Bases de dades objecte-relacionals

Aina del Tio Esteve

Índex

Introducció

Les bases de dades objecte-relacionals agrupen les particularitats de les bases de dades relacionals i les bases de dades orientades a objectes. Entre altres característiques, aquestes base de dades:

1. estableixen interconnexions entre les dades per mitjà de les relacions,
2. encapsulen les dades emmagatzemant-les en estructures complexes i protegint-les d'accessos il·lícits per part d'usuaris externs,
3. estenen la funcionalitat d'unes estructures de dades a unes altres fomentant la reutilització, i
4. permeten crear nous tipus d'usuari proporcionant flexibilitat.

Per desenvolupar les funcionalitats anteriors farem un recorregut al llarg de dos llenguatges. Un d'ells, el **llenguatge de definició de dades** (*data definition language,* DDL) permet a l'usuari definir tant estructures de dades com funcions o procediments que permetran consultar-les; l'altre, el **llenguatge de manipulació de dades** (*data manipulation language,* DML) permet a l'usuari dur a terme tasques de consulta o manipulació de dades.

Hi ha gestors de bases de dades objecte-relacionals que faciliten l'administració i gestió de les bases de dades. En aquesta unitat treballarem amb el **PostgreSQL** pel conjunt de funcionalitats avançades que suporta i pel seu funcionament multiplataforma. Cal tenir en compte que la migració de bases de dades allotjades en productes comercials a PostgreSQL es facilita gràcies al fet que suporta àmpliament l'estàndard SQL.

La versió del PostgreSQL que s'ha utilitzat durant la redacció d'aquest material, i en els exemples, és la 9.1, última versió estable en aquest moment.

Resultats d'aprenentatge

En finalitzar aquesta unitat l'alumne/a:

1. Gestiona la informació emmagatzemada en bases de dades d'objectes relacionals, avaluant i utilitzant les possibilitats que proporciona el sistema gestor.

- Descriu les característiques de les bases de dades objecte-relacionals.
- Crea tipus de dades objecte, els seus atributs i mètodes.
- Crea taules d'objectes i taules de columnes tipus objecte.
- Crea tipus de dades col·lecció.
- Realitza consultes.
- Modifica la informació emmagatzemada mantenint la integritat i consistència de les dades.

1. Bases de dades d'objectes relacionals

El terme *base de dades d'objectes relacionals* (BDOR) s'utilitza per descriure una base de dades que ha evolucionat des del model relacional cap a una altra més amplia que incorpora conceptes del paradigma orientat a objectes. Per tant, un**sistema de gestió d'objectes relacional**, *object database management system* (**ODBMS)**, conté totes dues tecnologies: la relacional i la d'objectes.

Un paradigma és una manera de representar i manipular el coneixement. Representa un enfocament particular o filosofia per a la construcció del programari. No és millor un que un altre, sinó que cada un té avantatges i desavantatges.

Les bases de dades **relacionals** han estat el model utilitzat fins a l'actualitat per modelitzar problemes reals i administrar dades dinàmicament. Compleixen el model relacional, i la seva idea fonamental és la utilització de relacions. Aquestes permeten establir interconnexions entre les dades i treballar-hi amb conjuntament.

Les bases de dades **orientades a objectes** tracten d'emmagatzemar dades d'objectes complets. Quan s'integren les característiques d'una base de dades amb les d'un llenguatge de programació orientat a objectes, el resultat és un ODBMS que estén els llenguatges amb dades persistents de manera transparent, control de concurrència, recuperació de dades i consultes associatives.

Les bases de dades orientades a objectes tracten d'emmagatzemar dades d'objectes complets.

Els sistemes de gestió d'objectes relacional, ODBMS, són una bona elecció per als sistemes que necessitin un bon rendiment en la manipulació de tipus de dades complexes. En relació amb altres sistemes de gestió, proporcionen els costos més baixos de desenvolupament i també el millor rendiment quan s'utilitzen estructures d'objectes, gràcies al fet que el model d'objecte utilitzat en l'aplicació s'emmagatzema exactament en el disc, de manera que té una integració transparent entre la base de dades i el programa escrit en el llenguatge de programació orientat a objectes. En emmagatzemar exactament el model d'objecte utilitzat en l'aplicació, els costos de desenvolupament i manteniment es redueixen.

1.1 Característiques de les bases de dades objecte-relacional

Una idea bàsica de les bases de dades objecte-relacional (BDOR) és que l'usuari pugui crear els seus propis tipus de dades, per ser utilitzats en la tecnologia que permeti la implementació de tipus de dades predefinits. A més, també permeten crear mètodes per a aquests tipus de dades, proporcionant flexibilitat i seguretat.

Un **objecte** és una instància d'un tipus objecte. Un **tipus objecte** és un paquet cohesionat que consisteix en un tipus concret de dada. S'usa per agrupar camps relacionats i mètodes, i descriu les regles que marquen el comportament de l'objecte.

Els sistemes gestors de BDOR són compatibles en sentit ascendent amb les bases de dades relacionals tradicionals, és a dir, es poden exportar les aplicacions sobre bases de dades relacionals al nou model sense que calgui reescriure-les.

Els **conceptes principals** de les BDOR són l'encapsulació, l'herència i el polimorfisme.

Seguidament veurem els conceptes principals:

1. **L'encapsulació** és l'ocultació de les dades d'un objecte de manera que només es poden modificar mitjançant les operacions definides per aquest objecte. L'aïllament protegeix les dades associades a un objecte de la modificació o eliminació sense permís per part d'un usuari extern.

2. **L'herència** és la relació entre un tipus objecte general i un altre de més especific. Un tipus objecte es deriva de l'altre estenent la seva funcionalitat, essent el mecanisme fonamental per implementar el polimorfisme i la reutilització.

3. **El polimorfisme** fa referència a la sobreescriptura de mètodes a l'hora d'implementar un tipus objecte utilitzant l'herència. No s'ha de confondre amb la sobrecàrrega de funcions que permeten altres llenguatges de programació.

La sobrecàrrega és la característica que permet tenir, en un mateix tipus objecte, diferents mètodes amb el mateix nom, mentre que la sobreescriptura (polimorfisme) és la característica que permet canviar la implementació d'un mètode, heretat del tipus objecte base, en el tipus objecte derivat.

1.1.1 Característiques del PostgreSQL

El PostgreSQL és un gestor de bases de dades relacionals amb suport per a objectes, que ens facilita l'administració i gestió de les bases de dades. Com que és un gestor que funciona àmpliament amb l'estàndard SQL, facilita la migració de dades allotjades en productes comercials.

Història

El PostgreSQL s'inicia el 1986 amb un projecte del professor Michael Stonebraker i un equip de desenvolupadors de la Universitat de Berkeley (Califòrnia). El nom original va ser Postgres. En el seu disseny es van incloure alguns conceptes avançats en bases de dades i suport parcial a l'orientació a objectes.

El Postgres va ser comercialitzat per Illustra, una empresa que posteriorment va formar part d'Informix. Va arribar un moment en què mantenir el projecte absorbia massa temps als investigadors i acadèmics, per la qual cosa el 1993 es va presentar la versió 4.5 i oficialment es va donar per acabat el projecte.

El 1994, Andrew Yu i Jolly Chen van incloure SQL a Postgres per posteriorment presentar-ne el codi al Web amb el nom de Postgres95. El projecte incloïa molts canvis en el codi original que en milloraven el rendiment i legibilitat.

El 1996 el nom va canviar a PostgreSQL reprenent la seqüència original de versions, per la qual cosa es va presentar la versió 6.0. L'última versió estable oficial, de 2004, és la 7.4.6, mentre que la versió 8.0 està en fase final d'estabilització.

Llicència BSD

És la llicència de programari principalment per als sistemes BSD (Berkeley Software Distribution), que pertany al grup de llicències de programari lliure. Aquesta té menys restriccions comparada amb d'altres com la GPL (General Public License), i és més propera al domini públic. El projecte PostgreSQL continua publicant versions principals anualment, com també versions menors de correcció d'errors (*bugs*), totes disponibles sota la llicència BSD, i basades majorment en contribucions de proveïdors comercials, empreses que hi donen suport i programadors de codi obert.

Prestacions

PostgreSQL destaca per l'àmplia llista de prestacions que el fan capaç de competir amb qualsevol sistema gestor de bases de dades comercial:

1. Esta desenvolupat en C, amb eines com Yacc i Lex.
2. Disposa d'un conjunt de tipus de dades, que en permet l'extensió mitjançant tipus i operadors definits i programats per l'usuari.
3. L'API d'accés a l'SGBD (Sistema Gestor de Bases de Dades) es troba disponible en C, C++, Java, Perl, PHP, Python i TCL, entre altres.
4. L'administració es basa en usuaris i privilegis.
5. Les seves opcions de connectivitat abasteixen TCP/IP, sòcols UNIX i sòcols NT.
6. És altament segur en relació amb l'estabilitat.
7. Es pot estendre amb biblioteques externes per suportar encriptació, cerques per similitud fonètica, etc.
8. Controla la concurrència multiversió, millorant sensiblement les operacions de bloqueig i transaccions en sistemes multiusuari.
9. Inclou suport per a vistes, claus foranes, integritat referencial, disparadors, procediments emmagatzemats, subconsultes i la majoria dels tipus i operadors suportats a SQL92 i SQL99.

   Limitacions

Trobareu més informació sobre l'ús de les transaccions, punts de recuperació i *tablespaces* en l'apartat "DDL i DML de les bases de dades objecte-relacionals" d'aquesta unitat.

Les limitacions d'aquest tipus de gestor de bases de dades objecte-relacionals se solen identificar en analitzar les prestacions que hi ha previstes per a les pròximes versions. Actualment, les limitacions principals són:

1. Punts de recuperació dins de transaccions. Actualment, les transaccions avorten completament si es troba un error durant l'execució. La definició de punts de recuperació permetria recuperar millor transaccions complexes.
2. No suporta *tablespaces* per definir on s'ha d'emmagatzemar la base de dades, l'esquema, els índexs, etc.
3. El suport a l'orientació a objectes és una extensió simple que ofereix prestacions com l'herència, però no dóna un suport complet.

1.1.2 El model orientat a objectes

En els llenguatges de programació orientats a objectes, els tipus objecte s'anomenen **classes**.

PostgreSQL inclou extensions d'orientació a objectes, tot i que no és un sistema gestor de bases de dades orientat a objectes complet. Els conceptes següents el relacionen amb aquest paradigma:

1. Objectes
2. Propietats
3. Encapsulació
4. Herència
5. Persistència
6. Adaptació a un sistema gestor de bases de dades

Objectes

En el món que ens envolta hi ha diferents objectes d'un mateix tipus -també podríem dir d'una mateixa classe. Per exemple, hi pot haver més d'un cotxe del mateix tipus. Si utilitzem la terminologia d'orientació a objectes, direm que el nostre cotxe és una instància del tipus objecte *cotxes*.

Un **objecte** és un tipus de dades que encapsula les dades necessàries i les funcions per accedir-hi.

Tots els cotxes tenen atributs (color, motor, potència) i mètodes (accelerar, frenar); tot i així cada cotxe tindrà un estat particular independent de la resta. Per això, podem dir que cada objecte té **identitat** pròpia: encara que dos objectes tinguin els mateixos valors seran entitats diferents. En els models orientats a objectes, la identitat es representa amb l'identificador d'objecte (*object identifier,* OID), que és únic per a cada objecte.

Els OID s'encarreguen de fer referència a un objecte des d'un altre; d'aquesta manera es creen les relacions entre objectes.

Un **tipus objecte** o classe defineix les característiques (atributs) i els comportaments (mètodes) que són comuns per a tots els objectes d'un cert tipus.

Tots els objectes que comparteixen les mateixes propietats pertanyen al mateix tipus objecte. Per mitjà d'aquest es defineixen les propietats dels objectes i s'utilitza com a plantilla per crear-los. Anomenem *instància de classe* la utilització de la definició d'un tipus objecte per obtenir un determinat objecte. Com a concepte, la instància és equivalent a un *tuple* (en català *n*-pla) concret en una taula d'una base de dades.

Els elements fonamentals en l'orientació a objectes són els objectes i els tipus objecte.

Propietats

Les propietats dels objectes poden ser dinàmiques o estàtiques. Els **atributs** representen una propietat estàtica d'un objecte. Els **mètodes** representen una propietat dinàmica, és a dir, una modificació sobre un atribut o una acció que es pot efectuar.

El conjunt de valors dels atributs en un moment donat es coneix com a **estat de l'objecte**. Els operadors actuen sobre l'objecte canviant-ne l'estat. La seqüència d'estats pels quals passa un objecte en executar els mètodes en defineixen el **comportament**.

Les propietats dels objectes són el seu estat i el seu comportament.

Encapsulació

Per a l'usuari l'estructura interna dels objectes ha d'estar oculta; no necessita conèixer-la per interactuar amb aquests.

El mètode és la implementació d'una operació.

En la **interfície** d'un objecte se'n defineixen els atributs i els mètodes, i és la part pública de l'objecte. L'estructura interna d'un objecte s'anomena **implementació**.

L'encapsulació comporta els avantatges següents:

1. La modificació interna de la implementació d'un objecte per modificar-lo o millorar-lo no afecta l'usuari.
2. Se simplifica l'ús de l'objecte, ja que se n'oculten els detalls del funcionament i aquest es presenta en termes de les seves propietats. En elevar el nivell d'abstracció es disminueix el nivell de complexitat d'un sistema.
3. Constitueix un mecanisme d'integritat. La dispersió d'un error per tot el sistema és menor, ja que en presentar una divisió entre interfície i implementació, els errors interns d'un objecte troben la barrera de l'encapsulació abans de propagar-se per tot el sistema.
4. Permet substituir objectes amb la mateixa interfície i diferent implementació.

   S'han de diferenciar els mètodes de les operacions. Les operacions no són exclusives dels tipus d'objecte; en canvi, els mètodes sí.

Una **operació** especifica què s'ha de fer, i un **mètode** com s'ha de fer.

Aquesta diferència permet tenir múltiples mètodes per a una mateixa operació.

Herència

En aquest apartat ens referirem als tipus objecte com a **classes**, nomenclatura que s'utilitza en el paradigma de l'orientació a objectes.

L'**herència** es defineix com el mecanisme pel qual s'utilitza la definició d'una classe anomenada *pare* per definir un nova classe anomenada *fill* que en pot heretar els atributs i les operacions.

L'herència de tipus

En l'herència de tipus el que hereta la subclasse són els atributs de la superclasse, però no necessàriament la seva implementació, ja que els pot tornar a implementar.

Les classes *fill* també es coneixen com a **subclasses**, i a les classes *pare* com a **superclasses**. La relació d'herència entre classes genera el que s'anomena j*erarquia de classes*.

Podem trobar diferents tipus d'herència. Parlem d'**herència de tipus** quan la subclasse hereta la interfície d'una superclasse, és a dir, els atributs i els mètodes. Parlem d'**herència estructural** quan la subclasse hereta la implementació de la superclasse, és a dir, les variables d'instància i els mètodes.

Exemple d'herència de tipus

Un cavall és-un animal. Totes les propietats de la classe "animal" les té la classe "cavall". Però un animal no-és necessàriament un cavall. Totes les propietats de cavall no les tenen tots els animals.

L'herència de tipus defineix relacions **és-un** entre classes, en què la subclasse té totes les propietats de la superclasse, però la superclasse no té totes les propietats de la subclasse.

Considerem una variable que és de tipus Animal, en el llenguatge de programació Java:

```
public class Animal {

Integer: velocitat;

public void correr(){

this.velocitat += 5;

}

}

Animal elmeuanimal = new Animal();

System.out.println(elmeuanimal.correr());
```

Sortida: 5

Pot fer referència a objectes de tipus Animal, o a tipus derivats d'aquest, com gos, gat o iguana.

```
public class Iguana extends Animal {

public void correr(){

```

```
5  this.velocitat +=1;
6
7  }
8
9  }
10
11 elmeuanimal = new Iguana();
12
13 System.out.println(elmeuanimal.correr());
```

Sortida: 1

Es construeix una nova variable Iguana i s'hi fa referència com a Animal.

La propietat de substituir objectes que descendeixen de la mateixa superclasse, com ja s'ha vist anteriorment, es coneix com a *polimorfisme* i és el mecanisme de reutilització.

La referència al tipus Animal és una referència polimòrfica, ja que es pot referir a tipus derivats d'Animal. A partir d'una referència polimòrfica es poden sol·licitar operacions sense conèixer-ne el tipus exacte:

```
1  elmeuanimal.correr();
```

El mètode correr() té el mateix significat per a tots els tipus Animal. Com ja hem vist, cadascú utilitzarà un mètode diferent per executar la instrucció.

Una classe pot heretar les propietats de dos superclasses mitjançant el que es coneix com a *herència múltiple*.

En una herència múltiple, podem trobar que en les dues superclasses hi hagi propietats amb els mateixos noms: això s'anomena **col·lisió de noms**. Podríem trobar els casos següents:

1. Els noms són iguals perquè es refereixen a la mateixa propietat. No hi ha conflicte perquè és la mateixa, només s'ha de definir una implementació adequada.
2. Els noms són iguals però tenen significats diferents. Hi ha conflicte i es pot resoldre canviant els noms de les propietats heretades que el tinguessin.

Persistència

La persistència es defineix com la capacitat d'un objecte per sobreviure al temps d'execució d'un programa. Per implementar-la, s'utilitza l'emmagatzematge secundari.

S'han proposat diferents mecanismes per implementar la persistència en els llenguatges de programació, entre els quals destaquen els següents:

1. **Arxius nets**. Es creen arxius per emmagatzemar els objectes en el format que vol el programador. Els objectes es carreguen en obrir el programa i es desen en finalitzar. Aquesta és l'opció més accessible per a tots els llenguatges de programació.
2. **Bases de dades relacionals**. Els objectes són mapats a taules; un mòdul del programa s'encarrega de fer les transformacions objecte-relacionals.

Aquest enfocament consumeix molt de temps a l'hora de dur a terme el mapatge. Hi ha eines que fan els mapatges de manera automàtica.

3. **Bases d'objectes**. Els objectes s'emmagatzemen de manera natural en una base d'objectes, i la consulta i recuperació és administrada pel gestor; d'aquesta manera les aplicacions no necessiten saber els detalls de la implementació.

4. **Persistència transparent**. El sistema emmagatzema i recupera els objectes quan ho creu convenient, sense que l'usuari hagi de fer cap sol·licitud explícita de consulta, actualització o recuperació d'informació a una base d'objectes. No es requereix un altre llenguatge per interactuar amb les bases de dades.

Adaptació a un sistema gestor de base de dades

Un dels conceptes fonamentals en l'orientació a objectes és el concepte de classe. Hi ha dos enfocaments per associar el concepte de classe amb el model relacional:

1. Les classes defineixen **tipus de taules**.

2. Les classes defineixen **tipus de columnes**.

Atès que en el model relacional les columnes estan definides per tipus de dades, el més correcte és associar les columnes amb les classes (vegeu la taula 1.1).

TAULA 1.1. Relació entre enfocaments i conceptes

	1r enfocament	2n enfocament
Objectes	Valors	n-plens
Classes	Dominis	Taules

Les bases de dades objecte-relacionals permeten fer una migració gradual de sistemes relacionals als sistemes orientats a objectes i que tots dos tipus d'aplicacions coexisteixin. No és fàcil aconseguir la coexistència dels dos models de dades, i és necessari equilibrar els conceptes de cadascun dels models sense que entrin en conflicte.

PostgreSQL implementa els objectes com a n-plans i les classes com a taules. Tot i que també és poden definir nous tipus de dades mitjançant els mecanismes d'extensió.

1.1.3 Mapatge d'objectes relacional

El **mapatge d'objectes relacional** (o*bject-relational mapping,* ORM) és una tècnica de programació per convertir dades de llenguatges de programació orientats a objectes en la seva representació en bases de dades relacionals, mitjançant la definició de les correspondències entre els diferents sistemes.

La gestió de dades en programació orientada a objectes es fa mitjançant la **manipulació d'objectes**, que quasi sempre presenten valors no escalables.

Considerant l'exemple d'una entrada d'una llibreta d'adreces, que representa una persona amb zero o més adreces i zero o més telèfons, en orientació a objectes això es representaria com un objecte "persona", del qual penjarien les seves dades personals, les adreces i els telèfons -aquestes dues darreres dades també serien objectes. Les bases de dades relacionals només poden emmagatzemar valors escalars, com cadenes de caràcters o enters i organitzar-los en taules. El programador ha de convertir els valors representats en forma d'objectes en valors simples o agrupats per emmagatzemar en la base de dades i implementar el procés invers. Alternativament podria treballar només amb valors escalars, i perdria els avantatges de la programació orientada a objectes. Per poder gaudir de la potència d'aquesta i estalviar-se els processos de conversió d'objecte a registre de la base de dades i el seu invers, s'ha creat l'automatització d'aquest procés, mitjançant el mapatge d'objectes relacional.

En utilitzar bases de dades objecte-relacional, és a dir, bases de dades dissenyades específicament per treballar amb valors d'objectes, eliminem la necessitat de convertir dades en objectes i des de la seva forma SQL, ja que les dades s'emmagatzemen de manera automàtica en la seva representació original.

El **mapatge d'objectes relacional**(ORM) elimina la necessitat de convertir dades en objectes, ja que aquestes s'emmagatzemen en la seva representació original.

La majoria de les eines ORM permeten crear el model mitjançant l'esquema de la base de dades; és a dir, en crear la base de dades l'eina llegeix automàticament l'esquema de les taules i les relacions, i crea un model ajustat. Una vegada el model està creat, el desenvolupament és més ràpid, ja que hi ha una abstracció que permet obviar les consultes a la base de dades, i el sistema ORM genera de manera automàtica les consultes a la base de dades per convertir els registres en objectes.

La característica més important dels sistemes ORM és el llenguatge propi que ofereix per a les consultes a la base de dades. Un conjunt de tipus objecte i mètodes permet extreure les dades de la base de dades deixant de banda la sintaxi de l'SQL (*sttructured query language*) i utilitzar el propi llenguatge de l'eina.

1.2 Gestió servidor-client PostgreSQL

El PostgreSQL està basat en una arquitectura client-servidor, la qual cosa vol dir que hi ha una relació entre les entitats client i servidor. Un procés servidor pot atendre exclusivament un sol client, és a dir, calen tants processos servidor com clients hi hagi. L'encarregat d'executar un nou servidor per a cada client que sol·liciti una connexió és el procés *postmaster*.

1.2.1 Instal·lació del programari

Per instal·lar el programari en primer lloc cal que descarregueu el paquet d'instal·lació de la pàgina web oficial: http://www.postgresql.org/download/.

La distribució PostgreSQL oficial està disponible en diferents formats binaris i en codi font. El paquet complet inclou:

1. El servidor PostgreSQL amb documentació completa: html, man.
2. Diverses eines de línia d'ordres: psql, pg_ctl, pg_dump, pg_restore, etc.
3. Biblioteca en C, libpq; i processador C incorporat, ecpg.
4. Nombrosos llenguatges per a funcions/procediments emmagatzemats: plpgsql, pltcl, plperl, plpython.
5. Nombrosos paquets addicionals: metaphone, pgcrypto, etc.

Una vegada tinguem el paquet descarregat, procedirem a la seva instal·lació i posterior execució.

1.2.2 Accés al servidor PostgreSQL

Abans d'intentar connectar-nos amb el servidor, ens hem d'assegurar que està funcionant i admet connexions, tant locals (l'SGBD s'està executant a la mateixa màquina que intenta la connexió) com remotes.

Una vegada comprovat el funcionament correcte del servidor, hem de disposar de les credencials necessàries per connectar-nos-hi. Per simplificar, suposarem que disposem de les credencials de l'administrador de la base de dades, que normalment són l'usuari PostgreSQL i la seva contrasenya.

El client psql

Per connectar-se amb un servidor es requereix, lògicament, un programa client. En la distribució de PostgreSQL s'inclou un client, psql, fàcil d'utilitzar, que permet la introducció interactiva d'ordres en mode text.

Per fer una connexió es requereixen les dades següents:

1. Servidor. Si no s'especifica, s'utilitza *localhost*.
2. Usuari. Si no s'especifica, s'utilitza el nom d'usuari UNIX que executa psql.
3. Base de dades.

Executem en el terminal l'SQL Shell psql que proporciona el programari i introduïm les dades que ens demana; si no n'especifiquem cap el client psql utilitza els valors per defecte (els que apareixen entre claudàtors):

```
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Password for user postgres: *****
psql (9.1.1)
Type "help" for help.
postgres=#
```

El símbol `#` significa que el psql està llest per llegir l'entrada de l'usuari.

Les sentències SQL s'envien directament al servidor per tal que les interpreti. Les ordres internes tenen la forma `\ordre` i ofereixen opcions que no estan incloses en l'SQL i que són interpretades internament pel psql.

Per acabar la sessió amb psql, utilitzem l'ordre `\q` o podem polsar les tecles Ctrl + D.

Introducció de sentències

Les sentències SQL que escriguem al client hauran d'acabar amb `;` o bé amb `\g`:

```
postgres=# select user;
current_user
--------------
postgres
(1 row)
```

Quan una ordre ocupa més d'una línia, l'indicador canvia de forma i va assenyalant l'element que encara no s'ha completat:

```
postgres=# select
postgres-# user\g
current_user
--------------
postgres
(1 row)
```

TAULA 1.2. Indicadors de PostgreSQL

Indicador	Significat
`=#`	Espera una nova sentència.
`-#`	La sentència encara no s'ha acabat amb `;` o `\g`.
`‘‘#`	Una cadena en cometes dobles no s'ha tancat.
`’#`	Una cadena en cometes simples no s'ha tancat.
`(#`	Un parèntesi no s'ha tancat.

El client psql emmagatzema la sentència fins que se li dóna l'ordre d'enviar-la a l'SGBD. Per visualitzar el contingut de la memòria intermèdia (*buffer*) en què ha emmagatzemat la sentència, disposem de l'ordre `\p`:

```
postgres=# select
postgres-# 2 * 10 + 1
postgres-# \p
select
2 * 10 + 1
postgres-# \g
?column?
----------
21
(1 row)
```

El client també disposa d'una ordre que permet esborrar completament la memòria intermèdia per començar de nou amb la sentència:

```
postgres=# select 'Hola'\r
Query buffer reset (cleared).
```

Expressions i variables

El client psql disposa de multitud de prestacions avançades, entre les quals (com ja hem comentat) hi ha el suport per a la substitució de variables, similar al dels *shells* de l'UNIX:

```
postgres=# \set var1 demostracio
```

Aquesta sentència crea la variable `var1` i li assigna el valor `demostració`. Per recuperar el valor de la variable, simplement l'haurem d'incloure precedida de `:` en qualsevol sentència o bé veure'n el valor mitjançant l'ordre `echo`:

```
postgres=# \echo :var1
demostracio
```

De la mateixa manera, psql defineix algunes variables especials que poden ser útils per conèixer detalls del servidor al qual estem connectats:

```
postgres=# \echo :DBNAME :ENCODING :HOST :PORT :USER;
postgres UTF8 localhost 5432 postgres ;
```

Procés per lots i formats de sortida

El psql pot processar ordres per lots emmagatzemats en un arxiu del sistema operatiu mitjançant la sintaxi següent:

```
$ psql postgres –f arxiu.psql
```

El mateix intèrpret psql ens proporciona mecanismes per emmagatzemar en un fitxer el resultat de les sentències:

1. Especificant el fitxer de destinació directament en finalitzar una sentència:

```
postgres=# select user \g /tmp/a.txt
```

2. Mitjançant una *pipe* a través de la qual enviem la sortida a una ordre UNIX:

```
postgres=# select user \g | cat > /tmp/b.txt
```

Pipe

És un dispositiu lògic destinat a comunicar processos.El seu funcionament és el d'una cua de caràcters, amb una longitud fixa en què els processos poden llegir i escriure.

3. Mitjançant l'ordre `\o`, que permet indicar quina ha de ser la sortida de les sentències SQL que s'executin en endavant:

```
postgres=# \o /tmp/sentencies.txt
postgres=# select user;
postgres=# select 1+1+4;
postgres=# \o
postgres=# select 1+1+4;
?column?
----------
6
(1 row)
```

Notació

A l'ordre \o se li ha d'especificar un fitxer o bé una ordre que anirà rebent els resultats mitjançant una pipe.Quan es vulgui tornar a la sortida estàndard (stdout) simplement es donarà l'ordre \o sense cap paràmetre.

4. Es pot especificar el format de sortida dels resultats d'una sentència –per defecte, psql els mostra en forma tabular mitjançant text. Per a canviar-lo s'ha de modificar el valor de la variable interna `format` mitjançant l'ordre `\pset`. En especificar que es vol la sortida en HTML, es redirigeix a un fitxer i es genera un arxiu HTML que permet veure el resultat de la consulta mitjançant un navegador web convencional.Vegem, en primer lloc, l'especificació del format de sortida:

```
postgres=# \pset format html
Output format is html.
postgres=# select user;
<table border="1">
<tr>
<th align="center">current_user</th>
</tr>
<tr valign="top">
<td align="left">postgres</td>
</tr>
</table>
<p>(1 row)<br />
</p>
```

Hi ha d'altres formats de sortida, com `aligned`, `unaligned`, `html` i `latex`. Per defecte, psql mostra el resultat en format `aligned`.

També tenim multitud de variables per ajustar els separadors entre columnes, el nombre de registres per pàgina, el separador entre registres, el títol de la pàgina HTML, etc. Vegem-ne un exemple:

```
postgres=# \pset format unaligned
Output format is unaligned.
postgres=# \pset fieldsep ','
Field separator is ",".
postgres=# select user, 1+2+3 as resultat;
current_user,resultat
postgres,6
(1 row)
```

Per a poder posar en pràctica els exemples de la resta d'aquest apartat, s'ha de processar el contingut del fitxer *arxiu.psql* tal com es transcriu a continuació.

> Des del terminal, situats en el directori en què hàgiu creat el fitxer, amb l'usuari per defecte postgres, executeu l'ordre `$ psql postgres -f arxiu.psql`.

El contingut del fitxer *arxiu.psql* és el següent:

```
--drop table productes;
--drop table proveidors;
--drop table preus;
--drop table guany;
create table productes (
partvarchar(20),
tipusvarchar(20),
especificacio varchar(20),
psuggeritfloat(6),
clauserial,
primary key(clau)
);
insert into productes (part,tipus,especificacio,psuggerit) values ('Processador
    ','2 GHz','32 bits',null);
insert into productes (part,tipus,especificacio,psuggerit) values ('Processador
    ','2.4 GHz','32 bits',35);
insert into productes (part,tipus,especificacio,psuggerit) values ('Processador
    ','1.7 GHz','64 bits',205);
insert into productes (part,tipus,especificacio,psuggerit) values ('Processador
    ','3 GHz','64 bits',560);
insert into productes (part,tipus,especificacio,psuggerit) values ('RAM','128MB
    ','333 MHz',10);
insert into productes (part,tipus,especificacio,psuggerit) values ('RAM','256MB
    ','400 MHz',35);
insert into productes (part,tipus,especificacio,psuggerit) values ('Disc Dur
    ','80 GB','7200 rpm',60);
insert into productes (part,tipus,especificacio,psuggerit) values ('Disc Dur
    ','120 GB','7200 rpm',78);
insert into productes (part,tipus,especificacio,psuggerit) values ('Disc Dur
    ','200 GB','7200 rpm',110);
insert into productes (part,tipus,especificacio,psuggerit) values ('Disc Dur
    ','40 GB','4200 rpm',null);
insert into productes (part,tipus,especificacio,psuggerit) values ('Monitor
    ','1024x876','75 Hz',80);
insert into productes (part,tipus,especificacio,psuggerit) values ('Monitor
    ','1024x876','60 Hz',67);

create table proveidors (
empresa varchar(20) not null,
credit bool,
efectiubool,
primary key(empresa)
);

insert into proveidors (empresa,efectiu) values ('Tecno-k', true );
insert into proveidors (empresa,credit) values ('Patito', true );
```

```
insert into proveidors (empresa,credit,efectiu) values ('Nacional', true, true
    );

create table guany(
vendavarchar(16),
factordecimal (4,2),
primary key (venda)
);

insert into guany values('al engros',1.05);
insert into guany values('al detall',1.12);

create table preus (
empresavarchar(20) not null,
clauint not null,
preufloat(6),
primary key (empresa, clau),
foreign key (empresa) references proveidors,
foreign key (clau)references productes
);

insert into preus values ('Nacional',001,30.82);
insert into preus values ('Nacional',002,32.73);
insert into preus values ('Nacional',003,202.25);
insert into preus values ('Nacional',005,9.76);
insert into preus values ('Nacional',006,31.52);
insert into preus values ('Nacional',007,58.41);
insert into preus values ('Nacional',010,64.38);
insert into preus values ('Patito',001,30.40);
insert into preus values ('Patito',002,33.63);
insert into preus values ('Patito',003,195.59);
insert into preus values ('Patito',005,9.78);
insert into preus values ('Patito',006,32.44);
insert into preus values ('Patito',007,59.99);
insert into preus values ('Patito',010,62.02);
insert into preus values ('Tecno-k',003,198.34);
insert into preus values ('Tecno-k',005,9.27);
insert into preus values ('Tecno-k',006,34.85);
insert into preus values ('Tecno-k',007,59.95);
insert into preus values ('Tecno-k',010,61.22);
insert into preus values ('Tecno-k',012,62.29);
```

Gestió de la base de dades

L'ordre següent informa sobre les bases de dades existents en l'SGBD:

```
postgres=# \l
List of databases
Name | Owner | Encoding
-----------+-----------+----------
postgres | postgres | UTF8
template0 | postgres | UTF8
template1 | postgres | UTF8
(3 rows)
```

L'ordre `\c` permet connectar-se a una base de dades:

```
postgres=# \c postgres
You are now connected to database "postgres" as user "postgres".
```

La taula que conté la base de dades *postgres* es consulta mitjançant l'ordre `\d`:

```
postgres=# \d

List of relations

Schema | Name| Type| Owner

————————+————————————+————————+————————

public | guanys| table | postgres

public | preus| table | postgres

public | productes| table | postgres

public | productes_clau_seq | sequence | postgres

public | proveidors| table | postgres

(5 rows)
```

> L'ordre `\d` és útil per mostrar informació sobre l'SGBD: taules, índexs, objectes, variables, permisos, etc. Podeu obtenir totes les variants d'aquesta sentència introduint `\?` en l'intèrpret d'ordres.

Vegeu com es mostra una consulta de les columnes de cada una de les taules:

```
postgres=# \d proveidors

Table "public.proveidors"

Column | Type | Modifiers

————————+————————————+————————

empresa | character varying(20) | not null

credit | boolean |

efectiu | boolean |

Indexes:

"proveidors_pkey" PRIMARY KEY, btree (empresa)

Referenced by:

TABLE ""preus CONSTRAINT ""preus_empresa_fkey FOREIGN KEY (empresa) REFERENCES
    proveidors (empresa)
```

Per crear una nova base de dades, utilitzarem la sentència `create database`:

```
postgres=# create database prova;
```

Per eliminar una base de dades, utilitzarem la sentència `drop database`:

```
postgres=# drop database prova;
```

2. Objectes i col·leccions

El tipus objecte és un paquet cohesionat que descriu les regles que marquen el comportament de l'objecte. Disposa tant d'una interfície com d'una estructura; la primera descriu com es pot interactuar amb el tipus objecte, mentre que la segona defineix com s'emmagatzemen les dades a la base de dades. En el moment que es vol agrupar un nombre indefinit d'elements, si aquests elements són del mateix tipus, l'anomenem *col·lecció*.

2.1 Tipus de dades objecte

Els sistemes gestors de bases de dades inclouen un conjunt important de tipus de dades que s'adapten a moltes aplicacions i permeten als usuaris definir els seus propis tipus de dades. Els tipus d'usuari s'emmagatzemen en les files d'una taula i tenen un identificador únic (OID). Aquest identificador es pot utilitzar per referenciar altres tipus d'usuari i així representar relacions d'associació i d'agregació.

Un **tipus** defineix una estructura i un comportament comuns per a un conjunt de dades.

2.1.1 Estructura d'un tipus objecte

Un tipus objecte representa una entitat del món real i es compon dels elements següents:

1. Un **nom** que serveix per identificar el tipus objecte.
2. Uns **atributs** que modelitzen l'estructura i els valors de les dades del tipus. Cada atribut pot ser d'un tipus de dades **bàsic** o d'un tipus **d'usuari**.
3. Uns **mètodes** que són procediments o funcions escrits en el llenguatge PL/PgSQL (emmagatzemats en la BDOR), o escrits en un llenguatge orientat a objectes (emmagatzemats externament).

   L'estructura d'un tipus objecte consta de dues parts: especificació i cos.

L'**especificació** és la interfície del tipus objecte a les aplicacions. És on es declaren les estructures de dades o conjunt d'atributs i els mètodes necessaris per manipular les dades. El **cos** defineix els mètodes, és a dir, implementa l'especificació.

Els tipus objecte es poden interpretar com a plantilles per als objectes de cada tipus.

En la figura 2.1 podeu veure un exemple de definició de tipus de dades en pseudocodi, i com s'utilitza aquest tipus de dades per definir altres tipus objecte:

FIGURA 2.1. Definició de tipus de dades en pseudocodi

```
tipus Model
        atributs
                enter: potència;
                real: consum;
                real: cilindrada;
                real: acceleració;
fi Model.
tipus Vehicle
        atributs
                cadena_caràcters: marca;
                Model: model_vehicle;
                cadena_caràcters: color;
                enter: rodes;
        mètodes
                accelerar();
                frenar();
fi Vehicle.
```

En les especificacions es troba tota la informació que un client necessita per utilitzar els mètodes, d'aquí que es consideri una interfície operacional. És possible modificar el cos sense necessitat de modificar l'especificació i d'aquesta manera no afectar les aplicacions client.

Una de les característiques de l'orientació a objectes és que en una especificació de tipus objecte els atributs s'han de declarar abans que qualsevol dels mètodes. Per tant, si una especificació de tipus només declara atributs, el cos és innecessari perquè no hi ha mètodes a implementar. Tampoc no es poden declarar atributs en el cos del tipus objecte.

En l'especificació del tipus totes les declaracions són públiques, és a dir, visibles fora del tipus objecte. En canvi, el cos pot contenir declaracions privades, que defineixen mètodes necessaris per al funcionament intern del tipus objecte. L'àmbit de les declaracions privades és local en el cos de l'objecte.

En les **especificacions** del tipus objecte les declaracions són **públiques**, en canvi en el **cos** les declaracions són **privades**.

Per entendre aquesta estructura utilitzarem l'exemple de la figura 2.2, en què es defineix un tipus objecte i una sèrie d'operacions associades.

FIGURA 2.2. Exemple d'especificació d'un tipus objecte Rectangle

```
// Especificació dels atributs i mètodes
tipus Rectangle
        atributs
                enter: x;
                enter: y;
                enter: ample;
                enter: alt;
        mètodes
                calcularArea() -> enter;
                desplaçar(enter dx, enter dy);
fi Rectangle.
// Implementació dels mètodes
mètode calcularArea()
inici
        retorna(ample*alt);
fi
mètode desplaçar (enter dx, enter dy)
inici
        x <- x + dx;
        y <- y + dy;
fi
```

La implementació dels mètodes utilitza els atributs declarats dins el tipus objecte. En l'exemple anterior es calcula l'àrea del rectangle amb els valors dels atributs `ample` i `alt`, i el resultat es retorna com a paràmetre de sortida. En canvi, per calcular el desplaçament es modifiquen els atributs del tipus objecte amb els valors dels paràmetres d'entrada del mètode.

2.1.2 Components d'un tipus objecte

Un tipus objecte encapsula dades i operacions, per la qual cosa en l'especificació només es poden declarar atributs i mètodes, però no constants, excepcions, cursors o tipus. Com a mínim en un tipus objecte hi ha d'haver un atribut; pel que fa als mètodes, són opcionals.

Atributs

Un atribut es declara mitjançant un nom i un tipus. El nom ha de ser únic dins del tipus objecte, i el tipus pot ser qualsevol que suporti l'SGBD. Els tipus de dades bàsics són els següents: lògics, numèrics, de caràcters, de dates i de taules (vegeu la figura 2.3).

FIGURA 2.3. Tipus objecte amb diferents atributs

```
tipus Persona
        atributs
                cadena_caràcters: nom_cognom;
                enter: edat;
                data: data_naixement;
                cadena_caràcters: telèfon;
fi Persona.
```

El tipus objecte `Persona` es representa com un *tuple*, en què apareixen atributs de diferents tipus de dades.

Les estructures de dades poden arribar a ser molt complexes, fins al punt que el tipus d'un atribut pot ser un altre tipus objecte, anomenat *tipus objecte imbricat*. Això permet construir tipus d'objectes complexos a partir d'objectes simples. Alguns objectes com cues, llistes i arbres són dinàmics, creixen a mesura que s'utilitzen (vegeu la figura 2.4).

FIGURA 2.4. Tipus objecte complex Client

```
tipus Adreça
        atributs
                cadena_caràcters: carrer;
                cadena_caràcters: ciutat;
                cadena_caràcters: codi_postal;
fi Adreça.
tipus Client
        atributs
                enter: numeroCli;
                cadena_caràcters: nomCli;
                Adreça: adreçaCli;
        mètodes
                númeroClient() -> enter;
                dadesClient() -> cadena_caràcters;
fi Client.
```

El tipus objecte `Client` és un tipus objecte complex, ja que conté el tipus objecte `Adreça`com a atribut.En aquest cas podem veure un exemple de reutilització, ja que el tipus `Adreça` s'ha encapsulat com a tipus independent i aquest es pot utilitzar en la declaració de tipus.

Mètodes

Un **mètode** és un **subprograma declarat en una especificació de tipus**. El mètode no pot tenir el mateix nom que el tipus objecte ni el de cap dels seus atributs.

Els mètodes consten d'especificació i cos, i l'especificació consisteix en el nom del mètode, una llista opcional de paràmetres i, en el cas de les funcions, un tipus de retorn. Pel que fa al cos, el codi que conté s'executa per portar a terme una operació específica.

Per cada especificació de mètode hi ha d'haver el cos corresponent del mètode. En un tipus objecte, els mètodes poden fer referència als atributs i als altres mètodes sense qualificador.

Els mètodes es poden executar sobre els objectes del seu mateix tipus. Si `client_jove` és una variable PL/PgSQL que emmagatzema objectes del tipus `Client`, llavors `client_jove.numeroClient()` és un mètode que retorna el número del client emmagatzemat a `client_jove`.

Sobrecàrrega

Els mètodes del mateix tipus es poden sobrecarregar. Perquè això passi haurem d'utilitzar el mateix nom en diversos mètodes si els seus paràmetres formals són

diferents en número, ordre o tipus de dades. Quan s'invoca un dels mètodes, el PL/PgSQL troba el cos adequat comparant la llista de paràmetres actuals amb cadascuna de les llistes de paràmetres formals.

L'operació de sobrecàrrega no és possible en els casos següents:

1. Si els paràmetres formals es diferencien només en el mode.
2. Si les funcions només es diferencien en el tipus de retorn.

2.2 Definició de tipus d'objecte

Un cop sabem com volem emmagatzemar les dades, hem de crear l'estructura que les acollirà. Per realitzar aquest procés el PostgreSQL ens ofereix diferents tipus d'estructures: el tipus simple i el tipus compost. Aquestes estructures es poden utilitzar per definir el tipus d'un atribut dins d'una taula i per definir una taula d'objectes on els camps del tipus també són els camps de la taula.

Atès que el gestor de bases de dades seleccionat dóna suport parcial a orientació a objectes, hi ha alguns conceptes que no es podran desenvolupar, com la definició de mètodes en l'especificació d'un tipus.

2.2.1 Creació de tipus

PostgreSQL permet a l'usuari crear nous tipus de dades, el nom del tipus de les quals ha de ser diferent del nom de qualsevol tipus o domini que hi hagi en la base de dades. Preveu dos tipus de dades definides per l'usuari:

1. Un tipus de dades **simple**, per utilitzar en les definicions de columnes de les taules.
2. Un tipus de dades **compost**, per utilitzar com a tipus de retorn en les funcions definides per l'usuari.

   Ofereix les declaracions següents a l'hora de crear un tipus:

```
1  CREATE TYPE nom AS
2  ( nom_columnatipus [, ... ] )
3
4  CREATE TYPE nom (
5  INPUT = funcio_entrada,
6  OUTPUT = funcio_sortida
7  [ , RECEIVE = funcio_rebre ]
8  [ , SEND = funcio_enviament ]
9  [ , TYPMOD_IN = modificador_tipus_funcio_entrada ]
10 [ , TYPMOD_OUT = modificador_tipus_funcio_sortida ]
11 [ , ANALYZE = funcio_analitzar ]
12 [ , INTERNALLENGTH = { longitud_interna | VARIABLE } ]
13 [ , PASSEDBYVALUE ]
14 [ , ALIGNMENT = alineament ]
15 [ , STORAGE = emmagatzematge ]
16 [ , LIKE = tipus_com ]
17 [ , CATEGORY = categoria ]
18 [ , PREFERRED = preferit ]
```

```
[ , DEFAULT = per_defecte ]
[ , ELEMENT = element ]
[ , DELIMITER = delimitador ]
)
```

Els **tipus objecte** es poden definir com a **tipus simples** o com a**tipus compostos**.

Tipus simple

Per al tipus de dades simple, la definició és més complexa, ja que a les funcions se'ls ha d'especificar que tractaran amb aquest tipus. D'aquesta manera les funcions el podran utilitzar en operacions, assignacions, etc.

Definirem els tipus objecte com a tipus simples, indicant-los els atributs i definint els mètodes que implementaran les seves operacions.

Tractar amb el tipus de dades simple
Habitualment, les funcions que tractaran amb aquest tipus de dades s'escriuran en llenguatge C.

A tall d'exemple, crearem el tipus `nombre complex`, tal com fa la documentació de PostgreSQL. En primer lloc, hem de definir l'estructura en què emmagatzemarem el tipus:

```
TYPEDEF STRUCTComplex {
DOUBLEx;
DOUBLEy;
} Complex;
```

Després, les funcions que el rebran o tornaran:

```
PG_FUNCTION_INFO_V1(complex_in);
Datum
complex_in(PG_FUNCTION_ARGS) {
char*str = PG_GETARG_CSTRING(0);
double x, y;
Complex *result;

if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
ereport(ERROR,(errcode(ERRCODE_INVALID_TEXT_REPRESENTATION), errmsg"(invalid
    input syntax for complex: "\%s""\, str)));

result = (Complex *) palloc(sizeof(Complex));
result->x = x;
result->y = y;
PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_out);
Datum
complex_out(PG_FUNCTION_ARGS) {
Complex *complex = (Complex *) PG_GETARG_POINTER(0);
char*result;
result = (char *) palloc(100);
snprintf(result, 100, "(%g,%g)", complex->x, complex->y); PG_RETURN_CSTRING(
    result);
}
```

Ara estem en condicions de definir les funcions, i el tipus:

```
CREATE FUNCTION complex_in(cstring)
RETURNS complex
AS 'filename'
LANGUAGE c IMMUTABLE STRICT;
CREATE FUNCTION complex_out(complex)
RETURNS cstring
AS 'filename'
LANGUAGE c IMMUTABLE STRICT;
CREATE TYPE complex (
internallength = 16,
input = complex_in,
output = complex_out,
alignment = double
);
```

Tipus compost

La primera forma de `CREATE TYPE` crea un tipus compost. El tipus compost s'especifica mitjançant una llista de noms d'atributs i tipus de dades, i representa l'estructura d'una fila o registre. És el mateix que el tipus fila d'una taula, però evita crear taules dins la base de dades quan només es vol definir un tipus.

El PostgreSQL permet que els tipus compostos s'utilitzin com a tipus simples. Per exemple:

1. Una columna d'una taula es pot declarar com a tipus de dades.
2. En els mètodes o funcions es poden utilitzar com a argument o com a tipus de retorn de la funció.

Tipus d'objectes

Els tipus objecte es poden definir com a tipus simples o com a tipus compostos.

Definirem els tipus objecte com a tipus compostos, encapsulant els atributs del mateix o diferent tipus sota un nom únic i sense mètodes. A continuació podem veure un exemple de definició de tipus objecte:

```
CREATE TYPE complex AS (
r double precision,
i double precision
);
```

La sintaxi de definició d'un tipus objecte (`CREATE TYPE ... AS`) és comparable amb la sintaxi de definició d'una taula (`CREATE TABLE`), excepte que només els noms i tipus de camp es poden especificar, **no admetrestriccions.**Tingueu en compte que la paraula clau AS és essencial; sense aquesta, el sistema produirà un error de sintaxi.

Una vegada definits els tipus, aquests es poden utilitzar per definir nous tipus i taules que emmagatzemen objectes d'aquest tipus, o per definir el tipus dels atributs d'una taula.

Agafem com a exemple el tipus objecte, `inventari_element`. Aquest conté tres atributs de diferents tipus, que es definiran en les columnes `nom`, `proveidor_id` i `preu`:

```
CREATE TYPE inventari_element AS (
nom text,
proveidor_id integer,
preu numeric
);
```

Aquest tipus objecte el podem utilitzar per definir el tipus d'un atribut dins d'una taula, és a dir, per definir la columna de la taula, amb la sentència SQL `CREATE TABLE`:

```
CREATE TABLE a_mà (
element inventari_element,
quantitat integer
);
```

També podem utilitzar el tipus objecte com a paràmetre d'una funció:

```
CREATE FUNCTION preu_extensió (inventari_element, integer) RETURNS numeric
```

La sentència SQL `DROP TABLE` permet eliminar taules:

```
DROP TABLE a_mà;
```

Les taules creades en PostgreSQL poden incloure diverses columnes ocultes que emmagatzemen informació sobre l'identificador de transacció en què poden estar implicades, la localització física del registre dins de la taula (per localitzar-la molt ràpidament) i, els més importants, l'OID i el TABLEOID.

Aquestes últimes columnes estan definides amb un tipus de dades especial anomenat **identificador d'objecte** (*object identifier,* OID) que s'implementa com un enter positiu de 32 bits. Quan s'insereix un nou registre en una taula s'hi pot assignar un número consecutiu com a `OID`, i el `TABLEOID` de la taula que li correspon.

> `L'OID` no s'assigna per defecte a les taules creades per l'usuari: s'ha d'especificar quan es crea la taula amb l'ordre `WITH OIDS` o activant la variable de configuració `default_with_oids`.

En la programació orientada a objectes, el concepte d'OID és de vital importància, ja que es refereix a la **identitat pròpia**de l'objecte, la qual cosa el diferencia dels altres objectes.

Si ho apliquem en l'exemple anterior, definim la taula de la manera següent:

```
CREATE TABLE a_mà (
element inventari_element,
quantitat integer
) WITH OIDS;
```

Per observar les columnes ocultes hi hem de fer referència específicament en l'ordre de consulta `select`, indicant els camps `oid` i `tableoid` com també la resta de camps de la taula `a_mà`. Veurem amb més deteniment les ordres de consulta i

inserció en apartats posteriors, però de moment suposem que hem inserit dades en la taula anterior amb la sentència SQL `insert into`. L'exemple següent mostra un possible retorn d'una consulta d'aquest tipus:

```
oid | tableoid | nom | proveidor_id | preu | quantitat

———————+——————————+———————+——————————————+——————————+——————————————

17242 | 17240 | galetes | 42 | 1.99 | 1000

(1 rows)
```

Aquestes columnes s'implementen per servir d'identificadors en la realització d'enllaços des d'altres taules.

Dominis

La creació d'un domini en PostgreSQL consisteix en un tipus, definit per l'usuari o inclòs en l'SGBD, més un conjunt de restriccions. A diferència del tipus objecte, en la definició de dominis es poden especificar restriccions. La sintaxi la podem veure en els exemples següents:

```
CREATE DOMAIN country_code char(2) NOT NULL;
CREATE DOMAIN complex_positiu complex NOT NULL CHECK
(complex_major(value,(0,0)))
```

1. El primer domini s'ha creat basant-se en un tipus definit pel sistema, `country_code`, en què l'única restricció és la seva longitud.
2. En el segon domini hauríem d'haver definit l'operador `complex_major` que rebés dos nombres complexos i indiqués si el primer és més gran que el segon.

2.2.2 Taules d'objectes

Una taula d'objectes és una classe especial de taula que emmagatzema un objecte en cada fila i que facilita l'accés als atributs d'aquests objectes com si fossin columnes de la taula. Prenem com a exemple el tipus `inventari_element`. Aquest conté tres atributs de diferent tipus, que es definiran en les columnes `nom`, `proveidor_id` i `preu`:

```
CREATE TYPE inventari_element AS (
nom text,
proveidor_id integer,
preu numeric
);
```

Aquest tipus el podem utilitzar per definir el tipus d'una taula, és a dir, indicar de quin tipus seran els objectes que s'emmagatzemaran a la taula. Utilitzarem la sentència SQL `CREATE TABLE/OF`:

```
CREATE TABLE a_mà OF inventari_element (
PRIMARY KEY (nom),
preu WITH OPTIONS DEFAULT 10
);
```

En crear la taula associada a un tipus, aquesta pren l'estructura del tipus, i fa que les columnes quedin determinades pels atributs del tipus. No obstant això, l'ordre `CREATE TABLE` pot afegir valors i restriccions a la taula.

La taula està lligada al seu tipus, de manera que si s'elimina el tipus també s'elimina la taula associada, per exemple amb la sentència `DROP TYPE ... CASCADE`.

Quan creem el tipus `inventari_element` no s'accepten atributs de tipus `serial`. La clau primària ja s'indica en crear la taula.

Trobareu més informació sobre l'ús del tipus `serial` en l'apartat "Herència i identificadors".

Suposem que sol·licitem a la base de dades que ens mostri totes les columnes dels registres de la taula `a_mà`. A continuació podem veure el retorn d'una consulta d'aquesta taula. Es pot observar com les columnes de la taula són els atributs del tipus `inventari_element`:

```
nom| proveidor_id | preu
———————+———————————+———————
galetes|42| 1.99
(1 rows)
```

2.3 Herència

PostgreSQL ofereix com a característica particular l'herència entre taules, que permet definir una taula que hereti (atributs i mètodes) d'una altra prèviament definida, segons la definició d'herència que hem vist anteriorment.

Utilitzem la taula següent, anomenada `persona`, especificant que volem treballar amb OID:

```
CREATE TABLA persona (
nom varchar (30),
adreça varchar (30)
) WITH OIDS;
```

A partir d'aquesta, declarem la taula `estudiant` com a derivada de `persona`:

```
CREATE TABLE estudiant (
carrera varchar (50),
grup char,
grau int
) INHERITS (persona);
```

En la taula `estudiant` es defineixen les columnes `carrera`, `grup` i `grau`, però si sol·licitem informació de l'estructura de la taula observem que també inclou les columnes definides en la taula `persona`, `nom` i `adreça`.

En aquest cas, la taula `persona` l'anomenem *pare* i la taula `estudiant`, *fill*.

Cada registre de la taula `estudiant` contindrà cinc valors perquè té cinc columnes, tres de la taula `estudiant` i dues que hereta de la taula `persona`.

L'herència no solament permet que la taula *fill* contingui les columnes de la taula *pare*, sinó que estableix una relació conceptual entre aquestes.

Suposem que prèviament hem inserit les dades següents en la taula `estudiant`:

1. un estudiant de nom: 'Joan Miquel',
2. amb adreça: 'Treboles 21',
3. que estudia la carrera: 'Eng. computacio',
4. pertany al grup 'A' i grau 3.

   Si a continuació fem una consulta del contingut de la taula `estudiant` se'ns mostrarà un sol registre. No s'hereten les dades, únicament els camps de l'objecte –els atributs–:

```
nom | adreca | carrera | grup | grau

——————————+————————— +————————————+———————+———————

Joan Miquel | Treboles 21 | Eng. Computació | A | 3

(1 rows)
```

I si fem una consulta de la taula `persona`, aquesta mostrarà un nou registre. Podem veure el retorn de la consulta:

```
nom | adreca
————————— + ————————
Joan Miquel | Treboles 21
(1 rows)
```

El registre que es mostra és el que es va inserir en la taula `estudiant`; tanmateix, l'herència defineix una relació conceptual en la qual un estudiant és una persona. Per tant, en consultar quantes persones estan registrades en la base de dades, tots els estudiants s'inclouen en el resultat.

No és possible esborrar una taula *pare* si no s'esborren primer les taules *fill*.

Com és lògic, en esborrar la fila del nou estudiant que hem inserit aquesta s'esborra de les dues taules, tant si l'esborrem des de la taula `persona` com si l'esborrem des de la taula `estudiant`.

2.3.1 Herència i identificadors

Els OID (identificadors d'objectes) permeten que es diferenciïn els registres de totes les taules, encara que siguin heretades: el nostre estudiant tindrà el

mateix OID en les dues taules, ja que es tracta d'una única instància de la taula `estudiant`que hereta de la taula*persona*:

Retorn de la consulta de la taula estudiant :

```
oid | tableoid | nom | adreca | carrera | grup | grau

———————+——————————+————————— +———————————+———————————————————+———————+———————

**16434 **| 16431 | Joan Miquel | Treboles 21 | Eng. Computació | A | 3

(1 rows)
```

Retorn de la consulta de la taula persona:

```
oid | tableoid | nom | adreca

———————+——————————+———————————————————+———————————————

**16434 | **16427 | Joan Miquel | Treboles 21

(1 rows)
```

Com que no es recomana l'ús d'OID en bases de dades gaire grans, i s'ha d'incloure explícitament en les consultes per examinar-ne el valor, és convenient utilitzar una seqüència compartida per a *pares* i tots els seus *descendents* si es requereix un identificador.

En el PostgreSQL, una alternativa per no utilitzar els OID és crear una columna de tipus `serial` en la taula *pare*, així serà heretada en la taula *fill*. El tipus `serial` defineix una seqüència de valors que s'anirà incrementant de manera automàtica i, per tant, constitueix una bona manera de crear claus primàries, igual que el tipus `AUTO_INCREMENT` en MySQL. Tornarem a crear la taula `persona` amb l'atribut `id` de tipus `serial`:

```
CREATE TABLA persona (
id serial,
nom varchar (30),
adreça varchar (30)
);
```

En definir un tipus serial, hem creat implícitament una seqüència independent de la taula. Una vegada la taula estigui creada, el sistema ens notificarà aquest missatge, indicant que la columna `id` es considera clau primària:

```
NOTICE: CREATE TABLE will create implicit sequence 'persona_id_seq' for SERIAL
    column 'persona.
NOTICE: CREATE TABLE / UNIQUE will create implicit index 'persona_id_key' for
    table 'persona'
```

Creem novament la taula `estudiant` heretant els atributs i mètodes de `persona`:

```
CREATE TABLE estudiant (
carrera varchar (50),
grup char,
grau int
) INHERITS (persona);
```

`Estudiant` heretarà la columna `id` i al ser de tipus serial el valor s'incrementarà utilitzant la mateixa seqüència. Suposem que inserim en la taula alguns registres d'exemple, ometent el valor per a la columna `id`:

- Inserim a la taula persona: ('Josep Claramunt' , 'Montoliu 12');
- Inserim a la taula persona: ('Lluís Arnau', 'Barcelona 3');
- Inserim a la taula estudiant: ('Anna Guillen' , 'Tarragona 19', 'Psicologia', 'C', 2);

Si fem una consulta sobre la taula `estudiant`, veurem que aquesta contindrà un sol registre, però el seu identificador serà el número 3:

```
id | nom | adreca | carrera | grup | grau

———+———————————+———————————+———————————+————————+————————

3 | Anna Guillen | Tarragona 19 | Psicologia | C | 2

(1 row)
```

Tots els registres de la taula `persona` segueixen una mateixa seqüència sense importar si són *pares* o *fills*:

```
id | nom | adreca
———+———————————+———————————
1 | Josep Claramunt | Montoliu 12
2 | Lluis Arnau | Barcelona 3
3 | Anna Guillen | Tarragona 19
(3 row)
```

L'herència és útil per definir taules que de manera conceptual mantenen elements en comú, però també requereixen dades que les fan diferents. Un dels elements que convé definir com a comuns són els identificadors de registre.

2.3.2 Herència en taula d'objectes

Si tenim el tipus següent d'objecte `estudiant`:

```
CREATE TYPE estudiant AS (
id int,
nom varchar (20),
adrecavarchar (30),
carrera varchar (50),
grup varchar(5),
grau int
);
```

Es pot definir una taula per emmagatzemar els estudiants del curs 2012 i una altra per emmagatzemar els estudiants que cursaran pràctiques durant l'any 2012 de la manera següent:

```
CREATE TABLE estudiants_2012 OF estudiant (
PRIMARY KEY (id)
);
CREATE TABLE estudiants_en_pràctiques (
empresa_pràctiques varchar(30),
) INHERITS (estudiants_2012);
```

La diferència entre la primera i la segona taula és que la primera emmagatzema objectes de tipus `estudiant`, i la segona és una especialització de la taula anterior. És a dir, la segona taula hereta les característiques de la primera i afegeix les seves pròpies.

Exemple d'herència en taula d'objectes

Podríem executar una de les instruccions següents. La taula estudiants_2012 es considera una taula amb diverses columnes en què els valors són els especificats.

En la taula estudiants_en_pràctiques inserim:

```
(1000,'Roger Llorac Arnau', 'Castalia 33', 'Psicologia', 'C', 3,
    'Software Catalunya');
```

En incloure el registre a la taula fillestudiants_en_pràctiques, aquest s'afegirà a les dues taules, ja que en heretar els atributs compleix totes dues especificacions: és alumne en el curs 2012 i fa pràctiques a l'empresa.

En la taula estudiants_2012 inserim:

```
(1005,'Lluís Boella Domenec', 'Montoliu 55', 'Antropologia', 'B',
     2);
```

En aquest cas el registre s'inclou només en la taula pareestudiants_2012, ja que no hi ha cap especialització, no s'especifica l'empresa de pràctiques.

Les regles d'integritat i de clau primària, com també la resta de propietats que es defineixin sobre una taula, només afecten els objectes d'aquesta taula, és a dir, no es refereixen a tots els objectes del tipus assignat a aquesta.

2.4 Identificadors, referències i restriccions

Com ja hem vist anteriorment, en els models orientats a objectes, hi ha el concepte d'identificador d'objecte (en anglès *object identifier,* OID), que representa la identitat de l'objecte, única per cada un.

Els OID s'encarreguen de fer referència a un objecte des d'un altre, creant d'aquesta manera les relacions entre objectes.

2.4.1 Identificadors

Hem vist que el PostgreSQL inclou en les taules creades els identificadors d'objecte, aquests es poden utilitzar per referenciar altres tipus d'usuari i així representar relacions d'associació i d'agregació entre objectes.

També som conscients que en la programació orientada a objectes, el concepte d'OID és de vital importància, ja que es refereix a la identitat pròpia de l'objecte, la qual cosa el diferencia dels altres objectes. A continuació veurem un exemple de la utilització d'OID per enllaçar dues taules.

Ens basem en la taula `persona` dels exemples anteriors:

```
CREATE TABLE persona (
nom varchar (15),
adreça varchar (30)
) WITH OIDS;
```

Definim un nou tipus i una nova taula per emmagatzemar els telèfons:

```
CREATE TYPE telefon AS (
tipusvarchar (10),
numero varchar (20),
propietarioid
);
CREATE TABLE llista_telefons OF telefon (
PRIMARY KEY(propietari)
) ;
```

La taula `llista_telefons` inclou la columna `propietari` de tipus OID, que emmagatzemarà la referència als registres de la taula `persona`. Suposem que agreguem dos telèfons a '`Joan Miquel`'; per fer la inserció utilitzem el seu OID, que és 16434. En la taula `llista_telefons`inserim les dades següents:

```
( 'mòbil' , '12345678', 16434 );
( 'casa' , '987654', 16434 );
```

Les dues taules estan vinculades per l'OID de persona, en aquest cas el `propietari`, que com podem comprovar és el mateix:

```
-- Consulta de la taula llista_telefons
tipus | numero | propietari
----------- +----------------- +----------------
mòbil | 12345678 | 16434
casa | 987654 | 16434
(2 rows)
```

Consulteu l'apartat "Criteris de selecció" per obtenir més informació sobre l'operació `join`.

Hi ha una operació que ens permet unir les dues taules, `join`.En aquest cas uneix `llista_telefons` i `persona`, utilitzant la igualtat de les columnes`llista_telefons.propietari` i `persona.oid`.El resultat de la consulta anterior seria:

```
tipus | numero | propietari | nom | adreca

------------ + ---------------- + ---------------- +
    ------------------+------------------

mòbil | 12345678 | 16434| Joan Miquel | Treboles 21

casa | 987654 | 16434 | Joan Miquel | Treboles 21

(2 rows)
```

Els OID del PostgreSQL presenten algunes deficiències:

1. Tots els OID d'una base de dades es generen a partir d'una única seqüència centralitzada, la qual cosa provoca que, en bases de dades amb molta activitat d'inserció i eliminació de registres, el comptador de 4 bytes es desbordi i pugui lliurar OID ja lliurats. Això passa, per descomptat, amb bases de dades molt grans.

2. En les taules enllaçades mitjançant OID no s'obté cap avantatge, en termes d'eficiència, d'utilitzar operadors de composició respecte a una clau primària convencional.

3. Els OID no milloren el rendiment. En realitat, són una columna amb un nombre enter com a valor.

Els desenvolupadors de PostgreSQL proposen l'alternativa següent per utilitzar OID d'una manera absolutament segura:

1. Crear una restricció de taula perquè l'OID sigui únic, almenys en cada taula. L'SGBD anirà incrementant de manera seqüencial l'OID fins a trobar-ne un sense usar.

2. Usar la combinació `OID-TABLEOID` si es necessita un identificador únic per a un registre vàlid en tota la base de dades.

2.4.2 Referències

Els identificadors únics assignats als objectes que s'emmagatzemen en una taula permeten que aquests es puguin referenciar des dels atributs d'altres objectes.

Si volem que un atribut emmagatzemi una referència a un objecte del tipus definit, l'ordre que utilitzem és `REFERENCES`, que implementa una relació d'associació entre els dos tipus objecte.

En l'exemple següent apliquem l'ordre `REFERENCES` a un atribut:

```
CREATE TYPE tipus_persona AS (
dni varchar (9),
nom  varchar (30),
adreça varchar (30)
);
CREATE TABLA persona OF tipus_persona (
PRIMARY KEY (dni)
);
CREATE TYPE telefon (
tipusvarchar (10),
numero varchar (20)
);
CREATE TABLE llista_telefons (
tlftelefon,
propietari varchar(9) REFERENCES persona
) ;
```

Quan s'afegeixin les dades d'un nou telèfon, el PostgreSQL verificarà que el valor de propietari faci referència a una persona, i en cas contrari emetrà un missatge d'error.

2.4.3 Restriccions

Les restriccions permeten especificar condicions que hauran de complir les taules o columnes per mantenir la integritat de les dades. Algunes restriccions vénen imposades pel model concret que s'implementa, mentre que altres tenen l'origen en les regles del client o els valors que poden prendre alguns camps, entre d'altres.

'Null' i 'Not Null'

Sovint el valor d'una columna és desconegut, no és aplicable o no existeix. En aquests casos, els valors zero, cadena buida o fals són inadequats, per la qual cosa utilitzarem `null` per especificar l'absència de valor. En definir la taula podem indicar quines columnes podran contenir valors nuls i quines no.

```
1 CREATE TABLE persona (
2 nom varchar(40) not null,
3 treball varchar(40) null,
4 correu varchar(20)
5 );
```

El nom d'una persona no pot ser nul, però és possible que la persona no tingui correu, ja que en no especificar una restricció `not null` s'assumeix que la columna pot contenir valors nuls.

'Unique'

Aquesta restricció s'utilitza quan no volem que els valors que conté una columna es puguin duplicar.

```
1 CREATE TABLE persona (
2 nom varchar(40) not null,
3 conjugevarchar(40) unique
4 );
```

Així, `conjuge` no pot contenir valors duplicats: no hem de permetre que dues persones tinguin simultàniament el mateix cònjuge.

'Primary key'

Aquesta restricció especifica la columna o columnes que escollim com a clau primària. Hi pot haver múltiples columnes `unique`, però només hi ha d'haver una clau primària. Els valors que són únics poden servir per identificar una fila de la taula de manera unívoca, per la qual cosa se les anomena *claus candidates*.

```
CREATE TABLE persona (
dni varchar(10) PRIMARY KEY,
conjugevarchar(40) UNIQUE
);
```

En definir una columna com a `primary key`, es defineix implícitament com a `unique`. El `dni` no solament és únic sinó que també l'utilitzarem per identificar les persones.

Referències i 'foreign key'

En el model relacional, establim les relacions entre entitats mitjançant la inclusió de claus foranes en altres relacions. El PostgreSQL i l'SWL ofereixen mecanismes per expressar i mantenir aquesta integritat referencial. En l'exemple següent, els `MeusAnimals` tenen com a propietari una `persona`:

```
CREATE TABLE MeusAnimals (
nom varchar(20),
propietari varchar(10) REFERENCES persona
);
```

Una referència per defecte és una clau primària, per la qual cosa `propietari` es refereix implícitament al `dni`de `persona`. Quan es capturen les dades d'un nou animal, el PostgreSQL verifica que el valor de `propietari` faci referència a un `dni` que existeixi de `persona`, en cas contrari emetrà un missatge d'error. No es permet assignar un nou animal a una persona que no existeix.

També és possible especificar a quina columna de la taula es fa referència:

```
CREATE TABLE MeusAnimals (
nom varchar(20),
propietari varchar(10) REFERENCES persona(dni)
);
```

O el seu equivalent:

```
CREATE TABLE MeusAnimals (
nom varchar(20),
propietari varchar(10),
FOREIGN KEY propietari REFERENCES persona(dni)
);
```

Es podria donar el cas que la clau primària de la taula referenciada tingués més d'una columna. En aquest cas, la clau forana també hauria d'estar formada pel mateix nombre de columnes:

```
CREATE TABLE t1 (
a integer PRIMARY KEY,
binteger,
c integer,
FOREIGN KEY (b,c) REFERENCES other_table (c1,c2)
);
```

Si no s'especifica una altra acció, la persona que tingui un animal no es podrà eliminar per omissió, perquè l'animal es quedaria sense propietari. Per poder

eliminar una persona, abans s'han d'eliminar els animals que pugui tenir. Aquest comportament no sembla el més adequat, així que per modificar aquest comportament disposem de les **regles d'integritat referencial** del llenguatge SQL, que PostgreSQL també suporta.

En l'exemple següent es permet que en eliminar una persona els animals quedin sense propietari:

```
CREATE TABLE MeusAnimals (
propietari varchar(10) REFERENCES persona ON DELETE SET NULL
);
```

Amb la clàusula `on delete`es poden especificar les accions següents:

1. `set null`. La referència pren el valor `null`. Si s'elimina `Persona`, el seu `Animal` es quedarà sense propietari.
2. `set default`. La referència pren el valor per omissió.
3. `Cascade`. L'acció s'efectua en cascada. Si s'elimina `Persona` automàticament s'eliminen els seus animals.
4. `Restrict`. No permet esborrar el registre. No es pot eliminar una `Persona` que tingui animals. Aquesta és l'acció que es pren per omissió.

Si es modifica la clau primària de la taula referenciada, es disposa de les mateixes accions que en el cas anterior, que especificarem amb la clàusula `ON UPDATE`.

La restricció`Check` fa l'avaluació prèvia d'una expressió lògica quan s'intenta efectuar una assignació. Si el resultat és cert, accepta el valor per a la columna; en cas contrari, emet un missatge d'error i rebutja el valor.

```
CREATE TABLE Persona (
edatint CHECK (edat > 10 and edat < 80),
correuvarchar(20) CHECK (correu Autoria desconeguda2012–08–28T11:18:35Al
    maquetador: aquesta és una expressió aleatòria d'error. Si dóna problemes
    en l'exportació parlem–ne, doncs probablement es puguin substituir els car
    àcters reservats per d'altres que no facin petar l'exportador.~ '.+@
    .+\..+' ),
ciutatvarchar(30) CHECK (ciutat <> '')
);
```

S'han restringit els valors que s'acceptaran en la columna de la manera següent:

1. `edat` ha de tenir un valor entre onze i setanta-nou anys.
2. `ciutat` no ha de ser una cadena buida.
3. `correu` ha de contenir una arrova.

Qualsevol d'aquestes restriccions pot tenir nom, de manera que es facilita la referència a les restriccions especifiques per esborrar-les, modificar-les, etc. Per assignar un nom a una restricció ho farem de la manera següent:

```
CONSTRAINT nom_de_restriccio <restriccio>
```

Restriccions de taula

Quan les restriccions s'indiquen després de les definicions de les columnes, algunes d'aquestes poden quedar afectades simultàniament. Llavors parlem de *restriccions de taula*:

```
CREATE TABLE persona (
dni int,
nom varchar (30),
conjuge varchar(30),
cap int,
correu varchar(20),
PRIMARY KEY (dni),
UNIQUE (conjuge),
FOREIGN KEY (cap) REFERENCES persona,
CHECK (correu ~ '@' )
);
```

Aquesta notació permet que la restricció pugui abarcar diverses columnes.

```
CREATE TABLE curs (
materia varchar(30),
grup varchar (4),
dia int,
hora time,
aula int,
PRIMARY KEY (materia,grup),
UNIQUE (dia, hora, aula)
);
```

Un curs s'identifica pel grup i la matèria, i dos cursos no poden estar a la mateixa aula el mateix dia i a la mateixa hora.

Igual que amb la restricció de columna, a les restriccions de taula se'ls pot assignar un nom:

```
CREATE TABLE persona (
dniint,
nom varchar (30),
conjuge varchar(30),
cap int,
correu varchar(20),
CONSTRAINT identificador PRIMARY KEY (dni),
CONSTRAINT monogàmia UNIQUE (conjuge),
CONSTRAINT un_cap FOREIGN KEY (cap) REFERENCES persona,
CHECK (correu ~ '@' )
);
```

La sentència `ALTER TABLE` permet afegir (`ADD`) o treure (`DROP`) restriccions que ja s'han definit:

```
ALTER TABLE persona DROP CONSTRAINT monogàmia
ALTER TABLE ADD CONSTRAINT monogàmia UNIQUE (conjuge);
```

2.4.4 Indexació

El PostgreSQL crea un índex per a les claus primàries de totes les taules. Quan necessitem crear índexs addicionals, utilitzarem l'expressió següent:

```
1 CREATE INDEX taula_camp_index ON taula (camp);
```

Prenent com a exemple la taula `persona`, quan es fa una consulta a la taula segons el valor d'un camp concret, el sistema ha d'escanejar-la sencera, fila per fila, per trobar totes les entrades coincidents. Aquest és un mètode ineficient, ja que no discrimina el fet que hi hagi poques o moltes entrades: trigarà molt a fer la consulta a l'haver de fer un registre exhaustiu. Però si al sistema se li indica que mantingui un índex en una de lescolumnes a mode d'identificador, es pot utilitzar un mètode més eficient per localitzar els registres coincidents. Per exemple, només hauria d'avançar uns nivells de profunditat en un arbre de cerca.

Es pot utilitzar l'ordre següent per crear un índex en la columna `dni`:

```
1 CREATE INDEX persona_dni_index ON persona (dni);
```

Per eliminar un índex, s'utilitza l'ordre següent:

```
1 DROP INDEX persona_dni_index;
```

2.5 Tipus de dades col·lecció

Per poder implementar relacions 1:N, l'SGBD ha de permetre definir *tipus col·lecció*. Un tipus col·lecció està format per un nombre indefinit d'elements, tots del mateix tipus. D'aquesta manera, és possible emmagatzemar en un atribut un conjunt de *tuples* en forma d'*array* o en forma de taula imbricada.

Relació 1:N

Representa la interrelació entre dos entitats dins del model entitat-relació, la cardinalitat un a varis (1:N) especifica el tipus de relació.

Les **col·leccions** emmagatzemen objectes del mateix tipus.

El tipus de dades *array* és una de les característiques especials del PostgreSQL. Permet l'emmagatzematge de més d'un valor del mateix tipus en la mateixa columna.

El PostgreSQL permet que les columnes d'una taula es defineixin com a taules multidimensionals de longitud variable. Es poden crear taules de tipus simples definits per l'usuari, tipus enumerats o tipus compostos, però les taules de dominis encara no són compatibles.

Per il·lustrar l'ús de tipus *array*, creem la taula següent:

```
CREATE TYPE telefon AS (
tipusvarchar (10),
numero varchar (20)
);
CREATE TABLE llista_telefons (
nom varchar (10),
tlftelefon[]
);
```

La taula `llista_telefons` conté un atribut de tipus *array* que emmagatzemarà un conjunt d'objectes de tipus `telefon`.

La sintaxi permet especificar la mida exacta del tipus *array*, per exemple:

```
CREATE TABLE llista_telefons (
nom varchar (30),
tlftelefon[10]
);
```

No obstant això, definir la mida dins la sentència `CREATE TABLE` no afecta el comportament en temps d'execució.

Una sintaxi alternativa que s'ajusta a l'estàndard SQL utilitza la paraula clau `ARRAY` per a taules d'una dimensió. Si ho apliquem a l'exemple anterior es podria definir de la manera següent:

```
tlftelefon ARRAY
tlf telefon ARRAY[10]
```

Els valors de l'*array* s'escriuen sempre entre claudàtors, i com passa amb qualsevol columna quan no s'especifica el contrari, s'accepten valors nuls. Vegeu un exemple de com s'inserieix un registre en la taula `llista_telefons`amb un objecte de tipus `telefon`:

```
INSERT INTO llista_telefons VALUES ('Agenda feina', ARRAY[ROW('mobil
    ','623533123')::telefon]);
```

Com veurem més endavant, en la sentència `INSERT INTO` s'utilitza l'ordre `ROW`, perquè el tipus de l'atribut `tlf` és un tipus compost.

Un cop tenim dades a `llista_telefons` podem executar algunes consultes sobre la taula. Per exemple, podem mostrar les dades que s'han inserit anteriorment. Ho faríem de la manera següent:

```
SELECT nom, tlf[1] FROM llista_telefons;
nom |tlf
————————————+————————————————————
Agenda feina| (mobil, 623533123)
```

PostgreSQL ofereix un conjunt d'operadors i funcions compatibles amb el tipus *array*. Per exemple, si utilitzem la funció `array_dims()` podem conèixer les dimensions d'una taula:

```
SELECT array_dims(tlf) FROM llista_telefons;
[1:1]
```

Aquests operadors de tipus *array* permeten comparar i concatenar els elements de les taules. La taula 2.1 mostra els tipus d'operadors que podem trobar.

TAULA 2.1. Tipus d'operadors

Operador	Descripció	Exemple	Resultat
=	igual	`ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]`	t
<>	diferent	`ARRAY[1,2,3] <> ARRAY[1,2,4]`	t
<	més petit que	`ARRAY[1,2,3] < ARRAY[1,2,4]`	t
>	més gran que	`ARRAY[1,4,3] > ARRAY[1,2,4]`	t
<=	més petit o igual que	`ARRAY[1,2,3] <= ARRAY[1,2,3]`	t
>=	més gran o igual que	`ARRAY[1,4,3] >= ARRAY[1,4,3]`	t
@>	conté	`ARRAY[1,4,3] @> ARRAY[3,1]`	t
<@	és continguda per	`ARRAY[2,7] <@ ARRAY[1,7,4,2,6]`	t
&&	superposició (tenen elements en comú)	`ARRAY[1,4,3] && ARRAY[2,1]`	t
||	concatenació taula-a-taula	`ARRAY[1,2,3] || ARRAY[4,5,6]`	{1,2,3,4,5,6}
||	concatenació element-a-taula	`3 || ARRAY[4,5,6]`	{3,4,5,6}
||	concatenació taula-a-element	`ARRAY[4,5,6] || 7`	{4,5,6,7}

També podem trobar funcions que faciliten a l'usuari la interacció amb les taules. En la taula 2.2 podeu veure les diferents funcions que hi ha.

TAULA 2.2. Funcions

Funció	Tipus Retorn	Descripció	Exemple	Resultat
`array_append(anyarray, anyelement)`	anyarray	Afegeix un element al final de la taula.	`array_append (ARRAY[1,2],3)`	{1,2,3}
`array_cat(anyarray, anyarray)`	anyarray	Concatena dues taules.	`array_cat(ARRAY[1,2,3], ARRAY[4,5])`	{1,2,3,4,5}
`array_ndims(anyarray)`	int	Retorna el nombre de dimensions de la taula.	`array_ndims(ARRAY [[1,2,3],[4,5,6]])`	2
`array_dims(anyarray)`	text	Retorna una representació de text de les dimensions de la taula.	`array_dims (ARRAY[[1,2,3],[4,5,6]])`	[1:2][1:3]
`array_fill(anyelement, int[],[,int[]])`	anyarray	Retorna una taula inicialitzada amb el valor i dimensions indicats.	`array_fill(7,ARRAY[3], ARRAY[2])`	[2:4]={7,7,7}
`array_length(anyarray,int)`	int	Retorna la longitud de la dimensió de la taula sol·licitada.	`array_length(array [1,2,3],1)`	3
`array_lower(anyarray,int)`	int	Retorna el límit inferior de la dimensió de la taula sol·licitada	`array_lower('[0:2]= {1,2,3}'::int[],1)`	0
`array_prepend(anyelement, anyarray)`	anyarray	Afegeix un element al principi d'una taula.	`array_prepend(1, ARRAY[2,3])`	{1,2,3}
`array_to_string(anyarray, text)`	text	Concatena elements d'una taula utilitzant el delimitador indicat.	`array_to_string(ARRAY[1, 2, 3],'~^~')`	1~^~2~^~3
`array_upper(anyarray,int)`	int	Retorna el límit superior de la dimensió de la taula sol·licitada.	`array_upper(ARRAY [1,2,3,4],1)`	4
`string_to_array(text, text)`	text[]	Divideix la cadena de caràcters en elements d'una taula utilitzant el delimitador indicat.	`string_to_array ('xx~^~yy~^~zz','~^~')`	{xx,yy,zz}
`unnest(anyarray)`	setof anyelement	Amplia una taula a un conjunt de files.	`unnest(ARRAY[1,2])`	1 2 (2 rows)

3. DDL i DML de les bases de dades objecte-relacionals

Moltes vegades no n'hi ha prou amb la gestió de dades que proporciona el llenguatge SQL, ja que sovint ens interessarà automatitzar processos repetitius i/o prendre decisions de gestió de dades en funció del seu contingut. Per fer-ho caldrà disposar d'una extensió procedimental al llenguatge SQL que el PostgreSQL proporciona amb el llenguatge PL/pgSQL.

PL/PgSQL és un llenguatge procedimental per a sistemes de bases de dades PostgreSQL.

Llenguatge de definició de dades

Un llenguatge de definició de dades, data definition language (DDL), és un llenguatge proporcionat pel sistema de gestió de bases de dades que permet als usuaris dur a terme tasques de definició de les estructures que emmagatzemaran les dades com també els procediments o funcions que permetran consultar-les.

Els llenguatges procedimentals executen i processen, mitjançant ordres DDL (*data definition language*, llenguatge de definició de dades) i DML (*data manipulation language*, llenguatge de manipulació de dades), diferents operacions directament en el servidor. Un dels avantatges principals d'executar programació en el servidor de base de dades és que les consultes i el resultat no han de ser transportats entre el client i el servidor, ja que les dades resideixen en el mateix servidor.

Llenguatge de manipulació de dades

Un llenguatge de manipulació de dades, data manipulation language (DML), és un llenguatge proporcionat pel Sistema de Gestió de Bases de Dades que permet als usuaris dur a terme les tasques de consulta o manipulació de les dades.

El llenguatge PL/pgSQL, desenvolupat en C, s'executa des del mateix client de PostgreSQL (pgsql). Els objectius del seu disseny com a llenguatge procedimental són:

1. Que s'utilitzi per crear funcions i disparadors (*triggers*).

2. Que afegeixi estructures de control al llenguatge SQL.

3. Que hereti totes les definicions d'usuari com tipus, funcions i operadors.

   El PL/pgSQL disposa d'estructures de control repetitives i condicionals, a més de donar la possibilitat de crear funcions que es poden cridar en sentències SQL o executar en esdeveniments de tipus disparador (*trigger*).

Les funcions escrites en PL/PgSQL poden acceptar com a arguments qualsevol dada escalar o *array* de dades que estiguin suportats per l'SGBD i poden retornar un resultat de qualsevol d'aquests tipus. També podem acceptar o retornar un tipus de dada compost (*row type*) especificat pel nom. Es pot declarar una funció PL/PgSQL que retorni un registre, la qual cosa significa que el resultat serà un tipus de fila les columnes de la qual han estat determinades per l'especificació en la crida de la consulta.

3.1 Declaració i inicialització d'objectes

Una vegada s'ha definit un tipus objecte es pot utilitzar en qualsevol bloc PL/pgSQL. Les instàncies dels tipus objecte es creen en temps d'execució, de manera queaquests objectes segueixen les regles normals d'àmbit i instància. En un bloc o subprograma, els objectes locals són instanciats quan s'entra en el bloc o subprograma i deixen d'existir quan se'n surt.

3.1.1 Estructura del PL/pgSQL

En el PostgreSQL, en ser un llenguatge estructurat en blocs, el codi de la definició d'una funció es considera un bloc. Un bloc es defineix de la manera següent:

```
[ <<etiqueta>> ]
[ DECLARE
<declaracions de constants i variables>]
BEGIN
<sentències executables>;
[EXCEPTION
<declaració d'excepcions>;]
END [ etiqueta ];
```

L'etiqueta només és necessària si volem identificar el bloc per utilitzar-lo en una sentència `EXIT`, o per qualificar els noms de les variables declarades en el bloc. Si una etiqueta s'escriu després d'`END`, ha de coincidir amb l'etiqueta al principi del bloc.

A continuació podem veure un exemple de bloc i el codi de la definició de la funció `taxa_vendes(subtotal real)`:

```
CREATE FUNCTION taxa_vendes(subtotal real) RETURNS real AS $$
BEGIN
RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

> Hi pot haver qualsevol nombre de subblocs en la secció de la sentència d'un bloc.

Hi ha dos tipus de comentaris a PL/pgSQL. Amb dos guions `(-)`comença un comentari que s'estén fins al final de la línia mentre que els caràcters `/*` inicien un bloc de comentaris que s'estén fins que es troba una altra vegada el caràcter `*/`.

A continuació podeu veure un exemple d'una funció escrita amb PL/pgSQL que conté subblocs, `aquestafunc()`:

```
CREATE FUNCTION aquestafunc() RETURNS integer AS $$
DECLARE
quantitat integer := 30;
```

```
BEGIN
RAISE NOTICE 'La quantitat aquí conté %', quantitat; -- Mostra 30
quantitat := 50;
-- Creem un sub-bloc
DECLARE
quantitat integer := 80;
BEGIN
RAISE NOTICE 'La quantitat aquí conté %', quantitat; -- Mostra 80
END;
RAISE NOTICE 'La quantitat aquí conté %', quantitat; -- Mostra 50
RETURN quantitat;
END;
$$ LANGUAGE plpgsql;
```

Al final de la funció es defineix el llenguatge procedimental que utilitzarem mitjançant la clàusula `LANGUAGE`.

És important no confondre l'ús de `BEGIN/END` per agrupar declaracions en PL/pgSQL amb els noms de les ordres SQL per al control de la transacció. `BEGIN/END` només són per a l'agrupació, no per iniciar o finalitzar una transacció.

Trobareu més informació sobre l'ús de la transacció en l'apartat "Modificació d'objectes" d'aquesta unitat.

També podem canviar la definició de la funció amb l'ordre `ALTER FUNCTION`. Podem modificar el nom d'una funció tal com es mostra tot seguit:

```
ALTER FUNCTION taxa_vendes(subtotal real) RENAME TO impost_vendes;
```

3.1.2 Declaració d'objectes

La declaració d'objectes es pot dividir en diferents parts: la que defineix els paràmetres d'entrada i de retorn d'una funció; la que defineix el tipus objecte com a *variable de fila*; la que defineix l'estratègia que s'utilitza per declarar les col·leccions; i la que ens permet copiar el tipus de dades d'una estructura a una altra.

Declaració de paràmetres d'una funció

Totes les variables i constants, `CONSTANT`, utilitzades en un bloc o en un subbloc s'han de declarar en la secció `DECLARE` del bloc. Les variables en PL/pgSQL poden emmagatzemar qualsevol tipus de dades de SQL i el seu valor per omissió és `null`.

Prenent com a exemple la funció anterior `taxa_vendes(subtotal real)`amb una declaració de variables:

```
CREATE FUNCTION taxa_vendes(subtotal real) RETURNS real AS $$
DECLARE
percentatge real := 0.06;
BEGIN
RETURN subtotal * percentatge;
```

```
END;
$$ LANGUAGE plpgsql;
```

Els paràmetres passats a les funcions s'anomenen amb els identificadors `$1`, `$2`, etc. De manera opcional, es poden declarar àlies utilitzant el número del paràmetre, `$n`, per a més llegibilitat. Qualsevol àlies o identificador numèric es pot utilitzar per fer referència al valor del paràmetre.

Es poden crear àlies de dues maneres diferents. La millor manera és donar un nom al paràmetre de la funció: aquest es pot definir en crear la funció com en l'exemple anterior en què se li assigna l'àlies `subtotal`.

```
CREATE FUNCTION taxa_vendes(subtotal real) RETURNS real AS $$
```

L'altra manera, que era l'única disponible abans de la versió del PostgreSQL 8.0, és declarar de manera explícita un àlies utilitzant la sintaxi de declaració següent:

```
nom ALIAS FOR $n;
```

L'exemple anterior es podria modificar com es mostra a continuació:

```
CREATE FUNCTION taxa_vendes (real) RETURNS real AS $$
DECLARE
percentatge real := 0.06;
subtotal ALIAS FOR $1;
BEGIN
RETURN subtotal * percentatge;
END;
$$ LANGUAGE plpgsql;
```

Aquests dos exemples no són perfectament equivalents. En el primer cas, subtotal es podria referenciar com a `taxa_vendes.subtotal`, en canvi, en el segon cas no es podria.

Quan una funció PL/pgSQL es declara amb paràmetres de sortida, als paràmetres de sortida se'ls dóna els noms `$n`i àlies opcionals de la mateixa manera que als paràmetres d'entrada. Un paràmetre de sortida és una variable que comença amb valor `null`, i s'hi ha d'assignar un valor durant l'execució de la funció. El valor final del paràmetre és el que es retorna. Per exemple, l'impost a les vendes també es podria fer com es mostra a continuació:

```
CREATE FUNCTION taxa_vendes(subtotal real, OUT taxa real) AS $$
DECLARE

percentatge real := 0.06;

BEGIN

taxa := subtotal * percentatge;

END;
$$ LANGUAGE plpgsql;
```

Declaració d'un tipus objecte

Una variable de tipus objecte s'anomena *variable de fila* (variable *row-type*). Aquesta pot contenir una fila sencera del resultat d'una consulta `SELECT` o `FOR`, mentre que el conjunt de consultes d'una columna coincideix amb el tipus declarat de la variable.

```
nomnom_taula%ROWTYPE;
nomnom_tipus_objecte;
```

Als valors dels camps individuals de la fila s'hi accedeix usant la notació de punts habitual, `variable_fila.camp`.

En cas que els paràmetres d'una funció siguin de tipus objecte, l'identificador corresponent `$n` serà una variable de fila, i els camps es podran seleccionar com a `$1.id_usuari`.

A continuació podem veure un exemple de l'ús dels tipus objecte. Els tipus `taula1` i `taula2` ja existeixen, i podem veure com apareix la sentència `select` per fer una consulta:

```
CREATE FUNCTION combinació(t_fila taula1) RETURNS text AS $$
DECLARE
t2_fila taula2%ROWTYPE;
BEGIN
SELECT * INTO t2_fila FROM taula2 WHERE ... ;
RETURN t_fila.f1 || t2_fila.f3 || t_fila.f5 || t2_fila.f7;
END;
$$ LANGUAGE plpgsql;
```

De retorn d'una funció

Les funcions poden retornar tipus bàsics o compostos, fins i tot es poden retornar estructures de taules. Les estructures de control són probablement la part més útil del PL/pgSQL. Amb aquestes estructures es poden manipular les dades PostgreSQL d'una manera molt flexible.

Dins les estructures de control trobem dues ordres disponibles que permeten retornar les dades d'una funció: `RETURN` i `RETURN NEXT`.

L'ordre `RETURN` amb una expressió acaba la funció i retorna el valor de l'expressió a la crida. Aquest mètode és utilitzat per les funcions PL/pgSQL que no retornen un conjunt.

```
RETURN expressió;
```

Quan es retorna un tipus escalar es pot utilitzar qualsevol expressió. El resultat de l'expressió es converteix automàticament en el tipus de retorn de la funció. Per retornar el valor d'un tipus objecte (fila), s'ha de declarar l'expressió amb una variable de registre o fila.

La funció següent incrementa un nombre enter fent ús d'un paràmetre d'entrada, i retorna el resultat següent:

```
CREATE FUNCTION increment(i integer) RETURNS integer AS $$
BEGIN
RETURN i + 1;
END;

$$ LANGUAGE plpgsql;
```

Quan una funció PL/pgSQL es declara com a retorn `SETOF`, el procediment a seguir és lleugerament diferent. En aquest cas, els elements individuals a retornar s'especifiquen mitjançant una seqüència d'ordres `RETURN NEXT` o `RETURN QUERY`, i a continuació s'utilitza una ordre final `RETURN` sense arguments per indicar que la funció s'ha acabat d'executar.

L'ordre `RETURN NEXT` es pot utilitzar amb tipus escalars i tipus objecte. Si s'utilitza com a resultat, un tipus objecte retornarà una "taula" de resultats. L'ordre `RETURN QUERY` afegeix els resultats de l'execució d'una consulta al conjunt de resultats de la funció. Aquestes dues ordres poden ser barrejades en un sol conjunt d'una funció, en aquest cas els seus resultats es concatenen.

Si declarem la funció amb paràmetres de sortida, hem d'escriure `RETURN NEXT` sense expressió. En cada execució, els valors actuals de la variable o variables de sortida es desaran per a un eventual retorn com una fila del resultat. Recordeu que hem de declarar la funció com a retorn de registre, `SETOF record`, quan hi hagi diferents paràmetres de sortida, o `SETOF sometype`, quan hi hagi un sol paràmetre de sortida de tipus *sometype*, amb la finalitat de crear una funció amb un retorn que sigui un conjunt dels paràmetres de sortida.

A continuació veurem l'exemple d'una funció que utilitza `RETURN NEXT`:

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);

CREATE OR REPLACE FUNCTION getAllFoo() RETURNS SETOF foo AS
$BODY$
DECLARE
r foo%rowtype;
BEGIN
FOR r IN SELECT * FROM foo
WHERE fooid > 0
LOOP
-- camp per realitzar les tasques de processament
RETURN NEXT r; -- retorna la fila actual del SELECT
END LOOP;
RETURN;
END
$BODY$
LANGUAGE 'plpgsql' ;
```

Declaració de col·leccions

El bucle `FOREACH` és molt semblant a un bucle `FOR`, però en comptes d'iterar mitjançant les files retornades per una consulta SQL, aquest itera mitjançant els elements d'una col·lecció. La instrucció `FOREACH` té la sintaxi que es mostra en la figura 3.1.

FIGURA 3.1. Sintaxi de 'FOREACH'

```
[ <<etiqueta>> ]
FOREACH variable destinació [ SLICE número ] IN ARRAY expressió LOOP
    declaracions
END LOOP [ etiqueta ];
```

Sense `SLICE`, o si s'especifica `SLICE 0`, el bucle es repeteix mitjançant elements individuals de l'*array* produïda per l'avaluació de l'expressió.

La variable de destinació, *target*, s'assigna a cada valor d'element en seqüència, i el cos del bucle s'executa per a cada element. A continuació veurem un exemple d'un bucle mitjançant els elements d'una col·lecció d'enters:

```
1  CREATE FUNCTION suma(int[]) RETURNS int8 AS $$
2  DECLARE
3  s int8 := 0;
4  x int;
5  BEGIN
6  FOREACH x IN ARRAY $1— $1: paràmetre d'entrada int[]
7  LOOP
8  s := s + x;
9  END LOOP;
10 RETURN s;
11 END;
12 $$ LANGUAGE plpgsql;
```

Els elements són iterats en ordre d'emmagatzematge, independentment del nombre de dimensions de la col·lecció. Tot i que la variable de destinació és en general única, pot ser una llista de variables que iteri per mitjà d'una col·lecció de valors compostos (registres). En aquest cas, per cada element de la matriu, les variables s'assignen a partir de columnes successives del valor compost.

Amb un valor positiu del paràmetre `SLICE`, el bucle `FOREACH` itera per mitjà de talls de la col·lecció en lloc d'elements individuals. El valor del segment, `SLICE`, ha de ser un enter constant no més gran que el nombre de dimensions de la col·lecció. La variable destinació ha de ser una col·lecció, i ha de rebre talls successius del valor de la col·lecció, en què cada sector és el nombre de dimensions especificades pel segment. A continuació es mostra un exemple d'iteració mitjançant un segment d'una dimensió:

```
1  CREATE FUNCTION cercar_files(int[]) RETURNS void AS $$
2
3  DECLARE
4  x int[];
5
6  BEGIN
7  FOREACH x SLICE 1 IN ARRAY $1
8
9  LOOP
10 RAISE NOTICE 'row = %', x; —— El que mostrarà la funció
11 END LOOP;
12
13 END;
14
15 $$ LANGUAGE plpgsql;
```

La crida següent utilitza la funció anterior passant-li per paràmetre una matriu:

```
SELECT cercar_files(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);
-- La sortida que mostra en executar-se la funció:
NOTICE: row = {1,2,3}
NOTICE: row = {4,5,6}
NOTICE: row = {7,8,9}
NOTICE: row = {10,11,12}
```

Tipus de còpia

L'ordre `%TYPE` ofereix el tipus de dades d'una variable o columna d'una taula. La podem utilitzar per declarar les variables que contindran els valors de la base de dades. Per exemple, si tenim una columna anomenada `id_usuari` en la taula d'usuaris, per declarar una variable amb el mateix tipus de dades faríem el següent:

```
id_usuari usuaris.id_usuari%TYPE;
```

Mitjançant l'ús de `%TYPE` no cal conèixer el tipus de dades de l'estructura a la qual fem referència, i el més important, si el tipus de dades de l'estructura referenciada canvia en el futur, no haurem de modificar la definició de la funció.

Polimorfisme

És la capacitat de diversos tipus objecte derivats d'un antecessor per utilitzar el mateix mètode de manera diferent.

L'ordre `%TYPE` és imprescindible en funcions polimòrfiques, ja que els tipus de dades necessàries per a les variables internes poden canviar d'una crida a la següent.

3.1.3 Inicialització d'objectes

Les variables declarades en la secció de declaracions (`DECLARE`) s'inicialitzen amb el seu valor per defecte cada vegada que s'inicia el bloc, no cada vegada que es fa la crida a la funció. Per exemple, l'assignació de la funció `ara()` a una variable de tipus `timestamp` fa que la aquesta emmagatzemi el temps del moment en què s'ha fet la crida, no el moment en què la funció es va compilar.

Quan s'inicialitzen les variables també es poden definir valors per defecte, `DEFAULT`, o com a constants, `CONSTANT`, tal com es pot veure tot seguit:

```
quantitat integer DEFAULT 32;
url varchar := 'http://elmeuespai.com';
id_usuari CONSTANT integer := 10;
```

També es pot dur a terme la inicialització mitjançant una funció,en què les dades del tipus objecte es passen per mitjà dels paràmetres d'entrada. Tot seguit es mostra el tipus de dades `tipus_persona`, la taula `persona` d'aquest tipus i la funció que l'inicialitza:

```
CREATE TYPE tipus_persona AS (
dnivarchar (9),
nom varchar (15),
adreçavarchar (30)
);

```

```
CREATE TABLA persona OF tipus_persona (
PRIMARY KEY (dni)
);

CREATE FUNCTION alta_persona(varchar (9), varchar (15), varchar (30)) AS $$
BEGIN
--Inserim els valors que es passen a través dels paràmetres
INSERT INTO persona VALUES ($1, $2, $3);
END;
$$ LANGUAGE plpgsql;
```

3.2 Ús de la sentència 'SELECT'

Les consultes de selecció s'utilitzen per indicar al servidor que retorni informació de les bases de dades. Aquesta informació retornada pot ser un valor, un *tuple* o una taula. La sintaxi bàsica d'una consulta de selecció és la que es mostra en la figura 3.2.

Figura 3.2. Sintaxi bàsica d'una consulta de selecció

```
SELECT camps FROM taula;
```

En aquesta sintaxi, *camps* representa la llista de camps que volem recuperar i *taula* és on hi ha la informació emmagatzemada.

Creem un tipus d'objecte `persona` i una taula d'objectes de tipus `persones` que inclourà diferents valors. Consultem els atributs `id`, `nom` i `telefon` dels objectes de la taula:

```
CREATE TYPE persona AS (
id integer,
nom varchar(15),
cognom varchar(20),
aniversaridate,
adreçavarchar(30),
telefonvarchar(15)
);
CREATE TABLE persones OF persona;
-- Consulta dels objectes de la taula persones
SELECT id, nom, telefon FROM persones;
```

També ens podem referir a tots els camps d'una taula amb el símbol `*`. De manera que la consulta anterior també es podria fer de la manera següent:

```
SELECT * FROM persones;
```

De manera addicional es pot especificar l'ordre en què es volen recuperar els objectes de la taula mitjançant la clàusula `ORDER BY llista_camps`, en què `llista_camps` representa els camps a ordenar:

```
SELECT id, nom, telefon FROM persones ORDER BY id;
```

Aquesta consulta retorna les dades de les persones emmagatzemades a la taula `persones` però ordenades en funció del seu identificador.

Per indicar que volem recuperar els objectes segons l'interval de valors d'un camp hem d'utilitzar l'operador `BETWEEN`. La seva sintaxi és la que es mostra en la figura 3.3.

Figura 3.3. Sintaxi de l'operador 'BETWEEN'

```
camp [NOT] BETWEEN valor1 AND valor2
```

En aquest cas la consulta retornaria els objectes que continguin a *camp* un valor inclòs en l'interval *valor1*, *valor2* (tots dos inclosos). Si avantposem la condició `NOT`, es retornaran els valors no inclosos en l'interval:

```
1 BEGIN
2 SELECT * FROM persones
3 WHERE id BETWEEN 1 AND 100;
4 END;
```

La consulta retornarà les persones que tinguin el valor de l'atribut identificador entre 1 i 100.

3.2.1 Accés a les dades

Cada tipus de funció pot prendre els tipus bàsics, els tipus compostos o combinacions d'aquests com a arguments (paràmetres). A més, cada tipus de funció pot retornar un tipus bàsic o un tipus compost.

Accés als tipus compostos

Prenem com a exemple el tipus objecte *inventari_element.* Aquest conté tres atributs de diferent tipus:

```
1 CREATE TYPE inventari_element AS (
2 nom text,
3 proveidor_id integer,
4 preu numeric
5 );
```

Aquest tipus objecte el podem utilitzar per definir el tipus d'un atribut dins d'una taula, és a dir, definir la columna de la taula.

```
1 CREATE TABLE a_mà (
2 element inventari_element,
3 quantitat integer
4 );
```

Tanmateix, una funció pot prendre com a argument un tipus compost.

Tot seguit podeu veure com una funció utilitza un tipus compost en el seu cos:

```
CREATE FUNCTION preu_extensió(inventari_element, integer) RETURNS numeric AS $$
BEGIN
RETURN $1.preu * $2;
END;
$$ LANGUAGE plpgsql;
```

I ara podeu veure una consulta que utilitza la funció anterior, on el retorn de la funció és el paràmetre de la consulta:

```
SELECT preu_extensió(t.element, 10) FROM a_mà t;
```

Per accedir al camp d'una columna composta, s'escriu un punt i el nom del camp. Per exemple, si volem seleccionar alguns subcamps de la taula `a_ma`, com el nom dels elements amb preus superiors a les 9,99 unitats, ho faríem amb la sentència següent:

```
SELECT element.nom FROM a_mà WHERE element.preu > 9.99;
```

Però això no funcionarà, ja que el nom de l'element es pren com a un nom de taula, no com a nom d'una columna de la taula `a_mà`, segons les regles de la sintaxi SQL. Hauríem d'escriure, doncs, el següent:

```
SELECT (element).nom FROM a_mà WHERE (element).preu > 9.99;
```

O, en cas que necessitem utilitzar el nom de la taula, l'escriuríem de la manera següent:

```
SELECT (a_mà.element).nom FROM a_mà WHERE (a_mà.element).preu > 9.99;
```

Ara l'objecte entre parèntesis s'interpreta correctament com una referència a la columna d'elements i, a continuació el subcamp es pot seleccionar des d'aquesta.

Problemes sintàctics similars tenen lloc cada vegada que seleccionem un camp d'un valor compost. Per exemple, per seleccionar un sol camp a partir del resultat d'una funció que retorni un valor compost, la sintaxi seria la següent:

```
SELECT (la_meva_funció(...)).camp FROM ...
```

Sobrenoms

En una base de dades amb tipus i objectes, el més recomanable és utilitzar sobrenoms per als noms de les taules. El sobrenom d'una taula ha de ser únic en el context de la consulta. Els sobrenoms serveixen per accedir al contingut de la taula, però s'han d'utilitzar adequadament en les taules que emmagatzemen objectes.

Vegeu tot seguit com s'han d'utilitzar els sobrenoms:

```
BEGIN
CREATE TYPE persona AS (nom varchar(15));
CREATE TABLE ptaula1 OF persona;
CREATE TABLE ptaula2 (p1 persona);
...
END;
```

La diferència entre les dues primeres taules és que la primera emmagatzema objectes del tipus `persona`, mentre que la segona té una columna en què s'emmagatzemen valors del tipus `persona`. Considerant les consultes següents, vegeu en la figura 3.4 com s'accedeix a aquestes taules.

FIGURA 3.4. Consultes utilitzant sobrenoms

1. SELECT nom FROM ptaula1;	Correcte
2. SELECT p1.nom FROM ptaula2;	Incorrecte
3. SELECT *sobrenom*.nom FROM ptaula2 *sobrenom*;	Correcte

En la primera consulta, `nom` es considera com una de les columnes de la taula `ptaula1`, ja que els atributs dels objectes es consideren columnes de la taula d'objectes. En canvi, en la segona consulta es requereix l'ús d'un sobrenom per indicar que `nom` és el nom d'un atribut de l'objecte de tipus `persona` que s'emmagatzema en la columna `p1`. Per resoldre aquest problema no es poden utilitzar els noms de les taules directament: `ptaula2.p1.nom` seria incorrecte.

Per facilitar la formulació de consultes i evitar errors es recomana utilitzar sobrenoms per accedir a totes les taules que continguin objectes amb identitat o sense i per accedir a les columnes de les taules en general.

3.2.2 Criteris de selecció

El PL/pgSQL tambépermet filtrar els objectes amb la finalitat de recuperar només els que compleixin unes condicions preestablertes.

La clàusula 'WHERE'

La clàusula `WHERE` es pot utilitzar per determinar quins objectes de les taules enumerades en la clàusula `FROM` apareixeran en els resultats de la sentència `SELECT`.

Per exemple, per obtenir només les persones que tinguin el cognom `arnau`, la consulta adequada seria:

```
BEGIN
SELECT * FROM persones
WHERE cognom LIKE '%arnau';
END;
```

En l'exemple següent podeu veure la unió de la taula `pel·licules` amb la taula `distribuidors`:

```
BEGIN
SELECT p.titol, p.did, d.nom, p.data_prod, p.tipus
FROM distribuidors d, pel·licules p
WHERE p.did = d.did
END;
```

Resultat de la consulta:

```
titol | did | nom | data_prod | tipus
------------------------+-----+-------------+-------------+----------
The Third Man | 101 | British Lion | 1949-12-23 | Drama
The African Queen | 101 | British Lion | 1951-08-11 | Romantica

...
```

Les clàusules 'GROUP BY' i 'HAVING'

Després de passar el filtre `WHERE`, la taula d'entrada resultant es pot agrupar, amb la clàusula `GROUP BY`, i es poden eliminar les files del grup amb la clàusula `HAVING`.

```
SELECT llista_selecció
FROM ...
[WHERE ...]
GROUP BY agrupació_referencies_columna [, ]...
```

Aquesta clàusula s'utilitza per agrupar les files d'una taula que tenen els mateixos valors en totes les columnes que es mostren. L'ordre en què les columnes s'enumeren no importa. L'efecte és combinar cada conjunt de files que tenen valors comuns a la fila grup que representa totes les files del grup. Això es fa per eliminar la redundància en la sortida. Per exemple:

```
SELECT * FROM test1;
x | y
---+---
a | 3
c | 2
b | 5
a | 1
(4 rows)
```

```
SELECT x, suma(y) FROM test1 GROUP BY x;
```

```
x | suma
---+-----
a | 4
b | 5
c | 2
(3 rows)
```

Un altre exemple: la consulta calcula les vendes totals de cada producte, en lloc de les vendes totals de tots els productes:

```
BEGIN
SELECT id_producte, p.nom, (suma(s.units) * p.preu) AS vendes
FROM productes p LEFT JOIN vendes s USING (id_producte)
GROUP BY id_producte, p.nom, p.preu;
END;
```

En aquest exemple, les columnes `id_producte`, `p.nom` i `p.preu` han d'estar en la clàusula `GROUP BY`, ja que se'n fa referència en la llista de selecció de la consulta. La columna `s.units` no ha d'estar en la llista `GROUP BY`, ja que només s'utilitza en una expressió d'agregació (`suma(...)`), que representa les vendes d'un producte. Per cada producte, la consulta retorna totes les vendes del producte.

Si una taula s'ha agrupat utilitzant `GROUP BY`, però només alguns grups són d'interès, la clàusula `HAVING` es pot utilitzar, de la mateixa manera que una clàusula `WHERE`, per eliminar grups del resultat. La sintaxi és:

```
SELECT llista_selecció FROM ... [WHERE ...] GROUP BY ... HAVING expressió
    _booleana
```

Les expressions amb la clàusula `HAVING` es poden referir tant a les expressions agrupades com a les no agrupades (que impliquen necessàriament una funció d'agregació). A continuació es pot veure un exemple seguint l'anterior:

```
SELECT x, suma(y) FROM test1 GROUP BY x HAVING suma(y) > 3;
x | suma
---+-----
a | 4
b | 5
(2 rows)
```

```
SELECT x, suma(y) FROM test1 GROUP BY x HAVING x < 'c';
```

```
x | suma
---+-----
a | 4
b | 5
(2 rows)
```

Prenent com a exemple, la taula `pel·licules` de l'apartat anterior, podem sumar la columna longitud (`long`) de totes les pel·lícules i agrupar els resultats per `tipus`:

```
SELECT tipus, suma(long) AS total FROM pel·licules GROUP BY tipus;
```

```
tipus | total
----------------+-------
Accio | 07:34
Comedia | 02:58
Drama | 14:28
Musical | 06:42
Romantica | 04:38
```

Si volem sumar la columna longitud, `long`, de totes les pel·lícules, cal que agrupem els resultats per tipus i mostrem només els grups amb una durada inferior a cinc hores. Ho faríem de la manera següent:

```
SELECT tipus, suma(long) AS total FROM pel·licules
GROUP BY tipus
HAVING suma(long) < interval '5 hores';
```

```
tipus | total
——————————+———————
Comedia | 02:58
Romantica | 04:38
```

L'operador 'JOIN'

Si distribuïm la informació en diferents taules evitem la redundància de dades i ocupem menys espai físic en el disc. L'operador `JOIN` relaciona dues o més taules per obtenir un resultat que inclogui dades (camps i registres) de totes dues, i les taules es combinen segons els camps comuns a totes dues taules.

Per exemple, la informació de `llibres` es podria separar en dues taules, una anomenada `llibre` i una altra `editorial`. A continuació definim les dues taules:

```
CREATE TABLE llibre (
codiserial,
titolvarchar(40) not null,
autor varchar(30) not null default 'Desconegut',
codi_editorialsmallint not null,
preudecimal,
PRIMARY KEY(codi)
);
CREATE TABLE editorial (
codiserial,
nomvarchar(20) not null,
PRIMARY KEY(codi)
);
```

D'aquesta manera evitem emmagatzemar cada vegada la informació de l'editorial en la taula `llibre`. Indiquem l'editorial de cada llibre amb el camp que fa referència al codi de l'editorial.

Quan fem una consulta de les dades de la taula llibre ens apareixerà el codi de l'editorial, però no obtindrem la informació de l'editorial. Per obtenir les dades del llibre i de l'editorial necessitem consultar les dues taules alhora. Tot seguit es mostra una consulta relacionant les dues taules:

```
SELECT * FROM llibre
JOIN editorial ON llibre.codi_editorial = editorial.codi;
```

L'operador 'UNION'

Aquest operador calcula la unió del conjunt de files retornades per les sentències `SELECT` involucrades. Una fila apareix en el conjunt de la unió si apareix en almenys un dels conjunts de resultats. Les dues sentències `SELECT` que representen els operands directes de la unió han de tenir el mateix nombre de columnes, i les columnes corresponents han de ser de tipus de dades compatibles.

La clàusula UNION té la sintaxi que es mostra en la figura 3.5.

FIGURA 3.5. Sintaxi de la clàusula 'UNION'

```
declaracio_select UNION [ ALL | DISTINCT ] declaracio_select
```

Així, declaracio_select fa referència a qualsevol declaració SELECT sense les clàusules ORDER BY, LIMIT, FOR UPDATE, o FOR SHARE. El resultat de l'operador UNION no conté registres duplicats tret que s'especifiqui l'opció ALL. Aquesta opció impedeix que s'eliminin els duplicats. En canvi, DISTINCT es pot escriure per especificar explícitament el comportament per defecte d'eliminar les files duplicades.

Prenent com a exemple la taula distribuidors de l'apartat anterior, tot seguit es mostra com es pot obtenir la unió de les taules distribuidors i actors, restringint els resultats als que comencen amb la lletra *W* en cada taula. Només interessen les files diferents, de manera que la paraula clau ALL s'omet.

```
distribuidors: actors:
did | nom id | nom
————+————————— ————+————————————
108 | Westward 1 | Woody Allen
111 | Walt Disney 2 | Warren Beatty
112 | Warner Bros. 3 | Walter Matthau
... ...
```

```
SELECT distribuidors. nom
FROM distribuidors
WHERE distribuidors. nom LIKE 'W%'
UNION
SELECT actors.nom
FROM actors
WHERE actors.nom LIKE 'W%';
```

```
nom
————————————
Walt Disney
Walter Matthau
Warner Bros.
Warren Beatty
Westward
Woody Allen
```

3.2.3 Subconsultes

Una subconsulta és una instrucció SELECT imbricada dins d'una instrucció SELECT, INSERT...INTO, DELETE o UPDATE o dins d'una altra subconsulta. Es pot utilitzar una subconsulta en lloc d'una expressió en la llista de camps d'una instrucció SELECT o en una clàusula WHERE.

En les subconsultes s'utilitza la instrucció SELECT per proporcionar un conjunt d'un o més valors per avaluar l'expressió de la clàusula WHERE de la consulta

principal. La subconsulta següent retorna com a resultat un conjunt de files que contenen només atributs dels objectes `persona`:

```
BEGIN
INSERT INTO treballadors
(SELECT * FROM persones WHERE cognom LIKE '%arnau');
END;
```

La subconsulta següent ens retorna el títol, l'autor i el preu del llibre més car:

```
BEGIN
SELECT titol, autor, preu FROM llibre
WHERE preu = (SELECT max(preu) FROM llibre);
END;
```

Fins ara hem vist que una subconsulta pot substituir una expressió. Aquesta subconsulta ha de retornar un valor escalar o una llista de valors d'un camp. Les subconsultes que retornen una llista de valors inclouen dins la clàusula `WHERE` la paraula clau `IN`.

El resultat de fer una subconsulta amb `IN` o `NOT IN` és una llista de valors. La sintaxi és la que es mostra en la figura 3.6.

FIGURA 3.6. Sintaxi d'una subconsulta amb "IN o 'NOT IN'

```
... WHERE expressió IN (subconsulta);
```

L'exemple següent mostra els noms de les editorials que han publicat llibres d'un determinat autor:

```
BEGIN
SELECT nom FROM editorial
WHERE codi IN (SELECT codi_editorial FROM llibre
WHERE autor = 'Joanne Kathleen');
END;
```

La subconsulta retorna una llista de valors d'un sol camp, en aquest cas el `codi`, que utilitzarà la consulta exterior.

També es poden buscar valors no coincidents, per exemple, les editorials que no han publicat llibres d'un autor específic:

```
BEGIN
SELECT nom FROM editorial
WHERE codi NOT IN (SELECT codi_editorial FROM llibre
WHERE autor = 'Joanne Kathleen');
END;
```

3.3 Inserció d'objectes

Per afegir objectes a una taula d'objectes s'utilitza la sentència `INSERT INTO`. La sintaxi és la que es mostra en la figura 3.7.

FIGURA 3.7. Sintaxi de la sentència 'INSERT INTO'

```
INSERT INTO taula (camp1, camp2, ..., campN) VALUES (valor1, valor2, ..., valorN);
```

Aquesta consulta emmagatzema en el camp1 el valor1, en el camp2 el valor 2 i així successivament. S'ha de tenir un tracte especial amb les cadenes de caràcters, ja que s'han de delimitar els valors literals utilitzant cometes simples (').

Les sentències següents són equivalents. En la primera no s'especifiquen els camps de la taula perquè s'insereixen tots els valors en l'ordre adequat de les columnes de la taula. En la segona podem veure que es remarca l'ordre dels camps que volem inserir. En aquest cas, els camps coincideixen:

```
1 BEGIN
2 CREATE TABLE persones OF persona;
3 ...
4 INSERT INTO persones VALUES (001, 'Lluis', 'Llatzer', 1985-04-17, 'Alabastros
      5', '612345421');
5 INSERT INTO persones (id, nom, cognom, aniversari, adreça, telefon) VALUES
      (001, 'Lluis', 'Llatzer', 1985-04-17, 'Alabastros 5', '612345421');
6 ...
7 END;
```

No cal introduir tots els valors dels atributs de l'objecte, però en aquest cas és necessari remarcar quins camps emmagatzemaran els valors que s'han introduït. Vegeu-ne un exemple tot seguit:

```
1 INSERT INTO persones (id, nom, cognom, telefon) VALUES (002, 'Isabel', 'Cervera
       Arnau', '634769835');
```

La sentència `INSERT INTO` utilitza l'ordre `ROW` quan un dels atributs de la taula és un tipus compost, i per accedir-hi s'han d'introduir les dades dins d'una estructura de tipus fila. Tal com mostra l'exemple següent:

```
1  BEGIN
2  CREATE TYPE persona AS (
3  id integer,
4  nom varchar(15),
5  cognom varchar(20),
6  ...
7  );
8  CREATE TABLE compte (
9  clientpersona,
10 sucursalinteger,
11 balançreal
12 );
13
14 INSERT INTO compte VALUES (ROW(002, 'Carme', 'Llatzer Roig'), 043, 2500.50);
15 END;
```

Un altre exemple podria ser una inserció en la taula `a_mà`, amb l'atribut `element` com a tipus compost, és a dir, com a columna de la taula:

```
1 BEGIN
2 CREATE TABLE a_mà (
3 element inventari_element,
4 quantitat integer
5 );
```

```
INSERT INTO a_mà VALUES (ROW('galetes', 42, 1.99), 1000);
END;
```

3.4 Modificació i esborrament d'objectes

Les **consultes d'acció** són les que no retornen cap element i modifiquen el valor d'un, alguns o tots els camps d'un, alguns o tots els registres d'una taula. Podem dir que la modificació i eliminació d'objectes són consultes d'acció.

Quan hi ha diverses consultes i es considera que s'han d'executar en bloc per mantenir la integritat de les dades, parlem de **transacció**.

3.4.1 Modificació d'objectes

Per modificar els valors dels atributs d'un objecte en una taula, basant-se en un criteri especific, s'utilitza la sentència `UPDATE`. La seva sintaxi és la que s'indica en la figura 3.8.

Figura 3.8. Sintaxi de la sentència 'UPDATE'

```
UPDATE taula SET camp1=valor1, camp2=valor2, ... campN=valorN
WHERE criteri;
```

Tot seguit en podeu veure un exemple:

```
BEGIN
CREATE TABLE persones OF persona;
...
UPDATE persones SET adreça = '341 Oakdene Ave'
WHERE nom = 'Isabel' AND cognom = 'Cervera Arnau';
...
END;
```

Aquesta sentència modifica l'adreça si es compleix el criteri assignat, que és que hi hagi la persona Isabel Cervera Arnau.

La sentència anterior no genera cap resultat. Per saber quins objectes es canviaran, s'ha d'examinar primer el resultat de la consulta de selecció que utilitzi el mateix criteri i després executar la consulta d'actualització.

```
BEGIN
SELECT * FROM persones WHERE nom = 'Isabel' AND cognom = 'Cervera Arnau';
UPDATE persones SET adreça = '341 Oakdene Ave'
WHERE nom = 'Isabel' AND cognom = 'Cervera Arnau';
END;
```

Si en una actualització esborrem la clàusula `WHERE` tots els objectes de la taula

indicada seran actualitzats; per tant, s'ha d'anar amb compte amb aquest tipus de sentència.

En determinades clàusules, i per a taules heretades, és possible limitar l'esborrament o l'actualització en la taula *pare* (sense que es propagui als registres de les taules *fill*) amb la clàusula `ONLY`. Recuperarem l'exemple utilitzat en l'herència de tipus:

Trobareu més informació sobre l'herència en l'apartat "Herència" d'aquesta unitat.

```
BEGIN
CREATE TABLA persona (
id **serial**,
nom varchar (30),
adreça varchar (30)
);

CREATE TABLE estudiant (
carrera varchar (50),
grup char,
grau int
) INHERITS (persona);

INSERT INTO persona (nom, adreca) VALUES ('Josep Claramunt' , 'Montoliu 12');
INSERT INTO persona (nom, adreca) VALUES ('Lluís Arnau', 'Barcelona 3');
INSERT INTO estudiant (nom, adreca, carrera, grup, grau) VALUES ('Anna Guillen'
     , 'Tarragona 19', 'Psicologia', 'C', 2);

SELECT * FROM persona;

END;
```

```
———————————————————————————— Sortida
id | nom | adreca
———+———————————— +——————————————
1 | Josep Claramunt | Montoliu 12
2 | Lluis Arnau | Barcelona 3
3 | Anna Guillen | Tarragona 19
(3 row)
```

Només actualitzem els camps de la taula *pare* `persona`:

```
BEGIN
...
UPDATE ONLY persona SET nom='Sr. ' || nom;
SELECT * FROM persona;
...
END;
———————————————————————————— Sortida
id | nom | adreca
———+———————————— +——————————————
1 | Sr. Josep Claramunt | Montoliu 12
2 | Sr. Lluis Arnau | Barcelona 3
3 | Anna Guillen | Tarragona 19
(3 rows)
```

Els valors de la tercera fila no es modifiquen perquè pertanyen a una instància de la taula *fill* `estudiant`; es modifiquen els objectes de la taula *pare* concatenant una paraula amb els noms.

La sintaxi bàsica per modificar objectes utilitzant subconsultes és la que es mostra en la figura 3.9.

FIGURA 3.9. Sintaxi bàsica per modificar objectes utilitzant subconsultes

```
UPDATE taula SET camp = nou_valor
        WHERE camp = (SUBCONSULTA);
```

La subconsulta següent actualitza el preu de tots els llibres de l'editorial Brooklyn:

```
BEGIN
UPDATE llibre SET preu = preu + (preu*0.2)
WHERE codi_editorial =
(SELECT codi FROM editorial
WHERE nom = 'Brooklyn');
END;
```

Transaccions

Les transaccions són un concepte fonamental en tots els sistemes de bases de dades. El punt essencial d'una transacció és que engloba múltiples operacions en un sòl pas; per exemple, si considerem la base de dades d'un banc que conté balanços per a diferents comptes de clients, com també el total dels dipòsits de les sucursals. Suposem que volem registrar el pagament de 100 € des del compte del `Client1` al compte del `Client2`, les sentències SQL a executar per a aquesta operació serien les següents:

```
UPDATE compte SET balanç = balanç - 100 WHERE nom = 'Client1';\\ UPDATE
    sucursal SET balanç = balanç - 100 WHERE nom = (SELECT nom_sucursal FROM
    compte WHERE nom = 'Client1');
UPDATE compte SET balanç = balanç + 100 WHERE nom = 'Client2';\\ UPDATE
    sucursal SET balanç = balanç + 100 WHERE nom = (SELECT nom_sucursal FROM
    compte WHERE nom = 'Client2');
```

Com es pot veure hi ha diferents actualitzacions involucrades per finalitzar l'operació. Els treballadors del banc han d'estar segurs que totes aquestes actualitzacions s'executen, o que en cas d'error no se n'executa cap, ja que es podria donar el cas que el `Client2` rebés 100 € sense que fossin extrets del compte del `Client1`. Agrupant les actualitzacions en una sola transacció es pot garantir que en cas d'error no s'executaria cap actualització.

La **transacció** és un conjunt d'ordres que s'executen formant una unitat de treball, és a dir, de manera indivisible o atòmica. No pot finalitzar en un estat intermedi i, si per alguna causa el sistema la cancel·la, es comencen a desfer les ordres executades fins a deixar la base de dades en el seu estat inicial, mantenint la integritat de les dades.

Atomicitat

Propietat que garanteix que una operació es dugui a terme, no quedant-se mai en estats intermedis.

En PostgreSQL les transaccions es configuren de manera que es tanquen en un bloc les ordres SQL que es volen incloure; el bloc ha de començar i acabar amb les ordres `BEGIN` i `COMMIT`, per exemple:

```
BEGIN
UPDATE compte SET balanç = balanç - 100 WHERE nom = 'Client1...';
COMMIT;
```

En el moment en què la clàusula `COMMIT` es passa a PostgreSQL és quan s'escriuen les actualitzacions en la base de dades. Si en algun moment no volem fer `COMMIT` d'alguna operació, es pot utilitzar la clàusula `ROLLBACK` i totes les actualitzacions dins de la transacció es cancel·laran.

Si no es vol fer un `ROLLBACK` complet de la transacció es poden definir marcadors (*save-points*) que indiquen fins on es vol que s'arribi en la transacció, per exemple:

```
1  BEGIN;
2
3  UPDATE compte SET balanç = balanç - 100 WHERE nom = 'Client1';
4  SAVEPOINT marcador1;
5  UPDATE compte SET balanç = balanç + 100 WHERE nom = 'Client2';
6  -- es vol descartar l'actualització per al Client2 i fer-la per al Client3
7  ROLLBACK TO marcador1;
8  UPDATE compte SET balanç = balanç + 100 WHERE nom = 'Client3';
9
10 COMMIT;
```

En l'exemple anterior es veu l'ús de marcadors i *rollbacks;* en aquest cas s'ha fet una actualització sobre el compte del `Client2`, però a continuació es decideix que no s'abonarà la quantitat al `Client2`, sinó que es farà sobre el `Client3`. Per tant, es fa un `ROLLBACK` fins al marcador anterior, `marcador1`, i es continua amb l'actualització següent.

3.4.2 Esborrament d'objectes

Per eliminar objectes d'una taula d'objectes s'utilitza la sentència `DELETE`. Aquesta sentència elimina els objectes complets, no és possible eliminar el contingut d'algun camp en concret. En podeu veure la sintaxi en la figura 3.10.

Figura 3.10. Sintaxi de la sentència 'DELETE'

```
DELETE FROM taula WHERE criteri
```

Una vegada s'han eliminat els objectes utilitzats durant l'esborrament d'objectes no es pot desfer l'operació. Si volem saber quins objectes s'eliminaran, primer hem d'examinar els resultats d'una consulta de selecció que utilitzi el mateix criteri i després executar la sentència d'esborrament.

Per eliminar objectes de manera selectiva s'utilitza la clàusula `WHERE` com es mostra en l'exemple:

```
1  BEGIN
2  DELETE FROM persona
3  WHERE nom = 'Josep Claramunt';
4  END;
```

La sintaxi bàsica per eliminar objectes utilitzant subconsultes és la que es mostra en la figura 3.11.

FIGURA 3.11. Sintaxi bàsica per eliminar objectes utilitzant subconsultes

```
DELETE FROM taula
        WHERE camp IN (SUBCONSULTA);
```

La subconsulta següent elimina tots els llibres de les editorials que tenen publicats llibres d'un autor determinat:

```
1 BEGIN
2 DELETE FROM llibre WHERE codi_editorial IN
3 (SELECT e.codi FROM editorial AS e
4 JOIN llibre ON codi_editorial = e.codi
5 WHERE autor = 'Antoni Llorac');
6 END;
```