TEMES 5, 6 I 7

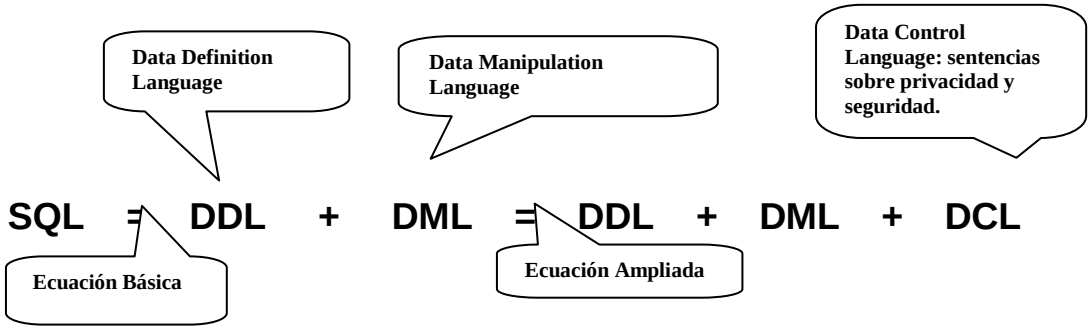
SQL

A. MOLL

1. INTRODUCCIÓ 2
2. DDL 3
2.1. Dominis 3
2.2. Definició de Taules 4
2.3. Vistes 6
2.4. Restriccions d'Integritat 7
3. DML 9
3.1. Lògica trivaluada del SQL 11
3.2. Visió conjuntista 12
3.3. Insercions, esborrats i canvis 13
4. DCL 14
5. EMBEDED SQL 15
5.1. El área de comunicaciones de SQL (SQLCA) 16
5.2. Cursors 17
5.3. Transaccions 18
5.3.1. Cicle de vida d'una transacció 20
5.4. SQL Dinàmic 20
6. PLPGSQL de Postgres 22
6.1. Funcions 22
6.1.1. Declaració de Variables 23
6.1.2. Sentències 24
6.1.3. Missatges i Excepcions 25
6.2. Triggers 25
7. Sobre L'estàndart SQL99 27

1. INTRODUCCIÓ

SQL ≡≡ Structured Query Language. El lenguaje de definición y manipulación de bases de datos relacionales más conocido, implantado, estudiado y utilizado. **Estudiaremos aquí la del estándar ANSI92** aunque al final del tema veremos un overview sobre el estándard objeto-relacional SQL99. Todavía existe una última versión del estándard: SQL 2003 que, anecdòticamente consta de unas 200 páginas.



La anterior ecuación ampliada nos delata el hilo conductor del resto del tema. Por otro lado, tenemos dos "tipos" de SQL según el uso que hagamos de él:

- **Interactivo**: Lanzamos interactivamente sentencias individuales. No obstante las podemos empaquetar en un script."Este SQL" no es procedural.
- **Interactivo con GUI**. Por ejemplo Navicat o pgAdminIII para interactuar con PostgreSQL.
- **Embebido**: "Metemos" las sentencias SQL , con algunos retoques sintácticos, en el código de un lenguaje de prg como C en aras de, aprovechando sus estructuras de control (if, bucles,...), aumentar la potencia expresiva del sistema sw.
- **Procedural (PL/SQL de ORACLE, PLPGSQL de Postgres):** No hace falta "salir de casa" incrustándose en otro lenguaje distinto. El propio SGBD proporciona funciones, procedimientos, estructuras de control, triggering..etc.

2. DDL

Esquema DDL ::= CREATE SCHEMA *NomEsquema* [AUTHORIZATION *usuario*]
<lista_elemento_esquema>

ElementoEsquema

- **Definición de dominios definidos por el usuario → CREATE DOMAIN**
- **Definición de tabla básica → CREATE TABLE**
- **Definición de tabla derivadas, virtuales o vistas → CREATE VIEW**
- **Definición de restricción de integridad → CREATE ASSERTION**
- **Etc…**

2.1. Dominis

Los tipos de datos especificados por el estándar son sólo un conjunto mínimo que la mayoría de sistemas SQL soportan y, además, amplian.

Tipos predefinidos en el estándar
CHAR (long)
INT o INTEGER
SMALLINT
NUMERIC (precisión, escala)
FLOAT (precisión)
REAL
DOUBLE PRECISION
.....

Con CREATE DOMAIN podemos enriquecer nuestro juego de tipos incorporando tipos o dominios definidos por el usuario. Veamos algunos ejemplos:

```
CREATE DOMAIN
CadenaLLarga
AS VARCHAR(80)
```

```
CREATE DOMAIN
MajorEdat
AS INTEGER
DEFAULT 18
CHECK VALUE
BETWEEN 18 AND 99
```

```
CREATE DOMAIN EstatCivil
AS VARCHAR(12)
CHECK VALUE IN
('solter'.'casat','separat','viduo','altre')
```

```
CREATE DOMAIN TipusCarnet AS
TipusMajuscules
DEFAULT 'B'
CONSTRAINT Carnet
CHECK VALUE IN ('A','B','C')
NOT DEFERRABLE
CONSTRAINT TipusAImplicaTipusB
CHECK NOT(VALUE='A') OR
(VALUE='B')
NOT DEFERRABLE
```

Exercici: Dedueix, a partir dels anteriors exemples, una sintaxi del *Create Domain* en notació BNF.

2.2. Definició de Taules

Estudiemos ahora la sentencia por antonomasia del DDL: la todopoderosa CREATE TABLE. Cuya sintaxis, en primera aproximación, es

```
CREATE TABLE tabla
<comalista_definición_columna>
[<comalista_definición_restricción_tabla>]
```

Bifurcaremos por el no terminal <comalista_definición_restricción_tabla> para expresar restricciones que afecten a dos o más columnas/campos como por ejemplo

- una clave ajena compuesta
- dos campos numéricos cuya suma no exceda de 100
- etc...

Por otro lado, *definición_columna* deriva en

columna {tipo_dato | *dominio*}
[DEFAULT {literal | función_sistema | NULL }]
[lista_definición_restricción_columna]

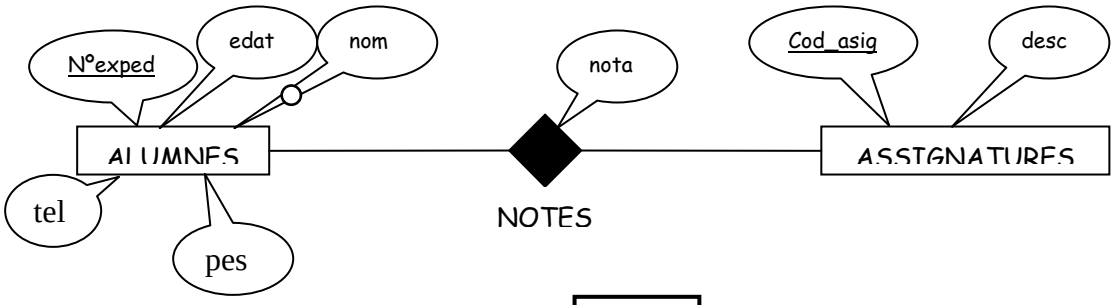
Especificado previamente con CREATE DOMAIN

Valor por defecto en caso de no indicar cláusula DEFAULT

La sintaxis para los no terminales restricción_columna o restricción_tabla son similares como pone de manifiesto la siguiente tabla:

restricción_columna::=	restricción_tabla::=
[CONSTRAINT restricción] {PRIMARY KEY | UNIQUE | REFERENCES tabla*[(columna*)] [TipoIntegridadReferencial] [DirectrizDeActualización] [DirectrizDeBorrado] | CHECK expression_condicional | NOT NULL} [[NOT] DEFERRABLE]	[CONSTRAINT restricción] {PRIMARY KEY (comalista_columna) | UNIQUE (comalista_columna) | FOREIGN KEY (comalista_columna) REFERENCES tabla*[(comalista_columna*)] [TipoIntegridadReferencial] [DirectrizDeActualización] [DirectrizDeBorrado] | CHECK expression_condicional} [[NOT] DEFERRABLE]

Example ,



Disseny Lògic

ALUMNE = Nºexped + edat + nom + tlf + pes.
ASIGNATURES = CodAsig + desc
NOTES = Nºexped + CodAsig + nota

C.Aj { Nºexped → ALUMNES
CodAsig → ASSIGNATUREs

Esquema lògic relacional

VNN(Nom)

Implementació en DDL (SQL) Postgres

Create Table *Alumnes* (Nexpd **char(4) Primary Key**, edat **int2**, Nom **VarChar**(30) not null, Tlf **char**(9), Pes **numeric(4,1)**;	**Create Table** *Assignatures* (CodAssig **char(3) Primary Key**, Desc **varchar(30)**);

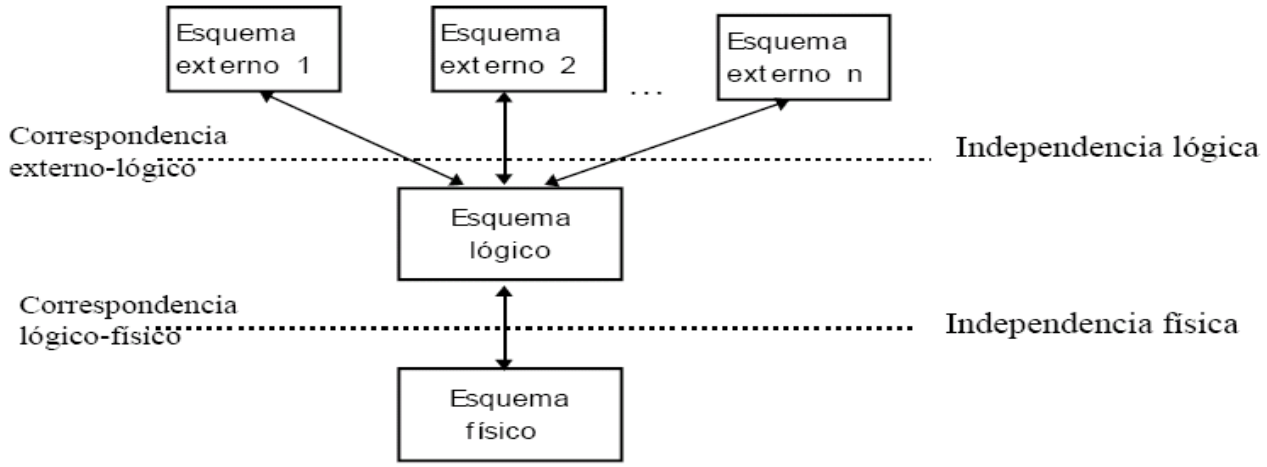
Create Table *Notes*

(Nexpd char(4) **References** *Alumnes*,
CodAssig **char**(3) **References** *Assignatures*,
Nota **numeric(4,2)**,

Primary Key (Nexpd, CodAssig))

2.3. Vistes

Con las vistas se promueve la independencia de datos a nivel lógico, separando el esquema lógico relacional de sus esquemas externos ➔ Arquitectura ANSI/SPARC



Una vista es una tabla "virtual" que no tiene existencia propia, es decir, no está materializada/almacenada expresamente en disco. Gráficamente la podemos interpretar como una ventana a partir de la cual, sólo podemos ver y actualizar una parte de las tablas y/o vistas (recursividad) sobre las que está definida.

Sintaxis	Ejemplo
CREATE VIEW vista [(comalista_columna)] AS expresión_tabla [WITH CHECK OPTION]	CREATE VIEW ASSIGNATURES_INTERESSANTS AS SELECT * FROM ASSIGNATURA WHERE desc='Interessant' WITH CHECK OPTION Filtrado Horizontal que no vertical (nos quedamos con todas las cols). Rechaza la inserción o modificación de tuplas que violen la def. de la vista (asignaturas interesantes).

(Ens veiem obligats a avançar esdeveniments: introduïm una sentència SELECT quan en el tema encara no hem parlat de DML. No queda més remei.)

Si lanzáramos una consulta sencilla sobre la vista ASSIGNATURES_INTERESSANTS ésta se traduciría en un acceso sobre la tabla básica que subyace.

SELECT CodAsig FROM ASSIGNATURES_INTERESSANTS WHERE CodAsig <> 'BDA'	⇒	SELECT CodAsig FROM **ASSIGNATURA** WHERE CodAsig <> 'BDA' AND **desc='Interessant'**

Esta es la sutil forma en la que los SGBD traducen "internamente" consultas sobre las vistas. El anterior ejemplo convence, definitivamente, por qué se trata de tablas "virtuales".

2.4. Restriccions d'Integritat

El DDL del estándar incorpora una sentencia específica para expresar rest. de integridad: ➔

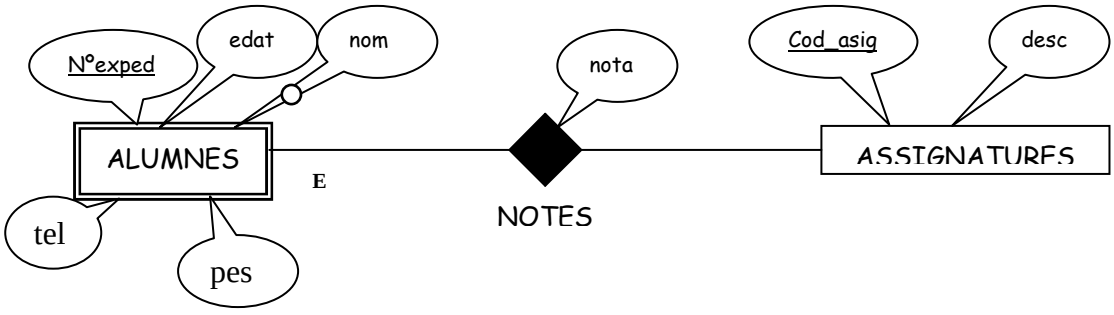
CREATE ASSERTION NomRestricción
CHECK (ExprBooleana)
[[NOT] DEFERRABLE]

Esta sentencia nos puede servir para expresar, en ppio, cualquier restricción "implementable". Pero algunas restricciones como las de acotación de valores (EdadLaboral entre 16 y 65) **se pueden representar en otras partes del código DDL**. Veamos ejemplos:

Ejemplo	Observaciones
Create Domain TedadLaboral AS integer Default 16 **Check value between 16 and 65**	Acotamos a nivel de dominio
Create Table Empleado (NumEmpleado Tcodigo primary key, …. Edad integer **check value between 16 and 65** …);	Acotamos a nivel de una columna/atributo concreto
Create Table A (a0 Tcodigo primary key, a1 real, a2 real, a3 varchar(4), …… **check a1 = 2*a2**);	Especificamos una restricción de integridad a nivel de una tabla (afecta a más de una de sus columnas).

Las anteriores restricciones también, como se ha indicado, se pueden captar con Create Assertion. Sin embargo, aquellas restricciones que afecten a información contenida en más de una tabla **necesariamente** se implementaran con el Create Assertion.

Pongamos un interesante ejemplo. Captar una **restricción de existencia** de una entidad respecto a una binaria con conectividad muchos a muchos.



Al transformar el anterior E/R a modelo relacional no podemos captar directamente que todo *alumno* participa de la relación "*NOTES*" , es decir, que todo alumno está matriculado. En el esquema lógico relacional se expresaría con una fórmula propia de los lenguajes teóricos del

modelo relacional (álgebra relacional, cálculo relacional de tuplas,…). Y en SQL lo captaríamos como sigue:

No queda más remedio que utilizar aquí el predicado EXISTS sin haberlo explicado. No pasa nada se sobreentiende su semántica. Estamos expresando:
ALUMNES[Nexpd] ⊂ NOTES[Nexpd]

```
Create Assertion Constraint RestExistenciaAlumnes
Check Not Exists (  Select  *
                    From Alumnes Al
                    Where Not Exists (  Select *
                                        From Notes N
                                        Where  Al.Nexp=N.Nexpd) )
```

<u>"No existen alumnos para los que no existen asignaturas de las que no se hayan matriculado" ➔ "Todos los alumnos se han matriculado".</u>

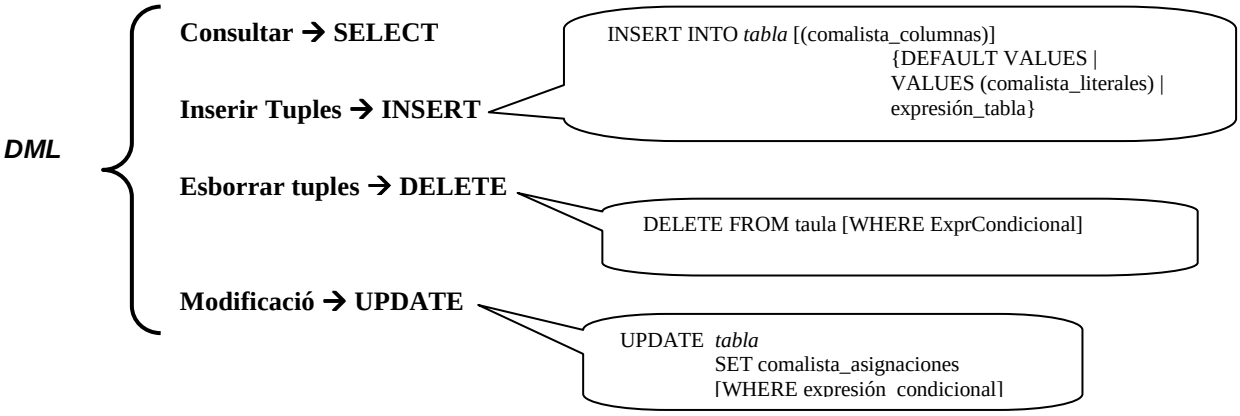
Por último nos queda decir que aquellas restricciones no "implementables" mediante Create Assertion se captaran a través de disparadores o via programa , y como último recurso, mediante el manual de usuario.

Y con esto terminamos el estudio de las restricciones de integridad y, por ende, el DDL del SQL. Sólo indicar como despedida que en el DDL también disponemos de sentencias de modificacición (ALTER) y eliminación (DROP) de las definiciones previamente establecidas con su pertinente CREATE. Algunas sintaxis son:

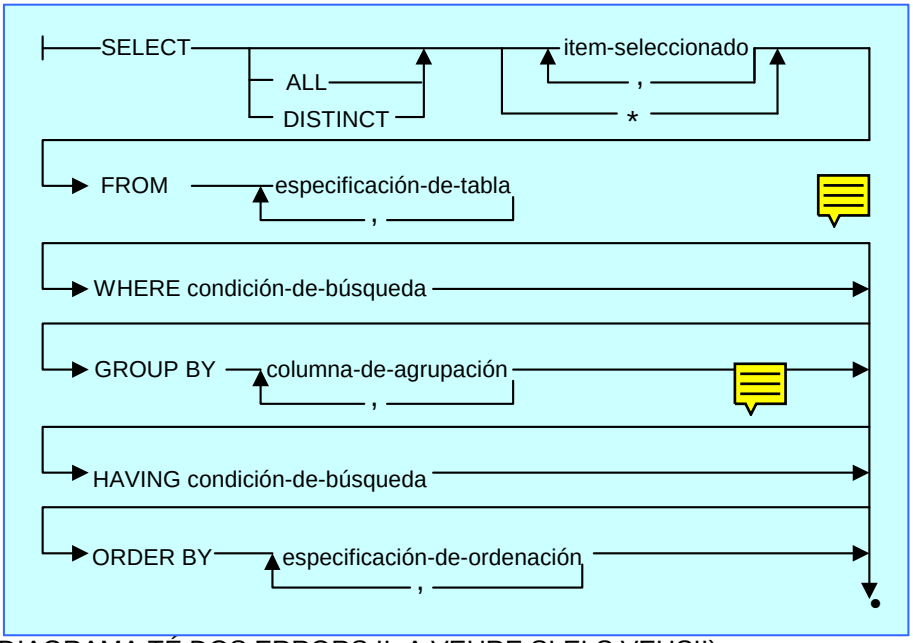
DROP TABLE *TaulaBàsica* {RESTRICT | CASCADE}	DROP ASSERTION *Restricció*	ALTER TABLE *TaulaBàsica* {ADD *DefColumna* | ALTER ……| DROP *columna* …}

3. DML

El lenguaje de manipulación de datos del SQL , es decir, el DML ofrece instrucciones para las operaciones nucleares de manipulación de datos.



Mención aparte merece la sintaxis de la todopoderosa SELECT. Y como en la variedad está el gusto, ahí va la sintaxis mediante un bonito diagrama sintáctico.



(EL DARRER DIAGRAMA TÉ DOS ERRORS !!. A VEURE SI ELS VEUS!!)

A partir de ahora la estrategia consiste en presentar ejemplos de sentencias SELECT de menor a mayor complejidad e intercalar, oportunamente, apuntes de tinte conceptual. El epígrafe acabará con ejemplos de INSERT, DELETE y UPDATE.

Las sentencias ejemplo se referirán al esquema lógico de siempre, a saber:

ALUMNES = Nºexped + edat + nom + tlf + pes
ASSIGNATURES = CodAsig + desc
NOTES = Nºexped + CodAsig + nota C.Aj { Nºexped → ALUMNES ; CodAsig → ASSIGNATURES

```
SELECT *
FROM ALUMNES
```

La més simple de totes. Tornem tota la informació de la taula ALUMNES

```
SELECT Nom, edat, pes*1000
FROM ALUMNES

WHERE nom IS NULL AND tlf LIKE '96%';
```

Podem ficar expressions en la projecció. Expressem el pes en grams.

Filtrat horitzontal. Incorrecte seria *nom=NULL* .
Predicat LIKE→ Els tlfs que comencen per 96

```
SELECT  1000*AVG(pes)
FROM ALUMNES
WHERE edat BETWEEN 19 AND 30 AND tlf  NOT LIKE '96%'
```

← Funció agrupada. També tenim MAX, MIN, COUNT i SUM

Sucre sintàctic: equival a un AND

```
SELECT Nexp, AVG(nota)
FROM NOTES
GROUP BY Nexp
```

Agrupem la taula. Cada grup conté el mateix Nexp. En una taula agrupada només puc projectar amb funcions agrupades, literals o expressions de literals i amb el (els) camp(s) d'agrupació.

```
SELECT AL.Nexp, AL.Nom, AVG(N.nota)

FROM ALUMNES AS AL, NOTES AS N

WHERE AL.Nexp=N.Nexp AND AL.edat<=30

GROUP BY AL.Nexp, AL.Nom

HAVING COUNT(*) = 4
```

Producte cartesià. Ambdós taules tenen els seus alias o variables de recorregut.

Afegim el camp ***nom*** aquí per a que puga sortir en l'agrupació.
No canvia, en aquest cas, la granularitat!!

Filtrem els grups!!. Sols queden aquells alumnes matriculats exactament en 4 assignatures

Primer es fa el producte cartesià. Després es filtren les tuples amb la clàusula WHERE. Amb les tuples supervivents s'agrupen per Nexp (el nom està com "espectador" sintàctic). Una vegada constituïts els grups es filtren amb la condició del HAVING i per als grups que queden es porta a terme la projecció.

```
SELECT DISTINCT Nexpd
FROM NOTES

WHERE nota IN (   select nota
                  from notes
                  where Nexpd='123')
```

El *DISTINCT* actua al final. Elimina duplicats

predicat IN::= operando [NOT] IN {(ComaListaValors) | subquery}

Subquery. No hi ha colisió entre els dos noms de la taula ***notes*** !!.

Tenemos muchos otros predicados, aparte del predicado IN, que aumentan considerablemente la expresividad de las condiciones de las cláusulas WHERE. A saber: ALL, ANY o SOME, [NOT] EXISTS, UNIQUE,...

Existen entre ellos equivalencias. Por ejemplo tenemos que

ExprEscalar < ALL(Subquery) ⇔ NOT(ExprEscalar >= ANY(Subquery))

En otras ocasiones presentan sutiles diferencias. Supongamos que deseamos listar la descripción de aquellas asignaturas donde todo el mundo ha sacado más de un nueve:

Solución buena	**Aquí sacaríamos también la descripción de aquellas asignaturas en las que nadie tiene nota, es decir, de las que nadie se ha matriculado**
SELECT desc FROM ASSIGNATURES A WHERE **9 < ALL** (select nota from notes n where .CodAsig=A.CodAsig)	SELECT desc FROM ASSIGNATURES WHERE CodAsig NOT IN (select CodAsig from notes where nota <=9)

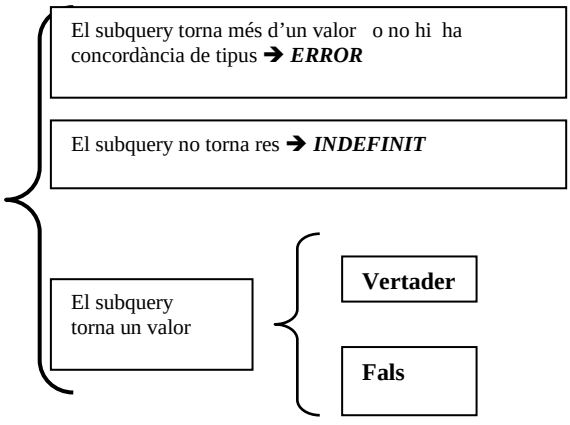
3.1. Lògica trivaluada del SQL

Siguiendo con las expresiones condicionales de las cláusulas WHERE cabe reseñar que **la lógica del SQL es trivaluada**. Las condiciones se evaluan a **cierto, falso e indefinido**. Sólo pasan el filtro de las WHERE aquellas tuplas que "emitan" un verdadero en la evaluación de la expresión condicional. La tabla de verdad del SQL queda como sigue:

F	G	F OR G	F AND G
cierto	cierto	cierto	cierto
cierto	falso	cierto	falso
cierto	indefinido	cierto	indefinido
falso	cierto	cierto	falso
falso	falso	falso	falso
falso	indefinido	indefinido	falso
indefinido	cierto	cierto	indefinido
indefinido	falso	indefinido	falso
indefinido	indefinido	indefinido	indefinido

F	-F
cierto	falso
falso	cierto
indefinido	indefinido

Suposem **WHERE 4 < (subquery)**



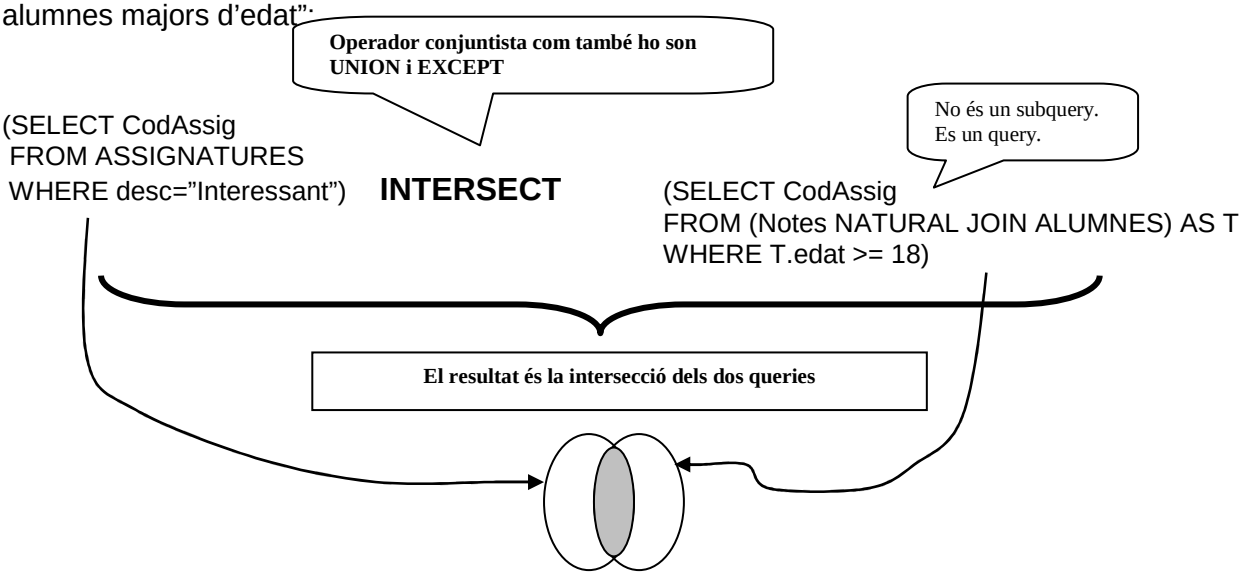
I que podem dir de la clàusula FROM. Generalment la gent fica el producte cartesià però hi ha altres possibilitats:

Dues solucions equivalents: "Expedient i edat de tots els matriculats en assignatures interessants"	
SELECT A.Nexp, A.edat FROM ALUMNES A, NOTES N, ASSIGNATURES ASSIG WHERE A.Nexp=N.Nexp AND AS.cod=N.cod AND ASSIG.desc='interessants'	SELECT A.Nexp, A.edat FROM ((ALUMNES ***NATURAL JOIN*** NOTES) ***NATURAL JOIN*** ASSIGNATURES) AS T WHERE T.desc='interessants'

Hem fet una concatenació natural i no un producte cartesià (també anomenat cross join) de les tres taules. No cal especificar cross join és suficient en separar amb comes les taules. Hi ha moltes més possibilitats de concatenació: LEFT JOIN, RIGHT JOIN, etc... però agafem la corbella !!.

3.2. Visió conjuntista

I acabem l'exploració de la SELECT amb una curiosa i oblidada ***visió conjuntista*** de les *selects*. "Volem obtenir els codis de les assignatures interessants en les que s'han matriculat alumnes majors d'edat":



Com ja podeu sospitar aquesta solució és equivalent a

SELECT S.CodAssig
FROM ALUMNES A, NOTES N, ASSIGNATURES S
WHERE A.edat >18 AND A.Nexp=N.Nexp **AND** S.CodAssig=N.cod AND S.desc='Interessant'

Fer servir l'operador conjuntista UNION equival a OR's. I l'operador EXCEPT equival a AND NOT.

3.3. Insercions, esborrats i canvis

I com acomiadament del DML vos oferisc uns examples de les altres sentencies:

Example	Observacions
UPDATE Notes SET nota = nota +1 WHERE CodAssig IN (select CodAssig from Assignatures where desc='Difícil")	Augmentem en un punt les notes de les assignatures difícils (on hi ha gent matriculada clar),
DELETE FROM ASSIGNATURA	Esborrem totes les tuples de la taula però mantenint l'estructura de la taula. DROP TABLE ASSIGNATURA esborraria la taula sencera.
INSERT INTO ALUMNES (Nexp,nom) VALUES ('123','Andreuet')	Inserim una tupla. La resta de camps no mencionats queden NULS (o al valor per defecte del hipotètic CREATE DOMAIN o de la definició de columnes en el CREATE TABLE)

4. DCL

DCL = Data Control Language. Sentencias para la gestión de privilegios, autorizaciones y permisos. En ocasiones y, abusando del lenguaje, se tiende a considerar estos términos como sinónimos. Pero pongamos los "puntos sobre las ies":

Con las sentencias GRANT (concesión) y REVOKE (revocación) del DCL gestionamos los privilegios sobre la base de datos. Veamos ejemplos:

GRANT para permisos	
GRANT *select* **ON PERSONA** **TO moll**	Se concede al usuario moll podrá consultar información de la tabla PERSONA.
GRANT *select,insert(nom,edat)* **ON PERSONA** **TO *moll***	Además de concederle permiso para consultar toda la información de la tabla PERSONA, podrá insertar tuplas rellenando solamente los campos *nom, edat*.
GRANT *select,insert(nom,edat)* **ON PERSONA** **TO *moll*** **WITH GRANT OPTION**	Lo mismo de antes pero además el usuario moll tiene la potestad de ceder todos o parte de estos permisos a terceros usuarios.
GRANT *ALL* **ON PERSONA** **TO moll, pep**	Oferim tots els permisos de la taula PERSONA als usuaris *moll* i *pep*
GRANT *ALL* **ON PERSONA** **TO PUBLIC**	El DBA concede todos los permisos (consultar, borrar, cambiar,..) del objeto tabla PERSONA a todos los usuarios.

La sintaxis de l'estàndart és,

GRANT <*ComaListapermiso*>
ON *tabla*
TO {<*ComaListaUsuarios*>| PUBLIC}
[WITH GRANT OPTION]

<permiso>::= {ALL|SELECT | INSERT[(<ComaListaCampo>)] | DELETE | UPDATE[(<ComaListaCampo>)]}

Cada sistema pot enriquir aquest llistat incorporant, per exemple, *triggers*, etc.Mirem com queda PostgreSQL…

```
GRANT
{{SELECT|INSERT|UPDATE|DELETE|REFERENCES|TRIGGER}[,..] | ALL[PRIVILEGES]}
    ON [ TABLE ] tablename [, ...]
    TO { [ GROUP ] rolename | PUBLIC } [, ...]
[ WITH GRANT OPTION ]
```

La sentència REVOKE revoca els permisos prèviament concedits…

5. EMBEDED SQL

Hasta ahora nos hemos centrado en el uso interactivo de SQL que, como sabemos, carece de estructuras típicas de control (IF,FOR, WHILE, etc). El diseño e implementación de una aplicación de gestión pasa por "conjugar/embeber" adecuadamente las declarativas y potentes sentencias de SQL en un lenguaje de 3ª generación que nos resulte familiar, por ejemplo C , al que etiquetaremos como **lenguaje anfitrión. El SQL se "sumergirá" como lenguaje huésped**.

Cualquier sentencia de SQL que se pueda utilizar de modo interactivo también se puede emplear en un programa de aplicación escrito en lenguaje anfitrión.. Esto, entre otras cosas, permite que las sentencias SQL a utilizar en un programa pueden ser probadas por primera vez utilizando SQL interactivo y luego codificadas en el programa.

A continuación se muestra un sencillo programa en C que utiliza SQL embebido. Se solicitará del usuario un **número de oficina** y se obtendrá de la base de datos la **ciudad, región, ventas y objetivo de ventas** de dicha oficina.

Área de comunicaciones de SQL. ORACLE escribe aquí "cómo le ha ido" al ejecutar una sentencia de SQL embebido

```
EXEC SQL INCLUDE sqlca.h;

void error_acceso (void);
void mal_numero (void);

main ( );
 {
        EXEC SQL BEGIN DECLARE SECTION
            int ofinum;             /* numero de oficina del usuario */
            char ciudad[15];                /* nombre de ciudad obtenido */
            char region[8];                 /* nombre de región obtenido */
            float objetivo;                 /* objetivo obtenido */
            flota ventas;           /* ventas obtenidas */
        EXEC SQL END DECLARE SECTION;
```

Sección de declaración: Estas variables se denominan *variables_sql* para distinguirlas del resto de variables "normales" del prg.
Oracle usa las variables_sql para pasar datos e información de estado al programa de aplicación y viceversa.

```
        /* prepara la gestión de errores */
        EXEC SQL WHENEVER SQLERROR DO error_acceso ( );
        EXEC SQL WHENEVER NOT FOUND DO mal_numero ( );
```

Gestión de errores. EXEC SQL WHENEVER <condición> <acción>. Se estudia detenidamente más adelante.

```
        /* pide al usuario que escriba un numero de oficina */
        printf (" Introduzca un numero de oficina: "); scanf ("%d", &ofinum);

        /* ejecuta la consulta SQL */
    EXEC SQL SELECT CIUDAD,REGION,OBJETIVO,VENTAS
        INTO :ciudad, :region, :objetivo, :ventas;
        FROM OFICINA
        WERE OFICINA.OFINUM = :ofinum
```

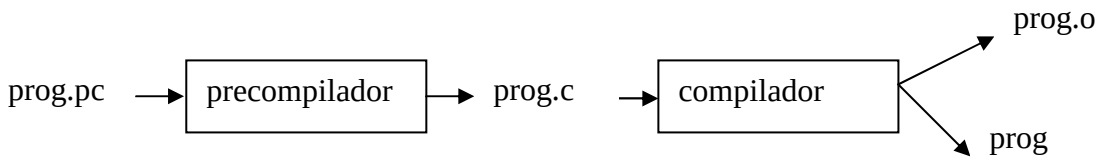
Las *variables_sql* deben ir precedidas por "dos ptos" cuando se usen en sentencias SQL.

```
        /* visualiza los resultados */
        printf("Ciudad %s\n", ciudad);
        printf("Region %s\n", region);
        printf("Objetivo %s\n", objetivo);
        printf("Ventas %s\n", ventas);
```

```
        exit (0);
  }
void error_acceso ( )
  {
     printf("error SQL:  %ld\n", sqlca.sqlcode);
     exit( 0 );
  }
void mal_numero ( )
  {
     printf ("numero de oficina no valido \n");
     exit (0);
  }
```

Un programa que contiene sentencias de SQL embebido no puede ser compilado directamente por un compilador del lenguaje de programación anfitrión. El SGBD nos proporcionará los *precompiladores*, uno para cada lenguaje distinto.

El precompilador produce un fichero de salida que es el programa fuente, en el que todas las sentencias de SQL embebido han sido sustituidas por llamadas a las rutinas "privadas" del SGBD. Este fichero fuente es el que se compila mediante el compilador del lenguaje anfitrión, obteniéndose el programa ejecutable. Gráficamente,



Veamos un ejemplo del trabajo llevado a cabo por el precompilador de C. Supongamos que declaramos,

```
EXEC SQL BEGIN DECLARE SECTION;
        VARCHAR tira[20];
EXEC SQL END DECLARE SECTION;
```

Varchar es un tipo predefinido de Oracle para almacenar cadenas de longitud variable. Tenemos aquí un vector de 20 elementos de tipo Varchar.

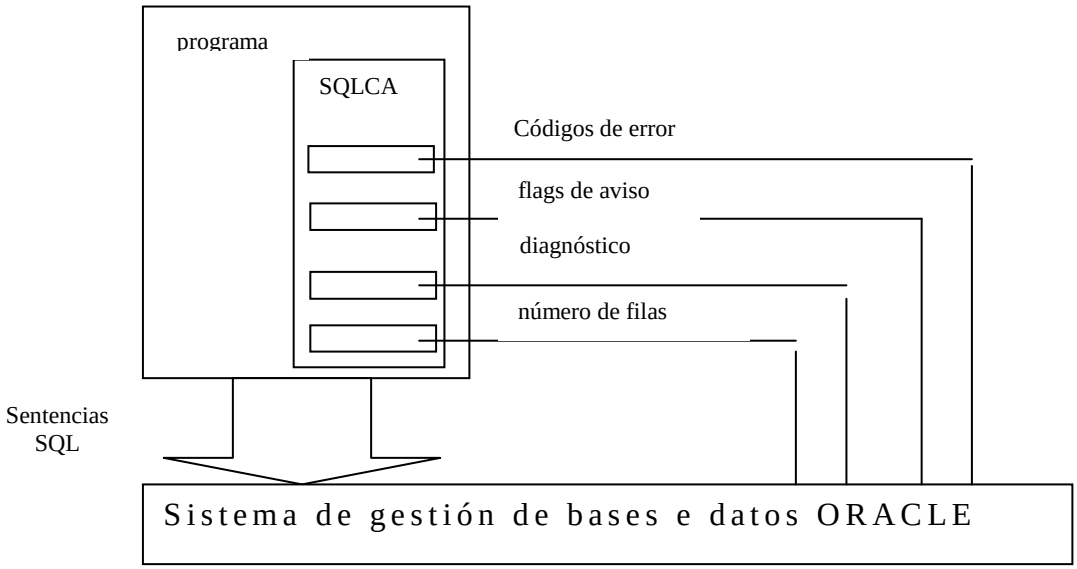
Resultado de la precompilación

```
Struct  {
     unsigned short len;
     unsigned char arr[20];
} tira;
```

5.1. El área de comunicaciones de SQL (SQLCA)

La SQLCA es una estructura de datos del lenguaje anfitrión que permite la detección de eventos (errores, estados, etc.). Cada vez que se ejecuta una sentencia SQL, Oracle escribe en la SQLCA información acerca de "como le ha ido" al ejecutar dicha sentencia.

La SQLCA se encuentra definida en el fichero **sqlca.h** **que siempre deberemos incluir** en nuestros programas de aplicación mediante la sentencia INCLUDE.



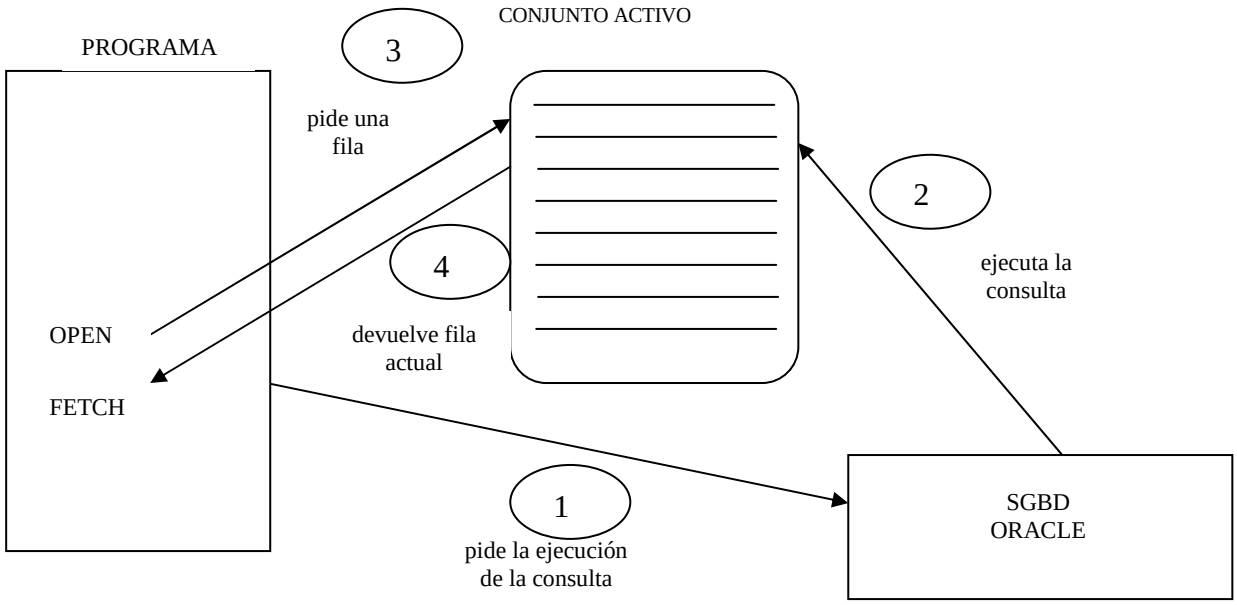
5.2. Cursors

Cuando una consulta devuelve más de una fila, podemos definir un cursor para procesar todas y cada una de las filas devueltas por la consulta.

Podemos ver el "cursor" tanto como la matriz que almacena las filas devueltas como el puntero que apunta a la *fila actual* de dicha matriz.

Las siguientes sentencias son las que permiten definir y utilizar cursores: DECLARE, OPEN, FETCH, CLOSE.

Primero se declara el cursor mediante DECLARE asociando el nombre del cursor con una consulta. Después mediante la sentencia OPEN se ejecuta la consulta. Las filas devueltas forman lo que se llama el *conjunto activo* del cursor. Con FETCH accedemos para leer la fila activa del cursor. Cada FETCH devolverá una fila distinta. Finalmente, se cierra el cursor mediante la sentencia CLOSE. Tras la ejecución de esta sentencia, el conjunto activo queda indefinido.



Ejemplos

```
EXEC SQL DECLARE mi_cursor
CURSOR FOR
        SELECT NOMBRE, CUOTA, VENTAS
        FROM REPVENTA
        WHERE REPVENTA.VENTAS > :venta_mes
        ORDER BY REPVENTA.NOMBRE;
.....
```

Esta sentencia define un cursor, asociándole una consulta. La sentencia SELECT asociada al cursor no lleva la parte del INTO.

```
EXEC SQL OPEN mi_cursor;
```

Ejecutamos la consulta asociada al cursor e identificamos el conjunto activo. El "***cursor"*** se sitúa justo antes de la primera fila del conjunto activo. En ese momento la consulta se ha realizado pero el programa todavía no tiene los datos devueltos por la consulta en ninguna de sus variables. Mientras el cursor esté abierto, el conjunto activo no cambia. Si las tablas de las que se extrajo el conjunto activo son modificadas, las modificaciones no se reflejan en el conjunto activo, con lo que se podría estar accediendo a datos "antiguos".

```
EXEC SQL FETCH mi_cursor
        INTO :repventa, :cuota, :ventas;
```

Antes de poder utilizar la sentencia FETCH el cursor debe haber sido declarado y abierto. La primera vez que se ejecuta un FETCH, el cursor se mueve a la primera fila del conjunto activo, siendo ésta la fila actual. Cada FETCH que se realice después hará que el cursor avance en el conjunto activo de una fila a la siguiente. El cursor sólo se puede mover hacia delante, por lo que si se desea acceder a una fila que ya ha sido accedida antes mediante un FETCH, se debe cerrar el cursor y volverlo a abrir, comenzando de nuevo desde la primera fila del conjunto activo.

Cuando el conjunto activo es vacío o se han terminado las filas, FETCH devuelve un código de error Oracle en la SQLCA.

```
....
....
EXEC SQL CLOSE mi_cursor;
```

Liberamos la memoria del cursor. No podemos efectuar una nueva FETCH. Sí podemos, en cambio, volver a abrir el cursor.
Pero los nuevos valores pueden, obviamente, ser distintos de los anteriores.

Finalment cal ressenyar que els cursors no sols es fan servir en SQL *"embebido"*, també els podem trobar en el PL/SQL

5.3. Transaccions

Una transacción es una secuencia de sentencias SQL consideradas como un todo de modo que, o se ejecutan todas sus sentencias o ninguna.

La mayoría de las transacciones se inician de forma implícita al utilizar alguna sentencia que empieza con `CREATE`, `ALTER`, `DROP`, `SET`, `DECLARE`, `GRANT` o `REVOKE`, aunque existe la sentencia SQL para iniciar transacciones, que es la siguiente:

```
SET TRANSACTION {READ ONLY|READ WRITE};
```

Ejemplo:

```
SET TRANSACTION READ WRITE;
UPDATE empleados SET sueldo = sueldo – 1000 WHERE num_proyec = 3;
UPDATE empleados SET sueldo = sueldo + 1000 WHERE num_proyec = 1;
COMMIT;
```

COMMIT hace permanentes todos los cambios realizados por la transacción en curso.

Aparte de COMMIT también tenemos las siguientes primitivas:

- ROLLBACK finaliza la transacción en curso y deshace todos los cambios realizados desde que comenzó la transacción.
- SAVEPOINT pone una marca en un punto intermedio de la transacción. Utilizado con ROLLBACK deshace parte de una transacción hasta ese pto intermedio.

Una transacción también termina cuando se produce un fallo del sistema. En este caso el SGBD deshace la transacción en curso.

En SQL embebido, si no se subdivide el programa con las sentencias COMMIT y ROLLBACK, Oracle, por ejemplo, interpreta el programa como una sola transacción (a menos que el programa tenga sentencias de definición de datos, ya que estas hacen un COMMIT automático).

Con la sentencia ***EXEC SQL SAVEPOINT MiPrimeraMarca;*** ponemos una marca en un punto intermedio de la transacción. **Estas marcas nos permiten subdividir transacciones que tienen muchas sentencias**, proporcionándonos un mayor control sobre las mismas. Por ejemplo, si una transacción realiza varias funciones podemos poner una marca en el principio de cada función. Entonces, si una de las funciones falla, podemos volver al estado en que nos encontrábamos antes de lanzar dicha función.

Para deshacer parte de una transacción, utilizamos marcas con la sentencia ROLLBACK:

```
EXEC SQL ROLLBACK TO SAVEPOINT MiPrimeraMarca;
```

Borramos todas las marcas posteriores a MiPrimeraMarca excepto ésta.

```
EXEC SQL ROLLBACK;
```

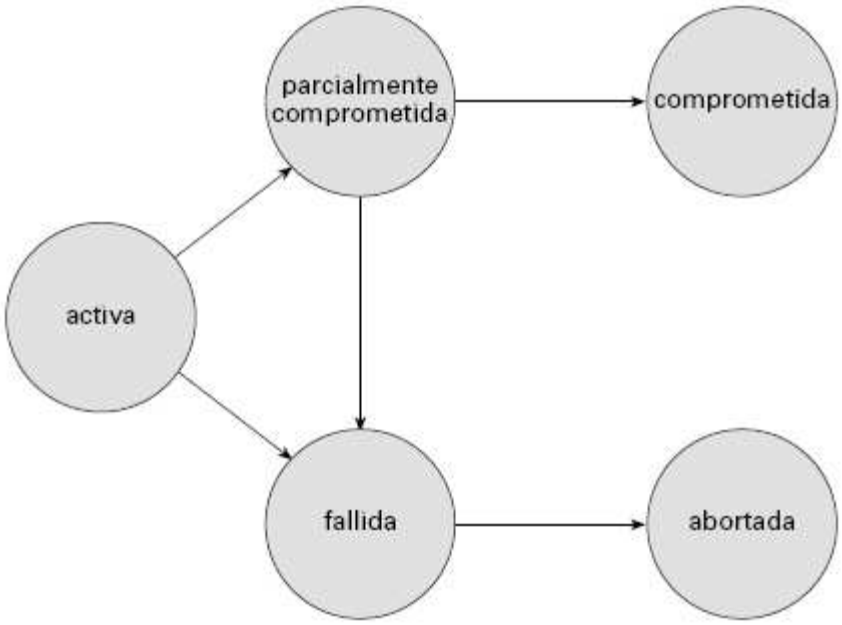
1.- Deshace todos los cambios hechos sobre la base de datos en la transacción en curso.
2.- Borra todas las marcas.
3.- Finaliza la transacción en curso.

Si un programa termina sin ejecutar COMMITT o ROLLBACK la norma no especifica si comprometer la transacción o retrocederla. Depende de cada SGBD concreto. En muchas implementaciones SQL, cada instrucción SQL es de manera predeterminada una transacción y se compromete tan pronto como se ejecuta.

El compromiso automático de las instrucciones SQL individuales se puede desactivar si es necesario ejecutar una transacción que conste de varias instrucciones SQL. La forma de desactivación del compromiso automático depende de la implementación SQL específica.

Una alternativa mejor, que es parte de la norma SQL:1999 (pero actualmente sólo soportada por algunas implementaciones SQL), es permitir encerrar varias instrucciones SQL entre las palabras clave ***begin atomic… end***. Todas las instrucciones entre las palabras clave forman así una única transacción.

5.3.1. Cicle de vida d'una transacció



5.4. SQL Dinàmic

Algunas aplicaciones precisan procesar distintas sentencias SQL en tiempo de ejecución. Por ejemplo, un prg generador de informes debe ejecutar distintas SELECT según el informe que se pretenda generar. En estos casos, la forma concreta de la consulta se desconoce en el momento de codificar el prg. Sólo, como hemos indicado, se conoce en tiempo de ejecución y, obviamente, los resultados cambian entre distintas ejecuciones ➔ **Utilizaremos sentencias SQL dinámicas**.

Las sentencias SQL dinámicas no aparecen embebidas en el prg fuente. Se almacenan en tiras de caracteres que se suministraran al prg en tiempo de ejecución. Dichas tiras se "rellenaran" interactivamente o leyendo desde ficheros.

El uso de SQL dinámico flexibiliza las aplicaciones a costa de aumentar la complejidad del código.

Exemple

...
char * prog_sql =«update cuenta set saldo = saldo * 1.05 where número-cuenta = ?»

{

...
EXEC SQL *prepare* prog_din *from* :prog_sql; /* S'exigeix guardar la tira que conté la consulta en la variable *prog_din* que no cal declarar*/

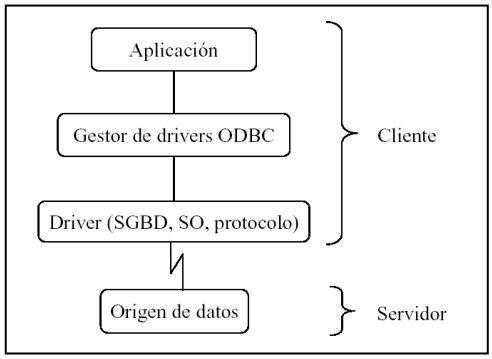
char cuenta[10] = «C-101»;

EXEC SQL execute prog_din using :cuenta;

...}

Este programa de SQL dinámico contiene una interrogación '?' que representa una variable que se debe proporcionar en la ejecución del programa.

Tanmateix, aquesta sintaxi requereix una extensió del llenguatge anfitrió i un preprocessador. Alternativament podem fer servir una API d'accés a la base de dades sense tocar el llenguatge ➔ API ODBC o JDBC (Java) ← TEMA 41



6. PLPGSQL DE POSTGRES

PL/pgSQL **(**ProceduraL PostGres SQL).

El PL/pgSQL ens permetrà construir funcions utilitzant un llenguatge procedimental. Podrem definir variables, crear bucles, introduïr sentències de SQL interactives; definir cursors.

En altres paraules no cal surtir fora de casa, fent servir un llenguatge anfitrió, per augmentar l'expressivitat del SQL. El propi sistema ens ho dona.

6.1. Funcions

```
CREATE [OR REPLACE] FUNCTION NomFunció [(llista_paràmetres)]
RETURNS tipus_del_valor_tornat AS
'
sentènciess
'
LANGUAGE [plpgsql | C | pltcl…];
```

Un primer exemple

```
CREATE OR REPLACE FUNCTION aeuro(numeric) RETURNS numeric AS
'
declare
   euro constant numeric:=166.386;
  begin
   return round($1/euro,2);
  end;
' LANGUAGE plpgsql;
/* Hem passat de pesetes a euros */
```

Segon exemple

```
create or replace function EsPar(numeric) returns text as
'
declare
 resultat text;
begin

  if abs($1) > 999 then raise notice ''El numero introduit es massa llarg'';
```

```
   /*NOTICE (write the message into the postmaster log and forward it to the
client application)*/
   /* Proveu després amb EXCEPTION i notareu la diferència amb NOTICE */

    elseif (mod($1,2)=0) then resultat:= ''El numero  introduit
''||to_char($1,999)||'' Es Par'';
                        else resultat:= ''El numero  introduit
''||to_char($1,999)||'' NO Es Par'';
  end if; /* Elseif en lloc   else if   permet fer servir només un endif*/
          /* Cura amb les cometes fins i tot dintre dels comentaris!!*/
 return resultat;
end;
'
language plpgsql;
```

Tercer exemple

```
CREATE OR REPLACE FUNCTION Max2(numeric,numeric) RETURNS numeric AS
'
declare
   aux numeric;
  begin
   aux:=$1;
   if $2>$1 then aux:=$2;
   end if; /* end if  observeu que va separat */
   return aux;
  end;
'
LANGUAGE plpgsql;
```

6.1.1. Declaració de Variables

S'han de declarar totes (excepte les variables comptador dels bucles FOR) en el bloc **DECLARE.** Aquesta és la sintaxi de declaració de variables:

nom [CONSTANT] tipus [NOT NULL] [{DEFAULT|:=}expressió];

Per exemple:

DECLARE
v1 VARCHAR;
v2 CONSTANT TEXT := 'Hola';
v3 NUMERIC(5) := 0;
v4 DATE NOT NULL DEFAULT '7-7-77';

6.1.2. Sentències

Assignació

Es pot fer de dues maneres:

- - Directa, amb l'operador := (a := 10; b := a * 0.5;)
- - Assignar el resultat d'una consulta (una única fila) a una variable o variables amb INTO. INTO pot anar abans o després de les columnes seleccionades:

SELECT Max(ram) INTO a FROM NODE;
SELECT INTO a Max(ram) FROM NODE;
SELECT Max(ram),Min(ram),Avg(ram) INTO a,b,c FROM NODE;

Condicional

IF *condició*
THEN
Sentències;
[ELSE
***sentències;*]**
END IF;

També es pot utilitzar ELSIF que equival a un ELSE seguit de IF. D'aquesta manera només hi haurà un END IF al final. És la cosa més pareguda al CASE.

Bucles

S'aconsegueixen per mig de **LOOP ..... END LOOP;**

- Bucle incondicional

  LOOP
  Sentències
  END LOOP;

  tal i com està, el bucle és infinit; per eixir **EXIT** o **EXIT WHEN *condició***.

- Bucle **FOR**

  FOR *variable* IN [REVERSE] *valor_mín .. valor_màx* LOOP
  Sentències
  END LOOP;

  La variable comptador no s'ha de declarar. Només té validesa dins del bucle.

- Bucle WHILE

  WHILE *condició* LOOP
  Sentències
  END LOOP;

Sentències SQL

Es poden executar sense problema totes aquelles sentències que no tornen cap resultat (INSERT, UPDATE, DELETE, CREATE TABLE, ...). La sentència SELECT, que sí que torna un valor s'ha d'utilitzar amb la clàusula INTO (com ja hem vist), o dins de PERFORM (com ara veurem).

Execució d'altres funcions

En moltes ocasions ens farà falta executar un altra funció. Si aquesta torna un valor la posarem dins d'una sentència (assignació, en una condició, ...). Però i si no torna cap valor? El mateix podríem dir d'una sentència SELECT (encara que la utilitat d'açò és més dubtosa). Ho farem per mig de PERFORM, que executa una sentència, ignorant el possible valor tornat per aquesta. Per exemple:

```
PERFORM cont_mussol();
PERFORM (SELECT * FROM NODE);
```

6.1.3. Missatges i Excepcions

RAISE *nivell* '*format*' [,*variable1*[,...]];

En nivell posarem un dels següents: DEBUG, LOG, INFO, NOTICE, WARNING o EXCEPTION, on la major part només donen un avís (amb l'encapçalament corresponent). Però EXCEPTION a més de traure l'avís avorta l'execució.

En format posarem una cadena amb el comentari que vulguem. Podrem posar **%** les vegades que vulguem, i se substituiran pel contingut de les variables que tinguem a continuació. Així, per exemple:

```
s1 := 'Pepet';
s2 := CURRENT_DATE;
RAISE NOTICE 'Hola, %. Avui és %',s1,s2;
```

Provocarà la següent eixida:

```
NOTICE: Hola, Pepet. Avui es 2004-07-18
```

6.2. Triggers

Los sistemas de bases de datos SQL usan ampliamente los disparadores, aunque antes de SQL:1999 no fueron parte de la norma. Por desgracia, cada sistema de bases de datos implementó su propia sintaxis para los disparadores, conduciendo a incompatibilidades.

Un TRIGGER o disparador és un procediment que se dispara quan s'acompleix un determinat event com ara, una actualització, una inserció o l'esborrany d'una tupla.

Per acabar, cal indicar que una BD amb disparadors sol dir-se d'ella que es tracta d'una BD activa.

Abans de l'estàndart SQL:1999 no formava part de la *norma* , així doncs, cada SGBD va implementar la seua pròpia sintaxi provocant incompatibilitats entre ells.

En el moment de crear el trigger especificarem una funció que s'executarà quan es produesca l'event. Aquesta funció pot estar escrita en qualsevol llenguatge de programació instal·lat. La sintaxi és:

```
CREATE [OR REPLACE] TRIGGER nom_trig {BEFORE | AFTER}
{INSERT | DELETE | UPDATE} [OR {INSERT | DELETE |UPDATE} ...]
ON nom_taula
[FOR EACH {ROW | STATEMENT}]
EXECUTE PROCEDURE nom_funció ([paràmetres]);
```

L'opció **OR REPLACE** ens permetrà no haver d'esborrar el trigger si volem refer-lo. **BEFORE** o **AFTER** indiquen quan s'ha d'activar el trigger: abans de produir-se l'acció d'inserir, esborrar o modificar, o després.

Una actualització pot afectar més d'una fila. Aleshores ens plantegem si s'ha de disparar el trigger per a cada actualització de cada fila, o si només una (abans o després d'actualitzar). Ho podrem especificar per mig de **FOR EACH ROW** o **FOR EACH STATEMENT**.

Exemple

Suposem que volem emmagatzemar la columna Nom de la taula Alumnes sempre en majúscules independentment de com ens envien el pertinent INSERT les aplicacions client..

```
create trigger InserirNomMajuscules
before insert
on alumnes
for each row
execute procedure FNomMajuscules();
```

I el codi de la funció associada al trigger…

```
create or replace function FNomMajuscules() returns trigger as
$$
begin
new.Nom:=upper(new.Nom);
return New;
end;
$$
language plpgsql;
```

Los disparadores se deberían escribir con sumo cuidado, dado que un error de un disparador detectado en tiempo de ejecución causa el fallo de la instrucción de inserción, borrado o actualización que inició el disparador. En el peor de los casos esto podría dar lugar a una cadena infinita de disparos.

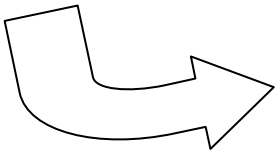
Por ejemplo, supóngase que un disparador de inserción sobre una relación realice otra (nueva) inserción sobre la misma relación. La acción de inserción dispara otra acción de inserción, y así hasta el infinito.

7. SOBRE L'ESTÀNDART SQL99

La norma SQL:1999 -también llamada SQL3- extiende el lenguaje de definición de datos, así como el lenguaje de consultas, y en particular da soporte a atributos de tipo estructurados, a la herencia, etc. En otras palabras, estamos ante el estándard **Objeto-Relacional**.

Tipos Estructurados

create **type** Editorial as
(nombre varchar(20),
sucursal varchar(20));



create type Libro as
(título varchar(20),
array-autores varchar(20) array [10],
fecha-pub date,
editorial **Editorial**,
lista-palabras-clave **setof**(varchar(20)))

create table libros of type Libro

Herencia

create type Persona
(nombre varchar(20),
dirección varchar(20)
departamento varchar(20));

create **type** Estudiante
under Persona
(curso varchar(20));

create **type** Profesor
under Persona
(sueldo integer);

create table persona of Persona

create table estudiantes of Estudiante
under persona

create table profesores of Profesor
under persona

- Los tipos de las subtablas deben ser subtipos del tipo de la tabla padre. Por tanto, cada atributo presente en *persona* debe estar también presente en las subtablas.
- Además, cuando se declaran *estudiantes* y *profesores* como subtablas de la tabla *persona*, cada tupla presente en *estudiantes o profesores* también están presentes implícitamente en *persona*. Así, si una consulta usa la tabla *persona*, encontrará no sólo las tuplas insertadas directamente en la tabla, sino también las tuplas insertadas en sus subtablas *estudiantes y profesores*.

Finalment i per no ésser exhaustius acabarem aquest overview sobre la norma objecte-relacional SQL 99 indicant que..


SQL99 No soporta herència múltiple


SQL99 defineix una sintaxi i semàntica per a la part procedural ➔ Funcions i procedimients