2 SQL – Un lenguaje de bases de datos relacionales

2.1 Introducción

El SQL (Structured Query Language) es un lenguaje estándar que se utiliza para definir y usar bases de datos relacionales. Originalmente se desarrolló en IBM's San Jose Research Laboratory a mediados y finales de los 70, en el proyecto System-R, y desde entonces ha sido adoptado y adaptado por muchos sistemas administradores de bases de datos relacionales. Ha sido aprobado como el lenguaje relacional oficial por el American National Standards Institute (ANSI) y la International Organization for Standardization (ISO).

El estándar más reciente es del año 2008 (17 de julio de 2008) y se conoce como SQL:2008. Sólo tiene algunas diferencias, no significativas en principio, con el estándar anterior SQL:1999 (también conocido como SQL3), en el cual se definieron por primera vez los conceptos del modelo objeto-relacional. La mayoría de los DBMS relacionales actuales principales (Oracle, DB2, SQL Server, Sybase, Informix) brindan las funcionalidades indicadas en este estándar; algunos, como Access, sólo proporcionan las establecidas en el estándar previo SQL-92 (del año 1992) que contempla sólo cuestiones del modelo relacional (sin objetos). En general, todos los DBMS brindan un conjunto central de facilidades definidas en el estándar, llamado SQL núcleo, agregando, o no, más facilidades según el manejador.

Este lenguaje es **declarativo** y se utiliza no sólo para hacer consultas a una base de datos, esto es para recuperar datos, sino también para definir la base de datos y los objetos que la componen, agregar, eliminar y/o modificar datos, así como para realizar otras funciones relacionadas con la manipulación de una base de datos (por ejemplo, cambiar su estructura).

El lenguaje brinda todos los elementos requeridos tanto para la programación como para la administración de las bases de datos, a fin de que cualquier usuario trabaje con una base de datos, sea éste programador de aplicaciones, administrador de las bases de datos, gerente o usuario final. Entre estos elementos se encuentran: los que definen o modifican la estructura de la base, los que manipulan la información (por ejemplo, hacer consultas), los que controlan las restricciones de integridad, los que manejan las transacciones, los que crean los grupos de usuarios, los que controlan la seguridad de la base, y muchos otros que permiten realizar casi cualquier actividad con una base de datos relacional.

El SQL no permite elaborar formas de edición de datos, reportes, menús de opciones, etc.; ya que éstas son funciones que corresponden a un lenguaje de 4a. generación. Entonces, el desarrollo normal de una aplicación completa que incluya la utilización de un DBMS de esta naturaleza implicaría, primeramente, el desarrollo de la base de datos usando este software y luego, la elaboración de las interfaces de entrada/salida (por ejemplo, formas y reportes) con un lenguaje de 4a. generación que se pueda acoplar al DBMS (por ejemplo, Visual Basic o Java). Naturalmente, este tipo de aplicaciones está relacionado con problemas que requieren de manejar volúmenes muy grandes de información; para cantidades pequeñas, basta con DBMS's para micros o estaciones de trabajo.

En las siguientes páginas se describirán las características principales del SQL estándar y se comentarán algunas extensiones que se manejan en el SQL de Oracle.

2.2 Elementos básicos

En esta sección se describen los elementos básicos que son utilizados en la mayoría de las instrucciones que conforman a SQL. Primeramente se comentan los tipos de datos principales que reconoce Oracle, después se hacen comentarios sobre las expresiones aritméticas y, finalmente, se describen las expresiones condicionales que acompañan a muchas de las instrucciones de SQL.

2.2.1 Tipos de datos

Como todo lenguaje de programación, SQL brinda un conjunto de tipos de datos a usar para la declaración de los dominios de las tablas, principalmente. Los principales tipos de datos que brinda el estándar, y que reconoce Oracle, son:

char(n) – para cadenas de longitud fija de *n* caracteres.
varchar(n) – para cadenas de longitud variable de hasta *n* caracteres.
number(p,s) – para números de punto decimal fijo con *p* cantidad de dígitos, de los cuales *s* están a la derecha del punto decimal.
int, **smallint** – para números enteros.
real, **double precision** – para números con punto flotante.
date – para fechas y horas.

2.2.2 Expresiones

Dentro de este término se engloba a una serie de expresiones especificadas en SQL. Dentro de ellas se pueden destacar a las expresiones aritméticas y a las de concatenación. Las primeras son las típicas de un lenguaje de programación y son las que están formadas por combinaciones de constantes, variables, nombres de columnas de tablas o funciones, unidas por los operadores aritméticos conocidos. Las segundas, son las que se utilizan para concatenar cadenas de caracteres requiriendo de un operador especial para ello (por ejemplo, en Oracle se usa ||). Ejemplos:

```
Sqrt(144) + 72
Avg(Historial.Calif)
Mater.Creds * 1000
('Clara ' || 'González')
```

(Los dos primeros ejemplos usan funciones de SQL; el segundo y tercero emplean columnas de tablas y el último es un ejemplo de concatenación).

Como caso particular de una expresión estarían las literales. Ejemplos en Oracle son:

numéricas: 123, 34.9, 6.023E23, -0.25
cadenas: 'ejemplo'
fecha hora: date '2011-09-20 12:00'

Como particularidad del último caso, cuando sólo se da la fecha, la hora por default es: 00:00.

2.2.3 Expresiones condicionales

Una expresión condicional es una expresión lógica que SQL evalúa a falso, verdadero o desconocido (esto último, cuando se encuentra un valor *null* en la expresión). Estas expresiones se utilizan en frases de SQL para expresar condiciones que se deben cumplir y que normalmente se relacionan con la manipulación de tuplas en relaciones. En SQL, estas expresiones también se conocen con el nombre de *predicados* y son las siguientes.

Expresión de comparación

Sirve para comparar dos expresiones utilizando un operador de comparación:

expresión operador_comparación expresión

donde *expresión* es una expresión aritmética tal como se describió en 2.2.2; y *operador_comparación* es uno de los operadores:

=, <>, <, <=, >, >=, !=, !>, !<

Expresión con *between*

Da un valor verdadero si *expresión1* es mayor o igual que *expresión2* y menor o igual que *expresión3*:

expresión1 [not] between *expresión2* and *expresión3*

Expresión con *in*

Da un valor verdadero si *expresión1* es igual a alguna de las expresiones que están en la lista especificada después de **in** (incluyendo subconsultas):

expresión1 [not] in {*expresión2* | ({*expresión3* | *subconsulta1*}
[,{*expresión4* | *subconsulta2*}]...)}

Expresión con *like*

Da un valor verdadero si el *patrón* especificado es encontrado en una subcadena de la primera expresión. En el *patrón* se pueden usar los caracteres especiales:

% para indicar cualquier cadena de cero o más caracteres
_ para indicar cualquier carácter

expresión [not] like *patrón*

Expresión con *null*

Da un valor verdadero si *expresión* produce un valor nulo:

expresión is [not] null

Expresión lógica

Sirve para conectar lógicamente a varias expresiones condicionales. Los operadores lógicos son los habituales: **and**, **or** y **not**. Su precedencia de evaluación, cuando aparecen en la misma expresión, es: **not**, **and** y **or**. Se pueden usar paréntesis para dar otro orden de evaluación:

[not] [(] *expre_cond* [)] [{ and | or } [not] [(] *expre_cond* [)]] ...

Expresión de cuantificación (operadores: *all*, *any*)

El operador **all** da un valor verdadero si *expresión operador_comparación* es cierto para todas las tuplas de la tabla de resultados de la subconsulta o si la tabla está vacía. El predicado **any** da un valor verdadero si *expresión operador_comparación* es cierto para al menos una tupla de la tabla de resultados de la subconsulta:

expresión operador_comparación { all | any } (*subconsulta*)

2.3 Definición del esquema lógico (conceptual) de una base de datos

2.3.1 Instrucción **create table**

Esta instrucción es la que se utiliza para definir las tablas del esquema lógico de una base de datos relacional. Cada tabla definida con esta instrucción corresponde a un esquema relacional de la base de datos; por lo tanto, deben definirse tantas tablas como esquemas relacionales se hayan obtenido previamente para la misma. Su sintaxis es la siguiente:

create table *nom_tabla*
(*columna_1 tipo_de_datos* [**default** {*literal* | **null**}]
[**primary key** , **not null** , **unique** ,
references *nom_tabla_n* [(*atributo_i*)] ,
check (*expre_cond*)]
[, *columna_2* .... , ...])

donde: *tipo_de_datos*: cualquiera de los válidos en el DBMS en cuestión.
literal: cualquiera de las válidas en el DBMS en cuestión.
expre_cond: cualquiera de las expresiones condicionales descritas en 2.2.3.

Existen variantes de algunos componentes de esta instrucción:

- Para una clave primaria de varios atributos: primary key (col1, col2, …)
- Para una clave externa de varios atributos: foreign key (col1, col2, …) references tabla
- Para un conjuntos de atributos con valores únicos por tupla: unique (col1, col2, …)

2.3.2 Definición de la base de datos del mini-sistema escolar

```
--      Esquema relacional de la base de datos:
--              Prof(IdProf, NomProf, Categoría)
--              Alum(CU, NomAl, Carr, Prom)
--              Mater(ClaveM, NomMat, Creds)
--              Grupo(ClaveG, Salón, IdProf(FK), ClaveM(FK))
--              Inscrito(CU(FK), ClaveG(FK))
--              Historial(Folio, Calif, Fecha, CU(FK), ClaveM(FK))
--
--      Definición de tablas:

create table Prof
    (IdProf         smallint        primary key,
    NomProf         char(30),
    Categoría       char(2)         check (Categoría in ('tc','mt','tp')));

create table Alum
    (CU             smallint        primary key,
    NomAl           char(30),
    Carr            char(3)         check (Carr in ('mat','com','eco','tel')),
    Prom            real            check (Prom is null
                                    or Prom between 6 and 10));

create table Mater
    (ClaveM         smallint        primary key,
    NomMat          char(30)        unique,
    Creds           smallint        check (Creds between 2 and 12));

create table Grupo
    (ClaveG         char(6)         primary key,
    Salón           char(8)         default null,
    IdProf          smallint        references Prof  null,
    ClaveM          smallint        references Mater not null);

create table Inscrito
    (CU             smallint        references Alum,
    ClaveG          char(6)         references Grupo,
                                    primary key (CU,ClaveG));

create table Historial
    (Folio          int             primary key,
    Calif           smallint        check (Calif between 5 and 10),
    Fecha           date            check (Fecha > Date '1990-01-01'),
    CU              smallint        references Alum not null,
    ClaveM          smallint        references Mater not null);
```

2.3.3 Creación genérica de tablas

En esta parte se muestra la creación genérica de tablas de los diversos conceptos de entidad-vínculo y modelo relacional, tomando como base los diagramas y esquemas relacionales de la sección 3.4 de C0ModeloRelacional. En los campos de las tablas se usan diversos tipos de datos simplemente para mostrar las alternativas que brinda SQL para aquellos.

1. Vínculo 1-1

Alternativa 1	**Alternativa 2**
create table A (A1 int primary key, A2 real, -- Vínculo 1-1: B1 int references B unique); -- Nota: primero se debe crear B y -- luego A. create table B (B1 int primary key, B2 real);	create table A (A1 int primary key, A2 real); create table B (B1 int primary key, B2 real, -- Vínculo 1-1: A1 int references A unique);

2. Vínculo 1-N

create table A (A1 int primary key, A2 date);	create table B (B1 int primary key, B2 number(10,2), -- Vínculo 1-N: A1 int references A);

3. Vínculo M-N

create table A (A1 int primary key, A2 char(30)); create table B (B1 int primary key, B2 varchar(30));	-- Vínculo M-N: create table R (A1 int references A, B1 int references B, primary key (A1, B1), -- Puede haber atributos adicionales: R1 smallint);

4. Vínculo ISA

-- Tipo base: create table A (A1 int primary key, A2 real, -- Atributos comunes. Tipo char(3)); -- Tipo derivado: create table B (A1 int primary key references A, B2 real, -- Atributos diferentes. B3 date);	-- Tipo derivado: create table Y (A1 int primary key references A, Y2 date, -- Atributos diferentes. Y3 varchar(30));

5. Entidad débil

```
-- Tipo dueño:
create table A
(A1      int        primary key,
 A2      real);
```

```
-- Tipo dependiente (débil):
create table B
(A1      int        references A,
 B1      int,
                     primary key (A1, B1),
 B2      number(10,2));
```

6. Vínculos recursivos

```
-- Vínculo 1-1:
create table A
(A1      int        primary key,
 A2      char(20),
 A1Bis   int        references A unique);
```

```
-- Vínculo 1-N:
create table A
(A1      int        primary key,
 A2      varchar(20),
 A1Bis   int        references A);
```

```
create table A
(A1      int        primary key,
 A2      real);
```

```
-- Vínculo M-N:
create table R
(A1      int        references A,
 A1Bis   int        references A,
                    primary key (A1, A1Bis),
-- Puede haber atributos adicionales:
 R1      number(10,2));
```

7. Vínculo ternario (sólo dos casos)

```
create table A
(A1      int        primary key,
 A2      char(30));

create table B
(B1      int        primary key,
 B2      varchar(30));

create table C
(C1      int        primary key,
 C2      date);
```

```
-- Vínculo 1-N-1 (un caso):
create table R
(A1      int        references A,
 B1      int        references B,
 C1      int        references C,
                    primary key (A1, B1),
 -- Puede haber atributos adicionales:
 R1      smallint);


-- Vínculo L-M-N:
create table R
(A1      int        references A,
 B1      int        references B,
 C1      int        references C,
                    primary key (A1, B1, C1),
 -- Puede haber atributos adicionales:
 R1      int);
```

2.4 Instrucciones para actualizar los datos de la base de datos

2.4.1 Instrucción **insert**

Esta instrucción se utiliza para agregar tuplas a una tabla. Su sintaxis (básica) es:

insert into *nom_tabla* [(*lista_de_columnas*)]
values (*lista_de_valores*)

donde:
lista_de_columnas: representa a las columnas de la tabla (separadas por coma).
lista_de_valores: representa los valores de la tupla a insertar.

Ejemplos:

```
insert into Grupo (ClaveG, Salón,ClaveM)
   values ('300-1','CC102',300)

insert into Prof
   values (5, 'Raúl', 'mt')
```

2.4.2 Instrucción **update**

Sirve para modificar los valores de las columnas de una tabla. Su sintaxis (básica) es:

update *nom_tabla*
set *columna_1 = valor_1* [, *columna_2 = valor_2*, ...]
[**where** *expre_cond*]

donde:
columna_1, *columna_2*, ...: son las columnas a modificar.

Ejemplos:

```
update Historial set Calif=8
   where CU=20 and ClaveM=620
```

2.4.3 Instrucción **delete**

Se emplea para eliminar tuplas de una tabla. Su sintaxis (básica) es:

delete from *nom_tabla*
[**where** *expre_cond*]

Ejemplos:

```
delete from Inscrito where CU = 60

delete from Inscrito
```

2.5 Instrucción select

SQL tiene una instrucción básica para recuperar información de una base de datos: la instrucción **select**. Esta instrucción genera una tabla de resultados que contiene las tuplas que produce la consulta. La instrucción presenta diversas opciones las cuales se analizarán gradualmente a continuación.

La sintaxis general de la instrucción select es:

select [**all** | **distinct**] *lista_de_columnas_s*
from *lista_de_tablas*
[**where** *expre_cond*]
[**group by** *lista_de_columnas_g*]
[**having** *expre_cond*]
[**order by** *columna* [**asc** | **desc**] [, *columna* [**asc** | **desc**]]...]

donde:

lista_de_columnas_s: es un lista de columnas (separadas con coma) cuyos valores van a ser recuperados por la consulta. Una columna puede ser:
[{*nombre_de_tabla* | *alias*}.]*nombre_de_columna*,
[{*nombre_de_tabla* | *alias*}.]* o *expresión*

lista_de_tablas: es una lista de tablas (separadas con coma) requeridas para procesar la consulta

expre_cond: es una de las expresiones condicionales que pueden especificarse en SQL

lista_de_columnas_g: es una lista de columnas (separadas con coma); para cada columna sólo se puede usar la sintaxis:
[{*nombre_de_tabla* | *alias*}.]*nombre_de_columna*,

columna: es de la forma: [{*nombre_de_tabla* | *alias*}.]*nombre_de_columna*

Conceptualmente, y sólo para construir la consulta, se puede considerar que cada una de las distintas cláusulas de la instrucción *select* genera una tabla intermedia que es usada para evaluar la cláusula siguiente. En la práctica, la ejecución no necesariamente va a ser así, ya que cada DBMS tiene subsistemas de optimización que pueden ejecutar la consulta de otra manera que resulte más óptima.

La cláusula **select** especifica las columnas cuyos valores se van a tomar para generar la tabla de resultados.

La cláusula **from** especifica las tablas (relaciones) de donde se van a tomar las columnas indicadas en la cláusula *select*.

La cláusula **where** sirve para especificar una condición que deben cumplir las tuplas que se seleccionen de las tablas indicadas en *from*.

La cláusula **group by** sirve para formar grupos con las tuplas que aparecen en la tabla intermedia producida por *where* o por *from*.

La cláusula **having** se emplea para aplicar alguna condición a los grupos formados con *group by*.

Finalmente, la cláusula **order by** sirve para ordenar las tuplas de la tabla de resultados.

El DBMS evalúa las cláusulas en la instrucción *select* en el siguiente orden: *from, where, group by, having, lista_de_columnas_s, order by*. Como se mencionó anteriormente, después de hacer una evaluación dada, conceptualmente se puede considerar que el DBMS produce una tabla intermedia la cual es usada para realizar la siguiente evaluación, y así hasta finalizar la ejecución de la instrucción.

Opcionalmente se puede usar la cláusula **union** para producir una sola tabla uniendo el resultado de varias instrucciones *select*. La sintaxis es la siguiente:

select_1
union [**all**]
select_2
[**union** [**all**]...]
[**order by** *resto_del_order_by*]

Por default, el resultado de *union* elimina tuplas duplicadas. Para conservar todas las tuplas resultantes de la unión se debe usar **all**. Cuando se usa *union* las instrucciones *select_1*, *select_2*, ... no deben contener *order by*.

Observaciones sobre las cláusulas que componen a *select*

- Cláusula *where*

El DBMS evalúa la expresión condicional dada en *where* para cada tupla de la tabla intermedia creada por *from*.
Las columnas especificadas en la expresión condicional de *where*:

- deben ser columnas de la tabla intermedia creada por *from*, o
- pueden ser una referencia externa (una *referencia externa* es una referencia dentro de una subconsulta a una tabla declarada en una (sub)consulta externa que englobe a dicha subconsulta). Por ejemplo, en la siguiente consulta

```
select  NomAl
from  Alum
where not  exists
    (select * from Inscrito where Alum.CU = Inscrito.CU)
```

Alum, dentro de la subconsulta, es una referencia externa con respecto a dicha subconsulta, ya que la tabla está declarada en la consulta externa.

- Cláusula *group by*

Especifica columnas que el DBMS usa para formar grupos con la tabla intermedia creada por *where*, si existe, o por *from.* Se pueden especificar varias columnas consecutivas, con lo cual se van formando subgrupos dentro de los grupos. Ejemplo:

```
select NomProf, g.ClaveG, count(*) Cant_Alumnos
from Prof p, Grupo g, Inscrito i
where p.IdProf=g.IdProf and g.ClaveG = i.ClaveG
group by NomProf, g.ClaveG
```

En este ejemplo se forman grupos por cada *profesor* que imparte cursos y, para cada uno, se forman subgrupos con las *claves de los grupos* en los cuales están inscritos los alumnos. Todos los valores nulos para una columna dada son agrupados juntos.

Cuando se usa esta cláusula, columnas que aparecen en *lista_de_columnas_s***, no especificadas en *group by*, deben colocarse siempre dentro de una función. En caso contrario, no pueden aparecer en dicha lista.** Por ejemplo, la siguiente consulta es incorrecta

```
select  Carr, NomAl, count(*)
from Alum
group by Carr
```

debido a que *NomAl* es una columna que no aparece dentro de *group by*, ni dentro de una función.

- Cláusula *having*

El DBMS aplica la expresión condicional de *having* a cada grupo de la tabla intermedia creada por la cláusula precedente (que en general es *group by*).

La cláusula *having* afecta grupos de la misma forma que *where* afecta tuplas individuales.
Si *having* no es precedida por *group by*, el DBMS aplica la expresión condicional a todas las tuplas de la tabla intermedia creada por *where* o por *from***,** como si se tratará de un solo grupo.

Las columnas especificadas en *having* deben:

- estar también en *group by*, o
- ser especificadas dentro de una función.

- Subconsultas

Cualquier consulta anidada sólo puede generar como resultado una tabla con **una sola** columna (la cual, por supuesto, puede contener varios valores). Cuando se tienen varias consultas anidadas, el orden de evaluación es de la más interna hacia la más externa.

Restricciones de uso

- En general, el predicado en *where* no puede contener una función de totales. La única excepción a esta restricción es cuando la función en *where* tiene como argumento una referencia externa.
- Las subconsultas sólo pueden entregar una tabla de una sola columna, aunque puede haber muchos valores en ésta.
- Si **no** se usa *group by*, la *lista_de_columnas_s* de la cláusula *select* debe:
  - ser únicamente una lista de funciones de totales, o
  - ser únicamente una lista de atributos de tablas.

  Por lo anterior, el siguiente ejemplo es incorrecto:

  ```
  select Carr, avg(Prom)
  from Alum
  ```

  dado que se está combinando una columna, *Carr*, con una función de totales, *avg(Prom)*, sin utilizar la cláusula *group by*.
- Si se usa *group by*, las columnas en la *lista_de_columnas_s* deben:
  - estar también especificadas en *group by*, o
  - ser especificadas dentro de una función de totales.
- Las funciones de totales, en general, no pueden estar anidadas. Es decir, normalmente no se debería escribir: count(sum(Creds)), aunque hay DBMS's que sí lo aceptan.

2.6 Lenguaje de programación y subprogramas almacenados

Todo DBMS de mediana o gran complejidad proporciona un lenguaje de programación el cual permite combinar instrucciones de SQL con instrucciones similares a las típicas de un lenguaje de procedimientos (como C), con el fin de poder expresar soluciones para aquellos problemas que no pueden resolverse directamente ─o es muy complicado hacerlo─ con las instrucciones que brinda SQL.

En la siguiente sección se estudiarán las características principales del lenguaje PL/SQL, que es el nombre que se le da al lenguaje proporcionado por Oracle; también se verán algunos ejemplos de programación con el mismo, con el objetivo de mostrar el uso de un lenguaje de esta naturaleza, ya que lenguajes parecidos existen en la mayoría de los DBMS comerciales, aunque con diferencias menores. Posteriormente se verá el manejo de cursores y de triggers (desencadenadores) dentro de Oracle, elementos en los cuales se aplican las frases de este lenguaje.

Definición de bloque

Un **bloque** es simplemente un conjunto de instrucciones de PL/SQL que Oracle puede ejecutar. Un bloque está constituido por tres partes: la declarativa (opcional), la ejecutable y la de manejo de excepciones (opcional; no se verá en estas notas). Las declaraciones son locales al bloque y cesan cuando el bloque se completa. Los **bloques básicos** son: bloques anónimos, procedimientos y funciones. A continuación se muestra un ejemplo de un bloque anónimo:

```
declare cant int;
begin
  select count(*) into cant from Alum;
  dbms_output.put_line('Cantidad de alumnos: ' || cant);
end;
```

La penúltima instrucción muestra en pantalla el valor de *cant* (si se está en SQL Developer o en SQL*Plus Worksheet, hay que dar antes: **set serveroutput on**, para poder ver en pantalla valores con esta instrucción). Los caracteres || sirven para concatenar cadenas en Oracle.

2.6.1 Lenguaje de programación PL/SQL

Como ya se comentó, está constituido por un conjunto de instrucciones muy similares a las de un lenguaje de 3ª generación, aunque dicho conjunto es bastante reducido con respecto al que se proporciona en uno de estos lenguajes. Estas instrucciones pueden usarse en los bloques, en los subprogramas almacenados y en los desencadenadores (triggers). A continuación se presentan las instrucciones más importantes de este lenguaje proporcionado por Oracle.

Declaración de variables

Se preceden por la palabra reservada: **declare**, y deben declararse indicando su tipo de datos (cualquiera de los de Oracle). Como ya se mencionó, son locales al bloque en el que se declaran.

La sintaxis de declaración es:

declare *nom_var1 tipo_datos*; [*nom_var2 tipo_datos*;] …

Ejemplos:

```
declare cant int; prom real;
```

Una vez declaradas las variables, se les puede asignar valores de tres maneras:

por asignación directa: prom := 8.5;
por medio de un select: select count(*) into cant from Alum;
usándolas como parámetro de salida en un subprograma (ver más adelante).

Si no se le asigna explícitamente un valor a una variable, ésta tendrá el valor **null**. Si por medio de un **select** se asigna más de un valor a un variable, el DBMS marcará error.
Nota: cuando se declaran variables o cursores (secc. 2.6.3), la palabra *declare* debe escribirse **explícitamente** en el caso de bloques anónimos y desencadenadores (secc. 2.7), y **no escribirse** en el caso de subprogramas almacenados (secc. 2.6.2).

begin … end

Se usan para enmarcar a un conjunto de instrucciones para que sean tratadas como si fueran una unidad. Su sintaxis es:

begin
instrucción; [*instrucción*;] …
end;

Ejemplo:

```
declare promGen real;
begin
  select avg(Prom) into promGen from Alum
     where Carr='mat';
  dbms_output.put_line('Promedio general de los alumnos de Matemáticas: ' || promGen);
end;
```

Select

La instrucción **select** sufre una ligera modificación cuando se usa dentro de un bloque. El resultado que ahora entrega debe ser una **sola** tupla de valores los cuales deben asignarse a tantas variables como valores haya en la tupla. La sintaxis de este **select** es:

select [**all** | **distinct**] *lista_columnas* **into** *lista_vars*
from *resto_select***;**

donde:

lista_columnas: es un lista de columnas o de funciones de totales (separadas con coma) cuyos valores van a ser recuperados por la consulta.
lista_vars: es la lista de variables (separadas por coma) a las cuales se asignará el resultado del select
resto_select: representa las demás cláusulas que se pueden especificar con select.

If … end if

Es la típica instrucción para ejecutar condicionalmente a otras instrucciones. Su sintaxis es:

if *expre_cond* **then**
  instrucción; [*instrucción*;] …
[**else**
  instrucción; [*instrucción*;] …]
end if;

donde:

expre_cond: es una de las expresiones condicionales que pueden especificarse en SQL, salvo: subconsulta y los operadores **exists, all, any**
instrucción: es cualquier instrucción del lenguaje de programación PL/SQL.

Ejemplo:

```
declare creds1 int;
begin
  select count(*) into creds1 from Mater where Creds=1;
  if creds1>=1 then  dbms_output.put_line('Hay ' || creds1 || ' materias de 1 crédito');
  else  dbms_output.put_line('No hay materias de 1 crédito');
  end if;
end;
```

Case … end case

Es la instrucción que se puede usar en lugar de **if** cuando se tienen varias alternativas para ejecutar condicionalmente a otras instrucciones. Su sintaxis es:

case *expre_num*
when *valor* **then**
  instrucción; [*instrucción*;] …
[**when** *valor* **then**
  instrucción; [*instrucción*;] …] …
[**else**
  instrucción; [*instrucción*;] …]
end case;

donde:

expre_num: es una expresión que debe producir un valor de tipo entero, flotante, lógico, carácter, cadena o fecha
valor: es un valor del mismo tipo de datos que *expre_num*
instrucción: es cualquier instrucción del lenguaje de programación PL/SQL.

Loop …exit when *expre_cond* **… end loop**

Instrucción iterativa de PL/SQL. El bloque de instrucciones especificado dentro de **loop** se ejecuta repetidamente hasta que la condición dada en **exit when** se vuelve verdadera. Su sintaxis es:

loop
instrucción; [*instrucción*;] …
exit when *expre_cond***;**
instrucción; [*instrucción*;] …
end loop;

donde:

expre_cond: es una de las expresiones condicionales que pueden especificarse en SQL, salvo: subconsulta y los operadores **exists, all, any**. También se pueden usar otros operadores los cuales se verán en la parte de *cursores*
instrucción: es cualquier instrucción del lenguaje de programación PL/SQL.

While … loop … end loop

Es otra de las instrucciones iterativas de PL/SQL. El bloque de instrucciones especificado con **while** se ejecuta repetidamente mientras la condición dada sea verdadera. Su sintaxis es:

while *expre_cond* **loop**
instrucción; [*instrucción*;] …
end loop;

donde:

expre_cond: es una de las expresiones condicionales que pueden especificarse en SQL, salvo: subconsulta y los operadores **exists, all, any**
instrucción: es cualquier instrucción del lenguaje de programación PL/SQL.

For … loop … end loop

For también forma parte del conjunto de instrucciones iterativas de PL/SQL. El bloque de instrucciones especificado se ejecuta una **cantidad fija** de veces. Su sintaxis es:

for *nom_const* **in** [**reverse**] *lim_inf* **..** *lim_sup* **loop**
instrucción; [*instrucción*;] …
end loop;

donde: *nom_const*: es una constante que sirve como contador del ciclo. No se requiere declararla explícitamente, puede ser usada en expresiones dentro del **for**, no se le puede asignar valor explícito y su alcance es local al **for**
lim_inf: valor inferior del contador
lim_sup: valor superior del contador (≥ *lim_inf*)
instrucción: es cualquier instrucción del lenguaje de programación PL/SQL.

Si se utiliza la palabra **reverse**, el valor del contador variará desde *lim_sup* hasta *lim_inf*.

2.6.2 Subprogramas almacenados

Los *subprogramas almacenados* son colecciones de instrucciones del lenguaje de programación PL/SQL. Estos subprogramas pueden:

- tomar parámetros
- llamar a otros subprogramas
- regresar valores al lugar que los llamó
- ser ejecutados en servidores SQL remotos

Los subprogramas almacenados normalmente son compilados por Oracle y guardados permanentemente en la base de datos, por lo que su rendimiento es muy alto y su ejecución es bastante rápida. Esto sugiere que cuando se tienen tareas muy bien definidas para la explotación de una base de datos, puede convenir implementarlas por medio de estos subprogramas para tener un procesamiento muy eficiente de las mismas. Como contraparte, dado que el DBMS es el que los ejecuta, se puede "sobrecargar" si muchos usuarios ordenan su ejecución al mismo tiempo.

Al igual que los bloques anónimos, los subprogramas tienen una parte declarativa (opcional). Tienen también una parte ejecutable y una parte para manejo de excepciones (opcional). Los subprogramas son de dos tipos: *funciones* y *procedimientos*.

Funciones

Son equivalentes a las funciones de cualquier lenguaje de programación por lo que el proceso que realizan debe ser tal que produzcan como resultado un solo valor. Su sintaxis de definición es:

[**create** [**or replace**]] **function** *nom_función* [(*parámetro* [, *parámetro*] …)]
    return *tipo_datos* **is**
[*declaraciones_locales*]
begin
  instrucción; [*instrucción*;] …
end;

donde:

nom_función: es el nombre (arbitrario) de la función
parámetro: representa a un parámetro formal con el cual se define la función. Su sintaxis de declaración es: *nombre_parámetro tipo_datos*
tipo_datos: es cualquiera de los tipos de datos reconocidos por Oracle
declaraciones_locales: aquí se pueden declarar variables, constantes, cursores y subprogramas anidados
instrucción: es cualquier instrucción del lenguaje de programación PL/SQL.

Si se usa **create**, entonces la función queda guardada permanentemente en la base de datos en la cual se define, esto es, se convierte en una *función almacenada*. Si no se usa **create**, la función es local al bloque en el cual se define. En el bloque **begin .. end;**, debe haber al menos un: **return** *expre*;, que es el resultado de la función.

Procedimientos

Son equivalentes a los procedimientos de cualquier lenguaje de programación. Pueden tener parámetros de entrada, de salida o de entrada/salida; a través de estos dos últimos es como se pueden regresar resultados al lugar en que se llama al procedimiento. Su sintaxis de definición es:

[**create** [**or replace**]] **procedure** *nom_proc* [(*parámetro* [, *parámetro*] …)] **is**
[*declaraciones_locales*]
begin
instrucción; [*instrucción*;] …
end;

donde:

nom_proc: es el nombre (arbitrario) del procedimiento

parámetro: representa a un parámetro formal con el cual se define el procedimiento. Su sintaxis de declaración es:
nombre_parámetro [**in | out | in out**] *tipo_datos*

declaraciones_locales: aquí se pueden declarar variables, constantes, cursores y subprogramas anidados

instrucción: es cualquier instrucción del lenguaje de programación PL/SQL.

La explicación de **create** para funciones, se aplica igual para procedimientos. En el caso de los parámetros: **in** (default) indica que es de entrada; **out**, de salida; e **in out**, de entrada/salida.

Como siempre, el llamado a las funciones debe hacerse dentro de una expresión y el llamado a los procedimientos como una instrucción sola.

Ejemplos:

1) Definición de una función:

```
-- Función para calcular el promedio de una carrera dada como parámetro.
create or replace function promCarr(carrera char) return real is
  auxProm real;
begin
  select avg(Prom) into auxProm from Alum where Carr=carrera;
  if auxProm is null then
    return 0;
  else return auxProm;
  end if;
end;
```

Ejecución:

```
begin
  dbms_output.put('Promedio de la carrera de Telecomunicaciones: ');
  dbms_output.put_line(promCarr('tel'));
end;
```

2) Definición de un procedimiento:

```
-- Procedimiento que lista la cantidad de alumnos que
--  están inscritos en una materia dada como parámetro.
create or replace procedure matCantAls(nom char) is
  cant int;
begin
   select count(*) into cant
   from Mater m, Grupo g, Inscrito i
   where m.NomMat=nom and m.ClaveM=g.ClaveM and g.ClaveG=i.ClaveG;
  dbms_output.put('La materia: ' || nom || ', tiene: ');
  dbms_output.put_line(cant || ' alumno(s)');
end;
```

Ejecución:

```
begin
  matCantAls('Matemáticas I');
end;
```

Parámetros de salida en un procedimiento

Un procedimiento puede ser definido también con parámetros formales de salida, o de entrada/salida, para regresar valores por medio de ellos al lugar en que se le llamó. Si un procedimiento se define así, en el llamado al mismo deben especificarse **variables** en los parámetros actuales correspondientes a dichos parámetros de salida. Ejemplo: dado el nombre de una materia como parámetro, contar cuántos alumnos la están llevando:

Definición:

```
create or replace procedure matCantAls2(nom char, cant out int) is
begin
  select count(*) into cant
  from Mater m, Grupo g, Inscrito i
  where m.NomMat=nom and m.ClaveM=g.ClaveM and g.ClaveG=i.ClaveG;
end;
```

Llamado:

```
declare
  nomMat varchar(20); cantAls int;
begin
  nomMat := 'Matemáticas I';
  matCantAls2(nomMat,cantAls);
  dbms_output.put('La materia: ' || nomMat || ', tiene: ');
  dbms_output.put_line(cantAls || ' alumno(s)');
end;
```

2.6.3 Cursores

Un *cursor* es un nombre simbólico asociado con una instrucción **select**. Consiste de:

- un conjunto resultado del cursor - es el conjunto (tabla) de tuplas resultante de la ejecución de la consulta asociada al cursor
- una posición del cursor - un apuntador (conceptual) a una tupla de dicho conjunto

Declaración del cursor

Se utiliza la siguiente sintaxis para declarar un cursor:

[**declare**] **cursor** *nom_cursor* **is** *instr_select*;

donde: *instr_select*: es la instrucción **select** de SQL (no la de PL/SQL, que es más restringida).

Apertura del cursor

Todo cursor debe abrirse antes de ser utilizado. Esto significa que se instruye al DBMS para que ejecute el **select** asociado al cursor con el fin de producir la tabla de resultados respectiva. La frase utilizada para esto es **open,** cuya sintaxis es:

open *nom_cursor*

Cuando se hace la apertura el cursor queda apuntando a la 1ª tupla del resultado.

Lectura de tuplas

Una vez abierto el cursor, ya se pueden leer las tuplas de la tabla de resultados. Para hacer esto se emplea la instrucción **fetch**, cuya sintaxis es:

fetch *nom_cursor* **into** *lista_vars*

La *lista_vars* es una lista de variables donde van a quedar guardados los valores de la tupla leída. Debe haber tantas variables como valores haya en la tupla; cada variable debe ser del mismo tipo de datos que su valor correspondiente (obviamente deben declararse antes de usarlas). Ya que los valores están en las variables, se pueden usar en cualquiera de las instrucciones de PL/SQL.

Con cada lectura el apuntador queda apuntando a la tupla siguiente a la leída. La lectura de las tuplas es secuencial; si se quieren volver a leer, hay que cerrar el cursor y volverlo a abrir.

Verificación del estatus del cursor

Se puede emplear una construcción como la siguiente para determinar si ya se llegó a la última tupla de la tabla de resultados:

```
loop
  fetch nom_cursor into …;
  exit when nom_cursor%notfound;
  -- Procesamiento de los valores de la tupla.
end loop;
```

Cierre del cursor

Para cerrar un cursor se emplea la instrucción **close**, cuya sintaxis es:

close *nom_cursor*

Una vez cerrado un cursor puede volver a abrirse si así se desea.

Uso del cursor por medio de for

Se puede utilizar la instrucción **for** para usar el cursor sin emplear las instrucciones **open**, **fetch** y **close**. La instrucción **for** implícitamente abre el cursor, accede a las tuplas una por una y, al final, cierra el cursor. La sintaxis seguida es:

for *var_tupla* **in** *nom_cursor* **loop**
  -- Procesamiento de los valores de la tupla.
end loop;

donde: *var_tupla*: es una variable, que no necesita declararse, la cual almacenará los valores de cada tupla, una a la vez. El acceso a cada valor de la tupla se hace con la notación de punto, por ejemplo: *var_tupla*.NomAl

Es conveniente usar esta construcción cuando se van acceder secuencialmente **todas** las tuplas del resultado. Si el acceso a las tuplas está sujeto a terminar cuando se cumpla cierta condición, entonces el acceso deberá hacerse con **loop … end loop**, saliendo del ciclo con las instrucciones: **exit when** *expre_cond.*

Varios

Se puede emplear **%type** para declarar una variable con el mismo tipo de datos que un atributo de una tabla. Esto tiene la ventaja de que si después se cambia el tipo de datos de dicho atributo, no es necesario cambiar explícitamente el tipo de datos de esa variable. Ej.:

```
declare  cursor DatosAlums is  select NomAl, Prom from Alum;
  nombre  Alum.NomAl%type;  p real;
begin
  open DatosAlums;
  fetch DatosAlums into nombre, p;
  dbms_output.put_line(nombre || ' ' || p);
  -- - -
end;
```

De igual manera, se puede usar **%rowtype** para declarar una variable que tenga la misma cantidad de atributos, con idéntico tipo de datos, que una tupla de un cursor. Se usa la notación de punto para acceder a cada atributo. Ejemplo:

```
declare  cursor DatosAlums is  select NomAl, Prom from Alum;
      tuplaAlum  DatosAlums%rowtype;
begin
   open DatosAlums;
   fetch DatosAlums into tuplaAlum;
   dbms_output.put_line(tuplaAlum.NomAl || ' ' || tuplaAlum.Prom);
   -- - -
end;
```

Las siguientes condiciones también pueden utilizarse en la instrucción exit when *condición*, cuando se utiliza loop … end loop para recorrer la tabla de resultados de un cursor, o en alguna instrucción de PL/SQL en que pueda especificarse una condición:

a) cursor%rowcount – que regresa la cantidad de tuplas del cursor.
b) cursor%found – que regresa true, si aún hay tuplas en la tabla de resultados.
c) cursor%isopen – que regresa true si el cursor está abierto.

2.7 Desencadenadores (triggers)

Un desencadenador (trigger) es una clase especial de subprograma almacenado el cual se activa cuando se llevan a cabo operaciones de actualización sobre las tablas de la base de datos. Los triggers pueden ayudar a conservar la integridad referencial de los datos manteniendo la consistencia entre datos lógicamente relacionados que estén en tablas diferentes. También pueden servir para llevar a cabo operaciones relacionadas con la actualización de las tablas. Los triggers son **automáticos**: se disparan cuando ocurre una actualización de datos en una tabla, ya sea causada por un usuario o por una aplicación. La sintaxis de definición de un trigger es la siguiente:

create or replace trigger *nombre_trigger*
{**before** | **after**} *instr_act* **on** *nombre_tabla*
[**for each row**] [**when** *expre_cond*]
bloque_PL/SQL

donde:

nombre_trigger: es el nombre para el desencadenador
instr_act: es cualquiera de las instrucciones de actualización de datos: **insert**, **update** o **delete**. Pueden especificarse de forma aislada o unidas con **or**
nombre_tabla: es la tabla que al actualizarse va a ocasionar que se dispare el trigger
expre_cond: es una de las expresiones condicionales que pueden especificarse en SQL, pero sin contener una subconsulta
bloque_PL/SQL: son instrucciones de PL/SQL que se ejecutarán al activarse el trigger.

Ejemplos:

```
-- Definición de triggers.
-- El siguiente trigger borrará tuplas relacionadas en la tabla Inscrito cuando se
-- elimine algún grupo de la tabla Grupo.

      create or replace trigger BorraInscGrupo
        before delete on Grupo
        for each row
        begin
          delete from Inscrito where :old.ClaveG=Inscrito.ClaveG;
        end;

-- Este trigger coloca un valor nulo en la columna IdProf, de la tabla Grupo, para aquellas
-- tuplas que estén relacionadas con algún profesor que se elimine de la tabla Prof.
      create or replace trigger QuitaProfGrupo
        before delete on Prof
        for each row
        begin
          update Grupo set IdProf=null where :old.Idprof=Grupo.IdProf;
        end;

-- Este trigger cambia la clave del profesor, en la columna IdProf de la tabla Grupo, en
-- aquellas tuplas que estén relacionadas con el profesor cuya clave cambie en la
-- tabla Prof.
      create or replace trigger CambiaClaveProf
        before update on Prof
        for each row
        begin
          update Grupo g set g.IdProf= :new.IdProf
       where g.IdProf= :old.IdProf;
          dbms_output.put('También se cambió en Grupo la clave anterior del prof. ');
          dbms_output.put_line(:old.IdProf || ' por su nueva clave: ' || :new.IdProf);
        end;
```

Observaciones sobre los triggers

1. Ayudan a **conservar la integridad referencial**, disminuyendo las posibles inconsistencias que se puedan presentar cuando se hace un actualización a las tablas de una base de datos.
2. **No hay un orden específico** en el cual se ejecuten, por lo que las actividades que se hacen con ellos pueden no dar los resultados pretendidos.
3. **Se pueden causar ciclos**, si hay triggers que se van disparando de manera encadenada y se llega a un punto en el cual se alcanza la tabla que disparó el primer trigger.
4. Cada que se actualiza una tabla sobre la cual hay triggers, éstos se procesan, por lo cual **se emplea tiempo extra** en el manejo de la tabla, pudiendo hacer lento el trabajo con ésta si hay muchas actualizaciones sobre ella.

5. Por ende, **se deben definir políticas** en la definición de triggers sobre las tablas y analizar si se deben tener o no (ejemplo: un trigger puede desencadenar la modificación o eliminación de tuplas que, en principio, no debían afectarse).
6. Lo anterior también está relacionado con el **establecer políticas sobre los permisos de operación** que deben tener los diferentes usuarios de una base de datos, ya que si un usuario tiene permiso de actualizar una tabla y ésta tiene triggers, puede causar que se afecten otras tablas para las cuales dicho usuario no tenía, en principio, permiso de actualización.

2.8 Definición de índices

Los índices son utilizados por el DBMS para acelerar el proceso de recuperación de datos. Su manejo es transparente para el usuario ya que el DBMS decide cuando usarlos o no al ejecutar una consulta. SQL sólo proporciona instrucciones para crear o eliminar índices sobre tablas. La sintaxis usada para crear índices es la siguiente:

create [**unique**] **index** *nombre_índice*
on *nombre_tabla* (*columna_1* [, *columna_2*]...)

En el caso de Oracle, y de muchos otros manejadores como Access, cada que se define una tabla se crea automáticamente un índice sin duplicados para los campos que constituyen la clave primaria de la misma. Se pueden definir más índices, con o sin duplicados, para los otros campos de una tabla, por ejemplo, para las claves externas o para campos comúnmente usados.
Una vez que un índice es definido sobre un campo, el DBMS lo usa siempre que se accede a dicho campo. El usuario no tiene control sobre esto y no puede indicarle al DBMS si lo quiere usar o no, por ejemplo en el procesamiento de una consulta.

Hay que tener presente que la ventaja de tener índices sobre campos de tablas es que el acceso a las tuplas de una tabla es más rápido, si se hace por medio de esos campos indizados; la desventaja, obviamente, es que se ocupa espacio extra y para ciertos casos –por ejemplo, en tablas con decenas o cientos de miles de tuplas- puede resultar en un costo excesivo.

```
--
-- Ejemplo de creación de índices
--
create index IndGrProf on Grupo(IdProf)
```

2.9 Eliminación de objetos de una base de datos

Para eliminar algún objeto (tabla, índice, trigger, etc.) de una base de datos se utiliza la instrucción **drop**. Cuando se elimina una tabla desaparecen, además de su estructura, los datos que almacena así como los índices y permisos que tiene asociados. La sintaxis de la instrucción es:

drop {**table | index | trigger | function | procedure**} *nombre*[, *nombre*] ...;

donde:

nombre: es el nombre de la tabla, índice, trigger, función almacenada o procedimiento almacenado que se va a eliminar de la base de datos.

2.10 Vistas

Una **vista** es una sola tabla que es derivada de otras tablas. Estas otras tablas pueden ser **tablas base**[1] u otras vistas definidas previamente. Una vista no necesariamente existe en forma física, por lo que también suele llamárseles **tablas virtuales**, en contraste con las tablas base cuyas tuplas sí están físicamente almacenadas en la base de datos.

Se puede pensar en una vista como una manera de especificar una tabla que se necesita constantemente, aunque pueda no existir físicamente. Normalmente la vista se define como resultado de una serie de equijuntas aplicadas a varias tablas. Posteriormente, se emiten consultas sobre una sola tabla, la vista, en lugar de usar dos, tres o más tablas para plantear la consulta equivalente.

De cualquier manera, la vista puede utilizarse como si fuera una tabla más de la base de datos; esto es, se puede usar para construir otras consultas nuevas. También, la vista puede **materializarse**, significando esto que temporalmente se puede crear físicamente una tabla con las tuplas que genera la vista; la desventaja de esto es que cuando se actualizan las tablas base, puede ser necesario actualizar la vista con el consecuente consumo de tiempo que esto implica. Después de un tiempo, o si la vista no se usa mucho, la tabla materializada puede borrarse de la base de datos. Si llegará a necesitarse posteriormente, el DBMS puede volver a generar las tuplas de la vista a partir de los datos actuales de las tablas base.

Las vistas también se usan como mecanismos de autorización y de seguridad para restringir el acceso a las diferentes partes de una base de datos. Se pueden definir vistas para que grupos de usuarios sólo "vean" una parte de la base de datos, sin que tengan conocimiento del resto de la base.

[1] Este término también se utiliza para referirse a las tablas que constituyen a una base de datos.

A continuación se muestra un ejemplo de cómo se definiría una vista en SQL (su objetivo es mostrar las materias que está cursando un alumno):

```
create view MateriasAlumno
as select NomAl, NomMat
  from Alum a, Inscrito i, Grupo g, Mater m
  where a.CU=i.CU and i.ClaveG=g.ClaveG and g.ClaveM=m.ClaveM;
```

Una vez definida la vista se puede emplear para consultarla, tal como se muestra en el siguiente ejemplo (el cual lista las materias que cursa Ana):

```
select *
from MateriasAlumno
where NomAl='Ana';
```

Resultado:

NomAl	NomMat
Ana	Estructuras de Datos
Ana	Economía I
Ana	Matemáticas I

Finalmente, cabe mencionar que una vista básicamente se usa para consultas ya que actualizarlas puede resultar sumamente complejo, ya sea porque en la vista se involucran muchas tablas, porque se hacen agrupaciones o porque se emplean funciones de totales en el resultado, lo cual hace extremadamente difícil el proceso de actualización de una vista.

2.11 Varios de SQL en Oracle

En esta sección se muestran variantes de algunas instrucciones de SQL que son válidas en Oracle. En otros DBMS pueden existir aunque, posiblemente, con sintaxis distinta.

a) Juntas externas.
   - Junta externa izquierda: incluye todas las tuplas de la primera tabla. En este ejemplo: lista todos los profesores, sin excepción, y las materias si es que están impartiendolas.

```
select nomprof, nommat, claveg
from prof p left outer join
    (grupo g left outer join mater m
            on g.clavem=m.clavem)
    on p.idprof=g.idprof;
```

   - Junta externa derecha: incluye todas las tuplas de la segunda tabla. En este ejemplo: lista todas las materias, sin excepción, y los profesores si es que están siendo impartidas.

```
select nomprof, nommat, claveg
from prof p right outer join
    (grupo g right outer join mater m
            on g.clavem=m.clavem)
    on p.idprof=g.idprof;
```

- Junta externa completa: incluye todas las tuplas de ambas tablas. En este ejemplo: lista todas las materias y todos los profesores, sin excpeción.

```
select nomprof, nommat, claveg
from prof p full outer join
    (grupo g full outer join mater m
            on g.clavem=m.clavem)
    on p.idprof=g.idprof;
```

b) Triggers simples en create table
Creación de tablas con actualización en cascada.

```
create table A
(A1 int primary key,
A2 int);
```

En lugar de cascade se podría poner set null; todo depende del tipo de la clave externa. En otros DBMS también se puede especificar update junto a (o en lugar de) delete.

```
create table B
(B1 int primary key,
B2 int references A on delete cascade);
```

c) Varios
- Especificación del resultado de una consulta como tabla en el *from* de otra consulta.

```
select nomal,nommat
from (select nomal,h.clavem from Alum a, Historial h where a.CU=h.CU) c, Mater m
where c.ClaveM=m.ClaveM
order by nomal;
```

- Inserción del resultado de una consulta en una tabla:

```
insert into A
  select distinct CU,10
  from alum;
```

- Una subconsulta sólo entrega una tabla con una columna, aunque en algunos DBMS se puede generar una tabla con más de una columna. Por ejemplo, la siguiente consulta no se puede ejecutar en SQL Server y sí en Oracle.

```
select nomal, carr
from Alum
where (NomAl,carr) in
  (select NomAl,carr from Alum);
```