sabría cuantas copias de la fila se podrían insertar, ya que al usar `DISTINCT` en la definición de la vista se eliminarían los duplicados.

- Borrado: Aunque sería teóricamente posible, borrando todas las copias de la tabla base que dan lugar a cada fila de la vista que se desea borrar, no se permite el borrado, ya que eso no sería el resultado deseado en muchos casos.

- Modificación: Por el mismo motivo, aún siendo teóricamente posible la actualización, no se permite en este tipo de vistas.

Consideremos ahora una vista que incluye agrupamiento y funciones de agregación:

```
SQL> CREATE VIEW EMP_POR_DEPT
  2     AS SELECT DEPTNO, COUNT(*) AS NUMEMP
  3         FROM EMP
  4         GROUP BY DEPTNO;
```

De nuevo, en este caso las tuplas de la vista no se corresponden una a una con las tuplas de la tabla base. Veamos caso por caso por qué las actualizaciones no se podrán realizar:

- Inserción: No es posible, ya que de nuevo al intentar trasladar la inserción a la tabla base no se sabría cómo realizar la inserción. Por ejemplo, intentar ejecutar `INSERT INTO EMP_POR_DEPT VALUES(50,2)` habría que insertar 2 filas en la tabla `EMP`, pero no se sabe qué poner en ninguno de sus atributos excepto el número de departamento.

- Borrado: Sería teóricamente posible borrar las filas de la vista, pero por ejemplo intentar ejecutar `DELETE FROM EMP_POR_DEPT  WHERE DEPTNO=30`, dado que la vista sólo nos dice cuántos empleados tiene el departamento, no parece lógico borrar todas las filas del la tabla `EMP` de ese departamento. Por lo tanto, el borrado no se permite en vistas definidas con agrupamientos y/o funciones de agregación.

- Modificación: Tampoco se permiten las modificaciones en este tipo de vistas. En algún caso sería teóricamente posible, por ejemplo, `UPDATE EMP_POR_DEPT SET DEPTNO=50 WHERE DEPTNO=30`, pero el resultado, que sería modificar el número de departamento de los empleados correspondientes, no sería lógico. Otro tipo de modificaciones, como `UPDATE EMP_POR_DEPT SET NUMEMP=2 WHERE DEPTNO=10`, se podría llevar a cabo borrando una fila de la tabla `EMP` de ese departamento (ya que actualmente tiene 3), pero tampoco sería una actuación correcta, por lo que no se permite.

Por lo tanto, como se ha visto, ninguna vista definida utilizando `DISTINCT` ni agrupamientos y/o funciones de agregación permite la actualización de los datos directamente a través de la vista.

Vistas definidas sobre más de una tabla Es común definir vistas sobre más de una tabla, haciendo joins entre varias. Por ejemplo, la siguiente vista nos daría los nombres de empleados y los nombres de los departamentos a los que pertenecen:

```
SQL> CREATE VIEW EMP_DEPT
  2     AS SELECT ENAME, DNAME
  3         FROM EMP E, DEPT D
  4         WHERE E.DEPTNO=D.DEPTNO;
```

Según el estándar SQL2 (el que *supuestamente* sigue Oracle), cuando hay más de una tabla en la cláusula `FROM` de la sentencia `SELECT` que define la vista, no es posible realizar ningún tipo de actualización directamente sobre la vista. La razón es que no hay forma genérica fiable de trasladar los cambios deseados de la vista a las tablas base. Oracle permite, bajo algunas circunstancias, hacer algunas actualizaciones sobre este tipo de vistas, pero ya que no es estándar y es algo que puede cambiar, no entraremos en detalles sobre eso y consideraremos que no se pueden realizar actualizaciones sobre vistas definidas sobre más de una tabla.

2.5.6. With check option

Hemos visto que a la hora de actualizar, insertar o borrar filas se pueden producir fenómenos extraños como insertar una fila en la vista que luego no es visible en dicha vista o actualizar una fila y que desaparezca de la vista.

Esto es así porque las filas existen en una vista porque satisfacen la condición `WHERE` de la consulta de definición. Si una fila es modificada de modo que ya no satisface la condición `WHERE`, entonces desaparecerá de la vista. Del mismo modo, aparecerán nuevas filas en la vista cuando una inserción o actualización haga que dichas filas satisfagan la condición `WHERE`. Estas filas que entran o salen de la vista se llaman *filas emigrantes*.

En el estándar de SQL se introdujo una cláusula opcional en la sentencia de creación de vistas para controlar estas filas. El formato la sentencia de creación con esta variación sería:

```
CREATE [OR REPLACE] VIEW <nombre_vista> [( <esquema_vista> )]
   AS <sentencia select>
   [WITH [CASCADED|LOCAL] CHECK OPTION]
```

Generalmente, la cláusula `WITH CHECK OPTION` prohíbe la migración de tuplas hacia o desde la vista. Los cualificadores `LOCAL/CASCADED` son aplicables a jerarquías de vistas: es decir, vistas que son derivadas de otras vistas. En este caso, si se especifica `WITH LOCAL CHECK OPTION` en la definición de una vista, entonces cualquier fila insertada o actualizada en la vista y sobre cualquier vista directamente o indirectamente definida sobre dicha vista no debe causar la migración de la fila en la vista, a no ser que la fila también desaparezca de la tabla/vista/s de la que es creada.

Si se especifica `WITH CASCADED CHECK OPTION` en la definición de la vista (modo por defecto), entonces la inserción o actualización sobre esta vista o sobre cualquier vista directamente o indirectamente definida sobre la vista en cuestión no puede provocar la migración de filas.

Esta opción es útil porque cuando un `INSERT` o un `UPDATE` sobre la vista viola la condición `WHERE` de la definición de la vista, la operación es rechazada. `WITH CHECK OPTION` sólo se puede especificar en vistas actualizables.

2.6. Seguridad y permisos (en Oracle)

2.6.1. Los conceptos de "Role" y "PUBLIC"

Un *role* en un sistema de bases de datos puede entenderse como cualquiera de los siguientes dos conceptos, que son equivalentes:

- Un conjunto de permisos.
- Un papel que un usuario desempeña ante la base de datos.

Por ejemplo, un usuario o conjunto de usuarios pueden desempeñar el papel de "facturador" en una empresa, con lo que entenderíamos el role como el segundo concepto, el papel que desempeñan esos usuarios. Equivalentemente, podríamos verlo como un conjunto de permisos, los necesarios para que estos usuarios realicen su labor. Por ejemplo, podrían acceder (seleccionar) datos en una tabla de facturas, y actualizar los campos que indican que la factura ha sido emitida o cobrada, pero no podrían acceder a la tabla de empleados para ver sus salarios.

Un role se crea con la sentencia `CREATE ROLE <nombre de role>`. Por ejemplo, `CREATE ROLE facturador`.

Como veremos, un usuario de la base de datos puede actuar bajo más de un role. Existe el comando `SET ROLE <nombre de role>` que hace que el usuario adquiera los permisos asociados a ese role (si tiene permiso para adquirir ese role) y pierda los permisos asociados al role anterior.

Existen varios roles predefinidos en Oracle. Entre ellos destacan `CONNECT` (el más básico), `RESOURCE`, o `DBA` (el máximo). Si un usuario intenta cambiar de role usando uno para el que no tiene permiso, recibirá un mensaje de error y el role no se cambia. Puede hacerse una prueba con un usuario normal al intentar ejecutar `SET ROLE DBA`.

Existe además un role especial llamado `PUBLIC`, al que tienen acceso todos los usuarios. Representa al usuario "genérico" de la base de datos, lo que engloba a todos los usuarios. Cuando un permiso se da a `PUBLIC`, se da a todos los usuarios de la base de datos. A veces se considera a `PUBLIC` como un usuario, sin embargo hay que tener en cuenta que no lo es realmente, ya que no sirve para identificarse ante la base de datos y conectarse a ella. Para esto es necesario disponer de un login y una clave "normales".

2.6.2. Gestión de permisos

Los permisos de un gestor de bases de datos se pueden clasificar en dos tipos:

- Permisos que afectan a todo el sistema: por ejemplo, la posibilidad de crear o borrar usuarios, crear una base de datos, etc.
- Permisos que afectan a un objeto de la base de datos: insertar datos en una tabla, ejecutar un procedimiento almacenado, etc.

De todas formas, las sentencias para dar o retirar permisos serán las mismas para ambos tipos, aunque con ligeras variaciones de sintaxis. Para dar un permiso se utiliza la sentencia `GRANT`, y para retirarlo la sentencia `REVOKE`. De la misma forma, dado que un role se puede ver como un conjunto de permisos, se usan las mismas sentencias para dar o retirar los permisos que tiene un role a otro role o a un usuario. Evidentemente, un usuario sólo podrá dar o quitar permisos que tenga (y que tenga el poder de dar o quitar).

2.6.3. Permisos globales

Los permisos que afectan a todo el sistema se pueden ver en la tabla `SYSTEM_PRIVILEGE_MAP` del catálogo de la base de datos. La sentencia `GRANT` para dar permisos de este tipo tiene el siguiente formato:

```
GRANT {Permiso|Role} [, {Permiso|Role}...]
   TO {usuario|Role|PUBLIC} [, ...]
   [WITH ADMIN OPTION]
```

Es decir, se puede dar un permiso, un conjunto de permisos individuales, o uno o varios roles, a una lista de usuarios, roles, o a `PUBLIC`.

Por ejemplo, la siguiente sentencia da los permisos `DROP ANY TABLE` (borrar cualquier tabla) y `CREATE USER` (crear usuarios), además del role `CONNECT` (permiso para conectarse a la base de datos) a los usuarios `scott` y `maria`.

```
GRANT CONNECT, DROP ANY TABLE, CREATE USER
   TO scott, maria
```

La cláusula opcional `WITH ADMIN OPTION` se usa para dar un permiso con la posibilidad de que el receptor pueda dar ese permiso a otros.

Para revocar un permiso se utiliza la sentencia `REVOKE`:

```
REVOKE {Permiso|Role|ALL PRIVILEGES} [, {Permiso|Role}...]
   FROM {usuario|Role|PUBLIC} [, ...]
```

La siguiente sentencia evita que el usuario `scott` se pueda conectar a nuestra base de datos:

```
REVOKE CONNECT
   FROM scott
```

Debemos tener además en cuenta un aspecto sobre los usuarios: los permisos individuales a un usuario toman precedencia sobre los roles, en particular sobre `PUBLIC`. Así, por ejemplo, si ejecutamos la siguiente secuencia:

```
GRANT CONNECT TO scott; REVOKE CONNECT FROM PUBLIC;
```

Impide que la generalidad de los usuarios se conecten a la base de datos, pero como el usuario `scott` tiene ese permiso dado explícitamente a él, sí podrá conectarse.

Si se utiliza `REVOKE ALL PRIVILEGES`, el usuario o role perderá todos los permisos.

Permisos a nivel de tabla El segundo tipo de permisos afectaban a objetos individuales de la base de datos. Dado que los más importantes afectan a las tablas, veremos estos en profundidad.

Cuando un usuario crea una tabla, será su dueño o *owner*, por lo que tiene todos los permisos sobre ella, y la potestad de traspasarlos a otros usuarios, roles o a `PUBLIC`. También podrá darlos aquél usuario que los reciba con el poder de traspasarlos. Los permisos que afectan a las tablas pueden verse en la tabla `TABLE_PRIVILEGE_MAP` del catálogo de la base de datos.

La sentencia básica para dar permisos es de nuevo `GRANT`, ahora con la siguiente sintaxis:

```
GRANT Permiso [, Permiso...]
   ON <nombre tabla>
   TO {usuario|Role|PUBLIC} [, ...]
   [WITH GRANT OPTION]
```

Los permisos a nivel de tabla son básicamente de seleccionar, insertar, modificar y borrar datos, alterar tablas, etc. (`INSERT`, `UPDATE`, `DELETE`, `ALTER TABLE`, ...). Para algunos de ellos, como por ejemplo el de `UPDATE`, se puede incluso especificar qué atributos de la tabla se pueden actualizar. En cambio, el permiso `SELECT` no tiene esa capacidad.

Si un permiso se da `WITH GRANT OPTION`, el usuario que lo recibe lo podrá dar a otros.

La siguiente sentencia da permisos para seleccionar cualquier dato y actualizar sólo el salario de la tabla `EMP`, al usuario `scott` y al role `facturador`. Además estos podrán a su vez dar esos permisos a otros.

```
GRANT SELECT, UPDATE (SAL)
   ON EMP
   TO scott, facturador
   WITH GRANT OPTION
```

Para sacar un permiso, de nuevo usaremos la sentencia `REVOKE`:

```
REVOKE {Permiso|ALL} [, {Permiso}...]
   ON <tabla>
   FROM {usuario|Role|PUBLIC} [, ...]
```

La siguiente sentencia impide que el usuario `scott` inserte datos en la tabla `EMP`.

```
REVOKE INSERT
   ON EMP
   FROM scott
```

De nuevo, si se indica `REVOKE ALL ...` el usuario o role perderá todos los permisos que tenía sobre la tabla.

Cuando se retira un permiso que fue dado con la opción `WITH GRANT OPTION` los permisos se retiran en cascada. Veamos un ejemplo, en el que se indica entre paréntesis el usuario que ejecuta cada sentencia:

```
(DBA)    GRANT SELECT ON EMP TO scott WITH GRANT OPTION;
(scott)  GRANT SELECT ON EMP TO miguel;
```

Ahora el usuario `miguel` puede acceder a la tabla `EMP`, pero tras la siguiente sentencia

```
(DBA)    REVOKE SELECT ON EMP FROM scott
```

tanto `scott` como `miguel` dejan de tener los permisos para acceder a la tabla `EMP`.

2.7. Transacciones

Un aspecto importante a decidir es el nivel o granularidad al que se bloquea. El nivel de bloqueo mayor es el que bloquea la base de datos completa, y que suele usarse sólo para tareas de administración y/o realización de copias de seguridad de la misma. Un nivel algo inferior será bloquear sólo la tabla afectada por la transacción, pero aún así es un nivel demasiado alto. Un nivel más bajo es el de bloqueo de página, que bloquea la página física en disco, y finalmente, el nivel de bloqueo de granularidad más fina es a nivel de tupla: sólo se bloquean aquellas tuplas que se van a leer o modificar. Los bloqueos pueden hacerse de forma explícita, pero normalmente, si no se indica, el gestor de bases de datos lo hará automáticamente dependiendo de las sentencias que ejecutemos contra la base de datos.

En Oracle, cualquier acceso a la base de datos está dentro de una transacción. La primera se inicia en el momento de la conexión (por ejemplo si usamos el comando `CONNECT` desde `SQL*Plus`). Cuando se acaba una transacción, automáticamente comienza la siguiente.

Hay dos formas de acabar una transacción:

- `COMMIT`: Acabar "aceptando los cambios". Todas actualizaciones indicadas desde el inicio de la transacción se graban definitivamente en la base de datos.

- `ROLLBACK`: Acabar descartando los cambios, es decir, todos las actualizaciones a la base de datos desde el inicio de la transacción se deshacen, quedando como antes del inicio de la transacción.

Los comandos `COMMIT` y `ROLLBACK` pueden ejecutarse desde `SQL*Plus`, con el significado anteriormente indicado. Además, existe a mayores un comando para crear puntos de seguridad:

```
SAVEPOINT <nombre de punto de seguridad>
```

que da un nombre a un punto específico dentro de una transacción. En cualquier momento de la transacción podemos volver al estado en que estaba la base de datos en ese punto, utilizando el comando

```
ROLLBACK TO <nombre de punto de seguridad>
```

Finalmente, debemos hacer notar que las transacciones sólo afectan a las sentencias DML de manipulación de datos, tales como `INSERT`, `UPDATE` o `DELETE`, pero no a sentencias DML como la de `CREATE TABLE`.

Ejemplo:

```
SQL> -- Creamos una tabla SQL> CREATE TABLE tabla(
  2       CAMPO int  );
Tabla creada.

SQL> -- la sentencia DML no se ve afectada por el siguiente rollback
SQL> ROLLBACK;

Rollback terminado.

SQL> DESCRIBE tabla;
 Nombre                                            ¿Nulo?   Tipo
 ------------------------------------------------- -------- -----------
 CAMPO                                                      NUMBER(38)

SQL> --Introducimos datos SQL> INSERT INTO TABLA (CAMPO) VALUES
(1);

1 fila creada.

SQL> INSERT INTO TABLA (CAMPO) VALUES (2);

1 fila creada.

SQL> SELECT * FROM TABLA;

     CAMPO
----------
         1
         2

2 filas seleccionadas.

SQL> -- Si hacemos ROLLBACK, no se insertan los datos anteriores
SQL> ROLLBACK;

Rollback terminado.

SQL> SELECT * FROM TABLA;

ninguna fila seleccionada

SQL> INSERT INTO TABLA (CAMPO) VALUES (10);

1 fila creada.

SQL> INSERT INTO TABLA (CAMPO) VALUES (20);
```

```
1 fila creada.

SQL> SELECT * FROM TABLA;

     CAMPO
----------
        10
        20

2 filas seleccionadas.

SQL> -- Validamos los datos, terminando la transacción
SQL> COMMIT;

Validación terminada.

SQL> -- Insertamos un dato más
SQL> INSERT INTO TABLA (CAMPO) VALUES(66);

1 fila creada.

SQL> -- Creamos punto de seguridad
SQL> SAVEPOINT Despues_del_66;

Punto de grabación creado.

SQL> INSERT INTO TABLA (CAMPO) VALUES(67);

1 fila creada.

SQL> -- Creamos otro punto de seguridad
SQL> SAVEPOINT Despues_del_67;

Punto de grabación creado.

SQL> INSERT INTO TABLA (CAMPO) VALUES(68);

1 fila creada.

SQL> -- Volvemos al segundo punto de seguridad SQL> ROLLBACK TO
Despues_del_67;

Rollback terminado.

SQL> SELECT * FROM tabla;
     CAMPO
----------
```

```
        10
        20
        66
        67
4 filas seleccionadas.

SQL> DISCONNECT; Desconectado de Oracle9i Enterprise Edition
Release 9.2.0.1.0 - Production With the Partitioning, OLAP and
Oracle Data Mining options JServer Release 9.2.0.1.0 - Production
```

Como nota final, debemos indicar que el comando `DISCONNECT`, que nos desconecta de la base de datos, hace un `COMMIT` implícito, pero si cerramos el programa que accede a la base de datos de forma anormal (por ejemplo, pulsando en la "X" que cierra la ventana de `SQL*Plus WorkSheet` en Windows, o si cerramos un `xterm` en donde estemos ejecutando `sqlplus`), se hace un `ROLLBACK` implícito.

Si estamos trabajando en `SQL*Plus` podemos activar la opción `AUTOCOMMIT`, que hace un `COMMIT` automático después de cada sentencia de actualización.

```
SQL> SHOW AUTOCOMMIT autocommit OFF
SQL> SET AUTOCOMMIT ON
SQL> INSERT INTO TABLA(CAMPO) VALUES (111);

1 fila creada. Validación terminada.
```

3. El catálogo

El catálogo o diccionario del sistema contiene multitud de tablas y vistas que permiten administrar el sistema. Las tablas y vistas varían de un producto a otro.

Nosotros sólo vamos a repasar unas pocas tablas del catálogo de Oracle que nos pueden resultar imprescindibles.

Podemos consulta las tablas que componen el catálogo accediendo a la tabla `DICT`:

```
SQL> describe dict
Nombre                                    ¿Nulo?   Tipo
----------------------------------------- -------- ----------------------------
TABLE_NAME                                         VARCHAR2(30)
COMMENTS                                           VARCHAR2(4000)
```

Como se puede observar, para cada tabla hay una descripción. Esta tabla, como todas las del catálogo se consulta con una sentencia `Select` normal.

```
SQL> select * from dict where TABLE_NAME='USER_TABLES';

TABLE_NAME  COMMENTS
----------- -------------------------------------------------
USER_TABLES Description of the user's own relational tables
```

USER_TABLES como vemos es otra tabla fundamental, contiene gran cantidad de información sobre las tablas del usuario.

```
SQL> describe USER_TABLES
 Nombre                                    ¿Nulo?   Tipo
 ----------------------------------------- -------- ----------------------------
 TABLE_NAME                                NOT NULL VARCHAR2(30)
 TABLESPACE_NAME                                    VARCHAR2(30)
 CLUSTER_NAME                                       VARCHAR2(30)
...
```

La tabla ALL_TABLES contiene la misma información pero de todas las tablas del sistema.

Para acceder a las restricciones que se han especificado sobre las tablas que hemos creado podemos acceder a la tabla USER_CONSTRAINTS:

```
SQL> describe USER_CONSTRAINTS
 Nombre                                    ¿Nulo?   Tipo
 ----------------------------------------- -------- ----------------------------
 OWNER                                     NOT NULL VARCHAR2(30)
 CONSTRAINT_NAME                           NOT NULL VARCHAR2(30)
 CONSTRAINT_TYPE                                    VARCHAR2(1)
 TABLE_NAME                                NOT NULL VARCHAR2(30)
...
```

Aquí podemos obtener el nombre de la restricción en caso de que no lo especificáramos cuando creamos la restricción. Este nombre es fundamental para poder manipular dicha restricción con ALTER TABLE como hemos visto.

El atributo CONSTRAINT_TYPE es un carácter con el siguiente significado:

Carácter	Constraint Type
C	Check constraint (incluye NOT NULL)
P	PRIMARY KEY constraint
R	FOREIGN KEY constraint
U	UNIQUE constraint
V	WITH CHECK OPTION constraint (para vistas)

Una vez más accediendo a ALL_CONSTRAINTS podemos consultas todas las restricciones del sistema.

Otras tablas interesantes son: ALL_USERS que contiene información de los usuarios del sistema, USER_INDEXES (ALL_INDEXES los del sistema en general) con información de los índices, TABLE_PRIVILEGES y COLUMN_PRIVILEGES que muestran los privilegios sobre tablas y columnas respectivamente y, USER_VIEWS (ALL_VIEWS) que muestran información sobre las vistas.