Llenguatge SQL. Extensió procedimental i immersió.

5

Isidre Guixà i Miranda
IES SEP Milà i Fontanals, d'Igualada

Sistemes gestors de bases de dades relacionals

Novembre del 2008
© Isidre Guixà i Miranda
IES SEP Milà i Fontanals
C/. Emili Vallès, 4
08700 - Igualada

En cas de suggeriment i/o detecció d'error podeu posar-vos en contacte via el correu electrònic `iguixa@xtec.cat`

Índex

Introducció

En aquests moments coneixem la potència del llenguatge SQL per efectuar consultes complexes i actualitzacions (insercions, modificacions i eliminacions) a les bases de dades. L'avantatja d'aquestes instruccions, que constitueixen el que s'anomena llenguatge SQL autosuficient, radica en que permeten un accés directe a la base de dades.

La possibilitat d'accedir directament a la base de dades per gestionar les dades no elimina la necessitat de continuar desenvolupant programes per gestionar les dades emmagatzemades. En l'actualitat hi ha diverses tècniques que ens permeten desenvolupar programes i/o subprogrames per automatitzar tasques de gestió de dades en les bases de dades.

Així, ens trobem amb que els principals SGBD faciliten uns llenguatges de tercera generació anomenats extensions procedimentals del llenguatge SQL, que permeten dissenyar petits programes a executar dins de guions i dissenyar subprogrames (funcions i accions) que s'emmagatzemen dins la base de dades i poden ser executats des de múltiples entorns.

Des de l'exterior, els actuals llenguatges d'alt nivell incorporen funcions per gestionar les dades de les bases de dades, fet que no forma part dels continguts d'aquest crèdit, però, en alguns casos, el propi SGBD facilita mecanismes per hostatjar sentències SQL dins llenguatges d'alt nivell. En tal situació es parla del llenguatge SQL hostatjat o immers en llenguatges d'alt nivell.

En el nucli d'activitat "Llenguatge PL/SQL. Fonaments" ens endinsem en el coneixement de l'extensió procedimental del llenguatge SQL que aporta el SGBD *Oracle* i, com és normal en l'estudi de qualsevol llenguatge de programació, n'estudiarem l'estructura dels programes, els tipus de dades i les estructures de control i, a més a més, el que és més important, com s'hi submergeix el llenguatge SQL.

En el nucli d'activitat "Codi PL/SQL dins la base de dades" aprofundim en la utilització del llenguatge PL/SQL per a escriure codi que queda emmagatzemat a la base de dades, fet que es produeix bàsicament en tres situacions: el disseny de subprogrames (funcions i accions), el disseny de paquets de subprogrames i el disseny de disparadors.

En el nucli d'activitat "SQL hostatjat en C" fem una petita introducció a com es desenvolupen programes amb SQL hostatjat en llenguatges de tercera generació pels que el fabricant del SGBD (*Oracle* en el nostre cas) en facilita la possibilitat.

Per aconseguir un bon aprenentatge del llenguatge PL/SQL amb totes les possibilitats que facilita i de la immersió de codi SQL en llenguatges d'alt nivell com el llenguatge C és necessari que aneu reproduint en el vostre ordinador tots els exemples incorporats en el text així com les activitats i els exercicis d'autoavaluació. I per a poder-ho fer continuarem utilitzant el SGBD *Oracle 11g* i les eines adequades seguint les instruccions del material web.

Objectius

En acabar la unitat didàctica heu de ser capaços del següent:

1) Emprar l'extensió procedimental del llenguatge SQL que aporten els SGBDR per tal d'incorporar les prestacions de la programació estructurada i modular.

2) Gestionar subprogrames i paquets de subprogrames emmagatzemats en la base de dades.

3) Dissenyar disparadors de base de dades.

4) Manipular paquets estàndards subministrats pel fabricant del SGBD.

5) Desenvolupar programes en llenguatge C utilitzant de format hostatjada el llenguatge SQL.

1. Llenguatge PL/SQL. Fonaments.

Els usuaris avantatjats de SGBD no en tenen prou amb la gestió de dades que proporciona el llenguatge SQL doncs en moltes ocasions interessarà automatitzar processos repetitius i/o prendre decisions de gestió de dades en funció del contingut de les pròpies dades i, per aconseguir-ho, cal disposar d'una extensió procedimental al llenguatge SQL, fet que l'*Oracle* proporciona amb el llenguatge PL/SQL.

Com en tot llenguatge procedimental, el domini del llenguatge PL/SQL passa pel domini de: estructura del programa, tipus de dades, estructures de control, interacció amb l'usuari, interacció amb el llenguatge SQL i tractament d'errors.

1.1. Extensions procedimentals al llenguatge SQL

Els SGBD relacionals faciliten el llenguatge SQL per executar diferents tipus de tasques en les bases de dades i s'acostuma a distingir, dins el llenguatge SQL, quatre subconjunts segons el tipus de tasca: llenguatge SQL-LC per a la consulta de dades (sentència `SELECT`), llenguatge SQL-LMD per a la manipulació de dades (sentències `INSERT`, `UPDATE` i `DELETE`), llenguatge SQL-LDD per a la definició de dades (sentències `CREATE`, `ALTER` i `DROP` aplicades a taules, índexs i altres estructures de dades) i llenguatge SQL-LCD per al control de dades (sentències `GRANT` i `REVOKE`).

Les instruccions facilitades pel llenguatge SQL, unes més complicades que altres, es poden posar en execució en una consola del SGBD de manera similar a les ordres de consola per a un sistema operatiu.

Exemples d'instruccions SQL executades des d'una consola

```
SQL> select emp_no, cognom, ofici
from emp;

EMP_NO                 COGNOM     OFICI
---------------------- ---------- ----------
7369                   SÁNCHEZ    EMPLEAT
7499                   ARROYO     VENEDOR
7521                   SALA       VENEDOR
...

14 files seleccionades

SQL> insert into dept (dept_no, dnom)
values (50, 'INFORMÀTICA');

1 fila creada
```

En ocasions pot interessar agrupar, en seqüència, diverses instruccions SQL que cal executar repetidament i per aquests casos els SGBD acostumen a facilitar la possibilitat de crear guions que agrupen les diverses sentències, de forma seqüencial.

Exemples de guions

Els fitxers `empresa.sql` i `sanitat.sql` són exemples de guions, doncs no són altra cosa que la seqüència ordenada d'instruccions SQL que faciliten:

- Connexió com a usuari `system`
- Eliminació d'usuari (esquema) corresponent si ja existia
- Creació d'usuari (esquema) corresponent
- Concessió de privilegis al nou usuari
- Connexió com al nou usuari
- Creació de taules i índexs amb inserció de dades

Podeu trobar els fitxers `empresa.sql` i `sanitat.sql` a la web del crèdit.

Però en multitud d'ocasions ens trobarem amb la necessitat d'executar un seguit d'instruccions i poder prendre decisions en funció dels seus resultats. Això no és factible amb la utilització de guions. Ens cal poder utilitzar nocions bàsiques de programació estructurada i modular (definició de variables, utilització d'estructures condicionals i iteratives i disseny de subprogrames) i, per aquest motiu, els SGBD acostumen a facilitar una **extensió procedimental** per al seu llenguatge SQL.

L'**extensió procedimental** d'un llenguatge SQL és un llenguatge de 3a. generació que permet dissenyar programes per a ser executats dins la base de dades i que inclouen sentències SQL.

Així, cada SGBD aporta la seva extensió procedimental que acostuma a tenir un nom. La taula 1 ens en presenta alguns.

Taula 1. Llenguatges procedimentals facilitats per diferents SGBD relacionals.

SGBD	Llenguatge procedimental
Oracle	PL/SQL
SQLServer	Transact-SQL
MySQL	Incorpora l'extensió procedimental a partir de les versions 5.0 (any 2006) i encara està en fase d'implementació. En el moment actual, no s'ha donat cap nom a aquesta extensió procedimental.

PL/SQL és l'extensió procedimental del llenguatge SQL implementat per *Oracle*, que combina la facilitat del llenguatge SQL per accedir a la base de dades amb les característiques de la programació estructurada i modular.

Els programes PL/SQL estan estructurats en blocs. Qualsevol programa està format, com a mínim, per un bloc, el qual pot contenir qualsevol nombre de sub-blocs.

Aplicacions importants del llenguatge PL/SQL són:

- Permet implementar programes senzills dins la base de dades, executables des de la consola textual del SGBD (*SQL*Plus*) i des d'altres eines gràfiques substitutòries de la consola textual (*SQL Developer*, *SQL Worksheet*,...)

- Permet dissenyar subprogrames (procediments i funcions), que s'emmagatzemen dins la pròpia base de dades, i que poden ser cridats des d'innumerables llocs.

- Permet desenvolupar disparadors dins la base de dades (conjunt d'instruccions que s'executen automàticament davant un cert esdeveniment).

- És el llenguatge de programació utilitzat en *Oracle Developer* per la programació dels esdeveniments.

***Oracle* Developer**

És una eina de 4a. generació amb la que es generen aplicacions visuals utilitzant objectes, dels que s'ha de programar l'actuació en els diferents esdeveniments. El llenguatge a utilitzar és *PL/SQL*.

1.2. Estructura d'un programa PL/SQL

Els programes PL/SQL s'estructuren en blocs lògics del següent tipus:

```
[declare
   <declaracions de constants i variables>;]
begin
   <sentències executables>;
[exception
   <declaració d'excepcions>;]
end;
```

Observem que es composa de tres zones clarament diferenciades, de les que una és obligatòria: la zona `begin ... end` (zona d'execució).

La zona `declare` conté, de forma similar a la majoria de llenguatges de programació, les constants i les variables a utilitzar en el programa.

La zona `exception` conté les instruccions per a tractar els errors d'accés a la base de dades que es puguin produir en el programa.

Cal saber que:

- Tota instrucció en PL/SQL finalitza en punt i coma

– Hi pot haver comentaris que, o han de començar amb dos guions (––) que indiquen comentari fins al final de la línia, o estar emmarcats entre `/*` i `*/`, que indiquen comentaris multiínia.

Cal distingir entre blocs anònims i blocs amb nom:

– Els blocs anònims són blocs que s'insereixen en altres eines, com *SQL*Plus* i *SQL Developer* i fins i tot, poden inserir-se en programes desenvolupats en llenguatges de tercera generació. S'executen quan l'eina i/o programa en qüestió executa el codi on és inserit el bloc.

– Els blocs amb nom són subprogrames (funcions i accions) que s'emmagatzemen a la base de dades i que poden ser invocats repetidament des de diversos indrets.

1.3. Elements bàsics del llenguatge PL/SQL

Entre els elements bàsics que cal conèixer en qualsevol llenguatge tenim els tipus de dades (simples i compostes) que facilita el llenguatge, la declaració de variables i constants, els operadors i les expressions que es poden construir, i les funcions que el llenguatge PL/SQL facilita i llur utilització en el desenvolupament de programes.

1.3.1. Tipus escalars (simples) de dades

La taula 2 ens mostra els tipus simples de dades que PL/SQL aporta.

Taula 2. Tipus de dades simples del llenguatge PL/SQL

	Tipus	Subtipus	Descripció
TIPUS NUMÈRICS	`BINARY_INTEGER`		Emmagatzema enters amb signe. El rang de valors permesos va de $-2^{31}-1$ a $2^{31}-1$, és a dir, de -2147483647 a 2147483647
		`NATURAL`	El rang de valors va de 0 a 2147483647
		`POSITIVE`	El rang de valors va de 1 a 2147483647
	`NUMBER [(precisió,escala)]`		Emmagatzema números en punt fix o flotant. La `precisió` indica el número total de dígits i l'`escala` determina el número de dígits a la dreta del punt decimal. El rang de valors permesos va de $1.0E^{-129}$ a $9.99E^{125}$ Per declarar números en punt flotant, no s'indica `precisió` ni `escala`. Per declarar valors enters, s'indica `escala` zero o, simplement, no s'indica `escala`. L'`escala` pot variar entre -84 i 127 i la precisió entre 1 i 38 tot i que en alguna plataforma pot ser inferior a 38.
	`DEC, DECIMAL, NUMERIC`		Per declarar números en punt fix amb una precisió màxima de 38 dígits decimals. Es mantenen per compatibilitat amb ANSI/ISO i altres sistemes.

	Tipus	Subtipus	Descripció
	DOUBLE PRECISION, FLOAT		Per declarar números en punt flotant amb una precisió màxima de 126 dígits binaris, que equival aproximadament a 38 dígits decimals. Es mantenen per compatibilitat amb ANSI/ISO i altres sistemes.
		REAL	Per declarar números en punt flotant amb una precisió de 63 dígits binaris, que equival aproximadament a 18 dígits decimals. Es manté per compatibilitat amb ANSI/ISO i altres sistemes.
	INTEGER, INT, SMALLINT		Per declarar enters amb una precisió màxima de 38 dígits decimals. Es mantenen per compatibilitat amb ANSI/ISO i altres sistemes.
TIPUS ALFANUMÈROIC	CHAR [(longitud màxima)]		Emmagatzema cadenes de caràcters de longitud fixa. La màxima longitud permesa és de 32767 octets. Si no s'especifica `longitud` es pren 1. Cal tenir en compte que les columnes `CHAR` de la base de dades tenen una longitud màxima de 255 caràcters.
		CHARACTER	És totalment equivalent a `CHAR`. Es manté per compatibilitat amb ANSI/ISO i altres sistemes.
		ROWID	Emmagatzema l'adreça física (columna `ROWID`) d'una fila dins la base de dades *Oracle*
		UROWID	Emmagatzema l'adreça (física, lògica o forana -no d'*Oracle*-) d'una fila dins la base de dades. Per les noves aplicacions és important utilitzar `UROWID` enlloc de `ROWID`, el qual es manté per compatibilitat amb versions anteriors.
	VARCHAR2 [(longitud_màxima)]		Emmagatzema cadenes de caràcters de longitud variable. La màxima `longitud` permesa és de 32767 octets. Cal anar en compte en intentar gravar valors `varchar2` en columnes `varchar2` de la base de dades, ja que les de la base de dades només admeten 4000 bytes.
		VARCHAR, STRING	Són totalment equivalents a `VARCHAR2`. Es mantenen per compatibilitat amb ANSI/ISO i altres sistemes.
	LONG		Emmagatzema cadenes de caràcters de longitud variable. La màxima longitud permesa és de 32760 bytes. Cal anar en compte en intentar assignar-hi valors `long` de columnes `long` de la base de dades, ja que les de la base de dades poden admetre fins a 2 GB.
	LONG RAW		Mateixes característiques que `LONG` però emmagatzema la informació en binari. Es pot utilitzar per emmagatzemar imatges, gràfics,...
	RAW [(longitud màxima)]		Mateixes característiques que `CHAR` però emmagatzema la informació en binari.
VARIS	BOOLEAN		Emmagatzema els valors `TRUE`, `FALSE` o `NULL`.
	DATE		Emmagatzema valors de dates

1.3.2. Declaració de variables i constants

Tenim dues formes de declarar variables i constants:

1) A partir de tipus de dades ja existents

La declaració de variables i constants ha d'estar en la secció `declare` del bloc PL/SQL, seguint la sintaxis:

```
<nom_variable> [constant] <tipus_dada> [not null] [{:=|default} <expressió_PL/SQL>];
```

És a dir:

- La declaració d'una constant és idèntica a la d'una variable però utilitzant la paraula clau `constant`.

- En la declaració d'una dada, si no s'indica `not null`, s'entén que pot contenir valors `NULL`.

- Es pot incloure una expressió vàlida en el llenguatge PL/SQL per tal de donar un valor inicial a la dada. Per defecte, la dada queda inicialitzada amb el valor `NULL` .

- És obligatori inicialitzar una dada si no pot contenir valors nuls o si es tracta d'una dada constant.

- La inicialització de variables es pot efectuar amb l'operador `:=` o la paraula `default`.

Exemples de declaració de variables i constants

```
declare
   taxa number;
   hi_ha_existencia boolean;
   pi constant number (5,4) := 3.1415;
   avui date;
   pi_quadrat number (6,4) := pi * pi;
   nom char(40) default 'Supercalifragilísticexpiralidos';
```

2) A partir de dades existents

PL/SQL facilita la possibilitat de declarar variables i constants basades en dades ja existents, de dues formes:

a) A partir d'altres dades (variables però no constants) o columnes de taules o vistes de la base de dades.

Aquesta possibilitat dona una gran potència a la programació en PL/SQL ja que, per les variables destinades a contenir valors de les taules o vistes de la base de dades, posteriors modificacions en la definició de les taules o vistes poden implicar, únicament, la nova compilació dels programes PL/SQL afectats, sense haver de canviar el seu codi font. Amb aquest tipus de declaració es redueix el manteniment dels programes.

La sintaxis d'aquesta declaració és:

```
<nom_variable> [constant] <nom_variable_PL/SQL>%type [not null] [:= <expressió>];
```

o

```
<nom_variable> [constant] <esquema.taula.columna>%type [not null] [:= <expressió>];
```

Exemples de declaració de variables i constants a partir de columnes de taules

```
declare
   v_dept_no constant dept.dept_no%type;
   aux_dept_no v_dept_no%type;
```

b) A partir d'una fila sencera d'una taula o vista. Això és possible per què PL/SQL facilita, també, la possibilitat de declarar variables tuple. En tal situació, els camps de la variable tenen els mateixos noms i tipus que les columnes de la taula o vista.

La sintaxis d'aquesta declaració és:

```
<nom_variable> <esquema.taula>%rowtype;
```

Exemples de declaració de tuples a partir de files de taules o vistes

```
declare
   r_dept dept%rowtype;
   r_emp emp%rowtype;
```

1.3.3. Àmbit d'existència de les dades

Les variables declarades en un bloc PL/SQL són considerades locals pel bloc i globals per tots els sub-blocs.

Exemple dels àmbits d'existència de variables segons la imbricació de blocs

```
declare
   va char;
   vb real;
begin
   -- variables disponibles en aquest punt: va(char) i vb (real)
   declare
      va integer;
      vc real;
   begin
      -- variables disponibles en aquest punt: va(integer), vb (real) i vc (real)
      ...
   end;
   -- variables disponibles en aquest punt: va(char) i vb (real)
   ...
   declare
      vd real;
   begin
      -- variables disponibles en aquest punt: va(char), vb (real) i vd (real)
      ...
   end;
   -- variables disponibles en aquest punt: va(char) i vb (real)
   ...
end;
```

1.3.4. Assignacions a variables

Es poden efectuar assignacions en qualsevol part del bloc PL/SQL (`declare`, `begin`, `exception`) i el valor assignat pot ser una constant o el resultat d'una expressió PL/SQL.

Hi ha dues formes d'emmagatzemar un valor en una variable:

– Amb la utilització de l'operador `:=`

– Amb la sentència `SELECT...INTO...` de SQL hostatjat en PL/SQL que permet emmagatzemar els valors de les columnes seleccionades per la clàusula `select` en les variables referenciades en la clàusula `into`.

Exemple de la sentència `SELECT...INTO...`

```
declare
   salari_total eo.emp.salari%type;
begin
   select sum(salari) into salari_total
   from eo.emp;
   ...
end;
```

En aquest tros de codi estem utilitzant una crida a una sentència `select` de SQL per a obtenir un resultat que emmagatzemem en la variable `salari_total`.

1.3.5. Expressions i operadors

Una expressió és, com en qualsevol llenguatge, una combinació de variables, constants, literals i operacions sobre els seus valors.

La taula 3 mostra els operadors que facilita PL/SQL i l'ordre de precedència per la seva avaluació.

Taula 3. Operadors facilitats pel llenguatge PL/SQL i llur ordre de precedència

Ordre	Operador	Descripció
1r.	`**, not`	potència, negació lògica
2n.	`+, -`	identitat, negació
3r.	`*, /`	multiplicació, divisió
4t.	`+, -, ||`	suma, resta, concatenació
5è.	`=, !=, <, >, <=, >=, is null, is not null, like, between, in`	comparació
6è.	`and`	conjunció
7è.	`or`	disjunció

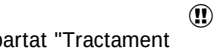
1.3.6. Funcions predefinides

Existeix un ampli conjunt de funcions predefinides en PL/SQL que pràcticament coincideixen amb les que *Oracle* incorpora en el seu llenguatge SQL. La taula 4 en presenta un conjunt important agrupades

per categories. No entrem en el seu estudi convidant-vos a cercar la corresponent informació en els manuals de referència de SQL i PL/SQL.

Totes aquestes funcions es poden utilitzar en sentències SQL excepte les de tractament d'error SQLCODE i SQLERRM. A més, totes les funcions són utilitzables en sentències del llenguatge PL/SQL excepte DECODE, DUMP i VSIZE.

Les funcions SQL de valors agregats (AVG, COUNT, MIN,...) no existeixen en PL/SQL. Per tant, es poden utilitzar en sentències SQL però no en sentències del llenguatge PL/SQL.



Al subapartat "Tractament d'errors" de l'apartat "Interacció amb el llenguatge SQL" d'aquest mateix nucli hi veurem les funcions de tractament d'error SQLCODE i SQLERRM.

Taula 4. Recull de funcions subministrades per *Oracle*

D'errors	Numèriques	De caràcter	De conversió	De dates	Altres
SQLCODE	ABS	ASCII	CHARTOROWID	ADD_MONTHS	DEREF
SQLERRM	ACOS	CHR	CONVERT	LAST_DAY	REF
	ASIN	CONCAT	HEXTORAW	MONTHS_BETWEEN	VALUE
	ATAN	INITCAP	RAWTOHEX	NEW_TIME	BFILENAME
	ATAN2	INSTR	ROWIDTOCHAR	NEXT_DAY	DECODE
	CEIL	INSTRB	TO_CHAR	ROUND	DUMP
	COS	LENGTH	TO_DATE	SYSDATE	EMPTY_BLOB
	COSH	LENGTHB	TO_MULTI_BYTE	TRUNC	EMPTY_CLOB
	EXP	LOWER	TO_NUMBER		GREATEST
	FLOOR	LPAD	TO_SINGLE_BYTE		LEAST
	LN	LTRIM			NLS_CHARSET_DECL_LEN
	LOG	NLS_INITCAP			NLS_CHARSET_ID
	MOD	NLS_LOWER			NLS_CHARSET_NAME
	POWER	NLSSORT			NVL
	ROUND	NLS_UPPER			SYS_CONTEXT
	SIGN	REPLACE			SYS_GUID
	SIN	RPAD			UID
	SINH	RTRIM			USER
	SQRT	SOUNDEX			USERENV
	TAN	SUBSTR			VSIZE
	TANH	SUBSTRB			
	TRUNC	TRANSLATE			
		TRIM			
		UPPER			

1.3.7. Col·leccions en PL/SQL

En el disseny de programes es fa necessari, en moltes ocasions, la utilització de tipus de dades que permetin la gestió de col·leccions de valors, com els vectors, els conjunts, les llistes, ... El llenguatge PL/SQL facilita els tipus de dades TABLE i VARRAY que ens permeten gestionar col·leccions de dades.

En concret, el llenguatge PL/SQL permet les col·leccions de dades:

- Vectors associatius (figura 1), també coneguts com a vectors indexats, que permeten emmagatzemar i cercar elements via índexs numèrics i alfanumèrics. Aquests vectors són similars a les taules *hash* facilitades en certs llenguatges de programació.

Figura 1. Exemples de vectors associatius en *Oracle*

Vector associatiu amb índex numèric

Índex	Valor
5	Maria Teresa
10	Guifré
17	Oriol
20	Regina
37	Olga

Vector associatiu amb índex alfanumèric

Índex	Valor
'A1'	Maria Teresa
'L2'	Guifré
'M3'	Oriol
'N1'	Regina
'P2'	Olga

- Taules imbricades (*nested tables* en terminologia *Oracle*), que permeten contenir un nombre arbitrari de valors, no obligatòriament en seqüència, i que es poden emmagatzemar en columnes de la base de dades *Oracle*.

- Vectors de longitud variable (*varray* en terminologia *Oracle*), pensats per a d'una determinada grandària la qual pot ser modificada en temps d'execució segons les necessitats del moment.

Totes aquestes col·leccions de dades permeten una única dimensió, però es permet la definició de col·leccions on els seus elements siguin, a la vegada, col·leccions, fet que possibilita la implementació de dades multidimensionals.

Per poder crear variables de qualsevol d'aquestes col·leccions, cal, en primer lloc, declarar el corresponent tipus de dada i, posteriorment, declarar-ne variables. Aprofundim una mica en els tres tipus de col·leccions.

a) Vectors associatius

Els vectors associatius són vectors d'una **única** columna, indexats com en tots els llenguatges, però a diferència del que és tradicional en els llenguatges informàtics on l'índex és un valor numèric correlatiu que comença per zero o 1, en els vectors associatius és un valor que pot ser de tipus numèric (`BINARY_INTEGER`) o alfanumèric (`VARCHAR2`), es gestiona per programa i no té per què emmagatzemar valors correlatius, tal i com es pot observar en els dos exemples de la figura 1.

- Per declarar un tipus de dada vector associatiu cal seguir la sintaxis:

```
type <nom_tipus> is table
     of {<nom_variable>%type|<tipus_existent>|<propietar.taula.columna>%type} [not null]
     index by {binary_integer | pls_integer | varchar2(grandària)}
```

Les opcions BINARY_INTEGER i PLS_INTEGER són equivalents a tots els efectes.

Exemples de declaració de tipus de dades corresponents a vectors associatius

```
declare
   type tl_noms is table of varchar2(40) index by binary_integer;
   type tl_salaris is table of emp.salari%type index by varchar2(5);
```

En la primera declaració, tl_noms permetrà definir vectors associatius de cadenes de fins a 40 caràcters indexades per valors numèrics.

En la segona declaració, tl_salaris permetrà definir vectors associatius de valors de mateix tipus que la columna emp.salari, indexats per valors alfanumèrics de fins a 5 caràcters.

Observem que en cap cas s'indica la grandària de la taula.

- Per declarar variables de tipus vector associatiu una vegada el corresponent tipus ha estat declarat, cal utilitzar la sintaxis normal de declaració de variables, amb l'excepció que no es poden inicialitzar.

Exemples de declaració de variables de tipus vector associatiu

```
declare
   type tl_noms is table of varchar2(40) index by binary_integer;
   v_noms tl_noms;
   type tl_salaris is table of emp.salari%type index by varchar2(5);
   v_salaris tl_salaris;
```

- Per accedir a les files d'una variable de tipus vector associatiu cal especificar el corresponent valor de l'índex.

Exemples d'accés a variables de tipus vector associatiu

```
v_noms(3):='Josep';
v_noms(17):='Maria';
v_salaris('Josep'):=100000;
v_salaris('Maria'):=100000;
```

- Les diferents posicions d'una variable de tipus vector associatiu es creen en el moment en que se li assigna valor. Fins aquest moment, no existeixen.

  Després de l'execució del darrer exemple, tenim les dues variables v_noms i v_salaris amb dues posicions cadascuna. Podem observar que no existeix el concepte de posicions correlatives segons l'índex del vector.

- L'accés a les posicions de les variables de tipus vectors associatius s'efectua en base als valors de l'índex. Tot intent d'accedir a una posició inexistent és causa d'error en temps d'execució.

  Per poder controlar l'accés a variables de tipus vector associatius, PL/SQL facilita un conjunt d'operadors, recollits a la taula 5.

Taula 5. Operadors facilitats per *Oracle* per controlar l'accés als vectors associatius

Operador	Descripció
`<nom_variable>.exists(valor)`	Retorna `TRUE` si existeix la cel·la amb valor d'índex `valor` i `FALSE` si no existeix
`<nom_variable>.first`	Retorna el valor de l'índex de la primera cel·la existent en el vector associatiu. Retorna `NULL` si el vector associatiu és buit.
`<nom_variable>.next(valor)`	Retorna el valor d'índex de la cel·la següent a la cel·la que té el valor d'índex `valor`. Retorna `NULL` si no hi ha cel·la següent.
`<nom_variable>.prior(valor)`	Retorna el valor d'índex de la cel·la anterior a la cel·la que té el valor d'índex `valor`. Retorna `NULL` si no hi ha cel·la anterior.
`<nom_variable>.last`	Retorna el valor de l'índex de la darrera cel·la existent en el vector associatiu Retorna `NULL` si el vector associatiu és buit.
`<nom_variable>.count`	Retorna el nombre d'elements del vector associatiu
`<nom_variable>.delete(m,n)`	Elimina les cel·les del vector associatiu existents des del valor `m` fins el valor `n` Si només s'indica un paràmetre, elimina la cel·la indicada del paràmetre Si no s'indica cap paràmetre, elimina totes les cel·les del vector associatiu

- Si en la definició del tipus no s'ha explicitat la clàusula `not null`, es permet l'assignació de valors `NULL` a les cel.les.

- Les cel·les s'ordenen pel valor de l'índex i no pas per l'ordre en que han estat inserides.

Exemple d'ubicació de les cel·les en vectors associatius

```
declare
   type tCadena is table of varchar2(40) index by varchar2(30);
   vNoms tCadena;
begin
   vNoms('N2'):=null;
   vNoms('A1'):='ISIDRE';
   vNoms('A0'):='TERESA';
   vNoms('P3'):='GUIFRÉ';
```

En aquestes assignacions, la taula `vNoms` ha quedat emplenada amb 4 valors. Vegem que malgrat l'ordre en que s'han efectuat les insercions, les cel·les resten ordenades segons els valors dels índexs:

- `vNoms.first` retorna 'A0'
- `vNoms.next('A0')` retorna 'A1'
- `vNoms.next('A1')` retorna 'N2'
- `vNoms.next('N2')` retorna 'P3'

- És permès efectuar assignacions entre variables de tipus vector associatiu.

Exemples d'assignació entre variables de tipus vectors associatius

```
declare
   type tl_noms is table of varchar2(40) index by binary_integer;
   v_noms1 tl_noms;
   v_noms2 tl_noms;
begin
   ...
   v_noms1 := v_noms2;
   ...
```

- Es pot declarar variables de tipus vectors associatius com a paràmetres de subprogrames

- No es pot fer referència a variables de tipus vectors associatius en la clàusula `into` de la sentència `SELECT`.

- El traspàs de dades entre variables de tipus vectors associatius i taules de la base de dades s'ha de fer posició a posició.

La declaració de variables de tipus vectors associatius com a paràmetres de subprogrames es veurà en la presentació de subprogrames en PL/SQL.

b) Taules imbricades

Les taules imbricades (*nested tables* en terminologia *Oracle*) són molt similars als vectors associatius però afegeixen algunes funcionalitats importants:

- Faciliten mètodes de gestió addicionals
- Una taula imbricada es pot emmagatzemar en un camp de taula a la base de dades.
- Es poden gestionar directament amb instruccions SQL d'*Oracle*.

Vegem les principals característiques de gestió de les taules imbricades:

- La declaració d'un tipus de dada taula imbricada és idèntica a la declaració d'un tipus de dades vector associatiu però sense indicar la clàusula `INDEX BY`. En no indicar el tipus d'índex, es considera com una declaració de tipus de dada taula imbricada. Així doncs, la sintaxis seria:

```
type <nom_tipus> is table
     of {<nom_variable>%type|<tipus_existent>|taula.columna%type} [not null]
```

- Per declarar variables de tipus taula imbricada una vegada el corresponent tipus ha estat declarat, cal utilitzar la sintaxis normal de declaració de variables.

  En el moment d'efectuar la declaració, es pot procedir a la inicialització utilitzant una funció (constructor) que coincideix, en el nom, amb el del corresponent tipus taula imbricada. Aquesta funció té una sintaxi similar a:

```
<nom_variable>:=<nom_constructor>(<valor1>,<valor2>...)
```

És imprescindible la utilització del constructor per inicialitzar una variable de tipus taula imbricada. Si no es fa en la declaració, s'haurà d'efectuar posteriorment.

Programació orientada a objectes. Constructors.

El concepte constructor s'utilitza en la programació orientada a objectes per a indicar una funció que permet construir objectes (variables) de una classe (tipus de dada).

Oracle implementa el tipus de dada de taula imbricada com a una classe d'objectes i, per tant, per a poder-ne declarar variables cal utilitzar un constructor. No us preocupeu si no sou coneixedors de les tècniques de la programació orientada a objectes, doncs no ens és necessari per a poder treballar amb tipus de dades de taula imbricada.

En aquests moments només ens cal saber que *Oracle* facilita de forma automàtica, per a cada tipus de dada taula imbricada definit, un constructor que coincideix, en nom, amb el del tipus de taula imbricada corresponent.

En inicialitzar una variable de tipus taula imbricada utilitzant el constructor, els elements de la taula es numeren correlativament a partir d'1 i fins el nombre d'elements especificat en la crida al constructor.

Exemples de declaració de variables de tipus de dades taula imbricada

```
declare
   type tCadena is table of varchar2(40);
   vNoms1 tCadena := tCadena('AAA');
   vNoms2 tCadena := tCadena('X', 'YY', 'ZZZ');
   vNoms3 tCadena := tCadena();
   vNoms4 tCadena;
```

En aquestes declaracions:

- `vNoms1` és una taula imbricada inicialitzada amb la única cadena 'AAA'. Ocupa la posició 1 de la taula.
- `vNoms2` és una taula imbricada inicialitzada amb tres cadenes: 'X', 'YY' i 'ZZZ'. Ocupen les posicions 1, 2 i 3.
- `vNoms3` és una taula imbricada buida (sense elements)
- `vNoms4` és una variable destinada a gestionar una taula imbricada que encara no existeix. Es diu que el valor de `vNoms4` és `NULL`.

- Si en la definició del tipus no s'ha explicitat la clàusula `not null`, es permet l'assignació de valors nulls a les cel.les.

- Els operadors `exists`, `first`, `last`, `prior`, `next`, `count` i `delete` aplicables en les variables de tipus vector associatiu, també són aplicables en les variables de tipus taula imbricada.

- En el cas d'eliminació de cel·les en una taula imbricada, es generen forats i, per tant, serà necessari utilitzar els operadors `last` i `prior` per a poder saltar els forats.

- Les variables de tipus taula imbricada no tenen grandària fixada però no es pot assignar un valor a una cel·la inexistent. Per a poder afegir cel·les cal emprar l'operador `extend`, que admet tres formes:

  `extend`, que afegeix una cel·la buida al final de la variable
  `extend(n)`, que afegeix `n` cel·les buides al final de la variable
  `extend(n,i)` que afegeix `n` cel·les, còpies de la cel·la `i`, al final de la variable

- És permès efectuar assignacions entre variables de tipus taula imbricada.

Exemples d'assignació entre variables de tipus taula imbricada

```
declare
   type tl_noms is table of varchar2(40);
   v_noms1 tl_noms;
   v_noms2 tl_noms;
begin
   ...
   v_noms1 := v_noms2;
   ...
```

- Es pot declarar variables de tipus taula imbricada com a paràmetres de subprogrames

La declaració de variables de tipus taula imbricada com a paràmetres de subprogrames es veurà en la presentació de subprogrames en PL/SQL.

- No es pot fer referència a variables de tipus taula imbricada en la clàusula `into` de la sentència `SELECT`.

- És possible tenir taules a la base de dades amb columnes preparades per emmagatzemar variables de taules imbricades, però això sobrepassa l'abast d'aquest crèdit. Podeu trobar informació al respecte en la documentació subministrada per *Oracle*.

c) Vectors variables

Els vectors variables (*varray* en terminologia *Oracle*) es manipulen de forma molt similar als vectors associatius i a les taules imbricades, però s'implementen de forma diferent.

Els elements, en els vectors variables, s'emmagatzemen començant per la posició 1 i fins la longitud màxima declarada.

- La sintaxis per declarar un tipus de dada vector variable és:

```
type <nom_tipus> is varray (<grandària_màxima>)
     of {<nom_variable>%type|<tipus_existent>|taula.columna%type} [not null]
```

- Per declarar variables de tipus vector variable una vegada el corresponent tipus ha estat declarat, cal utilitzar la sintaxis normal de declaració de variables.

  En el moment d'efectuar la declaració, es pot procedir a la inicialització utilitzant una funció (constructor) que coincideix, en el nom, amb el del corresponent tipus vector variable.

  És imprescincible la utilització del constructor per inicialitzar una variable de tipus vector variable. Si no es fa en la declaració, s'haurà d'efectuar posteriorment.
  En inicialitzar una variable de tipus vector variable utilitzant el constructor, els elements del vector es situen correlativament a partir de la posició 1 i fins el nombre d'elements especificat en la crida al constructor.

  Exemples de declaració de variables de tipus de dades vector variable

```
declare
   type tCadena is varray(5) of varchar2(40);
   vNoms1 tCadena := tCadena('AAA');
   vNoms2 tCadena := tCadena('X', 'YY', 'ZZZ');
   vNoms3 tCadena := tCadena();
   vNoms4 tCadena;
```

  En aquestes declaracions:

  - `vNoms1` és un vector variable d'1 posició (pot extendre's fins a 5 posicions) inicialitzat amb la única cadena 'AAA'. Ocupa la posició 1 del vector.
  - `vNoms2` és un vector variable de 3 posicions (pot extendre's fins a 5 posicions) inicialitzat amb tres cadenes: 'X', 'YY' i 'ZZZ'. Ocupen les posicions 1, 2 i 3.
  - `vNoms3` és un vector variable de zero posicions (pot extendre's fins a 5 posicions)
  - `vNoms4` és una variable destinada a gestionar un vector variable de fins a 5 posicions que encara no existeix. Es diu que el valor de `vNoms4` és `NULL`.

- Si en la definició del tipus no s'ha explicitat la clàusula `not null`, es permet l'assignació de valors nulls a les cel.les.

- Els operadors `first`, `last`, `prior`, `next` i `count` aplicables en les variables de tipus vector associatiu i taula imbricada, també són aplicables en les variables de tipus vector variable, però no ho són els operadors `exists` i `delete`. En els vectors variables també existeix l'operador `limit` que retorna el màxim nombre de posicions vàlides.

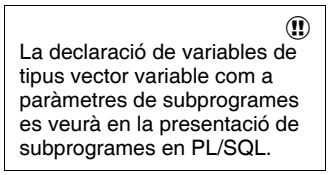
  Les variables de tipus vector variable tenen la grandària definida en la seva creació (constructor) que mai podrà ser major que la grandària màxima fixada en el tipus. Per a poder afegir cel·les fins a la grandària màxima, cal emprar l'operador `extend.`

- És permès efectuar assignacions entre variables de tipus vector variable.

Exemples d'assignació entre variables de tipus vector variable

```
declare
   type tl_noms is varray(5) of varchar2(40);
   v_noms1 tl_noms;
   v_noms2 tl_noms;
begin
   ...
   v_noms1 := v_noms2;
   ...
```

- Es pot declarar variables de tipus vector variable com a paràmetres de subprogrames


La declaració de variables de tipus vector variable com a paràmetres de subprogrames es veurà en la presentació de subprogrames en PL/SQL.

- No es pot fer referència a variables de tipus vector variable en la clàusula `into` de la sentència `SELECT`.
- És possible tenir taules a la base de dades amb columnes preparades per emmagatzemar variables de tipus vector variable, però això sobrepassa l'abast d'aquest crèdit. Podeu trobar informació al respecte en la documentació subministrada per *Oracle*.

1.3.8. Registres definits per l'usuari

El llenguatge PL/SQL aporta la possibilitat de definir el tipus de dada registre *(record)* de manera semblant a la majoria de llenguatges.
Per poder crear variables de tipus registre cal, en primer lloc, declarar el corresponent tipus de dada i, posteriorment, declarar-ne variables.

- Per declarar un tipus de dada registre cal seguir la sintaxis:

```
type <nom_tipus> is record
(  {<camp1> {<variable>%type|<tipus>|taula.col%type|taula%rowtype} [not null][:=<expr>],
   {<camp2> {<variable>%type|<tipus>|taula.col%type|taula%rowtype} [not null][:=<expr>],
   ...
);
```

Exemple de declaració de tipus de dades registres

```
declare
   type rg_persona is record
   (  dni char(9) not null:=' ',
      nom varchar2(40) not null:=' ',
      sexe char,
      edat int
   );
```

En aquest exemple s'ha declarat el tipus registre `rg_persona` format per quatre camps:

- `dni` com a una cadena de fins a 9 caràcters que no pot tenir valor `NULL` i inicialitzada amb un espai en blanc
- `nom` com a una cadena de fins a 40 caràcters que no pot tenir valor `NULL` i inicialitzada amb un espai en blanc
- `sexe` com a un caràcter
- `edat` com a un enter.

- Per declarar variables registre una vegada el corresponent tipus ha estat declarat, cal utilitzar la sintaxis normal de declaració de

variables, amb l'excepció que no es poden inicialitzar, ja que la inicialització cal fer-la en la declaració del tipus.

Exemple de declaració de variables de tipus de dades registre

```
declare
   r_pers1 rg_persona;
   r_pers2 rg_persona;
```

- Per fer referència els camps d'un registre cap utilitzar la notació de punts com en la majoria de llenguatges.

Exemples de referència als camps d'un registre

```
r_pers1.dni:='999999999';
r_pers1.nom:='Josep';
r_pers1.sexe:='H';
r_pers1.edat:=25;
```

- No es pot omplir els camps d'un registre amb una llista de valors, però sí pot ser utilitzat en la clàusula `into` de la sentència `SELECT`.
- És permès efectuar assignacions de registre a registre
- Es poden declarar variables de tipus registres com a paràmetres de subprogrames.

La declaració de variables de tipus registre com a paràmetres de subprogrames es veurà en la presentació de subprogrames en PL/SQL.

1.4. Estructures de control del llenguatge PL/SQL

Els programes PL/SQL són, com qualsevol programa desenvolupat en un llenguatge estructurat i modular, una seqüència d'instruccions en la que apareixen estructures iteratives i estructures condicionals.

1) Estructures condicionals

PL/SQL facilita la sentència `IF` en tres modalitats:

```
if <condició> then
   <conjunt_de_sentències_si_la_condició_s'avalua_com_a_cert>;
end if;

if <condició> then
   <conjunt_de_sentències_si_la_condició_s'avalua_com_a_cert>;
else
   <conjunt_de_sentències_si_la_condició_s'avalua_com_a_fals>;
end if;

if <condició1> then
   <conjunt_de_sentències_si_condició1_s'avalua_com_a_cert>;
elsif <condició2> then
   <conjunt_de_sentències_si_condició2_s'avalua_com_a_cert>;
elsif
```

```
   ...
else
   <conjunt_de_sentències_si_última_cond_s'avalua_com_a_fals>;
end if;
```

2) Estructures iteratives

PL/SQL facilita tres modalitats de bucles:

a) L'estructura `LOOP`, amb la sintàxis:

```
loop
   <conjunt_de_sentències>
exit when <condició_sortida>
   <conjunt_de_sentències>
exit when <condició_sortida>
   <conjunt_de_sentències>
...
end loop;
```

L'estructura `LOOP` permet diverses condicions de sortida (`exit when`). En arribar a la instrucció `END LOOP`, el programa repeteix el bucle.

b) L'estructura `WHILE ... LOOP`, amb la sintaxis:

```
while <condició> loop
   <conjunt_de_sentències>
end loop;
```

L'estructura `WHILE ... LOOP` és idèntica a la de la majoria de llenguatges.

c) La sentència `FOR`, amb la sintàxis:

```
for <variable> in [reverse] <rang_mínim>..<rang_màxim> loop
   <conjunt_de_sentències>
end loop;
```

L'estructura `FOR` no exigeix que `variable` estigui definida. Si no ho està, es defineix pel bucle i desapareix en finalitzar l'execució del bucle.

Els valors `rang_mínim` i `rang_màxim` han de ser expressions de resultat enter.

El conjunt d'instruccions d'un bucle `FOR` no poden modificar el contingut de `variable`.

El bucle `FOR ... IN` inicialitza el valor de `variable` amb `rang_mínim` i a cada repetició del bucle, incrementa el valor una unitat fins sobrepassar el `rang_màxim`.

El bucle `FOR ... IN REVERSE` inicialitza el valor de `variable` amb `rang_màxim` i a cada repetició del bucle, decrementa el valor una unitat fins ultrapassar el `rang_mínim`.

3) Control seqüencial

- PL/SQL incorpora la sentència `NULL` que indica no efectuar cap acció. Pot ser útil en certes ocasions.

  Exemple d'utilització de la sentència `NULL`

```
if i>0 then null;
else x:= y*2;
end if;
```

  Aquest exemple no té gaire sentit, per no dir que no en té gens, ja que la condició es podria escriure d'una altra forma de manera que no es necessités utilitzar la clàusula `else`. Però l'exemple serveix per a veure una possible utilització de la sentència `NULL`.

- PL/SQL incorpora, com la majoria de llenguatges estructurats i modulars, la sentència `GOTO`, la qual no s'hauria d'utilitzar mai en el flux normal d'un programa i s'hauria de deixar la seva utilització, únicament per a gestió de situacions d'error.

  La seva sintaxis és:

```
goto NOM_ETIQUETA;
```

  Per a poder-la utilitzar és necessari que en algun lloc del bloc PL/SQL hi hagi una línia iniciada amb una etiqueta amb el nom que acompanya la sentència `GOTO`. Les etiquetes s'emmarquen entre els símbols `<< ... >>`, és a dir, en algun lloc del programa hi ha d'haver:

```
<<NOM_ETIQUETA>>
```

1.5. Interacció de programes PL/SQL amb l'exterior

El llenguatge PL/SQL no aporta instruccions específiques d'entrada/sortida per permetre la interacció amb un usuari final, doncs cal tenir present que el llenguatge PL/SQL està especialment pensat per:

– Ser utilitzat per desenvolupadors d'*Oracle Developer* per programar els esdeveniments. *Oracle Developer* ja incorpora la interfície amb l'usuari.

– Ser utilitzat per administradors d'*Oracle* per desenvolupar programes que ajudin a l'automatització de les tasques administratives.

– Ser utilitzat per desenvolupadors i administradors per crear subprogrames, paquets i disparadors que queden emmagatzemats dins la pròpia base de dades.

Per tant, un usuari final no intereactua directament amb el codi PL/SQL sinó via interfícies especialment dissenyades. Però a banda d'un usauri final ens trobem en diferents situacions en les que necessitem que el codi PL/SQL es comuniqui amb l'exterior:

– Visualització del contingut de variables en els processos de depuració de codi.

– Visualització de missatges en pàgines web.

– Intercanvi d'informació entre el codi PL/SQL i el sistema operatiu (comandaments i fitxers).

– Comunicació amb servidors web i servidors de correu.

Oracle proporciona, per a donar resposta a les diverses necessitats de comunicació, un conjunt de paquets:

– `DBMS_OUTPUT`, per a mostrar missatges en l'execució de codi PL/SQL des d'eines com *SQL*Plus* i *SQL Developer*.

– `HTF` i `HTP` per a mostrar informació en una pàgina web.

– `DBMS_PIPE` per a intercanviar informació entre PL/SQL i comandaments del sistema operatiu.

– `UTL_FILE` per a llegir i escriure en el sistema de fitxers del sistema operatiu.

– `UTL_HTTP` per comunicar-se amb servidors web.

– `UTL_SMTP` per comunicar-se amb servidors de correu.

Paquets en *Oracle*

Un paquet d'*Oracle* és un objecte de la base de dades que permet l'agrupament de definicions de tipus de dades, variables, procediments, funcions i altres, comuns normalment sobre la base de la seva funcionalitat.

Oracle proporciona molts paquets per donar resposta a diferents necessitats i els desenvolupadors de codi PL/SQL poden crear-ne de nous.

A banda dels paquets esmentats, *Oracle* també facilita la tècnica de les variables d'enllaç (*bind variables*) que possibilita utilitzar variables declarades en *SQL*Plus* dins de codi PL/SQL.

1.5.1. Paquet DBMS_OUTPUT

En l'aprenentatge del llenguatge PL/SQL interessa emprar el paquet DBMS_OUTPUT per a tenir la possibilitat de visualitzar el contingut de les variables en temps d'execució així com poder mostrar missatges segons el flux d'execució dels programes.

El paquet DBMS_OUTPUT és molt potent doncs permet que els programes PL/SQL enviïn missatges a un *buffer* d'on poden ser llegits per altres programes PL/SQL. En aquest punt únicament ens interessar la funcionalitat que ens proporciona aquest paquet per a poder enviar missatges que puguin ser visualitzats per *SQL*Plus* o *SQL Developer* i així efectuar el seguiment de l'execució. La taula 6 ens mostra els procediments del paquet DBMS_OUTPUT del nostre interès ara mateix.

Taula 6. Procediments del paquet DBMS_OUTPUT adequats per a la depuració de codi PL/SQL des de *SQL*Plus* o *SQL Developer*

Procediment	Descripció	Arguments
put	Afegeix text a la línia actual.	<valor> {NUMBER | VARCHAR2 | DATE}
new_line	Marca un final de línia	---
put_line	Combinació de put i new_line	<valor> {NUMBER | VARCHAR2 | DATE}

Per a poder visualitzar, des de *SQL*Plus* o *SQL Developer* els missatges enviats amb els procediments put o put_line, cal activar el paràmetre SERVEROUTPUT, de la forma:

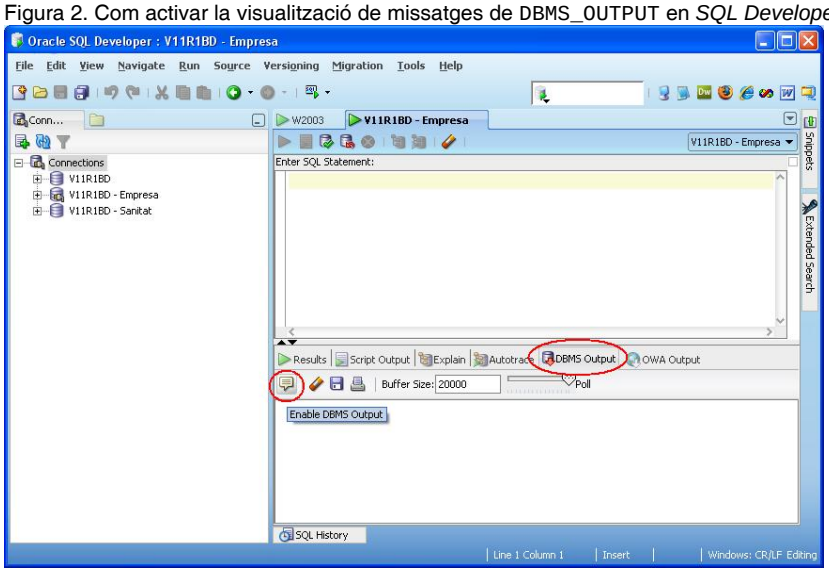
– Des de SQL*Plus:

```
set serveroutput on
```

– Des de SQL Developer, com es veu a la figura 2, activant el botó *Enable DBMS Output* de la pestanya *DBMS Output*. Tots els missatges enviats amb els procediments put o put_line es visualitzaran en aquesta pestanya.

És important saber que els missatges enviats pel paquet DBMS_OUTPUT al buffer no es visualtzen per *SQL*Plus* o *SQL Developer* fins que el codi PL/SQL ha finalitzat. No hi ha cap mecanisme per permetre que els missatges generats en el *buffer* passin immediatament (*flush*) cap a *SQL*Plus* o *SQL Developer*.

Figura 2. Com activar la visualització de missatges de DBMS_OUTPUT en *SQL Developer*



Trobareu el fitxer `u5n1p01.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Exemples d'utilització del paquet `DBMS_OUTPUT`

```
/* Programa: u5n1p01.sql (bloc anònim)
   Descripció: Utilització del paquet dbms_output per visualitzar el contingut
               de vectors associatius, taules imbricades i vectors variables
   Autor: Isidre Guixà
   Instruccions d'execució:
   - Des d'SQL Developer, activant el botó "Enable DBMS Output" de la pestanya "DBMS Output"
   - Des d'SQL*Plus, fent:
      set serveroutput on
      @u5n1p01.sql
      /
*/
declare
  type tva_noms is table of varchar2(40) index by binary_integer; -- vector associatiu
  type tti_noms is table of varchar2(40); -- taula imbricada
  type tvv_noms is varray(10) of varchar2(40); -- vector variable
  va_noms tva_noms;
  ti_noms tti_noms;
  vv_noms tvv_noms;
  i integer;
  n integer;
  clau integer;
begin
  dbms_output.put_line('Emplenem 2 posicions del vector associatiu va_noms');
  va_noms(4):='Guifré';
  va_noms(2):='Oriol';
  dbms_output.put_line('Creem (via constructor), la taula imbricada ti_noms amb 3 dades');
  ti_noms:=tti_noms('Miquel','Olga','Rosa','Regina');
  dbms_output.put_line('Ampliem amb 4 cel·les buides la taula imbricada ti_noms');
  ti_noms.extend(4);
  dbms_output.put_line('Omplim 2 de les 4 cel·les buides de la taula imbricada ti_noms');
  ti_noms(6):='Elvira';
  ti_noms(8):='Joan';
  dbms_output.put_line('Eliminem 1 de les cel·les de la taula imbricada ti_noms');
  ti_noms.delete(3);
  dbms_output.put_line('Creem (via constructor), el vector variable vv_noms amb 2 dades');
  vv_noms:=tvv_noms('Roger','Marc');
  dbms_output.put_line('Ampliem amb 4 cel·les buides el vector variable vv_noms ti_noms');
  vv_noms.extend(4);
  dbms_output.put_line('Omplim 2 de les 4 cel·les buides del vector associatiu vv_noms');
  vv_noms(4):='Marta';
  vv_noms(6):='Joan Esteve';

  -- Recorregut pel vector associatiu
  dbms_output.new_line();
  n:=va_noms.count;
  if n=0 then
    dbms_output.put_line('El vector associatiu va_noms és buit');
  else
    dbms_output.put_line('El vector associatiu va_noms té '||n||' dades, que són:');
    clau:=va_noms.first; i:=1;
    dbms_output.put_line(clau||': '||va_noms(clau));
    while i<n loop
```

```
      clau:=va_noms.next(clau); i:=i+1;
      dbms_output.put_line(clau||': '||va_noms(clau));
    end loop;
  end if;

  -- Recorregut per la taula imbricada;
  dbms_output.new_line();
  n:=ti_noms.count;
  if n=0 then
    dbms_output.put_line('La taula imbricada ti_noms és buida');
  else
    dbms_output.put_line('La taula imbricada ti_noms té '||n||' dades, que són:');
    clau:=ti_noms.first; i:=1;
    dbms_output.put_line(clau||': '||ti_noms(clau));
    while i<n loop
      clau:=ti_noms.next(clau); i:=i+1;
      dbms_output.put_line(clau||': '||ti_noms(clau));
    end loop;
  end if;
  /* Cal tenir molta precaució amb el recorregut per una taula imbricada, ja que pot tenir
     forats provocats per la utilització de l'operador delete. Per tant, caldrà utilitzar els
     operadors first i next quan no sapiguem si hi ha o no forats, en lloc d'utilitzar el típic
     índex numèric per accedir a totes les posicions de forma correlativa. */

  -- Recorregut pel vector variable;
  dbms_output.new_line();
  n:=vv_noms.count;
  if n=0 then
    dbms_output.put_line('El vector variable vv_noms és buit');
  else
    dbms_output.put_line('El vector variable vv_noms té '||n||' dades, que són:');
    for clau in 1..n loop
      dbms_output.put_line(clau||': '||vv_noms(clau));
    end loop;
  end if;
end;
```

S'obté la sortida:

```
Emplenem 2 posicions del vector associatiu va_noms
Creem (via constructor), la taula imbricada ti_noms amb 3 dades
Ampliem amb 4 cel·les buides la taula imbricada ti_noms
Omplim 2 de les 4 cel·les buides de la taula imbricada ti_noms
Eliminem 1 de les cel·les de la taula imbricada ti_noms
Creem (via constructor), el vector variable vv_noms amb 2 dades
Ampliem amb 4 cel·les buides el vector variable vv_noms ti_noms
Omplim 2 de les 4 cel·les buides del vector associatiu vv_noms

El vector associatiu va_noms té 2 dades, que són:
2: Oriol
4: Guifré

La taula imbricada ti_noms té 7 dades, que són:
1: Miquel
2: Olga
4: Regina
5:
6: Elvira
7:
8: Joan

El vector variable vv_noms té 6 dades, que són:
1: Roger
2: Marc
3:
4: Marta
5:
6: Joan Esteve
```

1.5.2. Variables d'enllaç

Les variables d'enllaç (*bind variables*) són variables creades en *SQL*Plus* que poden ser referenciades en blocs PL/SQL i sentències SQL. Una variable d'enllaç creada en *SQL*Plus*, pot ser utilitzada dins codi PL/SQL

com si hagués estad declarada dins el bloc PL/SQL. Les variables d'enllaç poden ser utilitzades per a emmagatzemar resultats i/o depurar codi PL/SQL.

- Per crear una variable d'enllaç en *SQL*Plus* cal utilitzar el comandament:

```
SQL> var[iable] <nom_variable> {number|char|char(n)|varchar2(n)|...}
```

- Per llistar totes les variables d'enllaç creades en una sessió, es pot executar el comandament `VARIABLE` sense cap argument.

- Per referenciar variables d'enllaç en el codi PL/SQL cal escriure la variable precedida del símbol ':'.

- Per visualitzar el valor d'una variable d'enllaç en *SQL*Plus*, es pot utilitzar el comandament:

```
SQL> pri[nt] <variable1> <variable2> ...
```

  Si no s'indica cap variable, el comandament `PRINT` mostra totes les variables existents a la sessió. En cas d'indicar varies variables, el comandament `PRINT` mostra el seu contingut una sota l'altra i això pot no agradar-nos; en tal situació, sempre tenim la possibilitat d'aconseguir la visualització en una mateixa línia utilitzant una sentència `SELECT` sobre les variables a la taula `DUAL`:

```
SQL> select :variable1, :variable2,... from dual;
```

  El format de visualització de les variables d'enllaç es modificable amb el comandament `COLUMN` de *SQL*Plus*.

- No hi ha manera d'eliminar les variables d'enllaç creades en una sessió SQL*Plus; només desapareixen en tancar la sessió de SQL*Plus.

- La declaració d'una variable d'enllaç prococa la desaparició de qualsevol variable d'enllaç amb mateix nom existent a la sessió de SQL*Plus.

Exemple d'utilització de variables d'enllaç en codi PL/SQL

```
/* Programa: u5n1p02.sql (guió)
   Descripció: Utilització, en codi PL/SQL, de variables d'enllaç
               definides en SQL*Plus
   Autor: Isidre Guixà
*/
set serveroutput on
var x number
var y number
begin
```

Trobareu el fitxer `u5n1p02.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
  :x :=10;
  :y :=20;
  dbms_output.put_line('Des de dins el 1r. bloc PL/SQL:');
  dbms_output.put_line('x = '||:x);
  dbms_output.put_line('y = '||:y);
end;
/
print x y
begin
  :x := :x*3;
  :y := :y*3;
  dbms_output.put_line('Des de dins el 2n. bloc PL/SQL:');
  dbms_output.put_line('x = '||:x);
  dbms_output.put_line('y = '||:y);
end;
/
print x y
```

Observem que aquest guió de SQL*Plus defineix les variables d'enllaç `x` i `y` que després són referenciades dins els dos blocs PL/SQL anònims inclosos dins el guió. El valor de les variables es manté al llarg de l'execució del guió, de manera que els valors calculats en el primer bloc PL/SQL es transmeten al segon bloc PL/SQL. Des de SQL*Plus, s'obté la sortida:

```
Des de dins el 1r. bloc PL/SQL:
x = 10
y = 20

El procediment PL/SQL ha finalitzat correctament.

         X
----------
        10

         Y
----------
        20

Des de dins el 2n. bloc PL/SQL:
x = 30
y = 60

El procediment PL/SQL ha finalitzat correctament.

         X
----------
        30

         Y
----------
        60
```

Observem que el comandament `PRINT` visualitza les dues variables `X` i `Y` una sota l'altra. Utilitzant "`select :x, :y from dual`" s'aconseguiria la visaultització a la mateixa línia.

1.6. Interacció de programes PL/SQL amb el llenguatge SQL

Arriba, per fi, el moment culminant que estàvem esperant: la utilització hostatjada de SQL en el llenguatge procedural PL/SQL, és a dir, la interacció del llenguatge PL/SQL amb les instruccions del llenguatge SQL.

En primer lloc cal saber que en totes les instruccions SQL hostatjades dins codi PL/SQL podem utilitzar les variables PL/SQL i les variables d'enllaç definides en *SQL*Plus*. En segon lloc ens cal conèixer el funcionament de les sentències `SELECT`, `INSERT`, `UPDATE` i `DELETE` en PL/SQL així com el control de transaccions i la gestió de cursors i errors.

1.6.1. Sentència SELECT

Dins el codi PL/SQL (i en general en qualsevol 3GL que hostatgi SQL) es pot utilitzar la sentència `SELECT...INTO...` sempre i quan hi hagi la seguretat que retorna **una i només una** fila. Del contrari es produeix un error que caldrà interceptar.

Els errors causats per no trobar cap fila o per a trobar-ne més d'una depenen de la situació de la base de dades en el moment d'executar la sentència. Per aquest motiu, els programadors han de tenir present si sempre el resultat serà una i només una fila o si hi pot haver situacions en les que això no sigui així i caldrà dissenyar la sentència de manera que sigui correcta en qualsevol circumstància. (!!)
Per altra banda, la clàusula `into` ha de tenir tantes variables com columnes o expressions formin part de la clàusula `select`. (!!)

(!!) A l'apartat "Tractament d'errors" es detallen les possibilitats que PL/SQL facilita per al tractament d'errors

Exemple de sentència `SELECT...INTO...` correcta dins codi PL/SQL

Considerem, a l'esquema *empresa* el guió següent:

```
/* Programa: u5n1p03.sql (guió)
   Descripció: Exemple de sentència SELECT dins codi PL/SQL
   Autor: Isidre Guixà
*/
variable nbDept number;
begin
  select count(*) into :nbDept
  from dept;
end;
/
print nbDept
```

(!!) Trobareu el fitxer `u5n1p03.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

L'execució de la sentència `SELECT` dins aquest bloc anònim no pot donar cap error doncs el comptatge (instrucció `count`) sempre dona un i només un resultat. L'execució d'aquest guió des de *SQL Developer* proporciona la sortida:

```
anonymous block completed

nbDept
-
5
```

Exemple de sentència `SELECT...INTO...` errònia dins codi PL/SQL

Considerem, a l'esquema *empresa*, el guió següent:

```
/* Programa: u5n1p04.sql (guió)
   Descripció: Exemple de sentència SELECT dins codi PL/SQL
               que pot provocar error segons les dades existents
   Autor: Isidre Guixà
*/
variable nomDept char(14);
variable locDept char(14);
begin
  select dnom, loc into :nomDept, :locDept
  from dept;
end;
/
print nomDept locDept
```

(!!) Trobareu el fitxer `u5n1p04.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

La sentència `SELECT` dissenyada en aquest bloc anònim anterior no és correcta, doncs si la taula `DEPT` és buida, no facilitarà cap fila i si la taula `DEPT` té més d'un departament, proporcionarà més d'una fila. En qualsevol de les dues situacions es produirà un error en temps d'execució. Aquesta sentència `SELECT` només tindria una execució correcta quan la

taula DEPT tingués una i només una fila. L'execució d'aquest guió des de SQL Developer quan la taula DEPT té més d'una fila proporciona la sortida:

```
Error starting at line 3 in command:
begin
  select dnom, loc into :nomDept, :locDept
  from dept;
end;
Error report:
ORA-01422: l'obtenció exacta retorna més que el nombre de files sol licitades
ORA-06512: a line 2
01422. 00000 -  "exact fetch returns more than requested number of rows"
*Cause:    The number specified in exact fetch is less than the rows returned.
*Action:   Rewrite the query or change number of rows requested

nomDept
------

locDept
------
```

Exemple de sentència SELECT dins codi PL/SQL que conté variables de substitució

Considerem, a l'esquema *empresa*, el guió següent:

```
/* Programa: u5n1p05.sql (guió)
   Descripció: Exemple de sentència SELECT dins codi PL/SQL
               que pot provocar error, segons la dada subministrada
               per l'usuari
   Autor: Isidre Guixà
*/
variable nomDept char(14);
variable locDept char(14);
set verify off
accept departament prompt "Número de Departament: "
begin
  select dnom, loc into :nomDept, :locDept
  from dept
  where dept_no=&departament;
end;
/
undefine departament
select :nomDept, :locDept from dual;
```

Trobareu el fitxer `u5n1p05.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Si responem 20 a la pregunta que el guió ens efectua a través de la instrucció accept, obtenim:

```
anonymous block completed
:NOMDEPT                       :LOCDEPT
------------------------------ ------------------------------
INVESTIGACIÓ                   MADRID
```

ja que hi ha un departament de codi 20 a MADRID que correspon a INVESTIGACIÓ. En canvi, si responent 25 a la pregunta, l'execució del guió ens dona el següent error, motivat per no existir cap departament de codi 25.

```
Error starting at line 4 in command:
begin
  select dnom, loc into :nomDept, :locDept
  from dept
  where dept_no=&departament;
end;
Error report:
ORA-01403: no s'ha trobat cap dada
ORA-06512: a line 2
01403. 00000 -  "no data found"
*Cause:
*Action:

:NOMDEPT                       :LOCDEPT
------------------------------ ------------------------------
```

El bloc anònim no està ben dissenyat (manca el control d'errors) doncs en funció de la resposta ijntroduïda per l'usuari i dels valors emmagatzemats a la base de dades, la seva

execució pot ser correcta o incorrecta, i això no es pot permetre. Un bloc de codi PL/SQL ha de proporcionar una correcta execució en qualsevol cas.

Observeu que en els tres exemples presentats s'està considerant un bloc PL/SQL, dins un guió de SQL*Plus, que conté una simple sentència `SELECT` quan aquesta podria haver estat inserida directament dins el guió (sense utilitzar, llavors, ni codi PL/SQL ni variables d'enllaç). Evidentment, per a executar les sentències `SELECT` dels exemples, no tenim cap necessitat del codi PL/SQL però penseu que es tracta simplement d'exemples introductoris i que en els programes reals, els blocs PL/SQL contenen sentències `SELECT`, càlculs a partir de les dades recuperades de la base de dades, sentències `INSERT`, `DELETE` i `UPDATE` i preses de decisions en funció de l'execució del codi PL/SQL.

1.6.2. Sentències `INSERT`, `UPDATE` i `DELETE`

PL/SQL (i en general en qualsevol llenguatge de tercera generació que hostatgi SQL) permet utilitzar aquestes sentències sense cap restricció que no sigui aplicar correctament la corresponent sintaxis.

Les sentències `UPDATE` i `DELETE` no provoquen cap error si no afecten a cap fila de la base de dades.

Trobareu el fitxer `u5n1p06.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Exemple d'utilització de sentències SQL-DML en blocs de codi PL/SQL

Considerem, a l'esquema *empresa*, el guió següent:

```
/* Programa: u5n1p06.sql (guió)
   Descripció: Exemple de sentències SELECT, INSERT, UPDATE i DELETE dins codi PL/SQL
   Autor: Isidre Guixà
*/
var nomDept char(14);
var locDept char(14);
set serveroutput on
set verify off
prompt "Efectuarem altes-baixes-modificacions dins un bloc PL/SQL."
prompt "Vegeu els departaments actualment existents a la base de dades."
select * from dept; -- (1)

prompt "Introdueixi les dades d'un departament inexistent a la base de dades." -- (2)
accept vs_numDept prompt "Codi de departament:"
accept vs_nomDept prompt "Nom de departament:"
accept vs_locDept prompt "Localitat de departament:"
begin
  insert into dept values (&vs_numDept, '&vs_nomDept', '&vs_locDept');
  dbms_output.put_line ('Hem inserit el departament a la base de dades');
  select dnom, loc into :nomDept, :locDept
  from dept
  where dept_no = &vs_numDept;
  dbms_output.put_line ('Comprovació de l''existència del departament &vs_numDept');
  dbms_output.put_line ('Nom.....:'||:nomDept);
  dbms_output.put_line ('Localitat:'||:locDept);
  delete dept where dept_no = &vs_numDept;
  dbms_output.put_line ('Hem eliminat el departament de la base de dades');
end;
/
undefine vs_numDept
undefine vs_nomDept
undefine vs_locDept
```

Suposem que l'usuari, una vegada observats els departaments visualitzats per la instrucció (1), veu que no hi ha el departament (60, 'COMPRES', 'GIRONA') i decideix treballar amb aquest, de manera que es produeix, a partir de (2), la següent seqüència d'accions:

```
"Introdueixi les dades d'un departament inexistent a la base de dades."
Codi de departament:60
Nom de departament:COMPRES
Localitat de departament:GIRONA
Hem inserit el departament a la base de dades
Comprovació de l'existència del departament 60
Nom......:COMPRES
Localitat:GIRONA
Hem eliminat el departament de la base de dades
```

En aquest exemple s'ha efectuat modificacions a la base de dades, encara que al final la taula resti igual que estava en un inici (doncs s'ha fet una alta, s'ha modificat el registre afegit i finalment s'ha eliminat), i no s'ha executat el `COMMIT` o el `ROLLBACK`. Per tant, si la sessió de treball està configurada amb `l'AUTOCOMMIT` desactivat, hi ha un bloqueig sobre el departament de codi 60 fins que no s'executi un `COMMIT` , un `ROLLBACK` o es surti de l'eina que provoca el bloqueig. Caldrà, doncs, recordar de posar les instruccions de validació que corresponguin.

1.6.3. Control de transaccions

Recordem que una transacció és un conjunt de sentències SQL que conjuntament modifiquen i alteren la base de dades i que els canvis poden convertir-se en permanents (sentència `COMMIT`) o es poden desfer (`ROLLBACK`).

PL/SQL suporta les següents instruccions pel control de transaccions:

```
commit;
rollback [to <nom_punt_salvaguarda>;
savepoint <nom_punt_salvaguarda>;
set transaction read only;
```

La sentència `SET TRANSACTION READ ONLY` permet assegurar que durant la transacció no s'efectuarà cap modificació a la base de dades. Aquesta instrucció, d'utilitzar-se, ha de ser la primera instrucció de la transacció.

Les instruccions `COMMIT`, `ROLLBACK` i `SAVEPOINT` tenen el mateix funcionament que en SQL.

1.6.4. Cursors

Els cursors, gran novetat del llenguatge SQL hostatjat en un llenguatge procedimental, permeten el tractament individualitzat del conjunt de files resultants d'una sentència `SELECT`.

Coneixem que la utilització de la sentència `SELECT...INTO...` és possible sempre i quan hi hagi la seguretat que el resultat contindrà una i només una fila. És de lògica pensar que en multitud d'ocasions es

necessitarà processar sentències SELECT que retornen cap o més d'una fila: els cursors ens ho permetran.

- Un cursor ha de declarar-se, a la zona declare, abans de ser utilitzat. La declaració consisteix en l'assignació d'un nom i una consulta en la que no ha d'aparèixer la clàusula into, seguint la sintaxis:

```
declare cursor <nom_cursor> is
<sentència_SELECT>;
```

- La utilització d'un cursor contempla tres operacions bàsiques: obertura, procés -una a una- de les corresponents files i tancament.

- L'obertura consisteix en l'execució de la consulta associada al cursor, el resultat de la qual queda emmagatzemada en una àrea associada al cursor.

La instrucció per l'obertura té la sintaxis:

```
open <nom_cursor>;
```

- El procés individual de les files resultants de la instrucció SELECT s'aconsegueix amb la instrucció FETCH:

```
fetch <nom_cursor> into <variables>;
```

En executar aquesta instrucció, els valors de les columnes de la sentència SELECT definida en el cursor, són copiats a les variables, que han de coincidir en nombre i en tipus i que poden estar agrupades en una variable registre.

Sabem declarar una variable registre a partir de la definició d'una taula:

```
<nom_variable> <esquema.taula>%rowtype;
```

però hem de saber que també es pot declarar a partir d'un cursor:

```
<nom_variable> <nom_cursor>%rowtype;
```

Cada instrucció FETCH provoca el tractament de la següent fila resultat de la sentència SELECT del cursor. No es pot tornar enrera!

- La instrucció CLOSE provoca la desactivació del cursor i, conseqüentment, la desaparició de l'àrea de treball associada, és a dir, s'alliberen els recursos del SGBD associats al cursor.

La instrucció de tancament té la sintaxis:

```
close <nom_cursor>;
```

Després de tancat, un cursor es pot tornar a obrir.

- Els cursors disposen d'uns determinats operadors per poder conèixer si s'ha arribat al final de les files, la quantitat de files recuperades i l'estat d'obertura o tancament. Així tenim:

```
<nom_cursor>.%found    -- A utilitzar després de fetch per saber si es té una fila
<nom_cursor>.%notfound -- A utilitzar després de fetch per saber si s'ha arribat al final
<nom_cursor>.%rowcount -- Comptador de les files tractades amb fetch
<nom_cursor>.%isopen   -- Per saber l'estat d'obertura del cursor
```

Exemple d'utilització de cursor via instruccions FETCH

Es demana, a l'esquema *empresa*, desenvolupar un programa per calculi el nombre d'empleats que té l'empresa i la suma dels seus salaris, sense utilitzar les funcions agregades COUNT i SUM.

Per aconseguir el nostre objectiu no podem fer altra cosa que declarar un cursor per a recorrer tota la taula d'empleats i anar sumaritzant els diferents salaris. Podem anar comptant els diferents empleats a mida que efectuem el recorregut o podem optar per a utilitzar l'operador rowcount a la fi del recorregut.

Trobareu el fitxer u5n1p07.sql en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n1p07.sql (guió)
   Descripció: Càlcul del nombre d'empleats de la taula EMP (esquema empresa)
               i de la suma dels seus salaris, sense utilitzar les funcions
               agregades COUNT i SUM.
   Autor: Isidre Guixà
*/
var sumSalaris number;
var nbEmpleats number;
declare
   cursor S is
      select nvl(salari,0) "salari" from emp;
   v_sal emp.salari%type;
begin
   :sumSalaris:=0;
   open S;
   fetch S into v_sal;
   while S%found loop
      :sumSalaris := :sumSalaris + v_sal;
      fetch S into v_sal;
   end loop;
   :nbEmpleats := S%rowcount;
  close S;
end;
/
select :sumSalaris, :nbEmpleats from dual;
```

- Com que la utilització de cursors acostuma a necessitar de les instruccions de control iteratives, PL/SQL proporciona una instrucció iterativa especial:

```
for <variable> in <cursor> loop
   <instruccions>
end loop;
```

Aquesta versió de bucle de cursor FOR declara, de forma implícita, una variable de tipus ROWTYPE, obre el cursor i de forma repetitiva realitza el FETCH de les files sobre la variable declarada. Finalment tanca el cursor quan totes les files han estat processades.

Exemple d'utilització de cursor via bucle de cursor FOR

Es demana, a l'esquema *empresa*, desenvolupar un programa per calculi el nombre d'empleats que té l'empresa, i la suma dels seus salaris, sense utilitzar les funcions agregades COUNT i SUM.

Per aconseguir el nostre objectiu no podem fer altra cosa que declarar un cursor per a recorrer tota la taula d'empleats i anar sumaritzant els diferents salaris. Si optem per utilitzar el bucle de cursor FOR no podrem utilitzar l'operador rowcount a la fi del recorregut, doncs el bucle de cursor FOR s'encarrega d'obrir el cursor, efectuar el recorregut i tancar el cursor i, per tant, a la fi del procés no podem preguntar per l'operador rowcount doncs el cursor ja restarà tancat.

Trobareu el fitxer u5n1p08.sql en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n1p08.sql (guió)
   Descripció: Càlcul del nombre d'empleats de la taula EMP (esquema empresa)
               i de la suma dels seus salaris, sense utilitzar les funcions
               agregades COUNT i SUM.
   Autor: Isidre Guixà
*/
var sumSalaris number;
var nbEmpleats number;
declare
   cursor S is
      select salari from emp;
begin
   :sumSalaris:=0;
   :nbEmpleats:=0;
   for e in S loop
      :sumSalaris := :sumSalaris + nvl(e.salari,0);
      :nbEmpleats := :nbEmpleats + 1;
   end loop;
end;
/
select :sumSalaris, :nbEmpleats from dual;
```

- En certes ocasions pot ser necessari especificar, en temps d'execució, alguna condició en la clàusula where de la sentència SELECT del cursor. En tal situació es necessita un **cursor parametritzat**.

En un cursor parametritzat, el paràmetre es declara en la definició del cursor:

```
declare
   cursor <nom_cursor> (<nom_argument> <tipus>) is
   <sentència_SELECT>;
```

Durant l'execució, cal comunicar al cursor el valor concret del paràmetre en cada obertura ja sigui de forma explícita en la instrucció OPEN o de forma implícita en la utilització d'un bucle de cursor FOR:

```
open <nom_cursor> (<expressió>);
```

```
for V in <nom_cursor> (<expressió>) loop
   ...
```

Exemple d'utilització de cursor amb paràmetre

Es demana, a l'esquema *empresa*, desenvolupar un programa que calculi el nombre d'empleats que té un departament, indicat via variable de substitució, i la suma dels seus salaris, sense utilitzar les funcions agregades COUNT i SUM.

Per aconseguir el nostre objectiu no podem fer altra cosa que declarar un cursor per a recorrer tota la taula d'empleats amb un paràmetre per a efectuar el filtratge pel camp departament.

Trobareu el fitxer u5n1p09.sql en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n1p09.sql (guió)
   Descripció: Càlcul del nombre d'empleats d'un departament (esquema empresa)
               passat via variable de substitució i de la suma dels seus salaris,
               sense utilitzar les funcions agregades COUNT i SUM.
   Autor: Isidre Guixà
*/
var sumSalaris number;
var nbEmpleats number;
set verify off
accept departament prompt "Introdueixi el codi de departament:"
declare
   cursor S (v_dept_no emp.dept_no%type) is
      select salari from emp where dept_no = v_dept_no;
begin
   :sumSalaris:=0;
   :nbEmpleats:=0;
   for e in S(&departament) loop
      :sumSalaris := :sumSalaris + nvl(e.salari,0);
      :nbEmpleats := :nbEmpleats + 1;
   end loop;
end;
/
undefine departament
select :sumSalaris, :nbEmpleats from dual;
```

- És interessant, en ocasions, poder assegurar que les files accessibles per un cursor no puguin ser actualitzades per altres processos. *Oracle* facilita la clàusula `for update` al final de la declaració de la sentència `SELECT` del cursor que provoca la sol·licitud de bloqueig de les files afectades pel cursor. Així, la seva sintaxis és:

```
<sentència SELECT>
for update [of <columna1>, <columna2>, ...] [nowait];
```

La clàusula `for update` identifica les files accessibles via el cursor i *Oracle* les bloqueja (totes les columnes i no només les indicades de manera optativa) en el moment d'obrir el cursor (no pas en el moment d'executar el `FETCH` sobre cada fila). Les files queden desbloquejades quan s'efectua un `COMMIT` o `ROLLBACK` de la transacció (no pas en el tancament del cursor). No es pot efectuar un `FETCH` per un cursor amb la clàusula `for update` després d'una instrucció `COMMIT`. (!!)

La clàusula opcional `nowait` permet a *Oracle* que no s'esperi si les files estan bloquejades per un altre usuari . Si s'utilitza aquesta clàusula i les files estan bloquejades, el control es retorna immediatament al programa que executa el cursor, amb un codi d'error susceptible de ser capturat, de manera que el procés pugui actuar en conseqüència. Si no s'utilitza la clàusula `nowait` i alguna de les files vinculades al cursor està bloquejada, el procés es queda en espera fins tenir accés a les files.

Observem que la clàusula `for update` permet indicar alguna columna. Això és especialment útil quan el cursor fa referència a diverses taules. En tal situació, la clàusula `for update` sense cap columna provoca el bloqueig de les files de les taules utilitzades per proporcionar les files del cursor. En utilitzar `for update of...` *Oracle* bloqueja les files de

les taules que contenen les columnes indicades. En qualsevol cas sempre es bloqueja la fila sencera.

Trobareu el fitxer `u5n1p10.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Exemple 1 de bloquetjos provocats per cursors

Considerem, a l'esquema *empresa*, l'execució del següent guió:

```
/* Programa: u5n1p10.sql (guió)
   Descripció: Exemple de guió (esquema empresa) que provoca el bloqueig de les files
               afectades per la sentència SELECT.
               Com que la sentència "for update" no va acompanyada de cap columna, es
               bloquegen les files de totes les taules implicades en la SELECT.
   Autor: Isidre Guixà
*/
accept departament prompt "Introdueixi el codi del departament:"
set verify off
declare
   cursor S (v_dept_no emp.dept_no%type) is
      select emp_no, cognom, dnom
      from emp inner join dept on dept.dept_no=emp.dept_no
      where emp.dept_no=v_dept_no
      for update;
begin
   for e in S(&departament) loop
      null;
   end loop;
end;
/
undefine departament
```

Observem que aquest guió és ben absurd, doncs aparentment no fa res... Però sí que fa quelcom: bloqueja totes les files de les taules `EMP` i `DEPT` corresponents al departament que indica l'usuari. Vegem-ho.

Executem aquest guió des d'una sessió *SQL*Plus* connectats a l'esquema *empresa*:

```
SQL> @u5n1p10
Introdueixi el codi del departament:10

El procediment PL/SQL ha finalitzat correctament.
```

Com es veu, l'usuari ha introduït el valor 10 com a codi de departament.

Ara, en una altra sessió *SQL*Plus* connectats a l'esquema *empresa*, intentem executar qualsevol de les sentències següents:

```
SQL> update emp set salari=salari * 2;
```

La sessió es queda penjada, doncs aquesta instrucció provoca la modificació de files algunes de les quals estan bloquejades pel guió `u5n1p10`.

```
SQL> update emp set salari=salari * 2 where dept_no=20;
```

Aquesta sentència no es queda penjada doncs cap de les files que es modifiquen no estan bloquejades per l'execució del guió `u5n1p10`.

```
SQL> update dept set loc='Lleida' where dept_no=10;
```

La sessió es queda penjada, doncs aquesta instrucció provoca la modificació d'una fila bloquejada pel guió `u5n1p10`.

```
SQL> update dept set loc='Girona' where dept_no=20;
```

Aquesta sentència no es queda penjada doncs la fila modificada no està bloquejada per l'execució del guió `u5n1p10`.

Les sessions penjades no continuen l'execució fins que no s'eliminen els bloquetjos de les files, fet que només es du a terme executant la sentència `ROLLBACK` o `COMMIT` a la sessió on s'executa el guió `u5n1p10`.

Exemple 2 de bloquetjos provocats per cursors

```
/* Programa: u5n1p11.sql (guió)
   Descripció: Exemple de guió (esquema empresa) que provoca el bloqueig de les files
               afectades per la sentència SELECT.
               Com que la sentència "for update" va acompanyada de columnes, només es bloquegen
               les files de les taules implicades en la SELECT que contenen les columnes.
   Autor: Isidre Guixà
*/
accept departament prompt "Introdueixi el codi del departament:"
set verify off
declare
   cursor S (v_dept_no emp.dept_no%type) is
      select emp_no, cognom, dnom
      from emp inner join dept on dept.dept_no=emp.dept_no
      where emp.dept_no=v_dept_no
      for update of emp_no;
begin
   for e in S(&departament) loop
      null;
   end loop;
end;
/
undefine departament
```

Observem que aquest guió és ben absurd, doncs aparentment no fa res... Però sí que fa quelcom: bloqueja totes les files de la taula EMP corresponent al departament que indica l'usuari (ja que la clàusula `for update of` va acompanyada d'una columna de la taula EMP). No es produeix cap bloqueig sobre la taula DEPT. Vegem-ho.

Executem aquest guió des d'una sessió *SQL*Plus* connectats a l'esquema *empresa*:

> Trobareu el fitxer `u5n1p11.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
SQL> @u5n1p11
Introdueixi el codi del departament:10

El procediment PL/SQL ha finalitzat correctament.
```

Com es veu, l'usuari ha introduït el valor 10 com a codi de departament.

Ara, en una altra sessió *SQL*Plus* connectats a l'esquema *empresa*, intentem executar qualsevol de les sentències següents:

```
SQL> update emp set salari=salari * 2;
```

La sessió es queda penjada, doncs aquesta instrucció provoca la modificació de files algunes de les quals estan bloquejades pel guió u5n1p10.

```
SQL> update emp set salari=salari * 2 where dept_no=20;
```

Aquesta sentència no es queda penjada doncs cap de les files que es modifiquen no estan bloquejades per l'execució del guió u5n1p11.

```
SQL> update dept set loc='Lleida' where dept_no=10;
```

Aquesta sentència no es queda penjada doncs cap fila de la taula DEPT no està bloquejada per l'execució del guió u5n1p11.

Les sessions penjades no continuen l'execució fins que no s'eliminen els bloquetjos de les files, fet que només es du a terme executant la sentència ROLLBACK o COMMIT a la sessió on s'executa el guió u5n1p11.

- Per finalitzar aquesta introducció al tractament dels cursors, comentar que les files seleccionades pels cursors es poden modificar i/o esborrar, sempre i quan la sentència SELECT que defineix el cursor incorpori la clàusula `for update`.

És a dir, *Oracle* permet utilitzar les sentències UPDATE i/o DELETE sobre la fila activa d'un cursor. Això s'aconsegueix afegint al final d'aquestes

instruccions la clàusula `where current of` per indicar que es vol modificar i/o eliminar la fila activa. La seva sintàxis és:

```
delete ... where current of <nom_cursor>;
update ... where current of <nom_cursor>;
```

Aquestes instruccions s'han d'executar amb el cursor obert i esborren o modifiquen únicament la fila seleccionada en el darrer FETCH.

Exemple de cursor que modifica valors de les files recorregudes

Es demana, a l'esquema *empresa*, desenvolupar un programa que permeti incrementar el salari dels empleats d'un departament. Cal, en temps d'execució, introduir el percentatge d'increment i el codi de departament afectat. S'exigeix efectuar el recorregut pels empleats afectats amb un cursor.

Trobareu el fitxer `u5n1p12.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n1p12.sql (guió)
   Descripció: Programa que incrementa el salari dels empleats d'un departament.
               Percentatge d'increment i departament són introduïts per l'usuari.
   Autor: Isidre Guixà
*/
accept departament prompt "Introdueixi el codi del departament:"
accept percentatge prompt "Introdueixi el percentatge d'increment:"
set verify off
declare
   cursor S (v_dept_no emp.dept_no%type) is
      select salari
      from emp
      where dept_no=v_dept_no
      for update of emp_no;
begin
   for e in S(&departament) loop
      update emp set salari=salari*(1+&percentatge/100) where current of S;
   end loop;
end;
/
undefine departament
undefine percentatge
```

Observem que estem modificant la columna `salari` la qual no s'ha emprat a la clàusula `for update of...` de la sentència `SELECT`. Recordem que la clàusula `for update of...` únicament s'utilitza per a indicar sobre quines taules de la sentència `SELECT` s'ha d'efectuar els bloquetjos. Com que en el nsotre exemple la sentència `SELECT` només conté la taula `EMP`, haguéssim pogut utilitzar la clàusula `for update` enlloc de `for update of...`

Observem també que haguéssim aconseguit el mateix resultat substituint el bloc anònim anterior pel bloc anònim següent:

```
begin
   update emp set salari=salari*(1+&percentatge/100)
   where dept_no=&departament;
end;
```

Evidentment el darrer bloc és més eficient que el primer que utilitza cursors, ja que amb una única sentència `UPDATE` efectua l'actualització desitjada, mentre que en la utilització del cursor, *Oracle* selecciona les files afectades i, posteriorment, executa l'actualització una a una. Per tant, utilitzarem cursors quan sigui imprescindible en no disposar d'una sentència SQL que permeti efectuar l'operació en el conjunt de files.

1.6.5. Tractament d'errors

L'Oracle anomena **excepcions** el tractament dels errors que puguin tenir lloc durant la part executable d'un bloc PL/SQL.

La zona `exception` (opcional) dels blocs PL/SQL està destinada a contenir el tractament que s'ha de donar a cada error que pugui esdevenir en temps d'execució. ‼

Així, quan es produeix un error (excepció) en execució, el control del programa passa automàticament al bloc d'excepcions, de manera que l'execució seqüencial de les sentències de la zona `begin` queda interrompuda definitivament.

Hi ha dos tipus d'excepcions:

- **Excepcions internes predefinides**. Són excepcions ja definides, associades a un tipus d'error *Oracle*, i que es poden utilitzar sense necessitat de definir-les. La taula 7 ens en presenta un recull.

Com exemple, podem considerar les excepcions que es produeixen quan s'intenta inserir una fila que viola la clau primària o la integritat referencial.

- **Excepcions definides pel programador**. Són excepcions que pot definir el propi programador, a la zona `declare`, i que poden tenir o no, relació amb errors *Oracle*.

La sintaxis per la zona `exception` d'un bloc PL/SQL és:

```
exception
   when <nom_excepció> then
      <seqüència_de_sentències>
   when <nom_excepció> then
      <seqüència_de_sentències>
   ...
   [when others then
      <seqüència_de_sentències>]
end;
```

En `<nom_excepció>` cal indicar l'excepció definida per *Oracle* o a una de les excepcions definides pel programador a la zona `declare` del bloc PL/SQL. Les excepcions més comunes ja estan predefinides pel sistema, però es poden definir nous noms per la resta d'excepcions.

Taula 7. Principals excepcions internes predefinides en PL/SQL

Excepció	Situació en que té lloc
`DUP_VAL_ON_INDEX`	Quan s'intenta inserir una fila i es repeteix una clau única
`INVALID_CURSOR`	Quan es fa referència a un cursor que no és vàlid
`INVALID_NUMBER`	Quan una conversió de tipus `CHAR` a `NUMBER` falla
`LOGON_DENIED`	Quan s'utilitza un incorrecte `USERNAME`

Excepció	Situació en que té lloc
NO_DATA_FOUND	Quan una consulta no retorna cap fila
NOT_LOGGED_ON	Quan s'intenta efectuar una crida a *Oracle* sense estar connectat
PROGRAM_ERROR	Quan ha tingut lloc un error intern de PL/SQL
STORAGE_ERROR	Quan PL/SQL està fora de memòria o la memòria està plena
TIMEOUT_ON_RESOURCE	Quan ha acabat un temps d'espera i el recurs no està disponible
TOO_MANY_ROWS	Quan uns sentència SELECT retorna més d'una fila
VALUE_ERROR	Quan qualsevol tipus de conversió, aritmètica, numèrica o cadena, és inconsistent
ZERO_DIVIDE	Quan es divideix per zero

Exemple 1 de captura d'excepcions PL/SQL

Considerem, a l'esquema *empresa*, el següent guió:

```
/* Programa: u5n1p13.sql (guió)
   Descripció: Exemple d'excepció no capturada si l'usuari introdueix un codi de departament
               inexistent a la taula DEPT (esquema empresa)
   Autor: Isidre Guixà
*/
var nom varchar2(14)
accept departament prompt "Introdueixi el codi del departament:"
set verify off
begin
   select dnom into :nom
   from dept
   where dept_no=&departament;
   update emp set salari=salari*(1+3/100);
end;
/
print nom
undefine departament
```

Posem-lo en execució en una consola *SQL*Plus* i introduïm, com a resposta a la sol·licitud de departament, el codi 99 inexistent a la base de dades. Vegem què succeeix:

Trobareu el fitxer `u5n1p13.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
SQL> @u5n1p13
Introdueixi el codi del departament:99
begin
*
ERROR a la línia 1:
ORA-01403: no s'ha trobat cap dada
ORA-06512: a line 2


NOM
--------------------------------
```

Com que l'usuari ha introduït un valor pel que no existeix cap fila a la taula DEPT, *Oracle* provoca una excepció NO_DATA_FOUND en la sentència SELECT i passa a la zona d'excepcions trencant la seqüència lògica del programa (és a dir, la sentència UPDATE ja no s'executa). I com que no hi ha zona d'excepcions, *Oracle* finalitza l'execució del programa amb l'error següent i mostrant el valor NULL de la variable d'enllaç nom.

```
ORA-01403: no s'ha trobat cap dada
```

En aquest punt hem de notar que tots els errors *Oracle* tenen una codificació i una descripció més o menys entenedora, en l'idioma en què ha estat instal·lat el producte. En la documentació subministrada per *Oracle* es pot localitzar explicació més o menys detallada sobre cada error.

Trobareu el fitxer `u5n1p14.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

L'error del codi PL/SQL anterior hauria d'haver estat previst pel programador, ja que és possible que l'usuari entri un departament inexistent. Així, caldria millorar el guió anterior:

```
/* Programa: u5n1p14.sql (guió)
   Descripció: Exemple d'excepció capturada si l'usuari introdueix un codi de departament
               inexistent a la taula DEPT (esquema empresa)
   Autor: Isidre Guixà
*/
var nom varchar2(14)
accept departament prompt "Introdueixi el codi del departament:"
set verify off
begin
   select dnom into :nom
   from dept
   where dept_no=&departament;
   update emp set salari=salari*(1+3/100);
exception
   when no_data_found then
      :nom:='?????';
end;
/
print nom
undefine departament
```

L'execució d'aquest guió, introduint com a codi de departament un valor inexistent a la taula DEPT causa l'excepció NO_DATA_FOUND que és capturada per la zona exception. La sentència UPDATE tampoc s'executa, però des del punt de vista de PL/SQL, el procediment ha finalitzat correctament, doncs el programador havia previst l'excepció. Vegem-ho:

```
SQL> @u5n1p14
Introdueixi el codi del departament:90

El procediment PL/SQL ha finalitzat correctament.

NOM
--------------------------------
?????
```

En el programa anterior és lògic no haver capturat l'excepció TOO_MANY_ROWS que es produeix quan una sentència SELECT pot donar més d'una fila. Aquí és impossible, doncs el codi de departament és clau primària de la taula DEPT i no es pot donar la situació, però en altres casos caldrà tenir-ho en compte.

Exemple 2 de captura d'excepcions PL/SQL

Es demana, a l'esquema *empresa*, un guió que demani un nom de localitat i informi del nom del departament ubicat en aquella localitat.

Analitzant el que ens demanen i estudiant la definició de la taula DEPT, veiem que en una localitat hi pot haver zero, un o més d'un departament. Per tant, tenim dues opcions:

a) Programa que controli les tres situacions possibles i en cas d'haver varis departaments a la localitat, doni un missatge informatiu sobre la circumstància.

Trobareu el fitxer u5n1p15.sql en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n1p15.sql (guió)
   Descripció: Programa (esquema empresa) que, donat un nom de localitat, cerca el nom del
               departament ubicat en aquella localitat.
               Resultats: - Informa que no hi ha cap departament en la localitat
                          - Informa del nom del departament si només n'hi ha un
                          - Informa de l'existència de diversos departaments si n'hi ha més d'un
   Autor: Isidre Guixà
*/
accept localitat prompt "Introdueixi el nom de la localitat:"
set verify off
set serveroutput on
declare
   nom varchar2(14);
begin
   select dnom into :nom
   from dept
   where upper(loc)=upper('&localitat');
   dbms_output.put_line('A la localitat &localitat hi ha el departament '||:nom);
exception
   when no_data_found then
      dbms_output.put_line('No hi ha cap departament a la localitat &localitat');
   when too_many_rows then
      dbms_output.put_line('Hi ha varis departaments a la localitat &localitat');
end;
/
```

```
undefine localitat
```

b) Programa que controli les tres situacions possibles i en cas d'haver varis departaments a la localitat, vagi mostrant els diferents noms de departament.

```
/* Programa: u5n1p16.sql (guió)
   Descripció: Programa (esquema empresa) que, donat un nom de localitat, cerca el nom de tots
               els departaments ubicats en aquella localitat.
   Autor: Isidre Guixà
*/
accept localitat prompt "Introdueixi el nom de la localitat:"
set verify off
set serveroutput on
declare
   nom1 dept.dnom%type;
   nom2 dept.dnom%type;
   cursor D (v_loc dept.loc%type) is
      select dnom
      from dept
      where upper(loc)=upper(v_loc);
begin
   open D ('&localitat');
   fetch D into nom1;
   if D%notfound then
      dbms_output.put_line('No hi ha cap departament a la localitat &localitat');
   else
      fetch D into nom2;
      if D%notfound then
         dbms_output.put_line('A la localitat &localitat hi ha un únic departament '||nom1);
      else
         dbms_output.put_line('Hi ha varis departaments a la localitat &localitat:');
         dbms_output.put_line(nom1);
         while (D%found) loop
            dbms_output.put_line(nom2);
            fetch D into nom2;
         end loop;
      end if;
   end if;
   close D;
end;
/
undefine localitat
```

Trobareu el fitxer `u5n1p16.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Si la zona `exception` d'un bloc no contempla el tractament d'error corresponent a l'excepció produïda, l'execució passa automàticament a la zona `exception` del bloc PL/SQL que conté el bloc en el que s'ha produït l'excepció i així successivament fins trobar un apartat `exception` que contempli l'excepció o fins arribar al nivell més extern de blocs PL/SQL.

Una excepció produïda dins la zona `exception` d'un bloc PL/SQL, provoca el salt immediat a la zona `exception` del bloc immediatament superior, abandonant definitivament la zona `exception` on s'ha provocat l'error.

Trobareu el fitxer `u5n1p17.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Exemple de la propagació de les excepcions al bloc immediatament superior

Considerem el guió següent:

```
/* Programa: u5n1p17.sql (guió)
   Descripció: Exemple de captura d'excepcions en el bloc on es produeix.
   Autor: Isidre Guixà
*/
set serveroutput on
begin
   dbms_output.put_line('Estem en el bloc més extern.');
   declare
      codiDept dept.dept_no%type;
   begin
      dbms_output.put_line('Estem en el bloc més intern.');
      select dept_no into codiDept from dept; --1
      dbms_output.put_line('La sentència SELECT s''ha executat sense cap problema.'); --2
```

```
   exception
      when too_many_rows then --3
         dbms_output.put_line('Estem a ''when too_many_rows'' del bloc intern.');
   end;
   dbms_output.put_line('Hem tornat al bloc extern.'); --4
exception
   when too_many_rows then
      dbms_output.put_line('Estem a ''when too_many_rows'' del bloc intern.');
end;
/
```

L'execució en una consola *SQL*Plus*, si la taula DEPT té més d'un departament, produeix:

```
SQL>  @u5n2p17
Estem en el bloc més extern.
Estem en el bloc més intern.
Estem a 'when too_many_rows' del bloc intern.
Hem tornat al bloc extern.

El procediment PL/SQL ha finalitzat correctament.
```

Fixem-nos que la instrucció `--1` provoca una excepció degut a que la taula DEPT conté més d'un departament i el flux d'execució salta a la instrucció `--3` (sense executar-se `--2`) on es captura l'excepció produïda. El bucle intern ha finalitzat correctament i en sortir es continua a la instrucci-o `–4` del bucle extern, que també finalitza correctament.

Considerem, el guió següent:

```
/* Programa: u5n1p18.sql (guió)
   Descripció: Exemple de propagació d'excepcions al bloc immediatament superior.
   Autor: Isidre Guixà
*/
set serveroutput on
begin
   dbms_output.put_line('Estem en el bloc més extern.');
   declare
      codiDept dept.dept_no%type;
   begin
      dbms_output.put_line('Estem en el bloc més intern.');
      select dept_no into codiDept from dept; --1
      dbms_output.put_line('La sentència SELECT s''ha executat sense cap problema.'); --2
   end;
   dbms_output.put_line('Hem tornat al bloc extern.'); --3
exception
      when too_many_rows then --4
         dbms_output.put_line('Estem a ''when too_many_rows'' del bloc extern.');
end;
/
```

La seva execució en una consola *SQL*Plus*, suposant que la taula DEPT tingui més d'un departament, produeix la sortida:

```
SQL> @u5n1p18
Estem en el bloc més extern.
Estem en el bloc més intern.
Estem a 'when too_many_rows' del bloc extern.

El procediment PL/SQL ha finalitzat correctament.
```

Fixem-nos que la instrucció `--1` provoca una excepció degut a que la taula DEPT conté més d'un departament i el flux d'execució es trenca (sense executar-se `--2`) cercant en el bloc `exception` la captura de l'excepció. Com que no hi és, el bloc intern finalitza amb excepcióp que es propaga al bloc extern i en trenca el flux (no s'executa `--3`) i es salta a la zona exception amb l'ànim de capturar l'excepció, fet que succeeix a la instrucció `--4`. El bucle extern finalitza correctament.

Trobareu el fitxer `u5n1p18.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Oracle facilita les **funcions** `SQLCODE` **i** `SQLERRM` per obtenir, en qualsevol zona `exception`, el codi d'error de l'excepció més recent i el corresponent missatge.

Per a les excepcions internes predefinides, SQLCODE retorna el valor de l'error *Oracle*, que sempre és negatiu exceptuant el cas de l'error NO_DATA_FOUND, cas en el que SQLCODE retornar +100. Per a les excepcions definides pel programador, SQLCODE retorna +1 (si és una excepció no vinculada a cap error *Oracle*) o el valor de l'error *Oracle* (si és una excepció vinculada a un error *Oracle*). La funció SQLCODE, fora d'una zona exception (és a dir, si no s'ha produït error) retorna el valor zero.

La funció SQLCODE s'invoca sense cap argument. En canvi, la funció SQLERRM pot ser invocada sense argument o amb un argument enter.

La funció SQLERRM retorna el missatge associat amb el seu argument. En cas d'omissió d'argument, retorna el missatge d'error associat al valor actual que retorna la funció SQLCODE. La utilització de la funció SQLERRM sense argument només té sentit en una zona exception, doncs en aquest cas és quan la funció SQLCODE dona un valor diferent de zero. La funció SQLERRM sense argument sempre retorna el missatge:

```
ORA-0000: normal, successful completion
```

Per a les excepcions internes, la funció SQLERRM retorna el missage associat amb l'error *Oracle*, el qual sempre comença amb el codi d'error *Oracle* (zero en el cas en que no hi hagi error), exceptuant el cas de l'error NO_DATA_FOUND en que el missatge comença amb el valor -1403. Per a les excepcions definides pel programador, SQLERRM retorna el missatge User-Defined Exception (si és una excepció no vinculada a cap error *Oracle*) o el missatge associat amb l'error *Oracle* (si és una excepció vinculada a un error *Oracle*).

Trobareu el fitxer u5n1p19.sql en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Exemple d'utilització de les funcions SQLCODE i SQLERRM

Considerem el guió següent:

```
/* Programa: u5n1p19.sql (guió)
   Descripció: Exemple d'utilització de les funcions SQLCODE i SQLERRM.
   Autor: Isidre Guixà
*/
set serveroutput on
declare
   a number;
begin
   a:=1/0;
exception
   when others then
      dbms_output.put_line('S''ha produït l''error: '||SQLCODE);
      dbms_output.put_line('Missagte d''Oracle: '||SQLERRM);
end;
/
```

La seva execució en una consola SQL*Plus produeix la sortida:

```
SQL> @u5nqp17
S'ha produït l'error: -1476
Missagte d'Oracle: ORA-01476: el divisor és igual a zero

El procediment PL/SQL ha finalitzat correctament.
```

Les sentències DELETE i UPDATE no generen cap excepció si no poden esborrar ni modificar, respectivament, cap fila. Com podem controlar-ho?

Oracle proporciona, per aquest menester, un cursor especial, de nom SQL, que podem utilitzar amb els coneguts operadors found, notfound i rowcount. (!!)

(!!) Trobareu el fitxer u5n1p20.sql en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Exemple d'utilització del cursor SQL per a control de la instrucció DELETE

Considerem el guió següent, a l'esquema *empresa*:

```
/* Programa: u5n1p20.sql (guió)
   Descripció: Exemple d'utilització del cursor SQL per controlar el resultat de la
               sentència DELETE
   Autor: Isidre Guixà
*/
accept departament prompt "Codi de departament a eliminar:"
set verify off
set serveroutput on
begin
   delete from dept where dept_no=&departament;
   if SQL%notfound then
      dbms_output.put_line('No s''ha eliminat cap fila.');
   else
      dbms_output.put_line('Fila eliminada.');
   end if;
end;
/
undefine departament
```

Per a comprovar-ne el seu correcte funcionament, sembla lògic executar-lo dues vegades, una indicant un valor existent a la taula DEPT de manera que es produeixi l'eliminació i una altra indicant un valor inexistent a la taula DEPT. Per a la primera prova, inserim prèviament un departament que després indicarem com a departament a eliminar. Vegem-ne quina sortida s'obté en una consola SQL*Plus:

```
SQL> insert into dept values (99,'OCI','SITGES');

1 fila creada.

SQL> @u5n1p20
Codi de departament a eliminar:99
Fila eliminada.

El procediment PL/SQL ha finalitzat correctament.

SQL> @u5n1p20
Codi de departament a eliminar:98
No s'ha eliminat cap fila.

El procediment PL/SQL ha finalitzat correctament.
```

Sembla que el funcionament és correcte, però recordem que els programes han d'estar pensats per a la correcta execució en qualsevol situació. Vegem què succeeix quan el departament a eliminar és el 10:

```
SQL> @u5n1p20
Codi de departament a eliminar:10
begin
*
ERROR a la línia 1:
ORA-02292: s'ha violat la restricció d'integritat (EMPRESA.EMP_FK_DEPT) - s'ha
trobat un registre fill
ORA-06512: a line 2
```

Fixem-nos que l'intent d'eliminació del departament 10 violaria la restricció d'integritat EMPRESA.EMP_FK_DEPT definida a la taula EMPRESA, fet que vol dir que a la taula EMPRESA hi ha files que referencien la fila de la taula DEPT que volem eliminar i això no és factible doncs la restricció d'integritat EMPRESA.EMP_FK_DEPT no està definida amb l'eliminació en cascada.

Observem que l'error que es produeix és el -2292. El programador d'aquest codi PL/SQL hauria d'haver previst la possibilitat d'aquest error i l'hauria d'haver interceptat en una zona `exception`. Així, el guió és millorable:

```
/* Programa: u5n1p21.sql (guió)
   Descripció: Exemple d'utilització del cursor SQL per controlar el resultat de la
               sentència DELETE, tenint en compte possible restricció d'integritat referencial
   Autor: Isidre Guixà
*/
accept departament prompt "Codi de departament a eliminar:"
set verify off
set serveroutput on
begin
   delete from dept where dept_no=&departament;
   if SQL%notfound then
      dbms_output.put_line('No s''ha eliminat cap fila.');
   else
      dbms_output.put_line('Fila eliminada.');
   end if;
exception
   when others then
      if SQLCODE=-2292 then
         dbms_output.put_line('No s''ha pogut eliminar la fila per restricció d''integritat
referencial');
      end if;
end;
/
undefine departament
```

Ara, vegem què es produeix en l'intent d'eliminació del departament 10:

```
SQL> @u5n1p21
Codi de departament a eliminar:10
No s'ha pogut eliminar la fila per restricció d'integritat
referencial

El procediment PL/SQL ha finalitzat correctament.
```

(!!) Trobareu el fitxer `u5n1p21.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

El programador pot provocar, en qualsevol moment, una excepció predefinida d'*Oracle* amb la sentència `RAISE`: (!!)

```
raise <nom_excepció>;
```

Exemple de provocació d'excepció per part del programador

El guió següent (esquema *empresa*) ens mostra com el programador provoca una excepció quan una sentència `DELETE` no pot eliminar cap fila de manera que es trencaria el flux normal del programa i es passaria a la zona `exception`:

(!!) Trobareu el fitxer `u5n1p22.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n1p22.sql (guió)
   Descripció: Exemple de control sobre una sentència DELETE
   Autor: Isidre Guixà
*/
accept departament prompt "Codi de departament a eliminar:"
set verify off
set serveroutput on
begin
   delete from dept where dept_no=&departament;
   if SQL%notfound then
      raise no_data_found;
   else
      dbms_output.put_line('Fila eliminada.');
   end if;
exception
   when no_data_found then
      dbms_output.put_line('No s''ha eliminat cap fila.');
   when others then
      if SQLCODE=-2292 then
         dbms_output.put_line('No s''ha pogut eliminar la fila per restricció d''integritat
referencial');
      end if;
end;
/
undefine departament
```

El programador pot definir excepcions que no tenen cap relació amb errors d'*Oracle*. ‼

És a dir, en certes ocasions pot interessar provocar un error, per aconseguir trencar el flux d'execució del programa i passar a la zona d'excepcions.

Aquestes excepcions s'han de declarar a la zona `declare` amb la sintaxis:

```
<nom_excepció> exception;
```

El programador decideix, en el bloc PL/SQL, quan provocar-la, amb la utilització de la sentència `raise`.

‼ Trobareu el fitxer `u5n1p23.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Exemple d'excepció definida del programador sense vinculació amb error Oracle

El guió següent (esquema *empresa*) ens mostra com el programador defineix una excepció `esborrat_no_efectuat` que utilitza quan una sentència `DELETE` no pot eliminar cap fila de manera que es trencaria el flux normal del programa i es passaria a la zona `exception`:

```
/* Programa: u5n1p23.sql (guió)
   Descripció: Exemple d'exception definida per l'usuari, per al control sobre una sentència
               DELETE, sense vinculació amb cap error Oracle
   Autor: Isidre Guixà
*/
accept departament prompt "Codi de departament a eliminar:"
set verify off
set serveroutput on
declare
   esborrat_no_efectuat exception;
begin
   delete from dept where dept_no=&departament;
   if SQL%notfound then
      raise esborrat_no_efectuat;
   else
      dbms_output.put_line('Fila eliminada.');
   end if;
exception
   when esborrat_no_efectuat then
      dbms_output.put_line('No s''ha eliminat cap fila.');
      dbms_output.put_line('SQLCODE: '||SQLCODE);
      dbms_output.put_line('SQLERRM: '||SQLERRM);
   when others then
      if SQLCODE=-2292 then
         dbms_output.put_line('No s''ha pogut eliminar la fila per restricció d''integritat
referencial');
      end if;
end;
/
undefine departament
```

Hem aprofitat el guió per a veure els valors que retornen les funcions `SQLCODE` i `SQLERRM` davant una excepció definida pel programador sense viculació amb cap error *Oracle*. Vegem la sortida que es produeix per una consola *SQL*Plus* en indicar l'eliminació d'un departament inexistent a la taula `DEPT`:

```
SQL> @u5n1p23
Codi de departament a eliminar:99
No s'ha eliminat cap fila.
SQLCODE: 1
SQLERRM: User-Defined Exception

El procediment PL/SQL ha finalitzat correctament.
```

El programador pot **definir excepcions relacionades amb errors d'Oracle**, de manera que es puguin interceptar en la zona d'excepcions. (!!)

Per a poder definir aquestes excepcions és necessari conèixer el codi d'error que *Oracle* associa a l'error i per a trobar-lo haurem de provocar l'error per recollir el seu codi per medi de la funció `SQLCODE`.

La sintaxis per la declaració d'aquest tipus d'excepció és:

```
<nom_excepció> exception;
pragma exception_init (<nom_excepció>, <codi_error_Oracle>);
```

Exemple d'excepció definida del programador amb vinculació amb error Oracle

Considerem l'error que es produeix en executar la següent sentència, motivat per que el mes no està ben introduït, ja que l'idioma actiu és el català i, per tant, cal escriure 'Desembre'.

```
d:=to_date('25 December 92','DD Month YY');
```

Detectem l'error executant el bloc PL/SQL:

```
declare
   d date;
begin
   d:=to_date('25 December 92','DD Month YY');
end;
/
```

L'execució provoca l'error:

```
ORA-01843: no és un mes vàlid
```

Per tant podem definir, en el bloc PL/SQL que ens interessi, una excepció associada a l'error -1843, amb un nom fàcil de recordar. El següent guió ens en mostra la utilització:

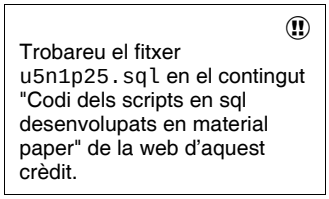
(!!) Trobareu el fitxer `u5n1p24.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n1p24.sql (guió)
   Descripció: Exemple d'exception definida per l'usuari, per al control sobre el mes del tipus
               data, amb vinculació amb l'error que provoca Oracle
   Autor: Isidre Guixà
*/
set serveroutput on
declare
   error_en_el_mes_de_la_data exception;
   pragma exception_init (error_mes, -1843);
   d date;
begin
   d:=to_date('25 December 92','DD Month YY');
exception
   when error_en_el_mes_de_la_data then
      dbms_output.put_line('SQLCODE: '||SQLCODE);
      dbms_output.put_line('SQLERRM: '||SQLERRM);
end;
/
```

En l'execució del guió podem veure que les funcions `SQLCODE` i `SQLERRM` retornen els valors de l'error *Oracle*. Sembla més fàcil pel programador (i més llegible en el codi) utilitzar el nom `error_en_el_mes_de_la_data` que no pas el valor -1843. Vegem-ho

```
SQL> @u5n1p24
SQLCODE: -1843
SQLERRM: ORA-01843: no és un mes vàlid

El procediment PL/SQL ha finalitzat correctament.
```

Per últim, remarcar que el control d'errors sempre s'efectua en la zona `exception` i que qualsevol intent de control d'error en la zona d'execució és un greu error de programació, doncs en cas de produir-se l'excepció, el flux d'execució normal s'avortaria, saltant a la zona `exception`, i no es passarà mai pel control d'error situat a la zona d'execució.


Trobareu el fitxer `u5n1p25.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Exemple d'intent erroni de control d'excepció en zona d'execució

Considerem el següent guió (esquema *empresa*):

```
/* Programa: u5n1p25.sql (guió)
   Descripció: Exemple d'intent de control d'error erroni (1) dins la zona d'execució.
               Si es produeix un error a la sentència SELECT a causa de no trobar cap departament
               amb el codi indicat per l'usuari, el flux del programa saltarà a la zona exception
               (inexistent en aquest cas) i MAI passarà pel punt (1)
   Autor: Isidre Guixà
*/
accept departament prompt "Introdueixi codi de departament:"
set verify off
set serveroutput on
declare
   nom dept.dnom%type;
begin
   select dnom into nom
   from dept
   where dept_no=&departament;
   if SQLCODE=100 then -- (1)
      dbms_output.put_line('No s''ha trobat el departament amb el codi &departament');
   else
      dbms_output.put_line('El nom del departament &departament és '||nom);
   end if;
end;
/
undefine departament
```

Vegem les diferents sortides que es produeixen per una consola *SQL*Plus* quan l'usuari introdueix un valor de departament existent (sortida correcta) i un valor de departament inexistent (es produeix una excepció no tractada):

```
SQL> @u5n1p25
Introdueixi codi de departament:10
El nom del departament 10 és COMPTABILITAT

El procediment PL/SQL ha finalitzat correctament.

SQL> @u5n1p25
Introdueixi codi de departament:99
declare
*
ERROR a la línia 1:
ORA-01403: no s'ha trobat cap dada
ORA-06512: a line 4
```

2. Codi PL/SQL dins la base de dades.

En aquests moments coneixem els fonaments del llenguatge PL/SQL amb els quals som capaços de desenvolupar blocs de codi PL/SQL que sabem inserir dins guions per a ser executats des d'una consola *SQL*Plus* o des de *SQL Developer*.

El llenguatge PL/SQL, però, no ha estat pensat per a ser utilitzat únicament en guions, sinó que té moltes més possibilitats algunes de les quals consisteixen en tenir codi PL/SQL dins la base de dades, en diverses formes: subprogrames, paquets i disparadors.

PL/SQL a l'exterior de les bases de dades

No hem d'oblidar que el llenguatge SQL també s'utilitza en eines externes a les bases de dades *Oracle*, com en *Oracle* Developer (*Oracle* Forms + *Oracle* Reports) i altres.

2.1. Subprogrames

El llenguatge PL/SQL permet definir subprogrames, és a dir, blocs PL/SQL, amb nom, que poden rebre paràmetres i ser directament invocats.

Com en la majoria de llenguatges, hi ha dos tipus de subprogrames: accions (`PROCEDURE`) i funcions (`FUNCTION`). Ambdós s'utilitzen per a efectuar accions i la diferència radica en que les funcions retornen un resultat que és utilitzat directament en expressions.

Oracle emmagatzema els subprogrames a la base de dades, des d'on poden ser invocats per a la seva execució des de multitud d'eines: guions de SQL*Plus, altres blocs PL/SQL, programes d'*Oracle* Developer,... Els subprogrames són un tipus d'objecte emmagatzemat en l'esquema de l'usuari propietari.

Un subprograma PL/SQL té obligatòriament una part declarativa (nom del subprograma, arguments i tipus retornat pel cas de les funcions) i una part executable. És optatiu incloure un apartat pel tractament d'excepcions.

L'edició dels subprogrames es pot efectuar des de diverses eines, entre les que podem considerar *SQL*Plus* (textual) i *SQL Developer* (visual). La forma d'edició és, evidentment, diferent segons l'eina emprada.

2.1.1. Edició de subprogrames des de *SQL*Plus*

La sintaxi per crear i/o modificar un PROCEDURE des de *SQL*Plus* és:

```
create or replace procedure [<esquema>.]<nom_procedure> [(<arguments)] {is|as}
   [<declaracions>]
begin
   <seqüència_d'instruccions>
[exception
   <tractament_d'excepcions>]
end [<nom_procedure>];
```

La sintaxi per crear i/o modificar una FUNCTION des de *SQL*Plus* és:

```
create or replace function [<esquema>.]<nom_function> [(<arguments)] return <tipus_dada>
{is|as}
   [<declaracions>]
begin
   <seqüència_d'instruccions>
[exception
   <tractament_d'excepcions>]
end [<nom_function>];
```

Evidentment, la sentència CREATE permet crear el subprograma i la sentència REPLACE permet modificar-lo amb una nova definició. Podem utilitzar una o altra sentència, però si utilitzem CREATE i ja existeix un subprograma amb el mateix nom, es produirà un error, de la mateixa manera que si utilitzem REPLACE i el subprograma no existeix.

En el moment d'enregistrar un subprograma a la base de dades (és a dir, en executar la sentència CREATE o la sentència REPLACE), el SGBD *Oracle* efectua una compilació per a detectar els possibles errors sintàctics i ens avisa en cas que la compilació hagi finalitzat amb errors. Aquests errors són visualitzables des de *SQL*Plus* pel comandament SHOW ERRORS.

L'eliminació d'un subprograma des de *SQL*Plus* s'efectua amb les sentències:

```
drop procedure [<esquema>.]<nom_procedure>;
drop function [<esquema>.]<nom_function>;
```

La definició dels arguments dels subprogrames ha de seguir la sintaxis:

```
<nom_argument> [in|out|in out] <tipus_dada> [default <valor>], ...
```

Les clàusules in i out permeten el tractament de pas per valor i pas per referència (o per variable) segons mostra la taula 8.

Taula 8. Significat de les partícules `in` i `out` en la definició dels arguments en subprogrames

Partícula	Descripció
`in`	El valor de l'argument prové del procés que crida el subprograma. En el procés que efectua la crida pot ser una constant o una variable, i en aquest segon cas, el seu valor no es veurà mai modificat pels càlculs que pugui sofrir l'argument dins el subprograma. Si en la definició de l'argument no s'indica la clàusula `in |out |in out`, llavors el valor per defecte és `in`.
`in out`	El valor de l'argument prové del procés que crida el subprograma. En el procés que efectua la crida ha de ser una variable i el seu valor és modificat amb el valor que tingui l'argument en finalitzar l'execució del subprograma (si no finalitza amb error).
`out`	L'argument no rep cap valor inicial, però el procés que crida el subprograma facilita una variable on recollirà el valor que l'argument tingui en finalitzar l'execució del subprograma (si no finalitza amb errror).

En `<tipus_dada>` es pot indicar qualsevol tipus de dada suportat per PL/SQL. No s'ha d'especificar, però, longitud ni precisió ni escala. Per exemple, `VARCHAR2(10)` no és vàlid i cal indicar `VARCHAR2`.

La clàusula `default` permet assignar un valor per defecte a les variables `in`. En efectuar una crida de subprograma que tingui varis arguments `in` amb valors per defecte, passant paràmetres per alguns d'ells i deixant-ne d'altres sense paràmetres, pot ser necessari indicar explícitament a quins arguments corresponen els paràmetres passats. Vegem-ho a partir de la declaració següent:

```
... XX (p1 IN integer default 10, p2 IN integer default 20) ...
```

Si es vol:

- prendre els valors per defecte, caldrà invocar el subprograma fent `XX`;
- passar el valor 30 com a primer paràmetre i prendre com a segon paràmetre el valor per defectre, calrà fer la invocació `XX(30)`;
- passar el valor 30 com a segon paràmetre i prendre com a primer paràmetre el valor per defecte, caldrà fer la invocació `XX(p2=>30);`

Com en la majoria de llenguatges que permeten pas de paràmetres per referència, una funció pot ser reescrita com un procediment que conté un argument `out` per recollir el resultat de la funció. En tal cas, però, el procediment no pot ser utilitzat directament en una expressió, sinó que es recull el resultat en el paràmetre i aquest és utilitzat en l'expressió. D'igual forma, un procediment amb múltiples arguments `out` pot ser reescrit com una funció que retorna un resultat de tipus `record`.

Les funcions han de retornar obligatòriament un valor del tipus especificat utilitzant, com en la majoria de llenguatges, la sentència `RETURN`, amb la sintàxis:

```
return <valor>;
```

Exemple de creació de funció des de SQL*Plus

Es demana una funció que esbrini, a l'esquema *empresa*, si existeix un departament amb un determinat codi. En cas afirmatiu, cal recollir les dades (nom i localitat) del departament.

La solució passa per executar directament des de *SQL*Plus* la sentència de creació de la funció, però és millor incorporar-la en un guió i executar el guió, de manera que si hem de fer alguna modificació, ja disposarem del guió amb el codi de la sentència.

Trobareu el fitxer `u5n2p01.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p01.sql (guió)
   Descripció: Creació de funció "depart" (esquema empresa) per esbrinar si existeix un
               departament amb un determinat codi. En cas afirmatiu, cal recollir les dades (nom
               i localitat) del departament.
   Autor: Isidre Guixà
*/
create or replace function depart (depcodi in dept.dept_no%type, depnom out dept.dnom%type,
                                   deploc out dept.loc%type) return number is
/* Retorna: 0 – S'ha trobat el departament. La informació resideix en els arguments out
            1 – Error: No hi ha cap departament amb el codi introduït
            Altres – Error Oracle no previst en el disseny d'aquesta funció
*/
begin
   depnom:=''; deploc:='';
   select dnom, loc into depnom, deploc
   from dept
   where dept_no=depcodi;
   return 0;
exception
   when no_data_found then
      return 1;
   when others then
      return SQLCODE;
end depart;
/
```

Fixem-nos que no hem controlat l'excepció `too_many_rows` doncs a la taula `DEPT` el codi de departament és identificador i, per tant, mai es podrà donar aquesta excepció.

Vegem, a continuació, l'execució del guió que provoca la creació de la funció (o substitució si ja hi era) a la base de dades.

No podem confondre l'execució d'un guió que inclou la creació d'un subprograma amb l'execució del subprograma.

```
SQL> @u5n2p01
Funció creada.

SQL> show errors
Cap error.
```

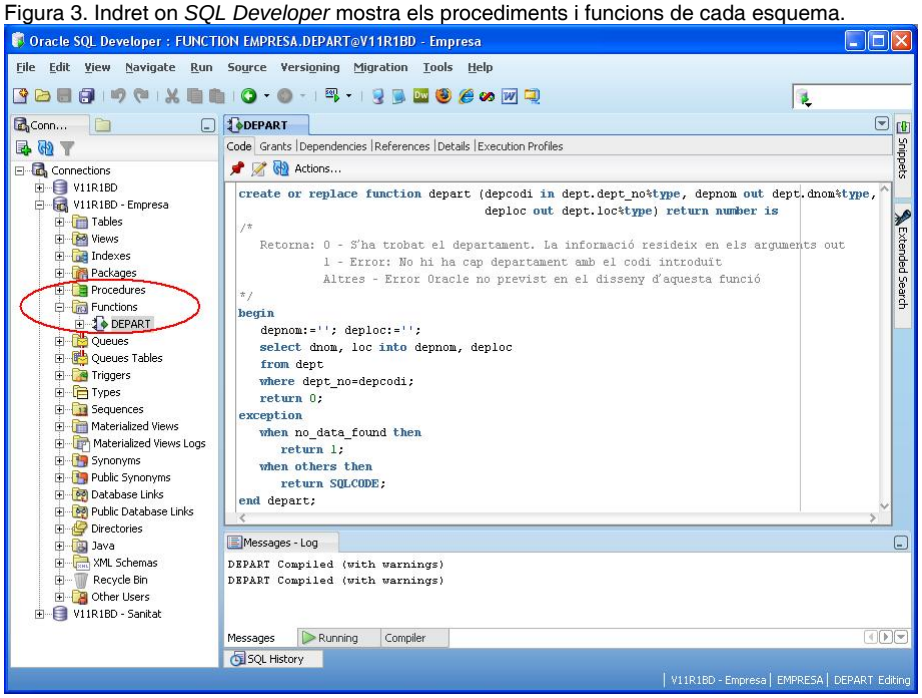
El missatge `Funció creada` ja ens diu que no hi ha cap error (doncs en cas d'algun error de compiulació el missatge hagués estat `Avís: Funció creada, però té errors de compilació`). Tot i així, si volem podem utiltizar el comandament `SHOW ERRORS` per a veure el llistat d'errors.

2.1.2. Edició de subprogrames des de *SQL Developer*

L'edició de subprogrames (creació i/o modificació) des de *SQL Developer*, s'efectua, via interfície visual, en les subcarpetes *Procedures* i *Functions* de l'esquema de l'usuari que hagi de ser el propietari dels subprogrames, com es pot aprociar a la figura 3.

A la figura 3 es veu, dins la carpeta Functions, la funció `DEPART`, i fent un clic damunt ella, se'ns visualitza el seu codi a la part dreta de la pantalla. Què podem fer des de *SQL Developer*? Doncs un munt d'accions:

Figura 3. Indret on *SQL Developer* mostra els procediments i funcions de cada esquema.



- Sobre cada funció i/o procediment disposem de diverses possibilitats. Cal situar-se damunt el subprograma i prémer el botó secundari del ratolí, fet que provocarà l'aparició d'un menú contextual amb diverses opcions interessants:

  – *Edit*, corresponent a l'edició del subprograma.
  – *Drop*, corresponent a l'eliminació del subprograma.
  – *Grant*, corresponent a la concessió de privilegis d'execució del subprograma, a altres usuaris de la base de dades.
  – *Revoke*, corresponent a la revocació de privilegis d'execució del subprograma als usuaris que el tenien concedit.
  – *Compile*, que efectua la compilació del subprograma, els missatges de la qual es poden observar en el panell inferior de la part dreta de la pantalla.
  – *Run*, per a executar el subprograma, emplenant els paràmetres corresponents als arguments `in` amb els valors que corresponguin i així poder efectuar la verificació de la correcta execució del subprograma sota diferents condicions.

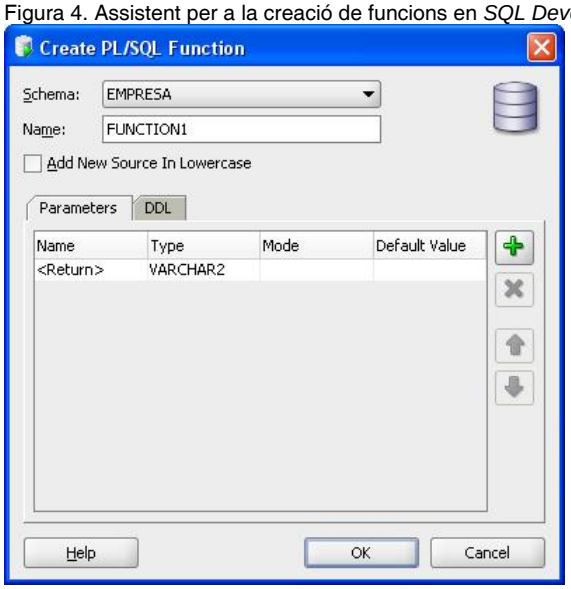
- Podem crear noves funcions i/o procediments tot situant-nos damunt la carpeta *Functions* i/o *Procedures* i prement el botó secundari del ratolí, fet que provocarà l'aparició d'un menú contextual on la primera opció és *New Function* o *New Procedure*.

  En sol·licitar la creació d'una nova funció, ens apareix un assistent, figura 4, que ens permet:

  – Assignar nom a la funció.

- Indicar els paràmetres de la funció (pestanya *Parameters*)
- Indicar el codi de la funció (pestanya *DDL*)

Figura 4. Assistent per a la creació de funcions en *SQL Developer*



2.1.3. Crides a subprogrames

Una vegada el procediment o funció està emmagatzemat a la base de dades, es pot invocar des d'una gran varietat d'entorns, passant els paràmetres necessaris. La taula 9 mostra com efectuar la crida en els entorns/eines en que nosaltres ens movem.

La taula 9 incorpora, en la darrera fila, la crida de subprogrames des de precompiladors. En el nucli d'activitat "SQL hostatjat en C" es veu com utilitzar *SQL* des de llenguatges de tercera generació via utilització de precompiladors.

Taula 9. Possibles formes d'invocar subprogrames PL/SQL des de eines i entorns.

Entorn	Sintaxis	Arguments
*SQL*Plus*	`execute :<variable>:=<nom_function>(<paràmetres>)` `execute <nom_procedure>(<paràmetres>)`	En qualsevol cas, variables d'enllaç d'*SQL*Plus* i també, pels arguments `in`, variables de substitució.
Bloc PL/SQL	`<nom_subprograma> (<paràmetres>)`	Variables locals PL/SQL
Subprogrames	`<nom_subprograma> (<paràmetres>)`	Variables locals PL/SQL o variables globals del paquet
Precompiladors	`Ordre EXEC SQL execute`	Variables *host* o variables locals PL/SQL

Exemple de com executar una funció des de diferents entorns

Considerem la següent funció a l'esquema *empresa*:

```
function depart (depcodi in dept.dept_no%type, depnom out dept.dnom%type,
                 deploc out dept.loc%type) return number is
/* Retorna: 0 – S'ha trobat el departament. La informació resideix en els arguments out
            1 – Error: No hi ha cap departament amb el codi introduït
            Altres – Error Oracle no previst en el disseny d'aquesta funció
*/
```

Vegem com executar-la des de diferents entorns:

a) Des de *SQL*Plus*:

Necessitarem variables d'enllaç per a poder recollir el resultat de la funció i per a poder veure els valors retornats pels paràmetres de tipus `out`.

```
SQL> var depnom varchar2(14)
SQL> var deploc varchar2(14)
SQL> var result number
SQL> execute :result := depart(20, :depnom, :deploc)
SQL> column
SQL> select :result, :depnom, :deploc from dual;

:RESULTAT :DEPNOM                       :DEPLOC
---------- ---------------------------- ----------------------------
         0 INVESTIGACIÓ                 MADRID
```

Si es vol, es pot confeccionar un guió que inclogui totes aquestes instruccions i, fins i tot, demani a l'usuari el vlaor del codi de departament a cercar: Tindriem un guió com:

```
/* Programa: u5n2p02.sql (guió)
   Descripció: Guió que efectua la crida a la funció "depart" que ha d'existir a la base de dades
   Autor: Isidre Guixà
*/
var depnom varchar2(14)
var deploc varchar2(14)
var resultat number
accept departament prompt "Introdueixi codi de departament: "
set verify off
execute :resultat := depart(&departament, :depnom, :deploc)
column :resultat heading 'Resultat'
column :depnom heading 'Nom' format 'A14'
column :deploc heading 'Localitat' format 'A14'
select :resultat, :depnom, :deploc from dual;
undefine departament
clear columns
```

Observem l'execució per una consola SQL*Plus quan l'usuari introdueix un departament existent a la taula DEPT:

```
SQL> @u5n2p02
Introdueixi codi de departament: 20

El procediment PL/SQL ha finalitzat correctament.

  Resultat Nom            Localitat
---------- -------------- --------------
         0 INVESTIGACIÓ   MADRID
```

Observem l'execució per una consola SQL*Plus quan l'usuari introdueix un departament inexistent a la taula DEPT:

```
SQL> @u5n2p02
Introdueixi codi de departament: 25

El procediment PL/SQL ha finalitzat correctament.

  Resultat Nom            Localitat
---------- -------------- --------------
         1
```

Trobareu el fitxer `u5n2p02.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

b) Des de blocs *PL/SQL*.

Necessitem crear un bloc PL/SQL des d'on efectuar la crida a la funció `depart`. Observem que si declarem les variables dins el bloc PL/SQL, llavors no són variables d'enllaç (com en el cas anterior) i, per tant, en efectuar la crida a la funció no s'ha d'incloure el prefix ":".

Trobareu el fitxer `u5n2p03.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p03.sql (guió)
   Descripció: Guió que efectua la crida a la funció "depart" que ha d'existir a la base de dades
               des d'un bloc de codi PL/SQL
   Autor: Isidre Guixà
*/
set serveroutput on
accept departament prompt "Introdueixi codi de departament: "
set verify off
declare
   resultat number;
   depnom dept.dnom%type;
   deploc dept.loc%type;
begin
   resultat := depart(&departament, depnom, deploc);
   if resultat=0 then
      dbms_output.put_line('S''ha trobat un departament de codi '||&departament);
```

```
      dbms_output.put_line('Nom.........: '||depnom);
      dbms_output.put_line('Localitat....: '||deploc);
   else
      dbms_output.put_line('L''execució de la funció ha retornat el codi d''error '||resultat);
   end if;
end;
/
undefine departament
```

Observem l'execució per una consola SQL*Plus quan l'usuari introdueix un departament existent a la taula DEPT:

```
SQL> @u5n2p03
Introdueixi codi de departament: 20
S'ha trobat un departament de codi 20
Nom.........: INVESTIGACIÓ
Localitat....: MADRID

El procediment PL/SQL ha finalitzat correctament.
```

Observem l'execució per una consola SQL*Plus quan l'usuari introdueix un departament inexistent a la taula DEPT:

```
SQL> @u5n2p03
Introdueixi codi de departament: 25
L'execució de la funció ha retornat el codi d'error 1

El procediment PL/SQL ha finalitzat correctament.
```

c) Des d'altres subprogrames.

Necessitem crear un subprograma des d'on efectuar la crida a la funció `depart`. Tant es podria fer amb un `procedure` com amb una `function`. Fem-ho amb un `procedure`.

Procedim, en primer generar un guió que contingui el codi d'un procediment que efectuï la crida a la funció `depart`:

> Trobareu el fitxer `u5n2p04.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p04.sql (guió)
   Descripció: Creació del procedure "crida" que efectua la crida a la funció "depart".
   Autor: Isidre Guixà
*/
create or replace procedure crida (depcodi IN dept.dept_no%type) is
   depnom dept.dnom%type;
   deploc dept.loc%type;
   resultat number;
begin
   resultat := depart(depcodi, depnom, deploc);
   if resultat=0 then
      dbms_output.put_line('S''ha trobat un departament de codi '||depcodi);
      dbms_output.put_line('Nom.........: '||depnom);
      dbms_output.put_line('Localitat....: '||deploc);
   else
      dbms_output.put_line('L''execució de la funció ha retornat el codi d''error '||resultat);
   end if;
end;
/
```

L'execució d'aquest guió provoca la creació del subprograma `crida` a la base de dades:

```
SQL> @u5n2p04

Procediment creat.
```

Una vegada tenim el `procedure` creat, per comprovar que la seva execució fa una crida correcta de la funció `depart`, hem de procedir a executar-lo des d'un guió que prepari adequadament l'entorn d'execució:

> Trobareu el fitxer `u5n2p05.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p05.sql (guió)
   Descripció: Guió que efectua la crida al procedure "crida" que ha d'existir a la base de dades
   Autor: Isidre Guixà
*/
set serveroutput on
accept departament prompt "Introdueixi codi de departament: "
set verify off
execute crida(&departament);
undefine departament
```

Observem l'execució per una consola SQL*Plus quan l'usuari introdueix un departament existent a la taula DEPT:

```
SQL> @u5n2p05
Introdueixi codi de departament: 20
S'ha trobat un departament de codi 20
Nom.........: INVESTIGACIÓ
Localitat....: MADRID

El procediment PL/SQL ha finalitzat correctament.
```

Observem l'execució per una consola SQL*Plus quan l'usuari introdueix un departament inexistent a la taula DEPT:

```
SQL> @u5n2p05
Introdueixi codi de departament: 25
L'execució de la funció ha retornat el codi d'error 1

El procediment PL/SQL ha finalitzat correctament.
```

2.1.4. Col·leccions i registres en arguments de subprogrames

Els arguments dels subprogrames poden ser qualsevol tipus de dada admès en PL/SQL i, per tant, es pot declarar arguments que siguin col·leccions o registres definits per l'usuari.

Per a la declaració d'un argument de tipus col·lecció o registre cal que el tipus estigui prèviament declarat i això es pot fer de diverses maneres:

1) Si es tracta d'un subprograma declarat localment dins un bloc PL/SQL, el tipus es declara prèviament a la mateixa zona `declare`.

2) Si es tracta d'un subprograma creat i emmagatzemat a la base de dades, cal que el tipus sigui declarat en algun paquet i que s'hi tingui accés.

Exemple d'utilització de col·leccions i tipus en arguments de subprogrames

El següent guió conté un bloc PL/SQL que defineix un parell de subprogrames interns que tenen com argument una taula imbricada i, per a tal motiu, cal haver declarat prèviament el tipus corresponent a la taula.

Trobareu el fitxer `u5n2p06.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p06.sql (guió)
   Descripció: Exemple d'utilització de col·leccions i registres en arguments de subprogrames
   Autor: Isidre Guixà
*/
declare
   type tCadena is table of varchar2(50);
   vCadena tCadena;
   procedure omplir_taula (x out tCadena) is
   begin
     x:=tCadena('Cad1','Cad2','Cad3');
   end;
   procedure visualitzar_taula (x in tCadena) is
     n number;
     i number;
   begin
     n:=x.count;
     for i in 1..n loop
       dbms_output.put_line(x(i));
     end loop;
   end;
begin
```

```
   omplir_taula (vCadena);
   visualitzar_taula (vCadena);
end;
/
```

La seva execució en una consola *SQL*Plus* provoca:

```
SQL> @u5n2p06
Cad1
Cad2
Cad3

El procediment PL/SQL ha finalitzat correctament.
```

2.1.5. Documentació de subprogrames

En primer lloc és molt important tenir el bon hàbit de documentar, en el mateix codi dels subprogrames:

- quina és la utilitat del subprograma;
- quins són els arguments, de quin tipus (`in` / `out` / `in out`) i llur significat;
- significat del valor retornat, en el cas de funcions;

En segon lloc, en treballar amb subprogrames emmagatzemats a la base de dades, és important disposar d'eines que permetin obtenir informació referent a subprogrames.

L'eina *SQL Developer* és una eina visual que permet veure les diferents funcions i procediments emmagatzemats per un usuari en el seu esquema i en permet la consulta, la modificació i l'eliminació. Per tant, una de les formes més simples de consultar el codi d'un subprograma és l'eina *SQL Developer*.

A vegades, però, no disposarem d'aquesta eina, i ens pot ser molt útil conèixer com accedir, des de *SQL*Plus*, als subprogrames (declaració i codi). Per aconseguir-ho disposem de diferents mecanismes:

a) Vistes `USER_OBJECTS` i `ALL_OBJECTS` proporcionades per *Oracle*.

Oracle facilita les vistes `USER_OBJECTS` i `ALL_OBJECTS` per visualitzar els objectes (taules, vistes, índexs, subprogrames,...) de la base de dades dels que l'usuari és propietari (`USER_OBJECTES`) i als quals té accés (`ALL_OBJECTS`).

Exemple d'utilització de la vista `USER_OBJECTS`

Es demana dissenyar una instrucció SQL que proporcini tots els subprogrames de l'esquema en el que estem connectats.

La informació que es demana cal cercar-la a la vista `USER_OBJECTS`. Seria convenient, però, abans d'efectuar una `SELECT` de totes les columnes, conèixer quines columnes té aquesta vista per a poder decidir sobre quines ens interessa consultar. Així doncs, fem:

```
SQL> desc user_objects;
 Nom                           Nul?     Tipus
 ----------------------------- -------- -------------------------
 OBJECT_NAME                            VARCHAR2(128)
 SUBOBJECT_NAME                         VARCHAR2(30)
 OBJECT_ID                              NUMBER
 DATA_OBJECT_ID                         NUMBER
 OBJECT_TYPE                            VARCHAR2(19)
 CREATED                                DATE
 LAST_DDL_TIME                          DATE
 TIMESTAMP                              VARCHAR2(19)
 STATUS                                 VARCHAR2(7)
 TEMPORARY                              VARCHAR2(1)
 GENERATED                              VARCHAR2(1)
 SECONDARY                              VARCHAR2(1)
 NAMESPACE                              NUMBER
 EDITION_NAME                           VARCHAR2(30)
```

De l'anàlisi d'aquesta informació sembla que ens cal utilitzar les columnes `object_name` (nom de l'objecte) i `object_type` (tipus d'objecte). No sabem, però, com estan codificats els diferents tipus d'objectes (taules, vistes, sinònims, funcions, procediments,...) per la qual cosa pot ser interessant descobrir-ho:

```
SQL> select distinct object_type from user_objects;

OBJECT_TYPE
-------------------
SEQUENCE
PROCEDURE
FUNCTION
TABLE
INDEX
VIEW

6 files seleccionades.
```

Hem de tenir en compte que el resultat d'aquesta sentència `SELECT` depèn dels objectes definits dins el nsotre esquema. És a dir, si apareixen els valors `FUNCTION` i `PROCEDURE` és per què hi tenim definits alguns objectes d'aquesta tipologia.

Ara que ja sabem com s'anomenen els diferents tipus d'objectes, procedim a executar la sentència que ens informarà sobre els subprogrames que tenim en el nostre esquema:

```
SQL> select object_name, object_type from user_objects
  2  where object_type='PROCEDURE' or object_type='FUNCTION';

OBJECT_NAME    OBJECT_TYPE
-------------- -------------------
CRIDA          PROCEDURE
DEPART         FUNCTION
```

El resultat dependrà, evidentment, dels subprogrames emmagatzemats a l'esquema.

b) El comandament `DESC` proporcionat per *SQL*Plus*.

El comandament `DESC` de *SQL*Plus* permet veure la informació corresponent als arguments d'un subprograma i, pel cas de les funcions, el tipus de valor resultant. Cal emprar-lo amb la sintàxis:

```
desc <nom_subprograma>
```

Exemple del comandament `DESC` per obtenir informació de subprogrames

Per obtenir informació sobre la funció `depart` i el procediment `crida`, podem fer:

```
SQL> desc depart

FUNCTION depart RETURNS NUMBER
 Nom d'argument                 Tipus                  E/S per omissió?
 ------------------------------ ---------------------- ---------------
 DEPCODI                        NUMBER(2)              IN
```

```
 DEPNOM                         VARCHAR2(14)           OUT
 DEPLOC                         VARCHAR2(14)           OUT

SQL> desc crida

PROCEDURE crida
 Nom d'argument                 Tipus                  E/S per omissió?
 ------------------------------ ---------------------- ---------------
 DEPCODI                        NUMBER(2)              IN
```

c) Vistes USER_SOURCE i ALL_SOURCE proporcionades per *Oracle*.

La vista USER_SOURCE permet visualitzar el codi font dels subprogrames dels que l'usuari és propietari. La vista ALL_SOURCE permet visualitzar el codi font dels subprogrames als quals un usuari té accés.

Exemple d'utilització de la vista USER_SOURCE

Es demana dissenyar una instrucció SQL per a visualitzar el codi de la funció depart a l'esquema *empresa*, on estem connectats.

La informació que es demana cal cercar-la a la vista USER_SOURCE. Seria convenient, però, abans d'efectuar una SELECT de totes les columnes, conèixer quines columnes té aquesta vista per a poder decidir sobre quines ens interessa consultar. Així doncs, fem:

```
SQL> desc user_source;
 Nom                    Nul?     Tipus
 ---------------------- -------- --------------------------
 NAME                            VARCHAR2(30)
 TYPE                            VARCHAR2(12)
 LINE                            NUMBER
 TEXT                            VARCHAR2(4000)
```

De l'anàlisi d'aquesta informació deduïm les columnes que hem de demanar a la sentència SELECT que ens ha de donar el codi de la funció depart:

```
SQL> select line, substr(text,1,100) from user_source where upper(name)='DEPART';

LINE SUBSTR(TEXT,1,100)
---- ----------------------------------------------------------------------------------------
   1 function depart (depcodi in dept.dept_no%type, depnom out dept.dnom%type,
   2                  deploc out dept.loc%type) return number is
   3 /*
   4    Retorna: 0 – S'ha trobat el departament. La informació resideix en els arguments out
   5             1 – Error: No hi ha cap departament amb el codi introduït
   6             Altres – Error Oracle no previst en el disseny d'aquesta funció
   7 */
   8 begin
   9    depnom:=''; deploc:='';
  10    select dnom, loc into depnom, deploc
  11    from dept
  12    where dept_no=depcodi;
  13    return 0;
  14 exception
  15    when no_data_found then
  16       return 1;
  17    when others then
  18       return SQLCODE;
  19 end depart;

19 files seleccionades.
```

d) Vistes USER_ERRORS i USER_OBJECT_SIZE proporcionades per *Oracle*.

Recordem que el comandament SHOW ERRORS permet visualitzar la informació corresponent als errors de la darrera compilació. Aquests errors resideixen a la vista USER_ERRORS.

Oracle també ens facilita la vista USER_OBJECT_SIZE per consultar la grandària en bytes de codi font i compilats de subprogrames.

2.1.6. Depuració de subprogrames

Oracle facilita el paquet estàndard DBMS_OUTPUT que inclou un seguit de funcions i procediments per facilitar la depuració de subprogrames. Ja coneixem els procediments PUT, NEW_LINE i PUT_LINE que proporciona aquest paquet per a mostrar informació des de blocs PL/SQL.

El paquet DBMS_OUTPUT permet que els subprogrames emmagatzemats (independents o formant part d'altres paquets) i els disparadors (que són un tipus especial de subprogrames) enviïn missatges a un *buffer*, d'on poden ser llegits per altres subprogrames i disparadors.

La utilització d'aquest paquet és especialment útil en la depuració de programes, ja que permet generar missatges durant l'execució del codi i així possibilita fer-ne el seguiment de l'execució.

La taula 10 mostra els procediments que formen el paquet.

Taula 10. Procediments incorporats en el paquet DBMS_OUTPUT.

Procediment	Descripció	Arguments
ENABLE	Activa la utilització del paquet. Les crides als procediments PUT, PUT_LINE, NEW_LINE, GET_LINE i GET_LINES són ignorades si el paquet és desactivat. L'activació en *SQL*Plus* del paràmetre SERVEROUTPUT és equivalent a l'execució d'aquest procediment.	<grandària_buffer> in INTEGER default 20000
DISABLE	Desactiva la utilització del paquet.	---
PUT	Afegeix text a la línia actual.	<valor> {NUMBER | VARCHAR2 | DATE}
NEW_LINE	Marca un final de línia	---
PUT_LINE	Combinació de PUT i NEW_LINE	<valor> {NUMBER | VARCHAR2 | DATE}
GET_LINE	Intenta recuperar una línia del *buffer*, la qual deixa en el primer paràmetre. El segon paràmetre permet saber si s'ha recuperat (valor 0) o no (valor 1) informació.	<variable> out VARCHAR2 <estat> out INTEGER
GET_LINES	Intenta recuperar una taula de línies del *buffer*. El segon paràmetre indica el número de línies a recuperar i en finalitzar l'execució conté el número real de línies recuperades.	<línies> out <taula de VARCHAR2(255)> <número_de_línies> in out INTEGER

Els missatges generats per PUT, PUT_LINE i NEW_LINE es mantenen en el *buffer* fins que són llegits o finalitza l'execució del programa principal. En finalitzar l'execució d'un programa des de *SQL*Plus*, es visualitzen els missatges existents en el *buffer* si la variable d'entorn SERVEROUTPUT de *SQL*Plus* està activada. Recordem que per activar-la:

```
SQL> set serverouput on
```

Després d'una crida als procediments GET_LINE i GET_LINES, totes les línies no recuperades abans de la següent crida als procediments PUT, PUT_LINE o NEW_LINE són descartades per evitar confusions entre elles i el següent missatge.

Trobareu el fitxer u5n2p07.sql en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Exemple d'utilització del paquet DBMS_OUTPUT

Considerem la seqüencia d'accions incorporades en el següent guió:

```
/* Programa: u5n2p07.sql (guió)
   Descripció: Proves de funcionament dels procediments del paquet dbms_output.
   Autor: Isidre Guixà
*/
set serveroutput on
var x01 varchar2(12)
var x02 varchar2(12)
var e01 number
var e02 number
column :x01 format 'A12'
column :x02 format 'A12'
column :e01 format 9
column :e02 format 9

create or replace procedure dbms_prova_intern (vx01 out varchar2, vx02 out varchar2,
                ve01 out number, ve02 out number) is
begin
   dbms_output.get_line(vx01,ve01); --3
   dbms_output.put_line('Missatge 3'); --4
   dbms_output.get_line(vx02,ve02); --5
end;
/

create or replace procedure dbms_prova (vx01 out varchar2, vx02 out varchar2,
                ve01 out number, ve02 out number) is
begin
   dbms_output.put_line('Missatge 1'); --1
   dbms_output.put_line('Missatge 2'); --2
   dbms_prova_intern(vx01,vx02,ve01,ve02);
   dbms_output.put_line('Missatge Final'); --6
end;
/

execute dbms_prova(:x01,:x02,:e01,:e02);
select :x01, :e01, :x02, :e02 from dual;
```

Fixem-nos que aquest guió crea dos procediments a la base de dades, el segon dels quals efectua una crida al primer. Els hem posat en l'ordre adequat per què a la primera execució del guió no hi hagi cap problema, ja que la compilació del segon comprova l'existència del primer.

L'execució d'aquest guió des d'una consola SQL*Plus provoca la sortida:

```
SQL> @u5n2p07
Procediment creat.

Procediment creat.

Missatge Final

El procediment PL/SQL ha finalitzat correctament.

:X01         :E01 :X02         :E02
------------ ---- ------------ ----
Missatge 1      0 Missatge 3      0
```

Reflexionem sobre els missatges que han aparegut per la consola i dels continguts finals de les variables d'enllaç.

Respecte els missatges que han aparegut per la consola, observem que només ha aparegut el missatge de la instrucció `--6`. Els anteriors no han aparegut doncs es sentències `get_line` anteriors els han eliminat del *buffer*.

Les instruccions `--1` i `--2` envien els corresponents missatges al *buffer*. La instrucció `--3` recupera el primer missatge del *buffer* on ja només queda el missatge enviat per la instrucció `--2`. La instrucció `--4` provoca l'esborrat del contingut del *buffer*, motiu pel que es perd el missatge enviat per la instrucció `--2`. La instrucció `--5` recupera el missatge enviat per la instrucció `--4`. Fixem-nos que el contingut del buffer s'ha compartit entre els diferents procediemnts.

Si comentem les instruccions `--3` i `--6` i tornem a executar el guió, obtenim:

```
SQL> @u5n2p07

Procediment creat.

Procediment creat.

Missatge 2
Missatge 3

El procediment PL/SQL ha finalitzat correctament.

:X01         :E01 :X02         :E02
------------ ---- ------------ ----
                  Missatge 1      0
```

En aquest cas, les instruccions `--1`, `--2` i `--4` envien missatges al *buffer*. La instrucció `--5` recull el primer missatge del *buffer* i, en finalitzar el procés, com que la variable d'entorn `SERVEROUTPUT` està activada, *SQL*Plus* visualitza la resta de missatges del *buffer*. Si no haguessim comentat la instrucció `--6`, en tractar-se d'una instrucció `PUT_LINE`, l'anterior `GET_LINE` hagués provocat l'eliminació de tots els missatges del buffer, i per tant no hagués aparegut `Missatge 2` i `Missatge 3` i en canvi hagués aparegut `Missatge Final`.

2.1.7. Beneficis d'utilització

La taula 11 ens mostra una síntesi dels beneficis que aporta la utilització de subprogrames.

Taula 11. Beneficis de la utilització de subprogrames

Nivell	Beneficis
Seguretat	S'executen sota el domini de seguretat del propietari del procediment Proporcionen a usuaris no privilegiats accés indirecte i controlat sobre els objectes de la base de dades
Integritat	Defineixen les operacions permeses sobre les dades Asseguren que les accions relacionades s'executin juntament
Rendiment	Redueixen el número de crides a la base de dades Redueixen el tràfic de la xarxa Porten incorporat un pre-anàlisi (compilació) de les sentències PL/SQL
Memòria	Requereixen una única còpia del codi per diferents usuaris
Productivitat	Eviten codi redundant per procediments comuns en diferents aplicacions
Manteniment	Es canvien en tot el sistema amb una única actualització

2.2. Paquets

Un paquet o PACKAGE és un objecte de la base de dades que inclou un grup de construccions del tipus:

Procediments	Definicions de cursors	Variables
Funcions	Definicions d'excepcions	Constants

Les variables i cursors d'un paquet conserven el seu estat (les variables retenen el seu valor i els cursors el seu context i posició) durant tota la sessió de l'usuari.

2.2.1. Edició i documentació de paquets

Un paquet es composa de dues parts ben diferenciades: especificació i cos. La seva edició es pot efectuar des de *SQL*Plus* i des d'eines visuals com *SQL Developer*.

- L'**especificació** és la secció de declaració específica del paquet i inclou les declaracions de les construccions públiques, és a dir, aquelles que estan disponibles per a tots els usuaris autoritzats.

  Pels subprogrames únicament s'hi facilita la capçalera, és a dir, nom, arguments i tipus resultant (si es tracta de funcions).

  La sintaxis per l'edició de l'especificació d'un paquet des de *SQL*Plus* és:

```
create or replace package <nom_paquet> as|is
   <capçaleres_subprogrames_públics>|<variables_públiques>
end [<nom_paquet>];
```

- El **cos** és la secció on es declaren tots els subprogrames, públics i privats.

  En primer lloc s'ha de declarar els subprogrames privats i posteriorment els subprogrames públics. *Oracle* distingeix els subprogrames públics per que la seva capçalera és inclosa en l'especificació del paquet.

  Els subprogrames privats únicament poden ser utilitzats dins el cos del paquet.

  La sintaxis per l'edició del cos d'un paquet des de SQL*Plus és:

```
create or replace package body <nom_paquet> as|is
   <definició_subprogrames_privats>
```

```
   <definició_subprogrames_públics>
end [<nom_paquet>];
```

El cos d'un paquet pot ser canviat sense afectar la corresponent especificació.

Exemple de disseny de paquet

Es demana, a l'esquema *empresa*, dissnyar un paquet per poder dur a terme les quatre operacions fonamentals a nivell de la taula COMANDA: consulta, inserció, modificació i eliminació d'una comanda.

A continuació es presenta un guió que conté les instruccions de creació del paquet.

Trobareu el fitxer `u5n2p08.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p08.sql (guió)
   Descripció: Disseny del paquet "p_comanda" a l'esquema "empresa"
   Autor: Isidre Guixà
*/
create or replace package p_comanda as
   /* En aquest paquet pretenem tractar diferents operacions sobre la taula "comanda" */

   function consultar (com in out comanda%rowtype) return number;
   /* Funció per consultar una comanda
      Arguments : com -> Registre que en la crida ha de contenir el numero de comanda en
                         el corresponent camp
                         En retornar conté, si s'ha trobat, la resta d'informació en els
                         altres camps
      Retorna 0 : OK
              1 : No existeix la comanda
              altres : Error SQLCODE
   */

   function esborrar (num in comanda.com_num%type, cascada in boolean) return number;
   /* Funció per eliminar una comanda
      Arguments : num -> Número de comanda a eliminar
                  cascada -> Cert : Eliminació en cascada (línies de detall)
                             Fals : No eliminació en cascada
      Retorna 0 : OK
              1 : No existeix la comanda
              2 : La comanda té línies i s'ha demanat eliminació sense cascada
              altres : Error SQLCODE
   */

   function crear (com in comanda%rowtype) return number;
   /* Funció per crear una nova comanda
      Arguments: com -> Registre que ha de contenir tota la informació de la capçalera
                        de comanda
      Retorna 0 : OK
              1 : Ja existia una comanda amb el número indicat
              2 : No existeix el client de la comanda (integritat referencial)
              altres : Error SQLCODE
   */

   function modificar (com in out comanda%rowtype) return number;
   /* Funció per modificar una comanda existent
      Arguments: com -> Registre que ha de contenir tota la informació modificada de la
                        comanda
      Retorna : 0 : OK
                1 : No existeix la comanda amb el número indicat
                2 : No existeix el nou client (integritat referencial)
                altres : Error SQLCODE
   */
end p_comanda;
/
create or replace package body p_comanda as
   function consultar (com in out comanda%rowtype) return number as
   begin
      select * into com from comanda where com_num=com.com_num;
      return 0;
   exception
      when no_data_found then
         return 1;
      when others then
         return SQLCODE;
   end consultar;
```

```
   function esborrar (num in comanda.com_num%type, cascada in boolean) return number as
      error_FK_Detall exception;
      pragma exception_init (error_FK_Detall, -2292);
   begin
      if cascada then
         delete from detall where com_num=num;
      end if;
      delete from comanda where com_num=num;
      if sql%notfound then
         return 1;
      else return 0;
      end if;
   exception
      when error_FK_Detall then
         return 2;
      when others then
         return SQLCODE;
   end esborrar;

   function crear (com in comanda%rowtype) return number as
      error_FK_Client exception;
      pragma exception_init (error_FK_Client, -2291);
   begin
      insert into comanda values (com.com_num, com.com_data, com.com_tipus,
                                  com.client_cod, com.data_tramesa, com.total);
      return 0;
   exception
      when dup_val_on_index then
         return 1;
      when error_FK_Client then
         return 2;
      when others then
         return SQLCODE;
   end crear;

   function modificar (com in out comanda%rowtype) return number as
      error_FK_Client exception;
      pragma exception_init (error_FK_Client, -2291);
   begin
      update comanda set com_data=com.com_data, com_tipus=com.com_tipus,
                         client_cod=com.client_cod, data_tramesa=com.data_tramesa,
                         total=com.total
             where com_num=com.com_num;
      if sql%notfound then
         return 1;
      else return 0;
      end if;
   exception
      when error_FK_Client then
         return 2;
      when others then
         return SQLCODE;
   end modificar;
end p_comanda;
/
```

Comentar que, en referència al tractament d'excepcions, on hem pogut, hem utilitzat les excepcions predefinides d'*Oracle* i on no ha estat possible, hem definit unes excepcions vinculades a errors *Oracle* amb la utilització de `PRAGMA EXCEPTION INIT`. Fixem-nos en que hem associat l'excepció a definir amb els codis d'error -2291 (quan s'intenta assignar un codi de client inexistent a la taula de clients) i l'error -2292 (quan s'intenta eliminar una comanda que té línies de detall). Com hem conegut aquests codis d'error? Doncs provocant-los des de *SQL*Plus* executant manualment les instruccions que els provoquen i veient el codi d'error que retorna *Oracle*.

En l'el·laboració dels paquets cal seguir les bones pràctiques de documentació recomanades pels subprogrames, tenint en compte que en els paquets, a més de subprogrames, hi pot haver altres tipus d'objectes com cursors, variables, tipus de dades,...

Per tal de conèixer els paquets als quals té accés un usuari, podem utilitzar eines visuals com *SQL Developer* o, des de *SQL*Plus*, utilitzar

les vistes i comandaments que ens proporciona *Oracle*, que són els mateixos que els emprats per a obtenir informació dels subprogrames:

- Les vistes `USER_OBJECTS` i `ALL_OBJECTS`, que ens facilitaran el nom dels paquets tenint en compte que són objectes identificats pels tipus `'PACKAGE'` i `'PACKAGE BODY'`.
- El comandament `DESC`, que ens facilitarà informació sobre els arguments.
- Les vistres `USER_SOURCE` i `ALL_SOURCE`, que ens facilitaran el codi dels paquets.

Exemple d'obtenció d'informació sobre els paquets d'un usuari

La sentència per a conèixer els paquets d'un usuari és:

```
SQL> select object_name, object_type from user_objects
  2  where object_type like 'PACKAGE%';

OBJECT_NAME          OBJECT_TYPE
-------------------- -------------------
P_COMANDA            PACKAGE BODY
P_COMANDA            PACKAGE
```

Per a obtenir les declaracions dels objectes continguts en el paquet, utilitzem el comandament `DESC`:

```
SQL> desc p_comanda;
FUNCTION CONSULTAR RETURNS NUMBER
 Nom d'argument                 Tipus                  E/S per omissió?
 ------------------------------ ---------------------- ----------------
 COM                            RECORD                 IN/OUT
   COM_NUM                      NUMBER(4)              IN/OUT
   COM_DATA                     DATE                   IN/OUT
   COM_TIPUS                    VARCHAR2(1)            IN/OUT
   CLIENT_COD                   NUMBER(6)              IN/OUT
   DATA_TRAMESA                 DATE                   IN/OUT
   TOTAL                        NUMBER(8,2)            IN/OUT
FUNCTION CREAR RETURNS NUMBER
...
```

Per accedir al cos del paquet, utilitzem la sentència:

```
SQL> select line, substr(text,1,100) from user_source where upper(name)='P_COMANDA';

LINE SUBSTR(TEXT,1,100)
---- ----------------------------------------------------------------------------------------------------
   1 package body p_comanda as
   2    function consultar (com in out comanda%rowtype) return number as
   3    begin
   4       select * into com from comanda where com_num=com.com_num;
   5       return 0;
   6    exception
   7       when no_data_found then
   8          return 1;
   9       when others then
  10          return SQLCODE;
  11    end consultar;
...
```

2.2.2. Crida als subprogrames dels paquets

La crida dels subprogrames dels paquets és molt simple. Segueix la mateix sintaxis que la crida dels subprogrames independents amb la precaució que cal indicar el nom del paquet davant el nom del

subprograma de manera semblant a quan volem referenciar un camp d'un registre:

```
<nom_paquet>.<nom_subprograma>[(<arguments>)]
```

Exemple de crides a subprogrames de paquets.

El següent guió ens mostra alguns exemples de crides sobre el subprograma ESBORRAR del paquet P_COMANDA.

```
/* Programa: u5n2p09.sql (guió)
   Descripció: Guió que efectua crides al subprograma "esborrra" del paquet "p_comanda"
   Autor: Isidre Guixà
*/
var resultat number
execute :resultat := p_comanda.esborrar(900,false)
print resultat
execute :resultat := p_comanda.esborrar(610,false)
print resultat
execute :resultat := p_comanda.esborrar(610,true)
print resultat
```

La seva execució des d'una consola SQL*Plus ha provocat la sortida:

```
SQL> @u5n2p09

El procediment PL/SQL ha finalitzat correctament.

  RESULTAT
----------
         1

El procediment PL/SQL ha finalitzat correctament.

  RESULTAT
----------
         2

El procediment PL/SQL ha finalitzat correctament.

  RESULTAT
----------
         0
```

Trobareu el fitxer u5n2p09.sql en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

2.2.3. Beneficis d'utilització

La taula 12 ens mostra una síntesi dels beneficis que aporta la utilització de paquets.

Taula 12. Beneficis de la utilització de paquets

Nivell	Beneficis
Organització	Permeten definir mòduls genèrics per àrees funcionals Minimitzen conflictes de noms de subprogrames dins l'organització, ja que possibilita tenir subprogrames amb igual nom en paquets diferents.
Seguretat	Simplifiquen la concessió i revocació de privilegis sobre els subprogrames, ja que es pot utilitzar la sentència GRANT EXECUTE sobre tot el paquet. Permeten distingir entre parts públiques (accessibles per tot usuari autoritzat) i parts privades (no accessibles per cap usuari).
Canvis parcials	Permeten efectuar canvis en el cos d'un subprograma sense canviar l'especificació
Variables globals	Permeten definir variables globals accessibles a tot usuari que estigui autoritzat
Rendiment	Redueixen les E/S físiques a disc (en efectuar-se una crida a un subprograma, el paquet sencer és carregat en memòria)

Nivell	Beneficis
Estat persistent	Retenen el valor de la construcció i de les variables per tota la sessió de treball

2.2.4. Paquets interessants: STANDARD, DBMS_STANDARD, DBMS_SQL.

Oracle subministra molts paquets que aporten diferents funcionalitats. Alguns d'ells s'instal·len de forma automàtica en la instal·lació del SGBD. És molt recomanable efectuar una ullada a la documentació que *Oracle* aporta al respecte.

Degut a les necessitats de depuració de subprogrames, ja coneixem el paquet DBMS_OUTPUT. A continuació presentem altres paquets que ens poden ser de molta utilitat.

Paquet STANDARD

Aquest paquet existeix en qualsevol instal·lació d'*Oracle* que disposi de PL/SQL. Conté, entre altres, les declaracions dels tipus de dades, excepcions i subprogrames utilitzables automàticament des de qualsevol programa PL/SQL. Així, totes les funcions predefinides de PL/SQL són dins aquest paquet.

Paquet DBMS_STANDARD

Aquest paquet subministra facilitats que ajuden a la interacció entre aplicacions i *Oracle*. Ens interessa, especialment, el procediment:

```
procedure raise_application_error (codi_error binary_integer, text_error varchar2)
```

Permet generar, des de qualsevol programa, un error codificat amb qualsevol valor entre -20000 i -20999 i amb un text associat, de manera que les funcions SQLCODE i SQLERRM el tracten com un error d'*Oracle*.

Exemple d'utilització del procediment RAISE_APPLICATION_ERROR

Considerem el guió següent que conté la creació d'un procediment CONSULTA (a l'esquema *empresa*) que utilitza el procediment RAISE_APPLICATION_ERROR:

Trobareu el fitxer u5n2p10.sql en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p10.sql (guió)
   Descripció: Disseny de procediment "consulta" (esquema "empresa") que efectua una
               crida al procediment "raise_application_error"
   Autor: Isidre Guixà
*/
create or replace procedure consulta (depcodi in dept.dept_no%type, depnom out dept.dnom%type,
                                      deploc out dept.loc%type) is
begin
   depnom:=''; deploc:='';
   select dnom, loc into depnom, deploc from dept where dept_no=depcodi;
   raise_application_error (-20000, 'Consulta finalitzada correctament');
exception
```

```
   when no_data_found then
      raise_application_error (-20001, 'No s''ha trobat cap departament');
   when too_many_rows then
      raise_application_error (-20002, 'S''ha trobat més d''un departament');
end consulta;
/
```

Vegem què succeeix davant diverses execucions del procediment des de *SQL*Plus*:

```
SQL> var depnom varchar2(14)
SQL> var deploc varchar2(14)
SQL> execute consulta (90, :depnom, :deploc)
BEGIN consulta (90, :depnom, :deploc); END;
*
ERROR a la l¡nia 1:
ORA-20001: No s'ha trobat cap departament
ORA-06512: a "EMPRESA.CONSULTA", line 8
ORA-06512: a line 1

SQL> execute consulta (10, :depnom, :deploc)
BEGIN consulta (10, :depnom, :deploc); END;
*
ERROR a la l¡nia 1:
ORA-20000: Consulta finalitzada correctament
ORA-06512: a "EMPRESA.CONSULTA", line 5
ORA-06512: a line 1
```

Com podem veure, el procediment `RAISE_APPLICATION_ERROR` genera un error (hagi estat o no error real d'*Oracle*) que es traspassat al nivell des d'on s'ha efectuat la crida.

Paquet `DBMS_SQL`

Aquest paquet permet l'execució, des de PL/SQL de sentències LDD de SQL (les quals no és permès executar des de PL/SQL) i de sentències SQL dinàmiques (construïdes en temps d'execució).

Taula 13. Principals subprogrames del paquet `DBMS_SQL`.

Subprograma	Descripció	Arguments i resultat
`OPEN_CURSOR (funció)`	Intenta crear un cursor `DBMS_SQL` per poder executar sentències SQL de l'àmbit LDD i/o dinàmiques.	`No té arguments` `Retorna el número INTEGER identificador del cursor`

`PARSE (procediment)`
`Versió per sentències SQL fins a 32 KB. Hi ha una altra versió per sentències "llargues"`
	Compila la instrucció SQL continguda en el segon paràmetre i si es tracta d'una sentència LDD l'executa. Prèviament cal haver obert un cursor `DBMS_SQL` al qual es referencia pel primer paràmetre. El tercer paràmetre indica la versió en què *Oracle* ha d'interpretar la sentència (pot necessitar-se quan estant en versió igual o superior a 8 es vulgui obligar a que *Oracle* actuï com en versió 6 o 7)	`<identificador_cursor> in INTEGER`
		`<sentència> in VARCHAR2`
		`<versió_SGBD_Oracle> in INTEGER`
		`0 : Versió 6 (Constant simbòlica: v6)`
		`1 : Versió nativa (Constant simbòlica: native)`
		`2 : Versió 7 (Constant simbòlica: v7)`

`EXECUTE (funció)`	Executa la instrucció SQL dinàmica compilada prèviament amb el procediment `PARSE`. Cal recordar que les instruccions LDD són executades pel procediment `PARSE`.	`<identificador_cursor> in INTEGER` `Retorna el nombre INTEGER de files processades en les instruccions INSERT, UPDATE i DELETE. En la resta d'instruccions pot retornar qualsevol valor que ha de ser ignorat.`
`CLOSE_CURSOR (procediment)`	Tanca el cursor `DBMS_SQL` obert	`<identificador_cursor> in INTEGER`
`FETCH_ROWS (funció)`	Recupera una fila del cursor, que ha de contenir una sentència `SELECT`. Es pot anar executant aquesta funció mentre hi hagi files pendents de ser recuperades.	`<identificador_cursor> in INTEGER` `Retorna el nombre INTEGER de files recuperades pel cursor des de que s'ha executat la sentència.`

Subprograma	Descripció	Arguments i resultat
EXECUTE_AND_FETCH (funció)	És la combinació de l'execució de la funció EXECUTE seguida de la funció FETCH,	<identificador_cursor> in INTEGER Retorna el nombre INTEGER de files recuperades pel cursor des de que s'ha executat la sentència.
DEFINE_COLUMN (procediment)	Defineix una de les columnes que podrà ser accedida pel cursor, que ha de contenir una sentència SELECT. Cal executar aquest procediment per cada columna que vulgui ser accedida. El tercer paràmetre ha de ser una varialble del tipus corresponent, però no té per què ser la variable que després s'ompli.	<identificador_cursor> in INTEGER <posició_de_la_columna_dins_la_SELECT> in INTEGER <variable_del_tipus_adequat> in <tipus_dada>
COLUMN_VALUE (procediment)	Recull la columna indicada de la darrera fila accedida pel cursor, que ha de contenir una sentència SELECT. i deixa el valor en la variable indicada.	<identificador_cursor> in INTEGER <posició_de_la_columna_dins_la_SELECT> in INTEGER <variable_del_tipus_adequat> out <tipus_dada>

Per aconseguir aquesta execució és imprescindible, com a mínim, obrir (funció OPEN_CURSOR) un cursor DBMS_SQL, compilar (procediment PARSE) la instrucció SQL, executar (funció EXECUTE) i tancar el cursor (procediment CLOSE_CURSOR). Aquestes quatre accions són imprescindibles en qualsevol sentència i, a més, en les sentències de recuperació de dades (sentències SELECT) se'n necessiten d'altres per a poder recuperar els valors de les diverses columnes de diverses files. La taula 13 mostra els principals subprogrames a considerar. Però n'hi ha molts més. Podeu trobar informació al respecte en la documentació subministrada per *Oracle*.

Exemple d'utilització del paquet DBMS_SQL per una sentència SELECT d'una única fila

Es demana una funció, d'accés per a tothom, que calculi el nombre de files de qualsevol taula de qualsevol esquema a indicar via paràmetres en la crida.

Està clar que la sentència és simple:

```
select count(*) from <nomEsquema>.<nomTaula>:
```

Però els noms de l'esquema i de a taula s'han de passar per paràmetre i, per tant, estem obligats a utilitzar SQL dinàmic, fet que ens ho permet el paquet DBMS_SQL.

El guió següent procedeix a crear la funció i a concedir el permís d'execució a qualsevol usuari, fet que s'assoleix concedint el privilegi EXECUTE a PUBLIC.

Trobareu el fitxer u5n2p11.sql en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p11.sql (guió)
   Descripció: Exemple d'utilització del paquet DBMS_SQL per a SQL dinàmic.
               Disseny de funció pública que calcula el nombre de files de qualsevol taula de
               qualsevol esquema a què l'usuari tingui accés.
   Autor: Isidre Guixà
*/
create or replace function comptar_files (nomEsquema in varchar2, nomTaula in varchar2)
                  return number is
/* Retorna: >=0 : Nombre de files que té la taula
            -1  : L'esquema o la taula no existeixen per a l'usuari (poden existir però no
                  tenir-hi accés)
*/
  c integer;
  aux number;
  resultat number;
  sentencia varchar2(1000);
begin
  sentencia:='select count(*) from '||nomEsquema||'.'||nomTaula;
  dbms_output.put_line('Sentència: '||sentencia);
-- Aquesta instrucció només hi és per poder comprovar la sintaxi de la
-- sentència construïda que s'executarà en el cursor
```

```
  c:=dbms_sql.open_cursor;
  dbms_sql.parse(c,sentencia,dbms_sql.native);
  dbms_sql.define_column(c,1,resultat);

  aux:=dbms_sql.execute_and_fetch(c); -- La variable aux no té cap utilitat
  dbms_sql.column_value(c,1,resultat);

  dbms_sql.close_cursor(c);
  return resultat;
exception
  when others then
    if SQLCODE=-942 then
      dbms_sql.close_cursor(c);
      return -1;
    end if;
end;
/
grant execute on comptar_files to public;
```

Observem que en SQL dinàmic cal aconseguir construir la sentència a executar, per la qualcosa en multitud d'ocasions s'utilitza el operador de concatenació.

Observem també que la funció controla, amb el codi d'error -942, el fet de que no existeixi l'esquema o la taula indicats. Per esbrinar el codi d'aquest error no hem de fer altra cosa que provocar-lo des d'una consola *SQL*Plus*.

Fixem-nos que les quatre accions imprescindibles hi són presents: `OPEN_CURSOR`, `PARSE`, `EXECUTE` i `CLOSE_CURSOR`. En aquest cas, en tractar-se d'una sentència `SELECT`, cal més accions per preparar la informació a recuperar. Així, hem necessitat:

- El procediment `DEFINE_COLUMN` per definir les columnes que recuperarem
- El procediment `FETCH_ROWS` (en realitat l'hem juntat amb `EXECUTE` emprant `EXECUTE_AND_FETCH`) per a recuperar una fila. No hem hagut d'executar, en aquesta ocasió, cap bucle, ni comprovar si hi havia com a mínim una fila, doncs el resultat d'uns instrucció `SELECT COUNT` sempre té un i només una fila resultant.
- El procediment `COLUMN_VALUE` per a recuperar el valor d'una columna de la darrera fila recuperada.

La funció `comptar_files` dissenyada pot ser executada des de diversos llocs. El següent guió ens serveix per a comprovar-ne el seu funcionament. Pensem, però, que aquesta funció ha estat creada en algun esquema (suposarem `empresa`) i, per tant, en executar-la cal indicar l'esquema on resideix.

```
/* Programa: u5n2p12.sql (guió)
   Descripció: Guió que permet comprovar el funcionament de la funció comptar_files
               (dissenyada en el guió u5n2p11.sql) i creada a l'esquema "empresa"
   Autor: Isidre Guixà
*/
set serveroutput on
var nbFiles number;
column :nbFiles heading "Nre. Files"
accept nomEsquema prompt "Nom de l'esquema on efectuar el recompte: "
accept nomTaula prompt "Nom de la taula on efectuar el recompte: "
execute :nbFiles:=empresa.comptar_files('&nomEsquema','&nomTaula')
select :nbFiles from dual;
undefine nomEsquema
undefine nomTaula
clear columns
```

Observem algunes execucions d'aquest guió:

```
SQL> @u5n2p12
Nom de l'esquema on efectuar el recompte: empresa
Nom de la taula on efectuar el recompte: dept
Sentència: select count(*) from empresa.dept

El procediment PL/SQL ha finalitzat correctament.

Nre. Files
----------
         6

SQL> @u5n2p12
```

Trobareu el fitxer `u5n2p12.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
Nom de l'esquema on efectuar el recompte: sanitat
Nom de la taula on efectuar el recompte: hospital
Sentència: select count(*) from sanitat.hospital

El procediment PL/SQL ha finalitzat correctament.

Nre. Files
----------
         4

SQL> @u5n2p12
Nom de l'esquema on efectuar el recompte: sanitat
Nom de la taula on efectuar el recompte: especialistes
Sentència: select count(*) from sanitat.especialistes

El procediment PL/SQL ha finalitzat correctament.

Nre. Files
----------
        -1
```

En primer lloc, comentem que gràcies a que el guió activa el paràmetre SERVEROUTPUT, podem veure la sentència que s'executa i que s'ha construït dinàmicament.

En segon lloc, observem que a les dues primeres execucions, el nombre de files resultant ha estat superior a zero, fet que indica que l'usuari que utilitza el guió té accés al comptatge del nombre de files de la taula indicada. En canvi, la tercera execució dona resultat -1 que cal interpretar com que no existeix o l'usuari no té accés a la taula indicada: SANITAT.ESPECIALISTES.

Exemple d'utilització del paquet DBMS_SQL per una sentència SELECT de diverses files

Es demana una funció, d'accés per a tothom, que retorni una taula imbricada amb els valors diferents que hi ha en quasevol columna alfanumèrica de qualsevol taula de qualsevol esquema a indicar via paràmetres en la crida.

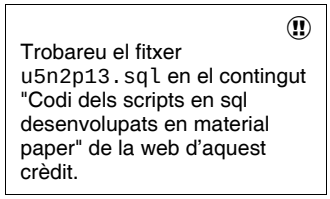
Està clar que la sentència és simple:

```
select distinct <nomCamp> from <nomEsquema>.<nomTaula>:
```

Però els noms del camp, de l'esquema i de a taula s'han de passar per paràmetre i, per tant, estem obligats a utilitzar SQL dinàmic, fet que ens ho permet el paquet DBMS_SQL.

El guió següent procedeix a crear un paquet que conté la funció demanada i a concedir el permís d'execució a qualsevol usuari, fet que s'assoleix concedint el privilegi EXECUTE a PUBLIC.

Per què un paquet i no només una funció? Pensem que aquesta funció ha de retornar una variable d'un tipus compost que cal declarar i això ens convé fer-ho en un paquet, ja que d'aquest manera, tenim agrupats el tipus i la funció.




```
/* Programa: u5n2p13.sql (guió)
   Descripció: Exemple d'utilització del paquet DBMS_SQL per a SQL dinàmic en una instrucció
               SELECT de varies files.
               Disseny de funció pública que retorna una taula imbricada amb els valors diferents
               que hi ha en una columna alfanumèrica d'una taula i esquema que han d'existir o
               als quals l'usuari ha de tenir accés.
   Autor: Isidre Guixà
*/
create or replace package vd as
   type tCadena is table of varchar2(1000);

   function vd_alfa (nomEsquema in varchar2, nomTaula in varchar2, nomCamp in varchar2)
            return tCadena;
   /* Retorna: Dada de tipus tCadena plena amb els diferents valors del camp indicat, el qual ha
               de ser alfanumèric
               La taula estarà buida en cas que el camp no contingui cap valor.
               null si no hi ha el camp o no hi ha la taula o no hi ha l'esquema
   */
end vd;
/
create or replace package body vd as

   function vd_alfa (nomEsquema in varchar2, nomTaula in varchar2, nomCamp in varchar2)
            return tCadena is
```

```
      c integer;
      aux number;
      resultat varchar2(1000);
      sentencia varchar2(1000);
      i number;
      vCadena tCadena;
   begin
      sentencia:='select distinct '||nomCamp||' from '||nomEsquema||'.'||nomTaula;
      dbms_output.put_line('Sentència: '||sentencia);
-- Aquesta instrucció només hi és per poder comprovar la sintaxi de la
-- sentència construïda que s'executarà en el cursor

      c:=dbms_sql.open_cursor;
      dbms_sql.parse(c,sentencia,dbms_sql.native);
      dbms_sql.define_column(c,1,resultat,1000);
      aux:=dbms_sql.execute(c); -- La variable aux no té cap utilitat

      i:=0;
      vCadena:=tCadena();
      while dbms_sql.fetch_rows(c)>0 loop
         dbms_sql.column_value(c,1,resultat);
         vCadena.extend(1);
         i:=i+1;
         vCadena(i):=resultat;
      end loop;
      dbms_sql.close_cursor(c);
      return vCadena;
   exception
      when others then
         if SQLCODE=-942 or SQLCODE=-904 then
            dbms_sql.close_cursor(c);
            return null;
         end if;
   end;
end vd;
/
grant execute on vd to public;
```

Observem que la funció controla, amb el codi d'error -942, el fet de que no existeixi l'esquema o la taula indicats i amb el codi -904 el fet de que no existeixi el camp indicat. Per esbrinar els codis d'aquests errors no hem de fer altra cosa que provocar-los des d'una consola *SQL*Plus*.

La funció `vd_alfa` dissenyada pot ser executada des de diversos llocs. El següent guió ens serveix per a comprovar-ne el seu funcionament.

Trobareu el fitxer `u5n2p14.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p14.sql (guió)
   Descripció: Guió que permet comprovar el funcionament de la funció vd_alfa (guió u5n2p13.sql)
   Autor: Isidre Guixà
*/
set serveroutput on
set verify off
accept nomEsquema prompt "Nom de l'esquema: "
accept nomTaula prompt "Nom de la taula: "
accept nomCamp prompt "Nom del camp: "
prompt Execució de la sentència SELECT corresponent:
select distinct &nomCamp from &nomEsquema..&nomTaula;
prompt Crida del procediment vd_alfa per aconseguir el mateix resultat:
declare
   vCadena vd.tCadena;
   i number;
begin
   vCadena:=vd.vd_alfa('&nomEsquema','&nomTaula','&nomCamp');
   if vCadena is null then
      dbms_output.put_line('Esquema o taula o camp NO són vàlids.');
   elsif vCadena.count=0 then
      dbms_output.put_line('No hi ha cap valor');
   else
      dbms_output.put_line('Valors diferents:');
      for i in 1..VCadena.count loop
         dbms_output.put_line(vCadena(i));
      end loop;
   end if;
end;
/
undefine nomEsquema
undefine nomTaula
undefine nomCamp
```

Observem algunes execucions d'aquest guió. Per a poder comprovar el cas en que no hi ha cap valor a la taula, procedim prèviament a la creació d'una nova taula sense assignar-li cap valor:

```
SQL> create table impostos (codi number, percentatge number(4,2));

Taula creada.

SQL> @u5n2p14
Nom de l'esquema: empresa
Nom de la taula: impostos
Nom del camp: percentatge
Execució de la sentència SELECT corresponent:

no s'ha seleccionat cap fila

Crida del procediment vd_alfa per aconseguir el mateix resultat:
Sentència: select distinct percentatge from empresa.impostos
No hi ha cap valor

El procediment PL/SQL ha finalitzat correctament.

SQL> @u5n2p14
Nom de l'esquema: sanitat
Nom de la taula: plantilla
Nom del camp: funcio
Execució de la sentència SELECT corresponent:

FUNCIO
----------
Infermer
Intern
Infermera

Crida del procediment vd_alfa per aconseguir el mateix resultat:
Sentència: select distinct funcio from sanitat.plantilla
Valors diferents:
Infermer
Intern
Infermera

El procediment PL/SQL ha finalitzat correctament.

SQL> @u5n2p14
Nom de l'esquema: empresa
Nom de la taula: dept
Nom del camp: nreEmpleats
Execució de la sentència SELECT corresponent:
select distinct nbEmpleats from empresa.dept
               *
ERROR a la línia 1:
ORA-00904: "NREEMPLEATS": identificador no vàlid

Crida del procediment vd_alfa per aconseguir el mateix resultat:
Sentència: select distinct nbEmpleats from empresa.dept
Esquema o taula o camp NO són vàlids.

El procediment PL/SQL ha finalitzat correctament.
```

Exemple d'utilització del paquet `DBMS_SQL` per una sentència `DDL`

Es demana una funció, d'accés per a tothom, que permeti efectuar l'eliminació de qualsevol tipus d'objecte a indicar via paràmetres en la crida.

Està clar que la sentència és simple:

```
drop <TipusObjecte> <nomEsquema>.<nomObjecte>:
```

Però els noms del tipus d'objecte, de l'esquema i de l'objecte s'han de passar per paràmetre i, per tant, estem obligats a utilitzar SQL dinàmic, fet que ens ho permet el paquet DBMS_SQL.

El guió següent procedeix a crear la funció i a concedir el permís d'execució a qualsevol usuari, fet que s'assoleix concedint el privilegi EXECUTE a PUBLIC.

Trobareu el fitxer `u5n2p15.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p15.sql (guió)
   Descripció: Exemple d'utilització del paquet DBMS_SQL per a SQL dinàmic en una instrucció DDL.
               Disseny de funció pública que permet efectuar un drop genèric per a qualsevol
```

```
              tipus d'objecte.
   Autor: Isidre Guixà
*/

create or replace function drop_generic (nomTipus in varchar2, nomEsquema in varchar2,
                                         nomObjecte in varchar2) return number is
/* Retorna: =0 : Execució correcta
            -1 : Codi Oracle del possible error
*/
  c integer;
  aux number;
  sentencia varchar2(1000);
begin
  sentencia:='drop '||nomTipus||' '||nomEsquema||'.'||nomObjecte;
  dbms_output.put_line('Sentència: '||sentencia);
-- Aquesta instrucció només hi és per poder comprovar la sintàxis de la
-- sentència construïda que s'executarà en el cursor

  c:=dbms_sql.open_cursor;
  dbms_sql.parse(c,sentencia,dbms_sql.native);

  aux:=dbms_sql.execute(c); -- La variable aux no té cap utilitat

  dbms_sql.close_cursor(c);
  return 0;
exception
  when others then
      dbms_output.put_line('SQLCODE: '||SQLCODE);
      dbms_output.put_line('SQLERRM: '||SQLERRM);
-- Aquestes instruccions només hi són per poder visualitzar la informació de l'error
      return SQLCODE;
end;
/
grant execute on drop_generic to public;
```

Fixem-nos que en aquest cas en tenim prou amb les quatre accions imprescindibles: OPEN_CURSOR, PARSE, EXECUTE i CLOSE_CURSOR.

La funció drop_generic dissenyada pot ser executada des de diversos llocs. El següent guió ens serveix per a comprovar-ne el seu funcionament. Pensem, però, que aquesta funció ha estat creada en algun esquema (suposarem empresa) i, per tant, en executar-la cal indicar l'esquema on resideix.

```
/* Programa: u5n2p16.sql (guió)
   Descripció: Guió per comprovar el funcionament de la funció drop_generic (guió u5n2p15.sql)
   Autor: Isidre Guixà
*/
set serveroutput on
var resultat number
column resultat heading "Resultat"
accept nomTipus prompt "Tipus de l'objecte a eliminar: ";
accept nomEsquema prompt "Nom de l'esquema: "
accept nomObjecte prompt "Nom de l'objecte: "
execute :resultat:=empresa.drop_generic('&nomTipus','&nomEsquema','&nomObjecte')
select :resultat from dual;
clear columns
undefine nomTipus
undefine nomEsquema
undefine nomObjecte
```

Observem algunes execucions d'aquest guió:

Trobareu el fitxer u5n2p16.sql en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
SQL> @u5n2p16
Tipus de l'objecte a eliminar: table
Nom de l'esquema: sanitat
Nom de l'objecte: especialistes
Sentència: drop table sanitat.especialistes
SQLCODE: -942
SQLERRM: ORA-00942: la taula o la vista no existeixen

El procediment PL/SQL ha finalitzat correctament.

 :RESULTAT
----------
      -942

SQL>  @u5n2p16
Tipus de l'objecte a eliminar: function
Nom de l'esquema: empresa
```

```
Nom de l'objecte: comptar_files
Sentència: drop function empresa.comptar_files

El procediment PL/SQL ha finalitzat correctament.

 :RESULTAT
----------
         0
```

2.2.5. Problemes d'injecció de SQL

La injecció de SQL és una tècnica per aconseguir efectes nocius en aplicacions que contenen sentències SQL construides dinàmicament amb incorporació de dades subministrades per l'usuari de l'aplicació.

Aquest és un problema que pot tenir lloc en aplicacions desenvolupades en qualsevol llenguatge que, en temps d'execució, construeixi les sentències SQL a executar, a partir d'informació subministrada per l'usuari.

No és només un problema de PL/SQL, sinó de qualsevol llenguatge que permeti construir, en execució, les sentències SQL a executar, com per exemple PHP, ASP, VB, Java,...

La nociva tècnica d'injecció SQL s'aprofita del fet de la construcció de la sentència SQL a partir de la concatenació de dades alfanumèriques introduïdes per l'usuari en temps d'execució. Vegem-ho amb exemples.

Exemple 1 d'injecció SQL

Es demana una funció que permeti actualitzar la localitat d'un departament a partir del seu codi subministrat via argument.

Una possible versió d'aquesta funció pot ser la que crea el següent guió:

Trobareu el fitxer `u5n2p17.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p17.sql (guió)
   Descripció: Exemple de funció que facilita la injecció de SQL
   Autor: Isidre Guixà
*/
create or replace function update_dept_by_codi_01 (codiDept in varchar2, novaLoc in varchar2)
         return number is
/* Retorna: Nombre de files canviades */
   c integer;
   resultat number;
   sentencia varchar2(2000);
begin
   sentencia:='update dept set loc='''||novaLoc||''' where dept_no='||codiDept;
   dbms_output.put_line('Sentència: '||sentencia);
-- Aquesta instrucció només hi és per poder comprovar la sintàxis de la
-- sentència construïda que s'executarà en el cursor  c:=dbms_sql.open_cursor;

   dbms_sql.parse(c,sentencia,dbms_sql.native);
   resultat:=dbms_sql.execute(c);
   dbms_sql.close_cursor(c);
   return resultat;
end;
/
```

Fixem-nos que per construir la sentència, com que és via concatenació:

- per aconseguir una cometa simple després de `loc=`, cal escriure doble cometa simple i la tercera cometa és per tancar la cadena iniciada en `update`

- per aconseguir una cometa simple abans de `where` que serveixi de fi de cadena per al nom de nova localitat introduït per l'usuari, ens cal escriure doble cometa simple

Dissenyem un guió que permeti comprovar el funcionament de la funció dissenyada:

```
/* Programa: u5n2p18.sql (guió)
   Descripció: Guió que permet comprovar el funcionament de la funció update_dept_by_codi_01
               (guió u5n2p17.sql)
   Autor: Isidre Guixà
*/
set serveroutput on
var nbFiles number;
column :nbFiles heading "Nre. files afectades"

prompt Mostrem els departaments actualment existents:
select * from dept;

prompt Procedim a canviar la localitat d'un departament.
accept novaLoc prompt "Introdueixi nova localitat: "
accept codiDept prompt "Introdueixi codi del departament a canviar: "
execute :nbFiles:=update_dept_by_codi_01('&codiDept','&novaLoc')
select :nbFiles from dual;

prompt Procedim a comprovar el resultat del canvi executat:
select * from dept;

clear columns
undefine novaLoc
undefine codiDept
```

Comprovem el correcte funcionament de la funció si l'usuari introdueix, com a codi de departament a modificar, el codi d'un departament existent, com el `40`:

```
SQL> @u5n2p18
Mostrem els departaments actualment existents:

   DEPT_NO DNOM           LOC
---------- -------------- --------------
        10 COMPTABILITAT  SEVILLA
        20 INVESTIGACIÓ   MADRID
        30 VENDES         BARCELONA
        40 PRODUCCIÓ      BILBAO
        50 INFORMÀTICA    IGUALADA
        60 COMPRES        IGUALADA

6 files seleccionades.

Procedim a canviar la localitat d'un departament.
Introdueixi nova localitat: VALÈNCIA
Introdueixi codi del departament a canviar: 40
Sentència: update dept set loc='VALÈNCIA' where dept_no=40

El procediment PL/SQL ha finalitzat correctament.


Nre. files afectades
--------------------
                   1

Procedim a comprovar el resultat del canvi executat:

   DEPT_NO DNOM           LOC
---------- -------------- --------------
        10 COMPTABILITAT  SEVILLA
        20 INVESTIGACIÓ   MADRID
        30 VENDES         BARCELONA
        40 PRODUCCIÓ      VALÈNCIA
        50 INFORMÀTICA    IGUALADA
        60 COMPRES        IGUALADA

6 files seleccionades.
```

Trobareu el fitxer `u5n2p18.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

El resultat ha estat l'esperat i el departament de codi 40 ha passat a estar ubicat a la ciutat de `VALÈNCIA`.. Com que en el guió tenim activat el paràmetre `SERVEROUTPUT`, també hem vist el codi de la sentència SQL que s'executa.

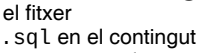
Però... què passa si l'usuari introdueix en lloc del valor `40` la cadena `100 or 1=1`?

```
SQL> @u5n2p18
Mostrem els departaments actualment existents:

   DEPT_NO DNOM           LOC
---------- -------------- --------------
        10 COMPTABILITAT  SEVILLA
        20 INVESTIGACIÓ   MADRID
        30 VENDES         BARCELONA
        40 PRODUCCIÓ      BILBAO
        50 INFORMÀTICA    IGUALADA
        60 COMPRES        IGUALADA

6 files seleccionades.

Procedim a canviar la localitat d'un departament.
Introdueixi nova localitat: VALÈNCIA
Introdueixi codi del departament a canviar: 100 or 1=1
Sentència:update dept set loc='VALÈNCIA' where dept_no=100 or 1=1

El procediment PL/SQL ha finalitzat correctament.


Nre. files afectades
--------------------
                   6

Procedim a comprovar el resultat del canvi executat:

   DEPT_NO DNOM           LOC
---------- -------------- --------------
        10 COMPTABILITAT  VALÈNCIA
        20 INVESTIGACIÓ   VALÈNCIA
        30 VENDES         VALÈNCIA
        40 PRODUCCIÓ      VALÈNCIA
        50 INFORMÀTICA    VALÈNCIA
        60 COMPRES        VALÈNCIA

6 files seleccionades.
```

Glups! Què ha passat? Vaja pifia, no? L'usuari, ni tant sols ha entrat un valor de departament correcte, però **ha injectat** codi SQL (`or 1=1`) que ha provocat que la sentència `UPDATE` s'executi per a tots els departaments existents i ara tots resideixen a la ciutat de `VALÈNCIA`.

Per tant, la funció `update_dept_by_codi_01` està afectada per la tècnia d'injecció SQL.

El guió següent crea una nova versió de la funció que no està afectada per la tècnia d'injecció SQL.



Trobareu el fitxer `u5n2p19.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p19.sql (guió)
   Descripció: Exemple de funció que no permet injecció SQL
   Autor: Isidre Guixà
*/
create or replace function update_dept_by_codi_02 (codiDept in dept.dept_no%type,
                 novaLoc in varchar2) return number is
/* Retorna: Nombre de files canviades */
   c integer;
   resultat number;
   sentencia varchar2(2000);
begin
   sentencia:='update dept set loc='''||novaLoc||''' where dept_no='||codiDept;
   dbms_output.put_line('Sentència: '||sentencia);
-- Aquesta instrucció només hi és per poder comprovar la sintàxis de la
-- sentència construïda que s'executarà en el cursor
   c:=dbms_sql.open_cursor;
   dbms_sql.parse(c,sentencia,dbms_sql.native);
   resultat:=dbms_sql.execute(c);
   dbms_sql.close_cursor(c);
   return resultat;
end;
/
```

I a continuació el guió que permet comprovar el funcionament de la funció dissenyada.

```
/* Programa: u5n2p20.sql (guió)
   Descripció: Guió que permet comprovar el funcionament de la funció update_dept_by_codi_02
               (guió u5n2p19.sql)
   Autor: Isidre Guixà
*/
```

```
set serveroutput on
var nbFiles number;
column :nbFiles heading "Nre. files afectades"

prompt Mostrem els departaments actualment existents:
select * from dept;

prompt Procedim a canviar la localitat d'un departament.
accept novaLoc prompt "Introdueixi nova localitat: "
accept codiDept prompt "Introdueixi codi del departament a canviar: "
execute :nbFiles:=update_dept_by_codi_02(&codiDept,'&novaLoc')
select :nbFiles from dual;

prompt Procedim a comprovar el resultat del canvi executat:
select * from dept;

clear columns
undefine novaLoc
undefine codiDept
```

L'execució d'aquest guió introduint un valor numèric per al codi de departament, continua sent correcta. Vegem què succeeix si l'usuari respon `100 or 1=1`:

Trobareu el fitxer `u5n2p20.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
SQL> @u5n2p20
Mostrem els departaments actualment existents:

   DEPT_NO DNOM           LOC
---------- -------------- --------------
        10 COMPTABILITAT  SEVILLA
        20 INVESTIGACIÓ   MADRID
        30 VENDES         BARCELONA
        40 PRODUCCIÓ      BILBAO
        50 INFORMÀTICA    IGUALADA
        60 COMPRES        IGUALADA

6 files seleccionades.

Procedim a canviar la localitat d'un departament.
Introdueixi nova localitat: VALÈNCIA
Introdueixi codi del departament a canviar: 100 or 1=1
BEGIN :nbFiles:=update_dept_by_codi_02(100 or 1=1,'VALÈNCIA'); END;
                                       *
ERROR a la línia 1:
ORA-06550: línia 1, columna 41:
PLS-00382: l'expressió és de tipus incorrecte
ORA-06550: línia 1, columna 7:
PL/SQL: Statement ignored
```

És a dir, la funció `update_dept_by_codi_02` no s'arriba e executar doncs el primer paràmetre (que és el que s'utilitza per el filtrat a la sentència `UPDATE`) està esperant un valor numèric i l'usuari li passa una cadena que no es correspon a un valor numèric.

Per a poder injectar codi SQL en un subprograma cal explorar les vulnerabilitats del subprograma a partir, normalment, de la tècnica "assaig i error" i això és feina difícil, però quan els resultats a aconseguir són importants, sempre trobarem usuaris disposats a efectuar-ne l'exploració. (!!)

Penseu en quants usuaris maliciosos hi ha disposats a intentar injectar codi en les crides a subprogrames amb l'únic objectiu de poder obtenir guanys de qualsevol àmbit.

Per desgràcia el nombre de subprogrames PL/SQL existents en els paquets subministrats per *Oracle* i tercers fabricants de programari, és immens i molts d'ells estan afectats per la vulnerabilitat de la injecció SQL. Es suposa que a mida que es detecten, *Oracle* i els corresponents fabricants de programari n'eliminen la vulnerabilitat en les successives versions dels paquets.

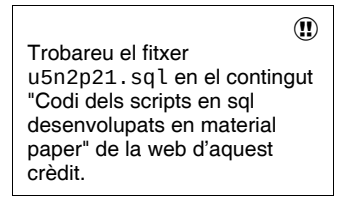
L'exemple anterior ha deixat ben clar que la vulnerabilitat pot trobar-se, també, en els subprogrames dissenyats per nosaltres i hem d'intentar

que això no succeeixi. Una manera de minimitzar les possibilitats d'injecció SQL és minimitzar la utilització d'arguments alfanumèrics. A l'exemple anterior, la vulnerabilitat ha desaparegut en declarar l'argument `codiDept` com a valor numèric enlloc de valor alfanumèric.

Exemple 2 d'injecció SQL

Es demana una funció que permeti actualitzar la localitat de tots els departaments que es troben en una determinada localitat, subministrada via argument.

Una possible versió d'aquesta funció pot ser la que crea el següent guió:



Trobareu el fitxer `u5n2p21.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

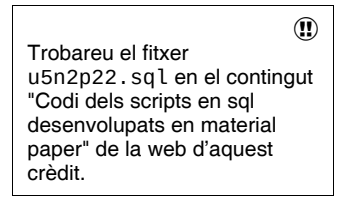
```
/* Programa: u5n2p21.sql (guió)
   Descripció: Exemple de funció que facilita la injecció de SQL
   Autor: Isidre Guixà
*/
create or replace function update_dept_by_loc (vellaLoc in varchar2, novaLoc in varchar2)
                  return number is
/* Retorna: Nombre de files canviades */
   c integer;
   resultat number;
   sentencia varchar2(2000);
begin
   sentencia:='update dept set loc='''||novaLoc||''' where loc='''||vellaLoc||'''';
   dbms_output.put_line('Sentència: '||sentencia);
-- Aquesta instrucció només hi és per poder comprovar la sintàxis de la
-- sentència construïda que s'executarà en el cursor
   c:=dbms_sql.open_cursor;
   dbms_sql.parse(c,sentencia,dbms_sql.native);
   resultat:=dbms_sql.execute(c);
   dbms_sql.close_cursor(c);
   return resultat;
end;
/
```

Fixem-nos que per construir la sentència, com que és via concatenació:

- per aconseguir una cometa simple després de `loc=`, cal escriure doble cometa simple i la tercera cometa és per tancar la cadena iniciada en `update`
- per aconseguir una cometa simple abans de `where` que serveixi de fi de cadena per al nom de nova localitat introduït per l'usuari, ens cal escriure doble cometa simple
- per aconseguir la cometa simple al final per tancar el nom de la vella localitat introduït per l'usuari, ens cal escriure una doble cometa simple dins una cadena que de per sí comença i finalitza en cometa simple i, per tant, ens trobem amb 4 cometes simples.

En aquest cas, l'argument que serveix per al filtre de la sentència `UPDATE` ha de ser obligatòriament alfanumèric.

Dissenyem un guió que permeti comprovar el funcionament de la funció dissenyada:



Trobareu el fitxer `u5n2p22.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p22.sql (guió)
   Descripció: Guió que permet comprovar el funcionament de la funció update_dept_by_loc
               (guió u5n2p21.sql)
   Autor: Isidre Guixà
*/
set serveroutput on
var nbFiles number;
column :nbFiles heading "Nre. files afectades"

prompt Mostrem els departaments actualment existents:
select * from dept;

prompt Procedim a canviar la localitat dels departaments d'una determinada localitat.
accept novaLoc prompt "Introdueixi nova localitat: "
accept vellaLoc prompt "Introdueixi nom de localitat a canviar: "
execute :nbFiles:=update_dept_by_loc_01('&vellaLoc','&novaLoc')
select :nbFiles from dual;

prompt Procedim a comprovar el resultat del canvi executat:
select * from dept;

clear columns
undefine novaLoc
undefine vellaLoc
```

L'execució d'aquest guió introduint un valor correcte pel nom de la localitat a canviar provoca una execució correcta de la funció `update_dept_by_loc`. Però, en la següent execució del guió, observis què introdueix l'usuari i, en conseqüència, quina és la sentència `UPDATE` executada dins la funció `update_dept_by_loc`.

```
SQL> @u5n2p22
Mostrem els departaments actualment existents:

   DEPT_NO DNOM           LOC
---------- -------------- --------------
        10 COMPTABILITAT  SEVILLA
        20 INVESTIGACIÓ   MADRID
        30 VENDES         BARCELONA
        40 PRODUCCIÓ      BILBAO
        50 INFORMÀTICA    IGUALADA
        60 COMPRES        IGUALADA

6 files seleccionades.

Procedim a canviar la localitat dels departaments d'una determinada localitat.
Introdueixi nova localitat: GIRONA
Introdueixi nom de localitat a canviar: Z'' or ''A''=''A
Sentència: update dept set loc='GIRONA' where loc='Z' or 'A'='A'

El procediment PL/SQL ha finalitzat correctament.


Nre. files afectades
--------------------
                   6

Procedim a comprovar el resultat del canvi executat:

   DEPT_NO DNOM           LOC
---------- -------------- --------------
        10 COMPTABILITAT  GIRONA
        20 INVESTIGACIÓ   GIRONA
        30 VENDES         GIRONA
        40 PRODUCCIÓ      GIRONA
        50 INFORMÀTICA    GIRONA
        60 COMPRES        GIRONA

6 files seleccionades.
```

És a dir, amb el codi `Z'' or ''A''=''A` injectat, s'ha aconseguit una condició de filtre que dona sempre cert per a qualsevol departament i, en conseqüència, s'actualitzen tots.

Ja sabem que en aquesta funció `update_detp_by_loc` l'argument `vellaLoc` ha de ser forçosament alfanumèric. Per eliminar la vulnerabilitat d'injecció SQL no tenim altre camí que validar, dins la mateixa funció, la correctesa de la cadena passada per paràmetre. Us animeu a fer-ne una versió?

2.3. Disparadors en Oracle

Un **disparador** (***trigger*** en anglès) de base de dades, és un bloc de codi associat a una taula específica, que s'executa de forma automàtica com a resposta a l'aparició de certs esdeveniments (execució de sentències LMD) sobre la mateixa taula.

Programació Orientada a Objectes

La Programació Orientada a Objectes (POO) consisteix, en poques paraules, en programar l'actuació dels objectes davant l'aparició de certs esdeveniments.

Per tant, la utilització de triggers en bases de dades és el primer intent d'aplicar les tècniques POO en els SGBD.

Els SGBD que incorporen alguna extensió procedimental al llenguatge SQL, també acostumen a incorporar la possibilitat de definir disparadors a la base de dades, éssent el llenguatge procedimental l'emprat per a definir els disparadors. *Oracle* segueix aquesta norma i PL/SQL és el llenguatge en el que cal definir els disparadors.

Oracle sap controlar fins a 12 esdeveniments diferents, resultat de totes les combinacions possibles entre:

- El tipus de sentència LMD que s'executa sobre la taula: `INSERT`, `UPDATE` i `DELETE`.
- Moment en què es dona resposta a l'esdeveniment: `BEFORE` **i** `AFTER`**.**
- Nivell en què s'intercepta l'esdeveniment: taula i fila.

L'acció d'un disparador pot provocar-ne l'aparició d'un altre. En tal cas es parla de **disparadors en cascada**.

Els disparadors de la base de dades no són els mateixos que els disparadors que puguin tenir les aplicacions i es disparen, si es dona l'esdeveniment, sigui quina sigui l'eina u aplicació que el provoca.

La taula 14 ens mostra les semblances i diferències entre els suprogrames emmagatzemats a la base de dades i els disparadors.

Taula 14. Semblances i diferències entre els subprogrames emmagatzemats a la base de dades i els disparadors.

Semblances	Estan formats per sentències SQL i PL/SQL El seguiment l'efectua *Oracle* de forma automàtica
Diferències	Els disparadors estan associats a taules Els disparadors s'invoquen de manera implícita, mentre que els procediments ho són de manera explícita Les sentències `COMMIT`, `ROLLBACK` i `SAVEPOINT` no es permeten en els disparadors ni en procediments cridats per disparadors Es creen amb diferents sentències SQL Els usuaris no necessiten el privilegi `EXECUTE` sobre els disparadors, sinó que aquests s'executen de forma automàtica, sense importar qui realitza els canvis a la taula. S'executen sobre els privilegis del propietari de la taula, el qual ha de tenir els accessos apropiats a tots els objectes referenciats en l'acció del disparador.

2.3.1. Edició i activació dels disparadors

L'edició dels disparadors es pot efectuar des de *SQL*Plus* o des d'eines visuals com *SQL Developer*.

La sintaxi per **crear i/o modificar** un disparador des de *SQL*Plus* és:

```
create or replace trigger [<esquema>.]<nom_trigger>
 {before|after}
 {delete|insert|update[of <llista_columnes>]}
 on [<esquema>.]<nom_taula>
 [for each row] [when <condició>]
<bloc_PL/SQL>
```

Els disparadors s'emmagatzemen separadament de la resta d'objectes i, per tant, poden tenir el mateix nom que la taula. Es recomana, però, noms únics.

La clàusula `update` sense cap columna indica que el disparador s'activarà en efectuar qualsevol `UPDATE` sobre la taula, sigui quina sigui la columna a actualitzar.

La clàusula `for each row` indica que es tracta d'un disparador a nivell de cada fila afectada per l'esdeveniment. La seva absència indica que es tracta d'un disparador a nivell de taula.

La clàusula `WHEN` indica, si existeix, una condició que cal verificar per a que el disparador s'executi en produir-se un dels esdeveniments controlats per ell.

Un disparador es pot **eliminar** amb la sentència:

```
drop trigger <nom_trigger>;
```

L'eliminació d'una taula que conté disparadors implica l'esborrat automàtic d'aquests.

Un disparador pot estar **activat o desactivat**. En la creació queda sempre activat. La sentència que permet modificar l'estat d'activació i, fins i tot, repetir la compilació d'un disparador és:

```
alter trigger <nom_trigger> {enable|disable|compile};
```

Hi ha diferents raons que poden provocar la desactivació d'un disparador:

- En efectuar una càrrega massiva de dades la qual es vol aconseguir de la forma més ràpida possible sense efectuar comprovacions.

- En efectuar un `INSERT`, `UPDATE` o `DELETE` sobre una taula que té referències i que en aquell moment no estan disponibles per diverses causes com error en la xarxa, error de disc o altres.

La sentència `ALTER TABLE` també pot utilitzar-se per activar o desactivar tots els disparadors de la taula:

```
alter table <nom_taula> {enable|disable} all triggers;
```

Els disparadors de bases de dades permeten fer referència al valor nou i al valor antic d'una columna a nivell de fila amb els prefixes `:old` i `:new`, utilitzant la sintaxi:

```
:old.<nom_columna> -- Per fer referència al valor antic
:new.<nom_columna> -- Per fer referència al valor nou
```

– Els valors `:old` i `:new` només estan disponibles a nivell de fila.

– Per la sentència `UPDATE` estan disponibles els dos valors `:old` i `:new`.

– Per la sentència `INSERT` el valor `:old` té el valor `NULL`.

– Per la sentència `DELETE` el valor `:new` té el valor `NULL`.

El disparador `BEFORE` a nivell de fila pot assignar un valor a `:new` encara que aquest vingués donat per la clàusula `set` de la instrucció `UPDATE` o per la llista `values` de la instrucció `INSERT`.

Si es desitja, els noms `:new` i `:old` poden canviar-se per uns altres noms correlacionats utilitzant la clàusula `referencing` en la creació del disparador:

```
create or replace trigger [<esquema>.]<nom_trigger>
 {before|after}
 {delete|insert|update[of <llista_columnes>]}
 on [<esquema>.]<nom_taula>
 [referencing {old [as] <nou_nom_per_OLD> | new [as] <nou_nom_per_NEW>}]
 [for each row] [when <condició>]
<bloc_PL/SQL>
```

Els prefixes `:old` i `:new` es poden utilitzar dins el bloc PL/SQL i en la condició `WHEN` de la capçalera del disparador. En aquest darrer cas no han de portar els dos punts al davant.

Per finalitzar, els disparadors de bases de dades permeten distingir el motiu d'execució del disparador, ja que un disparador pot estar dissenyat per donar resposta a més d'un tipus d'operacions LMD. Per aconseguir-ho, *Oracle* facilita els següents predicats condicionals:

```
if inserting ... -- Cert quan l'esdeveniment és d'inserció
if updating ... -- Cert quan l'esdeveniment és d'actualització
if deleting ... -- Cert quan l'esdeveniment és d'eliminació
```

Exemple de disparadors

Es desitja, a l'esquema *empresa*, automatitzar el càlcul del camp `import` de la taula `DETALL` a partir dels camps `preu_venda` i `quantitat`. De la mateixa manera, sembla lògic que el camp `total` de la taula `COMANDA` sigui calculat automàticament a mida que s'insereixen, eliminen i modifiquen les corresponents línies de la taula `DETALL`.

Per aconseguir aquestes automatitzacions, cal definir dos disparadors:

- Un disparador, que anomenarem `detall_before`, que s'executarà abans d'efectuar una inserció a la taula `DETALL` o una modificació en els camps `preu_venda`, `quantitat` o `import`, de manera que s'encarregarà de calcular correctament el valor del camp `import` en base als valors dels camps `preu_venda` i `quantitat`. És clar que aquesta actuació s'ha de fer abans de procedir a l'execució de la instrucció `INSERT` o `UPDATE`.
- Un disparador, que anomenarem `detall_after`, que s'executarà després d'haver inserit o eliminat una línia de detall a una comanda o haver-ne modificat el preu de

venda o la quantitat o haver canviat la línia a una altra comanda, de manera que s'encarregarà de propagar els canvis que corresponguin a les files afectades de la taula COMANDA.

- Amb aquests disparadors queden controlats els efectes que sobre les taules COMANDA i DETALL provoquen certes actuacions sobre la taula DETALL, però no queda controlat el fet de que en la inserció d'una nova comanda el camp `total` tingui sempre valor zero, independentment del que pugui dir la sentència. Així mateix tampoc queda controlat que ningú no pugui modificar el camp `total` (doncs volem que el valor d'aquest camp sigui únicament gestionat de forma automàtica). Ens cal, per tant, un tercer disparador que anomenarem `comanda_before`.

El següent guió conté les instruccions corresponents a la creació dels tres disparadors.

> Trobareu el fitxer `u5n2p23.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p23.sql (guió)
   Descripció: Guió que crea els disparadors per automatitzar el càlcul dels camps "import" a la
               taula DETALL i "total" a la taula COMANDA.
   Autor: Isidre Guixà
*/

create or replace trigger detall_before
      before insert or update of preu_venda, quantitat, import
      on detall
      for each row
   begin
      if inserting or updating then
         :new.import := :new.quantitat * :new.preu_venda;
      end if;
end detall_before;
/
create or replace trigger detall_after
      after delete or insert or update of com_num, preu_venda, quantitat
      on detall
      for each row
   begin
      if deleting or (updating and :new.com_num<>:old.com_num) then
         update comanda set total=total-:old.import where com_num=:old.com_num;
      end if;
      if inserting or (updating and :new.com_num<>:old.com_num) then
         update comanda set total=total+:new.import where com_num=:new.com_num;
      end if;
      if updating and :new.com_num=:old.com_num then
         update comanda set total=total+:new.import-:old.import where com_num=:new.com_num;
      end if;
end detall_after;
/
create or replace trigger comanda_before
      before insert or update of total
      on comanda
      for each row
   begin
      if inserting then :new.total := 0;
      end if;
      if updating then :new.total := :old.total;
      end if;
end comanda_before;
/
```

Per a comprovar el funcionament d'aquests disparadors, ens cal partir de dades coherents a les taules DETALL i COMANDA. Per aconseguir aquesta coherència, podem desactivar els disparadors, executar les instruccions necessàries per aconseguir la coherència i tornar a activar els disparadors.

```
SQL> alter table detall disable all triggers;

Taula alterada.

SQL> alter table comanda disable all triggers;

Taula alterada.

SQL> update detall set import=preu_venda*quantitat;

64 files actualitzades.

SQL> update comanda set total=(select sum(import) from detall
  2  where detall.com_num=comanda.com_num);

21 files actualitzades.
```

```
SQL> alter table detall enable all triggers;

Taula alterada.

SQL> alter table comanda enable all triggers;

Taula alterada.
```

Ara que tenim dades coherents, procedim a comprovar el funcioanament dels disparadors.

Per exemple, donada una comanda, efectuem modificacions en alguna de les línies de detall i comprovem com les modificacions, si cal, es propagan a la comanda:

```
SQL> select * from comanda where com_num=610;

   COM_NUM COM_DATA C CLIENT_COD DATA_TRA      TOTAL
---------- -------- - ---------- -------- ----------
       610 07/01/87 A        101 08/01/87      101,4
SQL> select * from detall where com_num=610 order by detall_num;

   COM_NUM DETALL_NUM   PROD_NUM PREU_VENDA  QUANTITAT     IMPORT
---------- ---------- ---------- ---------- ---------- ----------
       610          1     100860         35          1         35
       610          2     100870        2,8          3        8,4
       610          3     100890         58          1         58

SQL> update detall set quantitat=2*quantitat where com_num=610;

3 files actualitzades.

SQL> select * from detall where com_num=610 order by detall_num;

   COM_NUM DETALL_NUM   PROD_NUM PREU_VENDA  QUANTITAT     IMPORT
---------- ---------- ---------- ---------- ---------- ----------
       610          1     100860         35          2         70
       610          2     100870        2,8          6       16,8
       610          3     100890         58          2        116

SQL> select * from comanda where com_num=610;

   COM_NUM COM_DATA C CLIENT_COD DATA_TRA      TOTAL
---------- -------- - ---------- -------- ----------
       610 07/01/87 A        101 08/01/87      101,4
```

No hi veieu res d'estrany? La modificació de les quantitats a les línies de detall de la comanda 610 ha provocat el recàlcul de l'import de les files afectades (disparador `detall_before`), però en canvi no s'ha modificat el total de la corresponent comanda. Sembla que no hagi funcionat el disparador `detall_after` que és l'encarregat de recalcular l'import de la comanda en produïr-se alguna modificació en alguna línia de detall de la comanda en els camps `preu_venda` i/o `quantitat`.

La realitat és que el disparador `detall_after` ha funcionat, però com que modifica el camp `total` de la taula `COMANDA`, acte seguit es dispara el disparador `comanda_before` de la taula `COMANDA` que deixa el camp `total` com estava abans de fer la modificació. I aquest disparador ens interessa mantenir-lo per si algú intenta modificar el camp `total`, però clar, no voldríem que actués si la modificació prové del disparador `detall_after`. Com solucionar-ho? Hauríem d'establir algun tipus de comunicació entre els dos disparadors, fet que podem assolir amb la utilització del paquet `DBMS_OUTPUT`.

El següent guió conté una nova definició dels tres disparadors que soluciona la problemàtica. El primer disparador es manté intacte.

> Trobareu el fitxer `u5n2p24.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p24.sql (guió)
   Descripció: Guió que crea els disparadors per automatitzar el càlcul dels camps "import" a la
               taula DETALL i "total" a la taula COMANDA.
   Autor: Isidre Guixà
*/
create or replace trigger detall_before
      before insert or update of preu_venda, quantitat, import
      on detall
      for each row
   begin
      if inserting or updating then
         :new.import := :new.quantitat * :new.preu_venda;
      end if;
end detall_before;
```

```
/
create or replace trigger detall_after
      after delete or insert or update of com_num, preu_venda, quantitat
      on detall
      for each row
   begin
      dbms_output.enable;
      if deleting or (updating and :new.com_num<>:old.com_num) then
         dbms_output.put_line('Modificant detall');
         update comanda set total=total-:old.import where com_num=:old.com_num;
      end if;
      if inserting or (updating and :new.com_num<>:old.com_num) then
         dbms_output.put_line('Modificant detall');
         update comanda set total=total+:new.import where com_num=:new.com_num;
      end if;
      if updating and :new.com_num=:old.com_num then
         dbms_output.put_line('Modificant detall');
         update comanda set total=total+:new.import-:old.import where com_num=:new.com_num;
      end if;
end detall_after;
/
create or replace trigger comanda_before
      before insert or update of total
      on comanda
      for each row
   declare
      missatge varchar2(30);
      estat integer;
   begin
      if inserting then :new.total := 0;
      end if;
      if updating then
         dbms_output.enable;
         dbms_output.get_line(missatge,estat);
         if estat=0 and missatge='Modificant detall' then
            null;
         else
            :new.total := :old.total;
         end if;
      end if;
end comanda_before;
/
```

Suposant que tornem a tenir les dades coherents, procedim a comprovar el funcioanament de la nova versió dels disparadors.

```
SQL> select * from comanda where com_num=610;

   COM_NUM COM_DATA C CLIENT_COD DATA_TRA      TOTAL
---------- -------- - ---------- -------- ----------
       610 07/01/87 A        101 08/01/87      101,4

SQL> select * from detall where com_num=610 order by detall_num;

   COM_NUM DETALL_NUM   PROD_NUM PREU_VENDA  QUANTITAT     IMPORT
---------- ---------- ---------- ---------- ---------- ----------
       610          1     100860         35          1         35
       610          2     100870        2,8          3        8,4
       610          3     100890         58          1         58

SQL> update detall set quantitat=2*quantitat where com_num=610;

3 files actualitzades.

SQL> select * from detall where com_num=610 order by detall_num;

   COM_NUM DETALL_NUM   PROD_NUM PREU_VENDA  QUANTITAT     IMPORT
---------- ---------- ---------- ---------- ---------- ----------
       610          1     100860         35          2         70
       610          2     100870        2,8          6       16,8
       610          3     100890         58          2        116

SQL> select * from comanda where com_num=610;

   COM_NUM COM_DATA C CLIENT_COD DATA_TRA      TOTAL
---------- -------- - ---------- -------- ----------
       610 07/01/87 A        101 08/01/87      202,8
```

En aquest cas, el funcionament ha estat perfecte.

2.3.2. Restriccions en disparadors

Els disparadors tenen unes restriccions que passem a precisar. En primer lloc, un parell de definicions:

- **Taula mutant** és aquella que actualment està sent modificada per una sentència `INSERT`, `UPDATE` o `DELETE`.

- **Taula referenciada** és aquella que la sentència que dispara el disparador necessita llegir per verificar una restricció d'integritat referencial.

Les restriccions existents sobre els disparadors són:

- Els disparadors no poden accedir a una taula mutant (d'aquesta forma s'evita que els disparadors vegin inconsistències).

- Els disparadors no poden escriure, en taules referenciades, en aquelles columnes implicades en restriccions d'integritat referencial.

2.3.3. Flux d'execució d'un disparador

El flux d'execució d'un disparador és:

1) La sentència `INSERT/UPDATE/DELETE` sobre una taula arriba al SGBD.

2) S'executa el disparador `BEFORE` a nivell de taula

3) Per cada fila afectada per la sentència SQL:
   a) S'executa el disparador `BEFORE` a nivell de fila
   b) S'efectua el canvi en la fila i es comproven les restriccions d'integritat referencial.
   c) S'executa el disparador `AFTER` a nivell de fila

4) Es completa la comprovació de restriccions d'integritat referencial per sentències internes

5) S'executa el disparador `AFTER` a nivell de taula

6) Es retorna el control a l'aplicació que va llençar la sentència inicial

Si una sentència del disparador falla, el disparador executa un `ROLLBACK` total. En mig d'un disparador es pot introduir la sentència `CANCEL` que provoca, també el `ROLLBACK` total.

La taula 15 presenta casos d'utilització de diferents tipus de disparadors.

Taula 15. Casos d'utilització de diferents tipus de disparadors.

Tipus de disparador	Exemple d'utilització
`BEFORE` a nivell de taula	Per inicialitzar variables globals utilitzades per altres disparadors Per evitar actualitzacions abans que aquestes tinguin lloc
`BEFORE` a nivell de fila	Per obtenir camps calculats en la nova fila Per evitar actualitzacions abans que es aquestes tinguin lloc
`AFTER` a nivell de fila	Per auditories per valor i per fila
`AFTER` a nivell de taula	Per auditories una vegada l'acció s'ha desenvolupat

2.3.4. Aplicacions dels disparadors

Hi ha situacions en les que és apropiat treballar amb els disparadors de base de dades:

1) Mantenir camps calculats o derivats

Els disparadors de base de dades poden utilitzar-se per extraure columnes derivades basades en un valor que prové d'una sentència `INSERT` o `UPDATE`.

Així, per exemple, quan es vol assegurar que el camp `total` de la taula `COMANDA` sigui el sumatori del camp `import` de les corresponents files de la taula `DETALL`, definim un disparador sobre la taula `DETALL` de manera que en efectuar insercions, esborrats i/o actualitzacions dels camps `preu_venda` i/o `quantitat`, provoquin la corresponent actualització del camp `total` en la taula `COMANDA`.

En cas que una columna tingui el valor derivat de valors proporcionats per sentències `INSERT` o `UPDATE` sobre la mateixa taula, cal utilitzar el disparador `BEFORE ROW`, ja que:

– El valor ha de ser calculat abans de que s'executi la pròpia sentència `INSERT` o `UPDATE`.

– El disparador ha d'executar-se per cada fila afectada.

És la situació que té lloc en el càlcul del camp `import` de la taula `DETALL` a partir dels valors dels camps `preu_venda` i `quantitat` de la mateixa taula.

2) Implementar regles de seguretat més complexes

Els disparadors de base de dades es poden utilitzar per evitar que un usuari efectuï canvis en certes taules en intervals de temps determinats (caps de setmana, vacances de l'empresa, ...)

Aquest tipus de comprovació no es pot definir en declaracions de restriccions d'integritat sobre la taula i cal fer-ho via disparadors.

3) Forçar el compliment de les regles de negoci

S'anomenen regles de negoci les regles específiques de l'entorn de treball que no poden definir-se sobre la taula utilitzant restriccions.

Exemple de disparador per a controlar una regla de negoci

Suposem que a l'esquema empresa, els augments de salari dels empleats mai poden ser superiors a un 10%. Com controlar-ho?

Cal definir un disparador que no permeti l'augment de salari d'un empleat si és superior al 10%. El guió següent conté la definició del corresponent disparador que, molt adeqüadament, anomenem `control_canvi_salari`.

Trobareu el fitxer `u5n2p25.sql` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n2p25.sql (guió)
   Descripció: Guió que crea un disparador per a controlar que l'augment del salari d'un empleat
               no sigui superior al 10%
   Autor: Isidre Guixà
*/
create or replace trigger control_canvi_salari
      before update of salari on emp
      for each row
      when (new.salari<old.salari or new.salari>1.1*old.salari)
   begin
      raise_application_error (-20500,'Augment de salari entre el 0 i el 10%');
   end;
/
```

Una vegada executat el guió i, per tant, creat el deisparador a la base de dades, podem procedir a comprovar-ne el funcionament:

```
SQL> select emp_no, cognom, salari from emp where salari < 150000;

    EMP_NO COGNOM         SALARI
---------- ---------- ----------
      7369 SÁNCHEZ        104000
      7876 ALONSO         143000
      7900 JIMENO         123500

SQL> update emp set salari=150000 where salari < 150000;
update emp set salari=150000 where salari < 150000
       *
ERROR a la línia 1:
ORA-20500: Augment de salari entre el 0 i el 10%
ORA-06512: a "EMPRESA.CONTROL_CANVI_SALARI", line 2
ORA-04088: error en l'execució del disparador
''EMPRESA.CONTROL_CANVI_SALARI'

SQL> select emp_no, cognom, salari from emp where salari < 150000;

    EMP_NO COGNOM         SALARI
---------- ---------- ----------
      7369 SÁNCHEZ        104000
      7876 ALONSO         143000
      7900 JIMENO         123500
```

La nostra intenció ha estat augmentar el salari a 150000 a tots els empleats que el tenen inferior i, clar, com que en algun d'ells l'agument era superior al 10%, no s'ha pogut dur a

terme. Comprovem que si intenem actualitzar salaris d'empleats pels que l'augment no superi el 10%, no hi ha cap problema:

```
SQL> update emp set salari=150000 where emp_no=7876;

1 fila actualitzada.

SQL> select emp_no, cognom, salari from emp where salari < 150000;

    EMP_NO COGNOM         SALARI
---------- ---------- ----------
      7369 SÁNCHEZ        104000
      7900 JIMENO         123500
```

4) Realitzar auditories basades en valors

Les auditories en SGBD consisteixen en tenir constància de qui-què-quan ha executat instruccions sobre la base de dades.

Oracle incorpora un sistema d'auditoria amb el que es pot controlar els fets anteriors però no es pot deixar constància dels valors afectats. Els disparadors poden utilitzar-se per arribar on no arriba el SGBD i poder realitzar una auditoria basada en valors.

Amb els disparadors podem auditar sentències LMD, efectuar auditoria per cada fila i enregistrar els valors `new` i `old` en les sentències `UPDATE`. En canvi, no permeten auditar les sentències LDD, les sentències `SELECT` ni les connexions efectuades.

5) Realitzar canvis implícits a una acció

Els disparadors de la base de dades es poden utilitzar per realitzar una acció de forma transparent quan s'executa una sentència.

Així, per exemple, en efectuar una comanda, per cada línia de detall que s'insereix o s'esborra o se n'actualitza la quantitat, podem aconseguir que el corresponent producte augmenti, disminueixi o modifiqui, respectivament, la quantitat pendent de recepció. Aquest disparador hauria d'actuar sempre i quan la línia de la comanda no hagi estat ja servida, doncs és de suposar que en el moment de servir-se, la quantitat pendent de recepció va disminuir per augmentar la quantitat real emmagatzemada.

6) Mantenir taules replicades

Un disparador es pot utilitzar per replicar, sobre una taula replicada, totes les modificacions que s'efectuen en la taula original.

2.3.5. Documentació de disparadors

Per tal de conèixer els disparadors d'un esquema, es pot utilitzar eines visuals com *SQL Developer* o, des de *SQL*Plus*, utilitzar les vistes i comandaments que ens proporciona *Oracle*:

- Les vistes `USER_OBJECTS` i `ALL_OBJECTS`, que ens facilitaran el nom dels disparadors tenint en compte que són objectes identificats pels tipus `'TRIGGER'`.

- Les vistes `USER_SOURCE` i `ALL_SOURCE`, que ens facilitaran el codi dels disparadors.

- Les vistes `USER_TRIGGERS`, `USER_TRIGGER_COLS`, `ALL_TRIGGERS` i `ALL_TRIGGER_COLS` que faciliten informació més específica de disparadors que les vistes anteriors.

Exemple d'obtenció d'informació sobre els disparadors d'un esquema

La sentència per a conèixer els disparadors d'un esquema és:

```
SQL> select object_name, object_type from user_objects
  2  where object_type = 'TRIGGER';

OBJECT_NAME          OBJECT_TYPE
-------------------- -------------------
DETALL_BEFORE        TRIGGER
DETALL_AFTER         TRIGGER
CONTROL_CANVI_SALARI TRIGGER
COMANDA_BEFORE       TRIGGER
```

Per accedir el codi dels disparadors podem fer:

```
SQL> select line, substr(text,1,100) from user_source where upper(name)='CONTROL_CANVI_SALARI';

 LINE SUBSTR(TEXT,1,100)
----- ----------------------------------------------------------------------------------------
    1 trigger control_canvi_salari
    2       before update of salari on emp
    3       for each row
    4       when (new.salari<old.salari or new.salari>1.1*old.salari)
    5    begin
    6       raise_application_error (-20500,'Augment de salari entre el 0 i el 10%');
    7    end;
```

2.3.6. Beneficis d'utilització

La taula 16 ens mostra una síntesi dels beneficis que aporta la utilització de disparadors.

Taula 16. Beneficis de la utilització de disparadors

Nivell	Beneficis
Seguretat	Permeten sofisticats sistemes de control Permeten adaptar i personalitzar els sistemes d'auditoria a utilitzar

Nivell	**Beneficis**
Integritat	Asseguren que les operacions de dades relacionades entre sí s'executin conjuntament Reforcen la idea que cal utilitzar regles de negoci
Rendiment	Redueixen el número de crides a *Oracle* Redueixen el tràfic de la xarxa
Memòria	Estalvien memòria en utilitzar l'àrea de SQL compartida Necessiten una única còpia del codi per múltiples usuaris
Productivitat	Milloren l'escriptura i manteniment de programes en utilitzar una única còpia.

3. SQL hostatjat en C

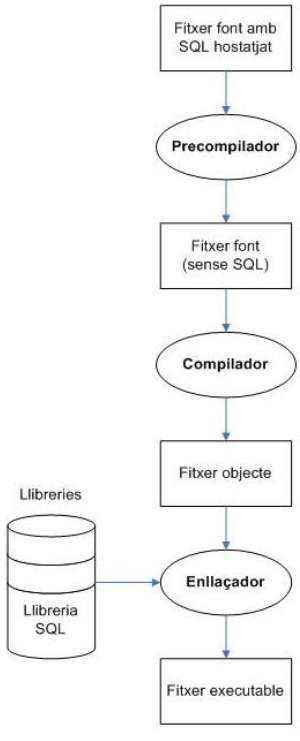
Els SGBD actuals faciliten el llenguatge SQL per a gestionar les dades i, la majoria de SGBD, faciliten alguna extensió procedimental que permet dissenyar subprogrames que s'emmgatzemen a la base de dades i automatitzar tasques (disparadors).

L'extensió procedimental (*PL/SQL* en *Oracle*, *Transact-SQL* en SQL Server,...) acostuma a ser un llenguatge de tercera generació que permet la inserció de crides a sentències SQL en el seu codi.

És necessari, però, poder executar sentències SQL des de programes desenvolupats en llenguatges d'alt nivell externs, totalment, al SGBD, com per exemple *Visual Basic*, *Java*, *C++*,... La majoria de llenguatges d'alt nivell incorporen mecanismes per a poder executar codi SQL i gestionar les bases de dades. Aquests mecanismes acostumen a consistir en funcions i/o accions proporcionades pel llenguatge a les quals es passa la instrucció SQL a executar via paràmetre.

En alguns llenguatges hi ha una altra possibilitat de funcionament, consistent en la immersió de sentències SQL dins els programes desenvolupats en un llenguatge d'alt nivell, fet que es coneix com a SQL immers o SQL hostatjat. ‼

Figura 5. Obtenció de programa executable a partir d'un fitxer font amb SQL hostatjat.



Ara bé, us imagineu com hostatjar, en un programa desenvolupat en el llenguatge C, sentències SQL? Oi que sembla impossible?

La tècnica de la immersió consisteix en dissenyar el programa en llenguatge C incloent, quan sigui necessari comunicar-se amb la base de dades, les sentències SQL que corresponguin, les quals s'han d'escriure seguint un determinat protocol, com per exemple, iniciar tota sentència SQL amb l'expressió `EXEC SQL`.

Una vegada es té el programa escrit en llenguatge C amb immersió de llenguatge SQL, es procedeix a efectuar una precompilació del font, amb un programa precompilador proporcionat normalment pel fabricant del SGBD, que efectua la traducció de totes les sentències SQL incloses dins el font en C a crides a la base de dades en llenguatge C. El protocol seguit per a escriure les sentències SQL (per exemple `EXEC SQL <sentència SQL>`) serveix per a que el programa precompilador detecti les instruccions que ha de traduïr. El resultat del programa precompilador és un programa font en C que ja no té cap sentència SQL.

Amb el fitxer font en C facilitat pel programa precompilador es procedeix a les fases normals de desenvolupament de qualsevol programa (compilació i enllaç). En la fase d'enllaç és normal haver d'utilitzar alguna llibreria proporcionada pel fabricant del SGBD ja que la traducció de les sentències SQL a llenguatge C possiblement hauran consistit en crides a funcions en llenguatge C que saben dialogar amb el SGBD.

La figura 5 visualitza el procés d'obtenció del programa executable a partir d'un fitxer font amb SQL hostatjat.

El nostre objectiu és efectuar una introducció en el desenvolupament de programes en C (llenguatge conegut per tots nosaltres) amb SQL hostatjat per accedir al SGBD *Oracle* i ho podrem practicar ja que *Oracle* ens facilita el precompilador per al llenguatge C i les llibreries per poder enllaçar els fitxers compilats en la plataforma *Visual C++* de *Microsoft*. Així doncs necessitarem aquesta plataforma per a poder aconseguir els programes executables.

Intentarem aconseguir el nostre objectiu tot comentant un programa desenvolupat en C amb SQL hostatjat. Es tracta d'un programa que permet afegir nous empleats a la taula `EMP` (esquema *empresa*), de manera que el número d'empleat és calculat automàticament a partir de sumar 10 al major número d'empleat existent. El programa demana, a l'usuari, les dades corresponents a cognom, ofici, salari i departament.

Trobareu el fitxer `u5n3p01.pc` en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

Exemple de programa en llenguatge C amb SQL hostatjat

```
/* Programa: u5n3p01.pc (Programa en llenguatge C amb SQL hostatjat)
   Descripció: Permet donar d'alta empleats (esquema empresa) als quals assigna com a codi el
               resultat del codi d'empleat més alt existent més 10.
   Autor: Isidre Guixà
*/

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <sqlda.h>
#include <sqlcpr.h>

int errrpt(void);
int llegir_cadena(char *, char *, int);
int llegir_enter(char *, int *);

EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR uid[80];                /* usuari que es connecta    */
   VARCHAR pwd[20];                /* contrasenya               */
   VARCHAR con[20];                /* cadena de connexió        */
   int     emp_no;
   VARCHAR cognom[11];
   int     dept_no;
   VARCHAR dnom[15];
   VARCHAR ofici[11];
   int     salari;

EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA.H;
```

```
void main()
{
/* Preparem variables i establim connexió amb la base de dades */

   uid.arr[0]='\0';
   do {printf ("Usuari.....: "); fflush(stdin); scanf ("%80[^\n]",uid.arr);}
   while (strlen(uid.arr)==0);
   uid.len = strlen (uid.arr);

   pwd.arr[0]='\0';
   do {printf ("Contrasenya: "); fflush(stdin); scanf ("%20[^\n]",pwd.arr);}
   while (strlen(pwd.arr)==0);
   pwd.len = strlen (pwd.arr);

   con.arr[0]='\0';
   do {printf ("Cadena.....: "); fflush(stdin); scanf ("%20[^\n]",con.arr);}
   while (strlen(con.arr)==0);
   con.len = strlen (con.arr);

   EXEC SQL WHENEVER SQLERROR GOTO errexit;
   EXEC SQL CONNECT :uid IDENTIFIED BY :pwd USING :con;

/* Cerquem el proper número d'empleat */

   EXEC SQL SELECT NVL(MAX(EMP_NO),0) + 10 INTO :emp_no FROM EMP;

/* Demanem les dades del nou empleat.
   L'entrada d'un nom en blanc indica finalització del procés.
   En l'entrada del departament comprovem la seva existència */

   for( ; ; emp_no+=10 )
   {
      int l;

      l = (short) llegir_cadena("Nom de l'empleat (en blanc per finalitzar) : ",
                                (char *)cognom.arr, 10);
      if ( l <= 0 ) break;

      cognom.len = (short) l;
      ofici.len = (short) llegir_cadena("Ofici de l'empleat: ", (char *)ofici.arr, 10);
      llegir_enter("Salari de l'empleat: ",&salari);

      for ( ; ; )
      {
         while ( llegir_enter("Codi de departament: ",&dept_no) < 0 );

         EXEC SQL WHENEVER NOT FOUND GOTO nodept;
         EXEC SQL SELECT DNOM
                    INTO :dnom
                    FROM DEPT
                    WHERE DEPT_NO = :dept_no;

         dnom.arr[dnom.len] = '\0';

         /* En aquest punt, tenim les dades correctes d'un empleat que
            hem d'afegir a la base de dades */

         EXEC SQL WHENEVER NOT FOUND STOP;

         EXEC SQL INSERT INTO EMP(EMP_NO,COGNOM,OFICI,SALARI,DEPT_NO)
                    VALUES (:emp_no,:cognom,:ofici,:salari,:dept_no);

         printf("\n%s afegit al departament %s com empleat número %d\n",
                 cognom.arr,dnom.arr,emp_no);
         break;

         nodept:
       printf("%d %s %d",sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc, sqlca.sqlerrm.sqlerrml);
            printf("\nNo existeix un tal departament.\n");
      }
      continue;
   }

   EXEC SQL COMMIT WORK RELEASE;
   printf ("\nFinal de l'entrada d'empleats.\n");
   return;

errexit:
   errrpt();
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL ROLLBACK WORK RELEASE;
   return;
}
```

```
int llegir_enter(char text[],int *variable)
/* Retorna -1 si no s'ha efectuat cap lectura
   i 0 si tot ha estat correcte */
{
   unsigned int i;
   printf(text);
   fflush(stdin);
   i=scanf("%d",variable);
   fflush(stdin);
   if ( i==0 ) return -1;
   return 0;
}

int llegir_cadena(char text[], char variable[], int llargada)
/* Retorna la longitud de la cadena llegida */
{
   char s[80];
   printf(text);
   fflush(stdin);
   gets(s);
   strncpy(variable, s, llargada); variable[llargada]='\0';
   return strlen(variable);
}

errrpt()
/* Visualitza el codi i missatge de l'error Oracle */
{
   printf("%.70s (%d)\n", sqlca.sqlerrm.sqlerrmc, -sqlca.sqlcode);
   return(0);
}
```

Per a obtenir el programa executable a partir de fitxers fonts en llenguatge C amb SQL hostatjat, vegeu en la secció "Recursos de continguts" del web d'aquest crèdit "Implementació de programes en C amb SQL hostatjat"

En aquest moment, amb els coneixements que tenim de PL/SQL i de llenguatge C estem en condicions d'efectuar una lectura força entenedora del programa. Efectuem aquesta lectura, procedim a executar els procediments per assolir el programa executable, l'executem i en comprovem el seu funcionament. Una vegada fet tot això n'efectuem l'estudi detallat. En el que segueix, farem referència al font anterior, que conté sentències SQL, i també al corresponent fitxer `.C` resultat de la precompilació.

D'entrada fixem-nos que les sentències SQL es distingeixen immediatament de les del llenguatge amfitrió per la utilització obligada de la partícula `EXEC SQL` com a prefix de la sentència.

En el programa anterior s'observa que totes les sentències SQL han estat escrites en majúscules. No hi ha cap obligatorietat al respecte. Únicament les hem escrit en majúscules per diferenciar-les, encara més, de les paraules reservades en el llenguatge C que, com molt be sabeu, acostumen a ser en minúscules obligatòriament.

El llenguatge SQL hostatjat disposa de dos tipus de sentències: **declaratives** i **executables**.

Les sentències **executables** són molt similars a les existents en SQL autosuficient. Les sentències **declaratives** són les que permeten definir variables i objectes que seran utilitzats per sentències SQL executables.

Totes les variables que hagin de ser utilitzades en sentències SQL (observem que s'utilitza la sintaxis `:<nom_variable>`) han d'estar

definides dins la secció declarativa de SQL, que apareix delimitada per les sentències:

```
BEGIN DECLARE SECTION;
END DECLARE SECTION;
```

Les dades declarades en la secció declarativa han de ser correctament interpretades pel llenguatge amfitrió que correspongui (C en aquest cas). Oi que en C no existeix el tipus VARCHAR?

VARCHAR és un tipus que es defineix en C a partir d'una estructura. Si observem el resultat de la precompilació del programa anterior, veiem que en l'arxiu `.c` hi apareixen les declaracions:

```
typedef struct { unsigned short len; unsigned char arr[1]; } VARCHAR;
typedef struct { unsigned short len; unsigned char arr[1]; } varchar;
```

Com es dedueix, tan es pot utilitzar el tipus VARCHAR com el tipus `varchar`.

Per cadascuna de les declaracions VARCHAR del programa, apareixen:

```
struct { unsigned short len; unsigned char arr[80]; } uid;
struct { unsigned short len; unsigned char arr[20]; } pwd;
struct { unsigned short len; unsigned char arr[20]; } con;
struct { unsigned short len; unsigned char arr[11]; } cognom;
struct { unsigned short len; unsigned char arr[15]; } dnom;
struct { unsigned short len; unsigned char arr[11]; } ofici;
```

La utilització de les variables VARCHAR no és obligatòria. Podem utilitzar les típiques `char[]` i `char *` del llenguatge C. L'avantatja d'utilitzar VARCHAR radica en que *Oracle* omple automàticament el camp `len` en les instruccions SELECT i FETCH i podem utilitzar aquest camp per fer coses semblants a afegir el caracter `'\0'` al final de la cadena (*Oracle* no li deixa).

La longitud a especificar en una declaració VARCHAR ha d'estar obligatòriament entre 1 i 65535. Així, la sentència següent és incorrecta:

```
VARCHAR cadena_nula[];
```

El tipus VARCHAR es pot tractar com un tipus del C, però hi ha algunes excepcions, com el fet de no poder-lo utilitzar per una altra declaració de tipus. Així, la següent declaració provoca errors de compilació:

```
typedef VARCHAR buffer[64];
```

L'arxiu `sqlca.h` incorpora les definicions necessàries per poder recuperar els errors *Oracle* des del programa. La seva inclusió es pot efectuar de dues maneres equivalents:

```
EXEC SQL INCLUDE SQLCA.H;
```

```
#include <sqlca.h>
```

Podeu mirar, amb deteniment, el contingut d'aquest arxiu (les declaracions que aporta). Ens interessa saber que tenim accés als errors *Oracle* mitjançant:

```
sqlca.sqlerrm.sqlerrmc /* Missatge d'error fins a 70 caràcters */
sqlca.sqlcode          /* Codi d'error Oracle */
```

Després de cada sentència SQL caldria analitzar l'estat d'error (amb les anteriors variables) i actuar en conseqüència. El llenguatge SQL hostatjat aporta la possibilitat d'efectuar un tractament d'excepcions semblant al que aporta el llenguatge PL/SQL. Hi ha, però, una clara diferència: en PL/SQL el tractament d'excepcions és obligatori, mentre que en SQL hostatjat és pot simular.

La sentència que SQL hostatjat aporta per efectuar el tractament d'errors és:

```
EXEC SQL WHENEVER <tipus_error> <acció>;
```

- `<tipus_error>` pot ser:
  - `SQLERROR`, que s'avalua com a certa si s'ha produït error
  - `NOT FOUND`, que té el mateix efecte que l'excepció predefinida `NO_DATA_FOUND` de PL/SQL.
- `<acció>` pot ser:
  - `CONTINIUE`, que indica que cal passar a la instrucció següent.
  - `GOTO <etiqueta>`, que provoca un salt incondicional al punt del programa que indica `<etiqueta>`.
  - `DO <funció>`, que provoca l'execució de la funció continuant el programa posteriorment.
  - `STOP`, que provoca l'avortament del programa.

La influència de la sentència WHENEVER és posicional i no lògica, és a dir, una sentència WHENEVER s'aplica a totes les línies que estan situades físicament després d'ella en el programa font i no necessàriament a totes les sentències SQL que es trobin després d'ella en el flux d'execució del programa. Això és degut a que WHENEVER no és una veritable sentència, sinó una directiva pel precompilador que provoca, que tota sentència EXEC SQL que trobi a continuació (físicament, és clar) sigui traduïda afegint el tractament que correspongui segons el contingut de la psudosentència WHENEVER. Exemplifiquem-ho:

Observem, en el codi .c resultat de la precompilació del programa anterior, que la pseudosentència

```
EXEC SQL WHENEVER NOT FOUND GOTO nodept;
```

provoca que, al final de la traducció de la sentència

```
EXEC SQL SELECT DNOM INTO :dnom FROM DEPT WHERE DEPT_NO = :dept_no;
```

aparegui:

```
...
if (sqlca.sqlcode == 1403) goto nodept; /* (1) */
if (sqlca.sqlcode < 0) goto errexit;
```

Posteriorment, la pseudosentència

```
EXEC SQL WHENEVER NOT FOUND STOP; /* (2) */
```

provoca que, al final de la traducció de la sentència

```
EXEC SQL INSERT INTO EMP(EMP_NO,COGNOM,OFICI,SALARI,DEPT_NO)
         VALUES (:emp_no,:cognom,:ofici,:salari,:dept_no);
```

aparegui:

```
...
if (sqlca.sqlcode == 1403) exit(1); /* (3) */
if (sqlca.sqlcode < 0) goto errexit;
```

La inexistència de la pseudosentència (2), hagués provocat que la instrucció (3) no hagués existit i en el seu lloc hi hauria una còpia de la instrucció (1).

No comentem la resta del programa ja que la seva comprensió ha de ser fàcil pel lector.

Per finalitzar aquesta petita introducció al llenguatge SQL hostatjat, comentar que a l'igual que el llenguatge PL/SQL, també facilita l'objecte CURSOR per la gestió de les diferents files resultants d'una sentència SELECT. Per la seva gestió cal saber, bàsicament:

– Els cursors es declaren, fora de la secció declarativa, amb la sentència:

```
EXEC SQL DECLARE <nom_cursor> CURSOR FOR <sentència_SELECT>;
```

– L'obertura d'un cursor s'executa amb la sentència:

```
EXEC SQL OPEN <nom_cursor>;
```

– El tancament d'un cursor s'executa amb la sentència:

```
EXEC SQL CLOSE <nom_cursor>;
```

– La lectura de les diferents files resultants del cursor s'efectura amb la sentència:

```
EXEC SQL FETCH <nom_cursor> INTO <llista_variables>;
```

Exemple d'utilització de cursors en SQL hostatjat

Desenvolupem un programa per calcular la suma del salari dels empleats de l'empresa, sense utilitzar la funció agregada SUM.

Trobareu el fitxer u5n3p02.pc en el contingut "Codi dels scripts en sql desenvolupats en material paper" de la web d'aquest crèdit.

```
/* Programa: u5n3p02.pc (Programa en llenguatge C amb SQL hostatjat)
   Descripció: Programa que calcula, a l'esquema empresa, la suma dels salaris dels empleats.
   Autor: Isidre Guixà
*/
#include <string.h>
#include <stdio.h>
#include <sqlca.h>
#include <sqlda.h>
#include <sqlcpr.h>

EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR uid[80];
   VARCHAR pwd[20];
   VARCHAR con[20];
   long v_sal;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE S CURSOR FOR
   SELECT salari FROM emp;

errrpt()
{
   printf("%.70s (%d)\n", sqlca.sqlerrm.sqlerrmc, -sqlca.sqlcode);
   return(0);
}

void main()
{
   long salari=0;

/* Preparem variables i establim connexió amb la base de dades */

   uid.arr[0]='\0';
   do {printf ("Usuari.....: "); fflush(stdin); scanf ("%80[^\n]",uid.arr);}
   while (strlen(uid.arr)==0);
   uid.len = strlen (uid.arr);
```

```
   pwd.arr[0]='\0';
   do {printf ("Contrasenya: "); fflush(stdin); scanf ("%20[^\n]",pwd.arr);}
   while (strlen(pwd.arr)==0);
   pwd.len = strlen (pwd.arr);

   con.arr[0]='\0';
   do {printf ("Cadena.....: "); fflush(stdin); scanf ("%20[^\n]",con.arr);}
   while (strlen(con.arr)==0);
   con.len = strlen (con.arr);

   EXEC SQL WHENEVER SQLERROR GOTO errexit;
   EXEC SQL CONNECT :uid IDENTIFIED BY :pwd USING :con;

   EXEC SQL OPEN S;
   EXEC SQL FETCH S INTO :v_sal;
   while (sqlca.sqlcode==0)
   {
      salari = salari + v_sal;
      EXEC SQL FETCH S INTO :v_sal;
   }
   EXEC SQL CLOSE S;

   printf ("La suma del salari és %ld\n",salari);
   return;

errexit:
   errrpt();
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL ROLLBACK WORK RELEASE;
   return;
}
```