Bases de dades en PostgreSQL

Marc Gibert Ginestà
Oscar Pérez Mora

P06/M2009/02152

UOC
www.uoc.edu

Índex

Introducció 5

Objetius 6

1. Característiques de PostgreSQL 7
1.1. Breu història 7
1.2. Prestacions 7
1.3. Limitacions 8

2. Introducció a l'orientació a objectes 9
2.1. El model orientat a objectes 9
2.2. Objectes: classe enfront d'instància 10
2.3. Propietats: atribut enfront d'operació 11
2.4. Encapsulament: implementació enfront d'interfície 11
2.4.1. Atribut enfront de variable d'instància 13
2.5. Herència: jerarquia de classes 14
2.5.1. Tipus i classe 17
2.6. Agregació: jerarquia d'objectes 17
2.7. Persistència 18
2.8. PostgreSQL i l'orientació a objectes 18

3. Accés a un servidor PostgreSQL 20
3.1. La connexió amb el servidor 20
3.2. El client psql 20
3.3. Introducció de sentències 21
3.3.1. Expresions i variables 22
3.4. Procés per lots i formats de sortida 23
3.5. Usar bases de dades 27

4. Creació i manipulació de taules 29
4.1. Creació de taules 29
4.2. Herència 32
4.3. Herència i OID 34
4.4. Restriccions 37
4.4.1. Restriccions de taula 40
4.5. Indexació 42
4.6. Consulta d'informació de bases de dades i taules 42
4.7. Tipus de dades 44
4.7.1. Tipus lògics 44
4.7.2. Tipus numèrics 44
4.7.3. Operadors numèrics 45

4.7.4. Tipus de caràcters 46
4.7.5. Operadors 47
4.7.6. Dates i hores 48
4.7.7. Arrays 48
4.7.8. BLOB 50
4.8. Modificació de l'estructura d'una taula 53

5. Manipulació de dades 54
5.1. Consultes 54
5.2. Actualitzacions e insercions 55
5.3. Transaccions 56

6. Funcions i disparadors 57
6.1. Primer programa 57
6.2. Variables 58
6.3. Sentències 58
6.4. Disparadors 61

7. Administració de PostgreSQL 63
7.1. Instal·lació 63
7.1.1. Internacionalització 64
7.2. Arquitectura de PostgreSQL 65
7.3. L'administrador de postgres 66
7.3.1. Privilegis 67
7.4. Creació de tipus de dades 68
7.5. Plantilla de creació de bases de dades 70
7.6. Copies de seguretat 71
7.7. Manteniment rutinari de la base de dades 72
7.7.1. *vacuum* 72
7.7.2. Reindexació 73
7.7.3. Fitxers de registre 73

8. Client gràfic: pgAdmin3 74

Resum 76

Bibliografia 77

Introducció

PostgreSQL és un gestor de bases de dades orientades a objectes (SGBDOO o ORDBMS en les seves sigles en anglès) molt conegut i usat en entorns de programari lliure perquè compleix els estàndards SQL92 i SQL99, i també pel conjunt de funcionalitats avançades que suporta, cosa que el situa al mateix nivell o a un de millor que molts SGBD comercials.

L'origen de PostgreSQL se situa en el gestor de bases de dades POSTGRES desenvolupat a la Universitat de Berkeley i que es va abandonar en favor de PostgreSQL a partir de 1994. Aleshores ja comptava amb prestacions que el feien únic al mercat i que altres gestors de bases de dades comercials han anat afegint durant aquest temps.

PostgreSQL es distribueix sota llicència BSD, la qual cosa en permet l'ús, la redistribució i la modificació amb l'única restricció de mantenir el *copyright* del programari dels seus autors, en concret el PostgreSQL Global Development Group i la Universitat de Califòrnia.

PostgreSQL pot funcionar en múltiples plataformes (en general, en totes les modernes basades en Unix) i, a partir de la pròxima versió 8.0 (actualment en la seva segona beta), també en Windows de forma nativa. Per a les versions anteriors hi ha versions binàries per a aquest sistema operatiu, però no tenen suport oficial.

Per al seguiment dels exemples i la realització de les activitats, és imprescindible disposar de les dades d'accés de l'usuari administrador del gestor de bases de dades. Encara que en alguns d'aquests els privilegis necessaris seran menors, per als capítols que tracten l'administració de l'SGBDOO serà imprescindible disposar de les credencials d'administrador.

Les sentències o ordres escrites per l'usuari seran en font monoespaiada, i les paraules que tenen un significat especial en PostgreSQL estaran en **negreta**. És important fer notar que aquestes últimes no sempre són paraules reservades, sinó ordres o sentències de psql (el client interactiu de PostgreSQL).

La versió de PostgreSQL que s'ha utilitzat durant la redacció d'aquest material, i en els exemples, és la 7.4, l'última versió estable en aquest moment, encara que no hi haurà cap problema per a executar-los en versions anteriors, fins a la 7.0.

Objectius

L'objectiu principal d'aquesta unitat és conèixer el gestor de bases de dades relacionals amb suport per a objectes PostgreSQL, i comentar tant les seves característiques comunes a altres gestors de bases de dades, com les que el distingeixen dels seus competidors de codi obert.

A més, s'ofereix l'oportunitat d'aplicar els coneixements apresos en el mòdul referent a SQL sobre un sistema gestor de base de dades real i examinar les diferències entre l'estàndard i la implementació que en fa l'SGBD.

Finalment, es presenten les tasques d'administració de l'SGBD més habituals que un usuari ha de dur a terme com a administrador de PostgreSQL.

1. Característiques de PostgreSQL

En aquest apartat comentarem les característiques més rellevants d'aquest SGBD amb suport per a objectes, tant les seves prestacions més interessants o destacades, com les limitacions en el seu disseny o en implementació dels estàndards SQL.

També és interessant conèixer una mica la seva història, ja que tant per les persones que han participat en el seu desenvolupament com per la seva relació amb altres gestors de bases de dades, ens ajudarà a tenir una millor visió de la seva evolució.

1.1. Breu història

La història de 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 del qual era POSTGRES. En el seu disseny es van incloure alguns conceptes avançats en bases de dades i suport parcial a l'orientació a objectes.

POSTGRES va ser comercialitzat per Illustra, una empresa que posteriorment va formar part d'Informix (que comercialitzava el conegut SGBD del mateix nom, recentment absorbida per IBM i el seu DB/2). Va arribar un moment en què mantenir el projecte absorbia massa temps als investigadors i acadèmics, per la qual cosa el 1993 es va alliberar 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 a alliberar-ne posteriorment el codi amb el nom de Postgres95. El projecte incloïa múltiples canvis en el codi original que en milloraven el rendiment i la llegibilitat.

El 1996 el nom va canviar a PostgreSQL reprenent la seqüència original de versions, per la qual cosa es va alliberar la versió 6.0.

1.2. Prestacions

PostgreSQL destaca per la seva amplíssima llista de prestacions que el fan capaç de competir amb qualsevol SGBD comercial:

Els desenvolupadors de projectes basats en programari lliure tenen molt en compte PostgreSQL quan els requisits d'un projecte exigeixen prestacions d'alt nivell.

- Està desenvolupat en C, amb eines com Yacc i Lex.

- L'API d'accés a l'SGBD està disponible en C, C++, Java, Perl, PHP, Python i TCL, entre d'altres.

- Té un ric conjunt de tipus de dades i, a més, en permet l'extensió mitjançant tipus i operadors definits i programats per l'usuari.

- La seva administració es basa en usuaris i privilegis.

- Les seves opcions de connectivitat inclouen TCP/IP, *sockets* Unix i *sockets* NT, a més de suportar completament ODBC.

- Els missatges d'error poden estar en espanyol i fer ordenacions correctes amb paraules accentuades o amb la lletra *ñ*.

- És altament confiable quant a estabilitat.

- Es pot estendre amb llibreries externes per a suportar encriptació, recerques per similitud fonètica (soundex), etc.

- Control de concurrència multiversió, cosa que millora sensiblement les operacions de bloqueig i transaccions en sistemes multiusuari.

- Suport per a vistes, claus foranes, integritat referencial, disparadors, procediments emmagatzemats, subconsultes i gairebé tots els tipus i operadors suportats en SQL92 i SQL99.

- Implementació d'algunes extensions d'orientació a objectes. En PostgreSQL és possible definir un nou tipus de taula a partir d'una altra prèviament definida.

1.3. Limitacions

Les limitacions d'aquest tipus de gestors de bases de dades se solen identificar molt fàcilment analitzant les prestacions que tenen previstes per a les pròximes versions. Trobem el següent:

- Punts de recuperació dins de transaccions. Actualment, les transaccions avorten completament si es troba un error durant la seva execució. La definició de punts de recuperació permetrà recuperar millor transaccions complexes.

- No suporta *tablespaces* per a definir on emmagatzemar la base de dades, l'esquema, els índexs, etc.

- El suport a orientació a objectes és una simple extensió que ofereix prestacions com l'herència, no un suport complet.

2. Introducció a l'orientació a objectes

Atès que PostgreSQL inclou extensions d'orientació a objectes (encara que no és, com ja hem comentat, un SGBDOO complet), és interessant repassar alguns dels conceptes relacionats amb aquest paradigma de programació i estructuració de dades.

2.1. El model orientat a objectes

El 1967, el llenguatge de programació Simula aplicava algunes idees per modelar aspectes de la realitat d'una manera molt més directa que els mètodes tradicionals. Des de llavors, l'orientació a objectes (OO) ha adquirit cada vegada més popularitat en demostrar els seus avantatges, entre els quals hi ha els següents:

- Permet un modelatge més "natural" de la realitat.
- Facilita la reutilització de components de programari.
- Ofereix mecanismes d'abstracció per a mantenir controlable la construcció de sistemes complexos.

Al mercat apareixen constantment eines i llenguatges de programació autodenominats *orientats a objectes* i els que ja hi ha evolucionen ràpidament incloent noves característiques d'orientació a objectes (OO). De la mateixa manera, s'han desenvolupat múltiples mètodes i metodologies amb aquest enfocament, cada una amb aportacions pròpies que arriben, de vegades, a resultar contradictòries entre si. S'ha aconseguit la creació d'un llenguatge unificat per al modelatge, anomenat precisament UML (*unified modeling language*). La intenció d'UML és ser independent de qualsevol metodologia i és precisament aquesta independència la que el fa important per a la comunicació entre desenvolupadors, ja que hi ha moltes metodologies i estan en constant evolució.

Lamentablement, malgrat que hi ha hagut avenços importants, les ciències de la computació no han creat encara una definició de **model d'objectes** com a tal. En un panorama com aquest, és indispensable, almenys, l'existència d'un *model informal d'objectes* que orienti l'evolució de la tecnologia i que tingui l'aprovació dels experts en la matèria. Un model així permetria el seu estudi consistent per part dels professionals de les tecnologies de la informació, facilitaria la creació de millors llenguatges i eines i, el que és més important, definiria els estàndards per a una metodologia de desenvolupament consistent i aplicable.

Tanmateix, aquest model no existeix, la qual cosa provoca inconsistències fins i tot en el tractament dels principis i conceptes bàsics de l'OO. Per això, és freqüent trobar errors greus en el desenvolupament de sistemes OO i, el que encara és pitjor, s'implementen solucions de dubtosa validesa en eines de desenvolupament que s'anomenen *orientades a objectes*.

Fins i tot sense haver assolit la maduresa, l'orientació a objectes és el paradigma que permet solucionar millor els molts i variats problemes que hi ha en el desenvolupament de programari. En els apartats següents analitzarem els conceptes bàsics d'aquest model per identificar alguns dels problemes que encara ha de resoldre, la qual cosa facilitarà la comprensió i avaluació de mètodes i eines OO.

2.2. Objectes: classe enfront d'instància

Els **objectes** són abstraccions que realitzem del món que ens envolta i que identifiquem per les seves propietats. Per a l'OO tot és un objecte.

Cada objecte té una existència una mica independent dels altres objectes; és a dir, té **identitat** pròpia. Encara que dos objectes tinguin exactament els mateixos valors, no per això seran el mateix objecte, continuaran essent entitats diferents. En els models OO, la identitat es representa amb l'identificador d'objecte, IDO (OID en anglès, d'*object identifier*). Teòricament, l'IDO d'un objecte és únic i irrepetible en el temps i l'espai.

Els IDO són el mecanisme que permet fer referència a un objecte des d'un altre. D'aquesta manera les referències teixeixen les relacions entre objectes.

Funció de l'IDO

L'IDO permet que dos objectes idèntics es puguin diferenciar, no és important que l'usuari conegui els IDO, el que és important és que els diferencïi el sistema.

Tots els objectes que comparteixen les mateixes propietats es diu que pertanyen a la mateixa **classe**. En els models OO, les classes li roben el paper central als objectes, ja que és per mitjà d'aquelles com es defineixen les propietats d'aquests i, a més, s'utilitzen com a plantilles per a crear objectes.

Elements fonamentals en OO

objecte	classe

En crear un objecte utilitzant la definició donada per una classe, obtenim un valor per a aquest, és el que s'anomena una **instància** de l'objecte. Durant l'execució dels programes es treballa amb instàncies. Com a concepte, la instància és equivalent a una tupla (fila) concreta en una taula d'una base de dades.

2.3. Propietats: atribut enfront d'operació

Les propietats dels objectes poden ser de dos tipus, dinàmiques i estàtiques. Un **atribut** representa una propietat estàtica d'un objecte (color, cost, edat, etc.). Una **operació** representa una propietat dinàmica; és a dir, una transformació sobre un atribut o una acció que pot realitzar.

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 el seu estat. La seqüència d'estats per la qual passa un objecte en executar operacions defineixen el seu **comportament**.

La possibilitat de definir comportaments complexos és el que fa diferent l'OO.

Propietats dels objectes

objecte (instància)	classe
atributs (estat)	atributs
operacions (comportament)	operacions

2.4. Encapsulament: implementació enfront d'interfície

L'estructura interna dels objectes ha de quedar oculta a l'usuari d'un objecte, no necessita conèixer-la per a interactuar-hi. Els objectes es conceben com una càpsula l'interior de la qual és ocult i no pot ser alterat directament des de l'exterior.

L'estructura interna d'un objecte s'anomena **implementació** i la part visible, la que es presenta a l'exterior, **interfície**. La interfície es defineix pels seus atributs i operacions.

La implementació d'una operació es coneix com a **mètode**. La implementació d'un atribut es realitza generalment amb **variables d'instància**.

Els tipus de dades abstractes

Els tipus de dades abstractes (TDA) obeeixen al mateix principi d'independència de la implementació. La diferència respecte als objectes és que aquests inclouen les dades i les operacions a la mateixa càpsula.

Encapsulament

Classe	
implementació	**interfície**
variables	atributs
mètodes	operacions

L'encapsulament comporta els avantatges següents:

- La modificació interna (de la implementació) d'un objecte per a corregir-lo o millorar-lo no afecta els seus usuaris.

- La dificultat inherent a la modificació de la implementació d'un objecte és independent de la mida total del sistema. Això permet que els sistemes evolucionin amb més facilitat.

- La simplificació en l'ús de l'objecte en ocultar els detalls del seu funcionament i presentar-lo en termes de les seves propietats. En elevar el nivell d'abstracció es disminueix el nivell de complexitat d'un sistema. És possible modelar sistemes de mida més gran amb menys esforç.

- Constitueix un mecanisme d'integritat. La dispersió d'un error a través de tot el sistema és menor, ja que en presentar una divisió entre interfície i implementació, els errors interns d'un objecte troben una barrera en l'encapsulament abans de propagar-se a la resta del sistema.

- Permet la substitució d'objectes amb la mateixa interfície i diferent implementació. Això permet modelar sistemes de mida més gran amb menys esforç.

Aquestes característiques de l'encapsulament han contribuït en gran manera a la bona reputació de l'OO.

Paradoxalment, l'encapsulament, tot i ser un dels conceptes bàsics en l'OO, no sempre s'interpreta i s'aplica correctament. Especialment pel que fa a encapsulament d'atributs.

Diferenciem operació i mètode per mitjà d'un exemple.

Considerem tres objectes: polígon, cercle i punt.

Als tres se'ls sol·licita l'operació d'imprimir. En aquesta situació, tenim que:

- L'operació sol·licitada és la mateixa, perquè el significat del resultat és el mateix.

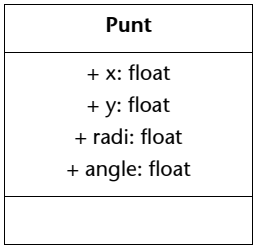
- Cada objecte executa l'operació de manera diferent; és a dir, amb un mètode diferent.

- Cada objecte, internament, pot tenir més d'un mètode i selecciona el més apropiat segons les circumstàncies.

Les operacions no són exclusives dels tipus d'objecte, 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.

Vegem ara la diferència entre atributs i variables d'instància, que pot semblar més subtil.

Un atribut és la vista externa d'una propietat estàtica d'un objecte. La representació interna pot variar, els atributs poden implementar-se també amb mètodes. Prenguem com a exemple l'objecte punt amb els atributs que es mostren a continuació:



Els atributs d'un punt es poden definir en coordenades angulars o rectangulars; en aquest cas, és possible conèixer ambdues representacions. En la implementació d'aquests atributs, dos poden ser variables d'instància i els altres dos s'implementen com a mètodes, que es calculen per mitjà dels primers.

Des de l'exterior no ha de ser possible conèixer la representació triada internament. Es pot canviar la implementació dels atributs sense alterar la interfície. En alguns casos es pot permetre fins i tot al sistema l'elecció de la representació interna d'un atribut de la mateixa manera que una operació tria entre diversos mètodes disponibles.

2.4.1. Atribut enfront de variable d'instància

Un atribut especifica una qualitat d'un objecte; una variable d'instància especifica com s'emmagatzemen els valors per a aquesta qualitat.

Els atributs

Un atribut pot ser emmagatzemat en una variable o ser calculat per un mètode.

Considerem tres objectes (nom, foto, vídeo), dels quals necessitem conèixer la mida i preveure, així, l'espai necessari per a emmagatzemar-los en disc.

En aquesta situació tenim que:

- L'atribut és el mateix, perquè la seva lectura té el mateix significat.
- Cada objecte implementa l'atribut de manera diferent. Sense importar la implementació, externament tots els atributs lliuren el mateix tipus de valor. Per exemple:
  - El nom pot utilitzar un byte com a variable d'instància, perquè la mida d'un nom no pot ser més gran de 255 caràcters, o es pot implementar un mètode que calculi la mida en temps d'execució.
  - La foto utilitzarà dos o quatre bytes.
  - El vídeo pot emmagatzemar el valor de mida en múltiples de K.
- Cada objecte pot tenir implementacions alternatives que s'adaptin a les circumstàncies.

Implementació del encapsulament

Classe	
implementació	**interfície**
variables	operacions de lectura/escriptura
mètodes	operacions

Lamentablement, els llenguatges de programació utilitzats comunament no implementen mecanismes adequats per a l'encapsulament dels atributs, si hi ha que, fins i tot, permeten l'accés públic a variables d'instància. A continuació, analitzarem les greus conseqüències d'aquest fet.

Accedir a un atribut és, en realitat, una operació que pot ser de lectura o d'escriptura. Per aquest motiu, freqüentment es defineix l'encapsulament com l'ocultació de totes les variables, cosa que permet l'accés de l'exterior només per a operacions. Quan el llenguatge de programació no ofereix independència de la implementació en els atributs, s'han de definir una variable d'instància i dos mètodes per cada atribut: *LlegirAtribut* i *EscriureAtribut*.

Les bases de dades relacionals tenen perfectament diferenciada la interfície de la implementació en els seus tipus de dades: la manera d'emmagatzemar-los és completament independent de la manera de consultar-los o guardar-los. No es conceben les operacions com a internes als objectes.

L'encapsulament ha estat considerat com un principi central de l'orientació a objectes i atemptar contra ell significa per a molts trencar amb les seves regles fonamentals. Tanmateix, les bases de dades orientades a objectes tenen entre les seves funcions la realització de consultes, que necessita accedir als atributs dels objectes. Atès que els objectes s'implementen amb variables, en accedir-hi es trenca l'encapsulament.

La majoria dels llenguatges orientats a objectes permeten trencar l'encapsulament de manera parcial, declarant variables com a públiques. L'encapsulament, en aquests casos, es proporciona com un mecanisme opcional, ja que l'usuari pot declarar totes les variables públiques i, per tant, accessibles directament.

Altres llenguatges implementen operacions de lectura/escriptura que permeten accedir a les variables sense trencar l'encapsulament.

2.5. Herència: jerarquia de classes

L'herència es defineix com el mecanisme mitjançant el qual s'utilitza la definició d'una classe anomenada "pare", per a definir una nova classe anomenada "fill", que pot heretar-ne els atributs i les operacions.

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

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

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 pot tornar a implementar-los.

L'herència de tipus defineix relacions **és-un** entre classes, on la classe "fill" té totes les propietats del "pare", però el "pare" no té totes les propietats del "fill".

Exemple

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

Considerem una referència mascota que és de tipus animal, en algun llenguatge de programació.

```
lamevamascota: Animal;
```

Pot fer referència a objectes de tipus animal, o tipus derivats d'aquest, com gos, gat o canari, per exemple.

```
lamevamascota = new Canari;
```

Es construeix un nou canari i s'hi fa referència com a mascota.

La propietat de substituir objectes que descendeixen del mateix pare es coneix com a **polimorfisme**, i és un mecanisme molt important de reutilització en l'OO.

La referència al tipus animal és una referència polimorfa, ja que es pot referir a tipus derivats d'animal. Per mitjà d'una referència polimorfa es poden sol·licitar operacions sense conèixer-ne el tipus exacte.

```
lamevamascota.menjar();
```

L'operació menjar té el mateix significat per a tots els animals. Com ja hem comentat, cada un utilitzarà un mètode diferent per a executar l'operació.

Per a conèixer el tipus exacte de l'objecte en qüestió, s'utilitza l'operador d'**informació de tipus**. D'aquesta manera es pot accedir a les propietats específiques d'un tipus d'objecte que no estan en els altres tipus.

En aquest exemple anomenem l'operador informació de tipus, **instància-de**.

```
if (lamevamascota instància-de Canari)
  lamevamascota.cantar();
```

Si la mascota és una instància del tipus Canari llavors se li sol·licitarà cantar, que és una propietat que no tenen totes les mascotes.

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

En una herència múltiple, pot ocórrer que en ambdues superclasses hi hagi propietats amb els mateixos noms, situació que es denomina **col·lisió de noms**. A continuació, es relacionen els possibles casos de col·lisió de noms en l'herència de tipus:

- Els noms són iguals perquè es refereixen a la mateixa propietat (ja hem vist exemples d'això: l'operació imprimir i l'atribut mida). En aquest cas no hi ha conflicte perquè el significat és clar: és la mateixa propietat, només cal definir una implementació adequada.

- Els noms són iguals però tenen significats diferents. Aquesta situació és possible perquè el modelatge és una tasca subjectiva i se soluciona canviant els noms de les propietats heretades que tinguin conflicte.

L'herència múltiple no comporta problemes per a l'herència de tipus, ja que no pretén la reutilització de codi, sinó el control conceptual de la complexitat dels sistemes mitjançant esquemes de classificació.

Pel que fa a l'herència estructural, que, recordem, consisteix en el fet que la subclasse hereta les variables d'instància i els mètodes de la superclasse –és a dir, la implementació–, la cosa canvia.

Per a entendre millor l'herència estructural, direm informalment que representa una relació **funciona-com-a**. Per exemple, es pot utilitzar per a definir un avió prenent com a superclasse au, d'aquesta manera la capacitat de volar de l'au queda implementada en l'avió. Un avió no **és-una** au, però podem dir que **funciona-com-a** au.

En aplicar l'herència d'aquesta manera es dificulta la utilització del polimorfisme: encara que un objecte funcioni internament com un altre, no es garanteix que externament pugui prendre el seu lloc perquè **funciona-com-a**.

L'objectiu de l'herència estructural és la reutilització de codi, encara que en alguns casos, com l'exemple anterior, pugui fer conceptualment més complexos els sistemes.

Sempre que és possible aplicar l'herència de tipus, es pot aplicar l'herència estructural, per la qual cosa la majoria dels llenguatges de programació no fan distinció entre els dos tipus d'herència.

Exemple

Si un canari **és-un** animal, llavors un canari **funciona-com-a** animal i, a més, té altres propietats especifiques de canari.

Els llenguatges de programació normalment no fan distinció entre l'herència estructural i l'herència de tipus.

L'herència estructural múltiple permet heretar variables i mètodes de diverses superclasses, però sorgeixen problemes que no són fàcils de resoldre, especialment amb les variables d'instància.

Per a resoldre el conflicte d'una variable d'instància duplicada, es pot optar per les solucions següents:

- Canviar els noms, cosa que pot provocar conflictes en els mètodes que les utilitzen.
- Eliminar una de les variables. Però pot passar que realitzin alguna funció independent i en aquest cas seria un error eliminar-ne una.
- No permetre herència múltiple quan hi ha variables duplicades.

Com es pot observar, no és fàcil solucionar conflictes entre variables d'instància. Per això molts llenguatges opten per diversos mecanismes incloent la prohibició de l'herència múltiple.

2.5.1. Tipus i classe

Hem d'advertir que la majoria de llenguatges de programació no diferencien els conceptes de tipus i classe i que la diferència que estableixen alguns autors no és gaire clara. De totes maneres, la tendència seria definir aquests conceptes de la manera següent:

- Un tipus és un conjunt d'objectes que comparteixen la mateixa interfície.
- Una classe és un conjunt d'objectes que comparteixen la mateixa implementació.

Una solució que s'aplica és incloure en el llenguatge el concepte d'**interfície** que defineix només les operacions d'una classe, però no ofereix alternatives per als atributs. Tanmateix, amb la diferenciació entre classes i interfícies no s'aconsegueix la diferenciació entre els dos tipus d'herència, ja que les classes s'utilitzen per a representar relacions **és-un**.

2.6. Agregació: jerarquia d'objectes

Els objectes són, per naturalesa, complexos; és a dir, estan compostos d'objectes més petits. Un sistema d'informació ha de reflectir aquesta propietat dels objectes de manera natural. En una base de dades relacional, un objecte complex ha de ser descompost en les seves parts més simples per a ser emmagatzemat. En extreure'l, és necessari acoblar cada una de les parts.

Exemple

Un automòbil està compost de carrosseria, motor, rodes, etc.

Per aquest motiu, el model relacional comporta problemes quan s'utilitza en aplicacions com el CAD, on els objectes que es processen són molt complexos.

Les bases de dades d'objectes han de proporcionar la facilitat d'obtenir un objecte complex en una sola consulta de manera transparent. En aquest cas, els

apuntadors són un mecanisme excel·lent per a representar composició, ja que permeten accedir ràpidament a les parts components d'un objecte, sense importar el seu lloc d'emmagatzemament.

Les bases de dades requereixen independència de l'aplicació, la qual cosa provoca un conflicte conceptual quan es treballa amb objectes compostos: les bases de dades han d'emmagatzemar informació independent de l'aplicació perquè noves aplicacions puguin fer diferents interpretacions de la informació original; però amb els objectes compostos això no és tan senzill, ja que solen tenir una sola interpretació, o més ben dit, una sola manera de ser consultats en una base de dades.

2.7. Persistència

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

S'han proposat diversos mecanismes per a implementar la persistència en els llenguatges de programació, entre els quals podem destacar els següents:

- Arxius plans. Es creen arxius per a emmagatzemar els objectes en el format desitjat pel programador. Els objectes es carreguen en obrir el programa i es guarden en finalitzar. Aquesta és l'opció més accessible per a tots els llenguatges de programació.

- Bases de dades relacionals. Els objectes són mapats en taules, un mòdul del programa s'encarrega de fer les transformacions objecte-relacionals. Aquest enfocament consumeix molt de temps en realitzar el mapatge. Hi ha algunes eines que realitzen mapatges semiautomàtics.

- Bases d'objectes. Els objectes són emmagatzemats 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 res sobre els detalls d'implementació.

- Persistència transparent. Els objectes són emmagatzemats i són recuperats pel sistema quan aquest 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, per tant, cap altre llenguatge per a interactuar amb les bases de dades.

2.8. PostgreSQL i l'orientació a objectes

L'argument a favor de les bases de dades objecte-relacionals sosté que permet realitzar una migració gradual de sistemes relacionals als orientats a objectes

i, en algunes circumstàncies, que coexisteixin ambdós tipus d'aplicacions durant algun temps.

El problema d'aquest enfocament és que no és fàcil aconseguir la coexistència de dos models de dades diferents com són l'orientació a objectes i el model relacional. És necessari equilibrar d'alguna manera els conceptes d'un model i de l'altre sense que entrin en conflicte.

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

1r. enfocament: les classes defineixen tipus de taules
2n. enfocament: les classes defineixen tipus de columnes

Atès que en el model relacional les columnes estan definides per tipus de dades, el més natural és fer correspondre les columnes amb les classes.

	1er. enfocament	2n. enfocament
Els objectes són	valors	tuples
Les classes són	dominis	taules

PostgreSQL implementa els objectes com a tuples i les classes com a taules. Encara que també és possible definir nous tipus de dades mitjançant els mecanismes d'extensió.

Els mecanismes d'extensió

No és habitual que l'usuari utilitzi els mecanismes d'extensió ja que es consideren mecanismes avançats.

Atès que les taules són classes, es poden definir com a herència d'altres. Les taules derivades són polimorfes i hereten tots els atributs (columnes) de la taula *pare* (inclosa la seva clau primària). Si no es manegen amb precaució, les taules polimorfes poden conduir a errors d'integritat en duplicar claus primàries. PostgreSQL suporta algunes extensions del llenguatge SQL per a crear i gestionar aquest tipus de taules.

Veurem aquests conceptes més en detall en el subapartat 4.2 d'aquesta unitat.

3. Accés a un servidor PostgreSQL

3.1. La connexió amb el servidor

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

Una vegada comprovat el correcte funcionament del servidor, hem de disposar de les credencials* necessàries per a la connexió. Per a simplificar, suposarem que disposem de les credencials de l'administrador de la base de dades (normalment, usuari PostgreSQL i la seva contrasenya).

*** Diferents tipus de credencials permeten diferents nivells d'accés.**

En l'apartat que fa referència a l'administració de PostgreSQL es comenten detalladament els aspectes relacionats amb el sistema d'usuaris, contrasenyes i privilegis de l'SGBD.

3.2. El client psql

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

El pas següent és conèixer el nom d'una base de dades resident en el servidor. L'ordre següent permet conèixer les bases de dades residents en el servidor:

```
~$ psql -l
List of databases
Name       | Owner    | Encoding
-----------+----------+-----------
demo       | postgres | SQL_ASCII
template0  | postgres | SQL_ASCII
template1  | postgres | SQL_ASCII
(3 rows)
~$
```

Per a fer una connexió, es requereixen les dades següents:

- Servidor. Si no s'especifica, s'utilitza *localhost*.
- Usuari. Si no s'especifica, s'utilitza el nom d'usuari Unix que executa psql.
- Base de dades.

Exemples de l'ús de psql per a connectar-se amb un servidor de bases de dades

```
~$ psql -d demo
~$ psql demo
```

Les dues formes anteriors executen psql amb la base de dades *demo*.

```
~$ psql -d demo -U yo
~$ psql demo yo
~$ psql -h servidor.org -U usuari -d basedades
```

A partir del fragment anterior, el client psql mostrarà una cosa similar al següent:

```
Welcome to psql, the PostgreSQL interactive terminal.
Type: \copyright for distribution terms
\h for help with SQL commands
\? for help on internal slash commands
\g or terminate with semicolon to execute query
\q to quit
demo=#
```

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

Notació

Hem utilitzat les dues ordres d'ajuda que ofereix el llenguatge:

- **\h** Explica la sintaxi de sentències SQL.
- **\?** Mostra les ordres internes disponibles.

Per a sortir d'ajuda, es pressiona la tecla 'q'.

Les sentències SQL s'envien directament al servidor per a la seva interpretació, les ordres internes tenen la forma *ordre* i ofereixen opcions que no estan incloses en SQL i són interpretades internament per psql.

Per a acabar la sessió amb psql, utilitzem l'ordre **\q** o podem polsar Ctrl-D.

3.3. Introducció de sentències

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

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

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

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

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 a visualitzar el contingut de la memòria intermèdia (*buffer*) on ha emmagatzemat la sentència, disposem de l'ordre '\p':

```
demo=> SELECT
demo-> 2 * 10 + 1
demo-> \p
   SELECT
   2 * 10 + 1
demo-> \g
   ?column?
   ----------
      21
   (1 row)
demo=>
```

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

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

3.3.1. Expresions i variables

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

```
demo=>\set var1 demostracio
```

Aquesta sentència crea la variable 'var1' i li assigna el valor 'demostració'. Per a recuperar el valor de la variable, simplement l'haurem d'incloure pre-

cedida de ': ' en qualsevol sentència o bé veure'n el valor mitjançant l'ordre 'echo':

```
demo=# \echo :var1
demostracio
demo-#
```

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

```
demo=# \echo :DBNAME :ENCODING :HOST :PORT :USER;
demo LATIN9 localhost 5432 postgres
demo=#
```

L'ús de variables pot ajudar en l'execució de sentències SQL:

```
demo=> \set var2 'lameva_taula'
demo=> SELECT * FROM :var2;
```

S'ha de tenir molta cura amb l'ús de les cometes i també és important tenir en compte que dins de cadenes de caràcters no se substitueixen variables.

3.4. Procés per lots i formats de sortida

A més de ser interactiu, psql pot processar ordres per lots emmagatzemats en un arxiu del sistema operatiu mitjançant la sintaxi següent:

```
$ psql demo -f demo.psql
```

Encara que l'ordre següent també funciona en el mateix sentit, no és recomanable usar-la perquè d'aquesta manera, psql no mostra informació de depuració important, com els números de línia on es localitzen els errors, si n'hi ha:

```
$ psql demo < demo.psql
```

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

- Especificant el fitxer destinació* directament en finalitzar una sentència:

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

*** Hem emmagatzemat el resultat en el fitxer '*/tmp/a.txt*'.**

- Mitjançant una *pipe* enviem la sortida a una ordre Unix:

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

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.

- Mitjançant l'ordre '\o' es pot indicar on ha d'anar la sortida de les sentències SQL que s'executin d'ara endavant:

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

- Es pot sol·licitar l'execució d'una sola ordre i acabar immediatament mitjançant la forma següent:

```
$ psql -d demo -c "ordre sql"
```

- 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'. Vegem, en primer lloc, l'especificació del format de sortida:

La sortida del fitxer

En haver especificat que es vol la sortida en HTML, la podríem redirigir a un fitxer (ja hem vist com s'ha de fer) i generar un arxiu HTML que permetés veure el resultat de la consulta mitjançant un navegador web convencional.

```
demo=# \pset format html
Output format is html.
demo=# 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>
demo=#
```

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

Tenim també multitud de variables per a 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:

La sortida d'aquest fitxer

Amb aquesta configuració, i dirigint la sortida a un fitxer, generaríem un fitxer *CSV* preparat per a ser llegit en un full de càlcul o un altre programa d'importació de dades.

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

Per a poder realitzar els exemples de la resta d'aquest apartat, s'ha de processar el contingut del fitxer *demo.sql* tal com es transcriu a continuació.

Contingut del fitxer *demo.psql*

```
--drop table productes;
--drop table proveïdors;
--drop table preus;
--drop table guany;

create table productes (
   part               varchar(20),
   tipus              varchar(20),
   especificació      varchar(20),
   psuggerit          float(6),
   clau               serial,

   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,
   crèdit        bool,
   efectiu       bool,
   primary key   empresa)
);

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

create table guany(
   venda      varchar(16),
   factor     decimal (4,2)
);

insert into guany values('a l'engròs',1.05);
insert into guany values('al detall',1.12);

create table preus (
   empresa                 varchar(20) not null,
   clau                    int not null,
   preu    float(6),

   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);
```

3.5. Usar bases de dades

L'ordre següent informa sobre les bases de dades actualment en l'SGBD.

```
demo=# \l
        List of databases
Name       |Owner     | Encoding
-----------+----------+----------
 demo      | postgres | LATIN9
 template0 | postgres | LATIN9
 template1 | postgres | LATIN9
(3 rows)
demo=#
```

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

```
demo=# \c demo
You are now connected to database "demo".
demo=#
```

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

```
demo=# \d
                List of relations
 Schema |        Name         | Type     |  Owner
-------+--------------------+---------+----------
 public | guany               | table    | postgres
 public | preus               | table    | postgres
 public | productes           | table    | postgres
 public | productes_clave_seq | sequence | postgres
 public | proveidors          | table    | postgres
(5 rows)
```

L'ordre **\d** és útil per a 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.

Consulta de les columnes de cada una de les taules:

```
demo-# \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)
```

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

```
mysql> create database prova;
```

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

```
mysql> drop database prova;
```

4. Creació i manipulació de taules

4.1. Creació de taules

Una vegada connectats a una base de dades, la sentència SQL **create table** permet crear les taules que necessitem:

```
demo=#     create table persona (
demo(#     nom varchar(30),
demo(#     adreça varchar(30)
demo(# );
CREATE
```

L'ordre **drop table** permet eliminar taules:

```
demo=# drop table persona;
```

La taula acabada de crear apareix ara en la llista de taules de la base de dades en ús:

```
demo=# \dt
List of relations
Name     |Type   |Owner
---------+-------+--------
 persona | table | quiron
(1 row)
```

Podem consultar la seva descripció mitjançant l'ordre ***\d taula:***

```
demo=# \d persona
Table "persona"
Column     |Type                    | Modifiers
---------- + ---------------------- +-----------
    nom    | character varying(30)  |
 adreça    | character varying(30)  |
```

La taula està preparada per a inserir-hi alguns registres.

```
demo=# insert into persona values ( 'Alexandre el Gran' , 'Babilònia' );
INSERT 24756 1
demo=# insert into persona values ( 'Federico García Lorca ' , 'Granada 65' );
INSERT 24757 1
```

El número amb què respon l'ordre *insert* es refereix a l'OID del registre inserit.

Aquest aspecte s'explicarà en detall més endavant.

Les consultes es realitzen amb la sentència `SQL` **`select`**. En aquest cas sol·licitem que ens mostri totes les columnes dels registres en la taula *persona*:

```
demo=# select * from persona;
nom                     |adreca
---------------------- +------------
 Alexandre el Gran      | Babilònia
 Federico García Lorca  | Granada 65
(2 rows)
demo=#
```

Les taules creades en PostgreSQL inclouen, per defecte, 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 a 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* (OID) que s'implementa com un enter positiu de 32 bits. Quan s'insereix un nou registre en una taula se li assigna un número consecutiu com a OID, i el TABLEOID de la taula que li correspon.

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.

Per a observar les columnes ocultes, hi hem de fer referència específicament en l'ordre *select*:

```
demo=# select oid, tableoid, * from persona;
oid    |tableoid  |nom                    |adreca
-------+----------+----------------------+------------
 17242 |    17240 | Alexandre el Gran    | Babilònia
 17243 |    17240 | Federico García Lorca| Granada 65
(2 rows)
demo=#
```

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

Exemple de la utilització d'OID per a enllaçar dues taules

Reprenem la taula *persona* i construïm una nova taula per emmagatzemar els telèfons:

```
demo=# create table telefon (
demo(# tipo char(10),
demo(# numero varchar(16),
demo(# propietari oid
demo(# );
CREATE
```

La taula *telèfon* inclou la columna *propietari* de tipus OID, que emmagatzemarà la referència als registres de la taula *persona*. Agreguem dos telèfons a 'Alexandre el Gran'; per a això utilitzem el seu OID que és 17242:

```
demo=# insert into telefon values( 'mòbil' , '12345678', 17242 );
demo=# insert into telefon values( 'casa' , '987654', 17242 );
```

Les dues taules estan vinculades per l'OID de persona.

```
demo=# select * from telefon;
tipus        |numero        | propietari
------------+ ------------+-------------
      mòbil |    12345678 |   17242
       casa |      987654 |   17242
(2 rows)
```

L'operació que ens permet unir les dues taules és *join*, que en aquest cas uneix *telèfon* i *persona*, utilitzant per a això la igualtat de les columnes *telefon.propietari* i *persona.oid*:

```
demo=# select * from telefon join persona on (telefon.propietari = persona.oid);
tipus        |numero        | propietari   |nom                 | adreca
----------- + -------------+ -------------+ ------------------ +-----------
      mòbil |    12345678  |       17242  | Alexandre el Gran  | Babilònia
       casa |      987654  |       17242  | Alexandre el Gran  | Babilònia
(2rows)
```

Els OID de PostgreSQL presenten algunes deficiències:

- 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 succeeix, per descomptat, amb bases de dades molt grans.

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

- Els OID no milloren el rendiment. Són, en realitat, una columna amb un nombre enter com a valor.

Els desenvolupadors de PostgreSQL proposen la següent alternativa per a usar OID d'una manera absolutament segura:

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

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

Pels motius anteriors, no és recomanable l'ús d'OID fins que noves versions de PostgreSQL els corregeixin. Si s'usen, convé seguir les recomanacions anteriors.

És possible crear taules que no incloguin la columna OID mitjançant la notació següent:

```
create table persona (
nom varchar(30),
adreça varchar(30)
)without oids;
```

4.2. Herència

PostgreSQL ofereix com a característica particular l'herència entre taules, que permet definir una taula que hereti d'una altra prèviament definida, segons la definició d'herència que hem vist en capítols anteriors.

Reprenguem la taula *persona* definida com segueix:

```
create table persona (
nom varchar (30),
adreça varchar (30)
);
```

A partir d'aquesta definició, creem la taula *estudiant* com a derivada de *persona*:

```
create table estudiant (
demo(# carrera varchar(50),
demo(# grup char,
demo(# grau int
demo(# ) inherits ( persona );
CREATE
```

En la taula *estudiant* es defineixen les columnes *carrera, grup* i *grau*, però en sol·licitar informació de l'estructura de la taula observem que també inclou les columnes definides en *persona*:

```
demo=# \d estudiant
Table "estudiant"
Column     |Type                    | Modifiers
-----------+------------------------+-----------
nom        | character varying(30)  |
adreça     | character varying(30)  |
carrera    | character varying(50)  |
grup       |           character(1) |
grau       |                integer |
```

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

Cada registre de la taula *estudiant* conté cinc valors perquè té cinc columnes:

```
demo=# insert into estudiant values (
demo(# 'Joan' ,
demo(# 'Treboles 21',
demo(# 'Enginyeria de computacio',
demo(# 'A',
demo(# 3
demo(# );
INSERT 24781 1
```

L'herència no només permet que la taula *fill* contingui les columnes de la taula *pare*, sinó que estableix una relació conceptual **és-un**.

La consulta del contingut de la taula *estudiant* mostrarà, sens dubte, un sol registre. És a dir, no s'hereten les dades, únicament els camps (atributs) de l'objecte:

```
demo=# select * from estudiant;
nom     |adreca       |carrera                  |grup    | grau
 ------- + ------------+ ------------------------+ ------ +-------
   Joan | Treboles 21 | Enginyeria en Computacio|     A  | 3
(1 row)
```

A més, la consulta de la taula *persona* mostrarà un nou registre:

```
demo=# select * from persona;
nom                      | adreca
 -------------------------+-------------
Federico García Lorca    | Granada 65
Alexandre el Gran        | Babilònia
Joan                     | Treboles 21
(3 rows)
```

L'últim registre mostrat és el que va ser inserit a 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, s'inclouen en el resultat tots els estudiants. Per a consultar només les persones que no són estudiants, podem utilitzar el modificador ONLY:

```
demo=# select * from only persona;
nom                    | adreca
 ---------------------- +------------
 Alexandre el Gran     | Babilònia
 Federico García Lorca | Granada 65
(2 rows)
demo=#
```

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

```
demo=# drop table persona;
NOTICE: table estudiant depèn de table persona
ERROR: no es pot eliminar table persona perquè altres objectes en depenen
HINT: Use DROP ... CASCADE per eliminar a més els objectes dependents.
```

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

4.3. Herència i OID

Els OID 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'única instància de la classe estudiant:

```
demo=# select oid,* from persona ;
oid     | nom                    | adreca
 ------- +------------------------+--------------
 17242  | Alexandre el Gran      | Babilònia
 17243  | Federico García Lorca  | Granada 65
 17247  | Joan                   | Treboles 21
(3 rows)
demo=# select oid,* from estudiant ;
oid     |nom       |adreca            |carrera                     |grup  | grau
 ------- +--------- + ----------------- + ---------------------------+ ---- +-------
 17247  | Joan     | Treboles 21      | Enginyeria de computació m | A    |     3
(1 row)
```

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

En PostgreSQL, una alternativa per a no utilitzar els OID és crear una columna de tipus **serial** a la taula *pare*, així serà heretada en la *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:

```
demo=# create table persona (
demo(# id serial,
demo(# nom varchar (30),
demo(# adreça varchar(30)
demo(# ) without oids;
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'
CREATE
```

La columna *id* es defineix com un enter i s'incrementarà utilitzant la funció *nextval()* tal com ens indica la informació de la columna:

```
demo=# \d persona
Table "persona"
Column     |Type                  |Modifiers
 ---------- + ---------------------+-----------------------------------------------------
id         |integer               |not null default nextval('"persona_id_seq"'::text)
nom        |character varying(30) |
adreça     |character varying(30) |
Unique keys: persona_id_key
```

En definir un tipus serial, hem creat implícitament una seqüència independent de la taula. Podem consultar les seqüències de la nostra base de dades mitjançant l'ordre '\ds':

```
demo=# \ds
                List of relations
 Schema |Name              |Type      |Owner
------- +----------------- +--------- +--------------
 public |productes_clau_seq|sequence  |postgres
(1 row)
```

Creem novament la taula *estudiant* heretant de *persona*:

```
create table estudiant (
demo(# carrera varchar(50),
demo(# grup char,
demo(# grau int
demo(# ) inherits ( persona );
CREATE
```

L'estudiant heretarà la columna *id* i s'incrementarà utilitzant la mateixa seqüència:

```
demo=# \d persona
Table "persona"
Column     |Type                   |Modifiers
 ---------- + --------------------- +-------------------------------------------------
id|         |integer                |not null default next-val('"persona_id_seq"'::text)
nom         |character varying(30)  |
adreça      |character varying(30)  |
carrera     |character varying(50)  |
grup        |character(1)           |
grau        |integer                |
```

Inserirem a la taula alguns registres d'exemple, ometent el valor per a la columna *id*:

```
demo=# insert into persona(nom,adreca)
values ( 'Federico García Lorca' , 'Granada 65' );
demo=# insert into persona(nom,adreca)
values ( 'Alexandre el Gran' , 'Babilònia' );
demo=# insert into estudiant(nom,adreca,carrera,grup,grau)
values ( 'Elizabeth' , 'Pino 35', 'Psicologia' , 'B' , 5 );
```

La taula *estudiant* contindrà un sol registre, però el seu identificador serà el número 3:

```
demo=# select * from estudiante;
id    | nom       | adreca  | carrera    | grup | grau
-----+---------- +-------- +---------- +----- +---------
3     | Elizabeth | Pino 35 | Psicologia| B    | 5
(1 row)
```

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

```
demo=# select * from persona;
 id   | nom                    | adreca
-----+----------------------- +------------
  1   | Federico García Lorca  | Granada 65
  2   | Alexandre el Gran      | Babilònia
  3   | Elizabeth              | Pino 35
(3 rows)
```

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

4.4. Restriccions

Com ja sabem, les restriccions permeten especificar condicions que hauran de complir taules o columnes per mantenir la integritat de les seves dades. Algunes de les restriccions seran imposades pel model concret que s'estigui implementant, mentre que d'altres tindran el seu origen en les regles de negoci del client, els valors que poden prendre alguns camps, etc.

Els valors que pot contenir una columna estan restringits en primer lloc pel tipus de dades. Aquesta no és l'única restricció que es pot definir per als valors en una columna, PostgreSQL ofereix les restriccions següents:

Exemple

Una columna definida com a **integer** no pot contenir cadenes de caràcters.

- **null** i **not null**. Moltes vegades 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 utilitzem **null** per a especificar l'absència de valor. En definir una taula podem indicar quines columnes podran contenir valors nuls i quines no en podran contenir.

```
create table Persona (
nom varchar(40) not null,
treball varchar(40) null,
correu varchar(20),
);
```

El *nom* d'una persona no pot ser nul i és possible que la persona no tingui *feina*. També és possible que no tingui *correu*, 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 continguts en una columna es puguin duplicar.

```
create table Persona (
nom varchar(40) not null,
conjuge varchar(40) unique,
);
```

cònjuge no pot contenir valors duplicats, no permetrem que dues persones tinguin simultàniament el mateix cònjuge.

- **primary key**. Aquesta restricció especifica la columna o columnes que triem 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 a identificar una fila de la taula de manera unívoca, per la qual cosa se'ls denomina *claus candidates*.

```
create table Persona (
nss varchar(10) primary key,
conjuge varchar(40) unique,
);
```

En definir una columna com a *primary key*, es defineix implícitament amb *unique*. L'nss (número de la seguretat social) no solament és únic, sinó que l'utilitzem per a identificar les persones.

- **references i foreign key**. En el model relacional, establim les relacions entre entitats mitjançant la inclusió de claus foranes en altres relacions. PostgreSQL i SQL ofereixen mecanismes per expressar i mantenir aquesta integritat referencial. En l'exemple següent, les *Mascotes* tenen com a amo una *Persona*:

```
create table Mascota(
nom varchar(20),
amo varchar(10) references Persona,
);
```

Una referència per defecte és a una clau primària, per la qual cosa *amo* es refereix implícitament a l'*nss* de *Persona*. Quan es capturin les dades d'una nova *mascota*, PostgreSQL verificarà que el valor d'*amo* faci referència a un nss que existeixi en *Persona*; en cas contrari, emetrà un missatge d'error. En altres paraules, no es permet assignar a una mascota un amo que no existeixi.

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

```
create table Mascota(
nom varchar(20),
amo varchar(10) references Persona(nss),
);
```

o el seu equivalent:

```
create table Mascota(
nom varchar(20),
amo varchar(10),
FOREIGN KEY amo references Persona(nss),
);
```

Podria donar-se 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,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

Si no s'especifica cap altra acció, per omissió la persona que tingui una mascota no pot ser eliminada, perquè la mascota es quedaria sense amo. Per a poder eliminar una persona, abans s'han d'eliminar les mascotes que tingui. Aquest comportament no sembla el més adequat per al cas.

Per a 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, les mascotes simplement es quedin sense amo:

```
create table Mascota (
amo varchar(10) references Persona on delete set null,
);
```

En clàusula **on delete** es poden especificar les accions següents:

– **set null**. La referència pren el valor NULL: si s'elimina *Persona* la seva *Mascota* es quedarà sense amo.

– **set default**. La referència pren el valor per omissió.

– **cascade**. L'acció s'efectua en cascada: si s'elimina *Persona* automàticament s'elimina la seva *Mascota*.

– **restrict**. No permet l'esborrament del registre: no es pot eliminar una *Persona* que tingui *Mascota*. 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.

- **check**. Aquesta restricció realitza l'avaluació prèvia d'una expressió lògica quan s'intenta fer una assignació. Si el resultat és vertader, accepta el valor per a la columna; en cas contrari, emetrà un missatge d'error i rebutjarà el valor.

```
create table Persona (
edat int check( edat > 10 and edat < 80 ),
correu varchar(20) check( correu ~ '.+@.+\..+' ),
ciutat varchar(30) check( ciutat <> '' )
);
```

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

- Edat ha de ser entre onze i setanta-nou anys.
- Ciutat no pot ser una cadena buida.
- Correu ha de tenir una arrova.

Comparació d'expressions regulars

L'operador ~ realitza comparacions de cadenes amb expressions regulars.
Les expressions regulars són patrons de recerca molt flexibles desenvolupats en el món Unix.

Qualsevol d'aquestes restriccions pot tenir nom i d'aquesta manera es facilita la referència a les restriccions específiques per a esborrar-les, modificar-les, etc., atès que es pot fer per noms. Per a donar nom a una restricció, utilitzem la sintaxi següent:

```
constraint nom_de_restriccio <restriccio>
```

4.4.1. Restriccions de taula

Quan les restriccions s'indiquen després de les definicions de les columnes, i poden afectar diverses d'aquestes simultàniament, es diu que són restriccions de taula:

```
create table Persona (
nss int,
```

```
nom varchar(30),
conjuge varchar(30),
cap int,
correu varchar(20),
primary key (nss),
unique (conjuge),
foreign key (cap) references Persona,
check (correu ~ '@' )
);
```

Aquesta notació permet que la restricció pugui incloure diverses columnes:

```
create table Curs (
materia varchar(30),
grup char(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 la restricció de columna, a les restriccions de taula se'ls pot assignar un nom:

```
create table Persona (
nss int,
nom varchar(30),
conjuge varchar(30),
cap int,
correu varchar(20),
constraint identificador primary key (nss),
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 ja definides:

```
alter table Persona drop constraint monogàmia
alter table add constraint monogàmia unique (conjuge);
```

4.5. Indexació

PostgreSQL crea índexs per a les claus primàries de totes les taules. Quan calgui crear índexs addicionals, utilitzarem l'expressió de l'exemple següent:

```
create index persona_nom_index on Persona ( nom );
```

4.6. Consulta d'informació de bases de dades i taules

Com ja sabem, el client **psql** ofereix diverses alternatives per obtenir informació sobre l'estructura de la nostra base de dades. En la taula següent es mostren algunes ordres de molta utilitat:

Ordre	Descripció
`\l`	Llista les bases de dades
`\d`	Descriu les taules de la base de dades en ús
`\ds`	Llista les seqüències
`\di`	Llista els índexs
`\dv`	Llista les vistes
`\dp \z`	Llista els privilegis sobre les taules
`\da`	Llista les funcions d'agregats
`\df`	Llista les funcions
`\g archivo`	Executa les ordres d'arxiu
`\H`	Canvia el mode de sortida HTML
`\! comando`	Executa una ordre del sistema operatiu

Per a obtenir la llista de taules de la base de dades *demo* fem el següent:

```
demo=# \d
List of relations
Name                | Type       | Owner
------------------ +---------- + ----------
guanys              |    table   | postgres
preus               |    table   | postgres
productes           |    table   | postgres
productes_clau_seq  | sequence   | postgres
proveidors          |    table   | postgres
(5 rows)
```

L'estructura de la taula *productes* se sol·licita de la manera següent:

```
demo=# \d productes
Table "productes"
Column        | Type                  | Modifiers
-------------+ --------------------- +-------------------------------------------------
part          |character varying(20) |
tipus         |character varying(20) |
especificacio |character varying(20) |
psuggerit     |real                  |
clau          |integer               |not null default nextval('"productes_clau_seq"'::text)
Primary key: productes_pkey
Triggers: RI_ConstraintTrigger_17342,
RI_ConstraintTrigger_17344
```

En l'exemple anterior podem observar que la columna *clau* conté dos modificadors:

- El primer especifica que no es poden assignar valors nuls.
- El segon especifica el valor per omissió que s'haurà d'assignar a la columna.

En aquest cas, el valor serà automàticament calculat per la funció *nextval()*, que pren com a argument la **seqüència*** *productes_clau_seq*.

*** Una seqüència és un nom especial que permet la producció de sèries numèriques.**

L'ordre següent mostra les seqüències creades en una base de dades:

```
demo=# \ds
List of relations
Name                | Type     | Owner
 ------------------ + --------- +----------
productes_clau_seq | sequence | quiron
(1 row)
```

Les seqüències es creen automàticament quan es declaren columnes de tipus serial.

En l'estructura de la taula productes també trobem una clau primària. PostgreSQL generarà sempre un índex per a cada taula utilitzant la clau primària. La llista dels índexs de la base de dades s'obté de la manera següent:

```
demo=# \di
List of relations
Name             | Type     | Owner
 --------------- +--------- +----------
productes_pkey   | index    | quiron
proveïdors_pkey  | index    | quiron
(2 rows)
```

El conjunt d'ordres proporcionades per **psql** que hem presentat permet obtenir informació sobre l'estructura de la nostra base de dades d'una manera directa i senzilla, i també és útil per a explorar bases de dades que no coneguem.

4.7. Tipus de dades

4.7.1. Tipus lògics

PostgreSQL incorpora el tipus lògic **boolean**, també anomenat **bool**. Ocupa un byte d'espai d'emmagatzemament i pot emmagatzemar els valors fals i vertader.

Valor	Nom
Fals	false, 'f', 'n', 'no', 0
Vertader	true, 't', 'y', 'yes', 1

PostgreSQL suporta els operadors lògics següents: **and**, **or** i **not**.

Encara que els operadors de comparació s'apliquen sobre pràcticament tots els tipus de dades proporcionades per PostgreSQL, atès que el seu resultat és un valor lògic, en descriurem el comportament en la taula següent:

Operador	Descripció
>	Major que
<	Menor que
<=	Menor o igual que
>=	Major o igual que
<> !=	Diferent de

4.7.2. Tipus numèrics

PostgreSQL disposa dels tipus enters **smallint**, **int** i **bigint** que es comporten com ho fan els enters en molts llenguatges de programació.

Els nombres amb punt flotant **real** i **double precisió** emmagatzemen quantitats amb decimals. Una característica dels nombres de punt flotant és que perden exactitud a mesura que creixen o decreixen els valors.

Encara que aquesta pèrdua d'exactitud no sol tenir importància en la majoria de les ocasions, PostgreSQL inclou el tipus **numeric**, que permet emmagatzemar quantitats molt grans o molt petites sense pèrdua d'informació. Sens dubte, aquest avantatge té un cost, els valors de tipus **numeric** ocupen un espai d'emmagatzemament considerablement gran i les operacions s'executen sobre ells molt lentament. Per tant, no és aconsellable utilitzar el tipus **numeric** si no es necessita una alta precisió o es prioritza la velocitat de processament.

Nom	Mida	Altres noms	Comentari
smallint	2 bytes	int2	
int	4 bytes	int4, integer	

Nom	Mida	Altres noms	Comentari
bigint	8 bytes	int8	
numeric(p,e)	11 + (p/2)		'p' és la precisió, 'e' és l'escala
real	4 bytes	float, float4	
double precision	8 bytes	float8	
serial			No és un tipus, és un enter autoincrementable

Serial

La declaració **serial** és un cas especial, ja que no es tracta d'un nou tipus. Quan s'utilitza com a nom de tipus d'una columna, aquesta prendrà automàticament valors consecutius en cada nou registre.

Exemple d'una taula que defineix la columna *foli* com a tipus **serial**.

```
create table Factura(
foli serial,
client varchar(30),
suma real
);
```

PostgreSQL respondria a aquesta instrucció amb dos missatges:

- En el primer avisa que s'ha creat una seqüència de nom *factura_foli_seq*:

```
NOTICE: CREATE TABLE will create implicit sequence 'factura_foli_seq' for SERIAL column '
```

- En el segon avisa de la creació d'un índex únic en la taula utilitzant la columna *foli*:

```
NOTICE: CREATE TABLE / UNIQUE will create implicit index 'factura_foli_key' for table 'factura'
CREATE
```

Si es declaren diverses columnes amb **serial** en una taula, es crearà una seqüència i un índex per a cada una d'elles.

4.7.3. Operadors numèrics

PostgreSQL ofereix un conjunt d'operadors numèrics predefinits, que presentem en la taula següent:

Símbol	Operador
+	Addició
-	Subtracció
*	Multiplicació
/	Divisió
%	Mòdul
^	Exponenciació
|/	Raíz cuadrada

Exemple

```
select |/ 9;
select 43 % 5;
select !! 7;
select 7!;
```

Símbol	Operador
||/	Arrel cúbica
!	Factorial
!!	Factorial com a operador fix
@	Valor absolut
&	AND binari
|	OR binari
#	XOR binari
~	Negació binària
<<	Corriment binari a l'esquerra
>>	Corriment binari a la dreta

4.7.4. Tipus de caràcters

Els valors de cadena en PostgreSQL es delimiten per cometes simples.

Recordeu

Les cometes dobles delimiten identificadors que contenen caràcters especials.

```
demo=# select 'Hola món';
?column?
------------
Hola món
(1 row)
```

Es pot incloure una cometa simple dins d'una cadena amb \' o ' ':

```
demo=# select 'ell va dir: ''Hola'' ';
?column?
-----------------
ell va dir: 'Hola'
(1 row)
```

Les cadenes poden contenir caràcters especials utilitzant les anomenades seqüències d'escapada que inicien amb el caràcter '\':

```
\n nova línia
\r retorn de carro
\t tabulador
\b retrocés
\f canvi de pàgina
\r retorn de carro
\\ el caràcter \
```

Les seqüències d'escapada se substitueixen pel caràcter corresponent:

```
demo=# select 'Això està en \n dues línies';
?column?
-----------------------------
Això està en
dues línies
(1 row)
```

PostgreSQL ofereix els tipus següents per a cadenes de caràcters:

Tipus	Altres noms	Descripció
char(n)	character(n)	Reserva *n* espais per emmagatzemar la cadena
varchar(n)	character var-ying(n)	Utilitza els espais necessaris per emmagatzemar una cadena menor o igual que *n*
text		Emmagatzema cadenes de qualsevol magnitud

4.7.5. Operadors

En la taula següent es descriuen els operadors per a cadenes de caràcters:

Operador	Descripció	Distingeix majúscules i minúscules?
||	Concatenació	-
~	Correspondència a expressió regular	Sí
~*	Correspondència a expressió regular	No
!~	No correspondència a expressió regular	Sí
!~*	No correspondència a expressió regular	-

En la taula següent es mostren algunes funcions d'ús comú sobre cadenes de caràcters:

Funció	Descripció
Length(cadena)	Torna la longitud de la cadena
lower(cadena)	Converteix la cadena a minúscules
ltrim(cadena,caracters)	Elimina de l'esquerra els caràcters especificats
substring(cadena from patró)	Extreu la subcadena que compleix el patró especificat

Bibliografia

És recomanable consultar el manual per a obtenir la referència completa de funcions.

Sobre les cadenes també podem utilitzar els operadors de comparació que ja coneixem.

Exemple

En aquest cas, el resultat de la comparació *menor que* és VERTADER:

```
demo=# select 'HOLA' < 'hola';
?column?
----------
t
(1 row)
```

4.7.6. Dates i hores

En la taula següent es mostren els tipus de dades referents al temps que ofereix PostgreSQL:

Tipus de dada	Unitats	Mida	Descripció	Precisió
date	dia-mes-any	4 bytes	Data	Dia
time	hrs:min:seg:micro	4 bytes	Hora	Microsegon
timestamp	dia-mes-any hrs:min:seg:micro	8 bytes	Data més hora	Microsegon
interval	second, minute, hour, day, week, month, year, decade, century, millennium*	12 bytes	Interval de temps	Microsegon

*** També admet abreviatures.**

Hi ha un tipus de dada **timez** que inclou les dades del tipus **time** i, a més, la zona horària. La seva sintaxi és la següent:

```
hh:mm[:ss[.mmm]][am|pm][zzz]
```

El tipus de dades **date** emmagatzema el dia, mes i any d'una data donada i es mostra per omissió amb el format següent: YYYY-MM-DD:

```
demo=# create table Persona ( naixement date );
CREATE
demo=# insert into persona values ( '2004-05-22' );
INSERT 17397 1
demo=# select * from persona;
naixement
------------
2004-05-22
(1 row)
```

Per a canviar el format de presentació, hem de modificar la variable d'entorn *datestyle*:

```
demo=# set datestyle = 'german';
SET VARIABLE
demo=# select * from persona;
naixement
------------
22.05.2004
(1 row)
```

Nom del format	Format	Exemple
ISO	Any-mes-dia	2004-05-22
GERMAN	Dia.mes.any	22.05.2004
POSTGRES	dia-mes-any	22-05-2004
SQL	mes/dia/any	05/22/2004

4.7.7. *Arrays*

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

Definició

Els *arrays* no compleixen la primera forma normal de Cood, per la qual cosa molts els consideren inacceptables en el model relacional.

```
demo=# create table Estudiant (
demo(# nom varchar(30),
demo(# parcials int [3]
demo(# );
CREATE
```

La columna *parcials* accepta tres qualificacions dels estudiants.

També és possible assignar un sol valor de l'*array*:

```
demo=# insert into Estudiant( nom, parcials[2]) values ( 'Pere' , '{90}');
INSERT 17418 1
demo=# select * from Estudiant ;
nom           | parcials
 -------------+----------------
John Lennon   |
Joan          | {90,95,97}
Pere          | {90}
(3 rows)
```

Els *arrays*, igual que qualsevol columna quan no s'especifica el contrari, accepten valors nuls:

```
demo=# insert into Estudiant values ( 'John Lennon ' );
INSERT 17416 1
demo=# insert into Estudiant values ( 'Joan' , '{90,95,97}' );
INSERT 17417 1
```

Els valors de l'array s'escriuen sempre entre claus.

```
demo=# select * from Estudiant;
nom           | parcials
 -------------+----------------
John Lennon   |
Joan          | {90,95,97}
(2 rows)
```

Per a seleccionar un valor d'un *array* en una consulta s'especifica entre claudàtors la cel·la que es desplegarà:

Només en Joan té qualificació en el tercer parcial.

```
demo=# select nom, parcials[3] from Estudiant;
nom             | parcials
 --------------- +-------------
John Lennon     |
Joan            | 97
Pere            |
(3 rows)
```

En molts llenguatges de programació, els *array* s'implementen amb longitud fixa, PostgreSQL permet augmentar la seva mida dinàmicament:

La columna *parcials* del registre Pau inclou quatre cel·les i només l'última té valor.

```
demo=# insert into Estudiant( nom, parcials[4]) values ( 'Pau' , '{70}');
INSERT 17419 1
demo=# select * from Estudiant;
nom          | parcials
 ------------+------------
John Lennon  |
Joan         | {90,95,97}
Pere         | {90}
Pau          | {70}
(4 rows)
```

Mitjançant la funció array_dims() podem conèixer les dimensions d'un *array*:

```
demo=# select nom, array_dims(parcials) from Estudiant;
nom          | array_dims
 ------------+------------
John Lennon  |
Joan         | [1:3]
Pere         | [1:1]
Pau          | [1:1]
(4 rows)
```

4.7.8. BLOB

El tipus de dades BLOB (Binary Large OBject) permet emmagatzemar en una columna un objecte de gran mida. PostgreSQL no coneix res sobre el tipus d'informació que s'emmagatzema en una columna BLOB, simplement ho considera com una seqüència de bytes. Per aquest motiu, no es tenen operacions sobre els tipus BLOB, amb l'excepció de l'operador de concatenació, que simplement uneix el contingut de dos BLOB en un.

Vegem com emmagatzemar una fotografia en una taula de persones mitjançant tipus BLOB.

Una primera manera de fer-ho és important el contingut de l'arxiu que conté la imatge mitjançant la funció **lo_import**():

```
demo=# select lo_import('/home/quiron/lameva-foto.jpg');
lo_import
-----------
17425
(1 row)
```

Aquesta funció torna com a resultat l'OID de l'objecte inserit. On s'ha emmagatzemat la fotografia si no hem utilitzat l'ordre *insert*? PostgreSQL manté una taula de nom **pg_largeobject** amb l'objectiu d'emmagatzemar BLOB. Podem utilitzar l'OID per a fer referenciar l'objecte en una taula:

```
demo=# create table persona (
demo(# nom varchar(30),
demo(# adreça varchar(30),
demo(# fotografia oid
demo(# );
CREATE
```

Els registres inserits en aquesta taula porten un nombre enter com a OID que, en el cas següent, s'ha obtingut sol·licitant-lo a la funció **lo_import()** anterior:

```
demo=# insert into persona values ( 'Juli' , 'Cedre 54', 17425);
```

La inserció anterior es podia haver fet en un sol pas:

```
demo=# insert into persona values
( 'Juli' , 'Cedre 54', lo_import('/home/quiron/lameva-foto.jpg'));
```

Per a extreure el contingut de la columna, s'utilitza la funció **lo_export()**

```
demo=# select lo_export(17425,'/tmp/lameva-foto.jpg');
lo_export
-----------
1
(1 row)
```

La funció **lo_unlink()** permet esborrar un BLOB emmagatzemat en *pg_largeobject:*

```
select lo_unlink(17425);
```

Vegem el format que utilitza PostgreSQL per a visualitzar BLOB.

S'ha retallat la sortida per a fer més comprensible la taula.

```
loid  | pageno | data
------+--------+-------------------------------------------------------------------
17425 | 0      | \377\330\377\340\000\020JFIF\000\001\001\001\000H\000H\000\000\37
17425 | 1      | \256-}\306\267\032s[\336)\245\231\370|L\206\275\364\224\321\237\2
17425 | 2      | \341\226;\0151\232\033f\\\371\251\0323\003t\307\207~\035GB\271\17
(3 rows)
```

La fotografia s'ha dividit en tres registres, que són les pàgines 0, 1 i 2 del BLOB. Els tres registres tenen el mateix *loid*, la qual cosa significa que són el mateix objecte. Observeu, també que els bytes s'han desplegat com a caràcters de la forma '*ddd*', on ddd són tres dígits octals.

Per a emmagatzemar un BLOB en una taula de manera directa; és a dir, sense utilitzar la taula del sistema *pg_largeobject* utilitzem el tipus de dada **bytea**:

En el primer cas s'està inserint un BLOB d'un sol byte, el caràcter ASCII zero.

En el segon cas s'estan inserint quatre bytes, els tres primers estan representats directament pels caràcters imprimibles ASCII i el tercer, el caràcter ASCII 30 octal, com que no és imprimible, s'escriu en notació '*ddd*'.

```
demo=# create table persona (
demo(# nom varchar(30),
demo(# adreça varchar(30),
demo(# fotografia bytea
demo(# );
CREATE
```

La columna fotografia és de tipus **bytea**, cosa que permet emmagatzemar objectes de gran mida, prop d'1 GB. Per a inserir valors en aquestes columnes, fem el següent:

```
demo=# insert into persona values ( 'Jordi' , 'Cirerer 55', '\\000');
INSERT 17436 1
demo=# insert into persona values ( 'Lluís' , 'Alsina 67', 'abc\\030');
INSERT 17437 1
```

La consulta es visualitza com segueix:

```
demo=# select * from persona;
nom     | adreca     | fotografia
 -------+ -----------+--------------
Jordi   | Cirerer 55 | \000
Lluís   | Alsina 67  | abc\030
(2 rows)
```

Els caràcters en notació octal es mostren amb una barra invertida i amb dues, tal com es van escriure. Això és perquè, en realitat, només porten una barra invertida, però per qüestions de disseny PostgreSQL, les literals BLOB s'han d'escriure amb doble barra invertida.

4.8. Modificació de l'estructura d'una taula

Per a modificar l'estructura d'una taula un cop construïda, disposem de la sentència SQL **alter table**.

Mitjançant aquesta sentència, podem dur a terme les operacions següents:

- Agregar una columna.

```
demo=# alter table persona add edat int ;
ALTER
```

- Eliminar una columna.

```
demo=# ALTER TABLE products DROP COLUMN description;
```

- Fixar el valor per omissió d'una columna.

```
demo=# alter table persona alter edat set default 15;
ALTER
```

- Eliminar el valor per omissió d'una columna.

```
demo=# alter table persona alter edat drop default;
ALTER
```

- Reanomenar una columna.

```
demo=# alter table persona rename adreca to dir;
ALTER
```

- Reanomenar una taula.

```
demo=# alter table persona rename to personal;
ALTER
```

5. Manipulació de dades

5.1. Consultes

Les consultes a la base de dades es realitzen amb l'ordre **select**, que s'implementa en PostgreSQL complint en gran part amb l'estàndard SQL:

Notació

Ometrem les referències comunes a SQL i només es mostraran algunes de les possibilitats de consulta amb PostgreSQL. Pel que fa a les funcions auxiliars, se n'han vist algunes en l'apartat de tipus de dades i, en tot cas, es recomana la consulta de la documentació del producte per a les operacions més avançades.

```
demo=# select part, tipus
demo-# from productes
demo-# where psuggerit > 30
demo-# order by part
demo-# limit 5
demo-# offset 3;
part        | tipus
 ----------- +----------
Monitor     | 1024x876
Monitor     | 1024x876
Processador | 2.4 GHz
Processador | 1.7 GHz
Processador | 3 GHz
(5 rows)
```

Igual que MySQL, PostgreSQL admet la sentència **explain** davant de **select** per examinar què està ocorrent durant una consulta:

Igual que en el mòdul de MySQL, veiem que PostgreSQL no aprofita els índexs (bàsicament perquè no en tenim cap de definit).

```
demo=# explain select productes.clau, part||' '||tipus||' '||especificacio as producte,
proveidors.empresa , preu from productes natural join preus natural join proveidors;
                                        QUERY PLAN
-------------------------------------------------------------------------------
Hash Join  (cost=45.00..120.11 rows=1000 width=104)
   Hash Cond: (("outer".empresa)::text = ("inner".empresa)::text)
   -> Hash Join  (cost=22.50..72.61 rows=1000 width=104)
      Hash Cond: ("outer".clau = "inner".clau)
      -> Seq Scan on preus (cost=0.00..20.00 rows=1000 width=32)
      -> Hash (cost=20.00..20.00 rows=1000 width=76)
         -> Seq Scan on productes  (cost=0.00..20.00 rows=1000 width=76)
   -> Hash (cost=20.00..20.00 rows=1000 width=24)
      -> Seq Scan on proveïdors  (cost=0.00..20.00 rows=1000 width=24)
(9 rows)
demo=#
```

Vegem com millorar el rendiment d'aquest **select**:

```
demo=# create index empresa_idx on preus (empresa);
CREATE INDEX
demo=# create index clau_idx on preus (clau);
CREATE INDEX
demo=# explain select productes.clau, part||' '||tipus||' '||especificacio as producte,
proveidors.empresa , preu from productes natural join preus natural join proveidors;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=29.00..56.90 rows=20 width=104)
   Hash Cond: ("outer".clau = "inner".clau)
   ->  Seq Scan on productes  (cost=0.00..20.00 rows=1000 width=76)
   ->  Hash  (cost=28.95..28.95 rows=20 width=32)
      ->  Hash Join  (cost=1.25..28.95 rows=20 width=32)
         Hash Cond: (("outer".empresa)::text = ("inner".empresa)::text)
         ->  Seq Scan on proveidors  (cost=0.00..20.00 rows=1000 width=24)
         ->  Hash  (cost=1.20..1.20 rows=20 width=32)
            ->  Seq Scan on preus (cost=0.00..1.20 rows=20 width=32)
(9 rows)
demo=#
```

5.2. Actualitzacions e insercions

PostgreSQL compleix l'estàndard SQL en tots els sentits per a les sentències d'actualització, inserció i esborrament. No defineix modificadors ni altres mecanismes per a aquestes sentències.

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**:

```
demo=# update only persona set nom='Sr. '||nom;
UPDATE 2
demo=# select * from persona;
nom                        | adreca
-------------------------- +---------------
 Sr. Alexandre el Gran     | Babilònia
 Sr. Federico García Lorca| Granada 65
 Joan                      | Treboles 21
(3 rows)
demo=#
```

5.3. Transaccions

Definim *transacció* com un conjunt d'operacions que tenen significat només en actuar juntes.

PostgreSQL ofereix suport a transaccions, garantint que ambdues operacions es realitzin o que no se'n realitzi cap. Per a iniciar una transacció, s'utilitza l'ordre **begin** i per a acabar-la, **commit**.

Exemple

Una compra pot ser una transacció que consti de dues operacions:

- Inserir un registre del pagament del producte.
- Inserir el producte en l'inventari.

No s'ha d'inserir un producte que no s'hagi pagat, ni pagar un producte que no sigui a l'inventari, per tant, les dues operacions formen una transacció.

```
demo=# begin;
BEGIN
demo=# insert into productes values ('RAM','512MB','333 MHz',60);
INSERT 17459 1
demo=# select * from productes;
part        | tipus    | especificacio  | psuggerit  | clau
 -----------+---------+ --------------- + -----------+--------
Processador | 2 Ghz    | 32 bits        |            | 1
Processador | 2.4 GHz  | 32 bits        | 35         | 2
Processador | 1.7 GHz  | 64 bits        | 205        | 3
Processador | 3 GHz    | 64 bits        | 560        | 4
RAM         | 128MB    | 333 MHz        | 10         | 5
RAM         | 256MB    | 400 Mhz        | 35         | 6
Disc Dur    | 80 GB    | 7200 rpm       | 60         | 7
Disc Dur    | 120 GB   | 7200 rpm       | 78         | 8
Disc Dur    | 200 GB   | 7200 rpm       | 110        | 9
Disc Dur    | 40 GB    | 4200 rpm       |            | 10
Monitor     | 1024x876 | 75 Hz          | 80         | 11
Monitor     | 1024x876 | 60 Hz          | 67         | 12
RAM         | 512MB    | 333 MHz        | 60         | 13
(13 rows)
demo=# insert into preus values ('Patito',13,67);
INSERT 17460 1
```

El nou registre té com a clau el 13 i, fins que no finalitzi la transacció, només el pot veure l'usuari que l'ha inserit.

Inserim un registre amb el preu del proveïdor Patito per al producte amb clau 13.

```
demo=# commit;
```

Les transaccions de PostgreSQL

No hi ha cap característica en les transaccions de PostgreSQL que el diferencii del que especifica l'estàndard. Ja hem vist en apartats anteriors que les files implicades en una transacció mantenen unes columnes ocultes amb informació sobre la mateixa transacció.

En tancar la transacció, els registres inserits ja són visibles per a tots els usuaris. Si per alguna raó, per exemple una caiguda del sistema, no s'executa el **commit**, la transacció es cancel·la. La manera explícita de cancel·lar una transacció és amb l'ordre **rollback**.

6. Funcions i disparadors

Com alguns dels gestors de bases de dades relacionals, comercials líders en el mercat, PostgreSQL pot incorporar múltiples llenguatges de programació a una base de dades en particular. Aquest fet permet, per exemple:

- Emmagatzemar procediments en la base de dades (***stored procedure***), que es podran llançar quan convingui.
- Definir operadors propis.

PL/pgSQL

PL/pgSQL (*procedual language/ postgreSQL*) és una extensió de l'SQL que permet la creació de procediments i funcions a l'estil dels llenguatges tradicionals de programació.

PostgreSQL ofereix per defecte suport per al seu propi llenguatge procedimental, el PL/pgSQL. Per a instal·lar-lo, n'hi ha prou d'invocar l'ordre **createlang** des del sistema operatiu, no des de la línia de psql.

```
$ createlang plpgsql demo
```

Mitjançant aquesta ordre, s'ha instal·lat el llenguatge PL/pgSQL en la base de dades *demo*.

PostgreSQL també suporta altres llenguatges directament, com PL/Tcl, PL/Perl i PL/Python.

6.1. Primer programa

Vegem el programa *HolaMón* en PL/pgSQL:

```
demo=# create function HolaMon() returns char
demo-# as ' begin return "Hola Món PostgreSQL" ; end; '
demo-# language 'plpgsql';
CREATE
```

La funció té tres parts:

- L'encapçalament que defineix el nom de la funció i el tipus de retorn.
- El cos de la funció, que és una cadena de text (per tant, sempre va entre cometes dobles).
- L'especificació del llenguatge utilitzat.

La funció acabada de crear té les mateixes característiques que les integrades.

Es pot sol·licitar mitjançant l'ordre **select**:

```
demo=# select HolaMon();
holamon
-----------------------
Hola Mon PostgreSQL
(1 row)
```

Es pot eliminar mitjançant l'ordre **drop function**.

```
demo=# drop function HolaMon();
DROP
```

6.2. Variables

Les funcions poden rebre paràmetres, només cal especificar els tipus de dades. PostgreSQL assigna els noms als paràmetres utilitzant la seqüència $1, $2, $3...

En aquest exemple veurem totes les possibles maneres de declarar variables en una funció.

La sentència `alias` crea un nou nom per a una variable. La sentència `rename` canvia el nom d'una variable.

```
create function mi_funcion(int,char) returns int
as '
declare -- declaració de variables locals
x int; -- x es de tipus enter
y int := 10; -- y té valor inicial de 10
z int not null; -- z no pot prendre valors nuls
a constant int := 20; -- a és constant
b alias for $1; -- El primer paràmetre té dos noms.
rename $1 to c; -- Canvia de nom el segon paràmetre
begin
x := y + 30;
end;
' language 'plpgsql';
```

6.3. Sentències

L'estructura bàsica d'una funció és el **bloc**, que inclou dues parts, la declaració de variables i la secció de sentències:

declare
secció de variables
begin
secció de sentències
end;

Sentència	Descripció
declare begin end	Bloc
:=	Assignació
select into	Assignació des d'un **select**
Sentències `sql`	Qualsevol sentència `sql`

Sentència	Descripció
perform	Fa una crida a ordre sql
execute	Interpreta una cadena com a ordre sql
exit	Acaba l'execució d'un bloc
return	Acaba l'execució d'una funció
if	Executa sentències condicionalment
loop	Repeteix l'execució d'un conjunt de sentències
while	Repeteix un conjunt de sentències **mentre**
for	Repeteix un conjunt de sentències utilitzant una variable de control
raise	Desplega un missatge d'error a advertència

La sentència d'**assignació** utilitza l'operador '**:=**' per a emmagatzemar els resultats d'expressions en variables. PostgreSQL proporciona una altra sentència per fer assignacions, select. Aquesta sentència ha d'obtenir com a resultat un sol valor perquè pugui ser emmagatzemat en la variable:

```
select into x psuggerit from productes where clau = 3;
```

L'execució d'ordres sql com **create**, **drop**, **insert** o **update** es pot fer sense cap sintaxi especial. L'excepció és l'ordre **select**, que requereix executar-se amb l'ordre **perform** a fi que el resultat de la consulta sigui descartat:

```
perform select psuggerit from productes;
```

La sentència **execute** també executa una ordre sql però a partir d'una cadena de text. Aquesta sentència comporta el problema que la seva sintaxi no es verifica fins a l'execució. Es pot utilitzar, per exemple, per a processar paràmetres com a ordres sql:

```
execute $1
```

L'ordre **exit** acaba l'execució d'un bloc. S'utilitza principalment per a trencar cicles.

La bifurcació, o execució condicional, es realitza mitjançant la sentència **if**:

```
if ( $1 > 0 ) then
resultat := 'Positiu';
else
resultat := 'Negatiu';
end if;
```

També es pot utilitzar **if** amb més de dues branques:

```
if ( $1 > 0 ) then
resultat := 'Positiu';
elsif ( $1 < 0 ) then
resultat := 'Negatiu';
else
resultat := 'Zero';
end if;
```

Amb referència als bucles, PL/pgSQL ofereix tres opcions:

- El bucle **loop** és infinit, per la qual cosa té una estructura molt simple. En general s'utilitza amb alguna sentència **if** per a acabar-lo:

```
cont := 0;
loop
if ( cont = 10 ) then
exit;
end if;
-- alguna acció
cont := cont + 1;
end loop;
```

- El bucle **while** inclou la condició a l'inici, per la qual cosa el control del seu acabament és més clar:

```
cont := 0;
while cont != 10 loop
-- alguna acció
cont := cont + 1;
end loop;
```

- El bucle **for** permet realitzar un nombre d'iteracions controlades per la variable del cicle:

```
for cont in 1 .. 10 loop
-- alguna acció
end loop;
```

La sentència **raise** permet enviar missatges de tres nivells de severitat:

- **debug**. El missatge s'escriu a la bitàcola del sistema (logs).
- **notice**. El missatge s'escriu a la bitàcola i en el client psql.
- **exception**. El missatge s'escriu a la bitàcola i avorta la transacció.

El missatge pot incloure valors de variables mitjançant el caràcter ‘ % ‘:

- **raise debug** ‘funcion(): executada amb èxit;
- **raise notice** ‘El valor % es va prendre per omissió’, variable;
- **raise excepció** ‘El valor % està fora del rang permès’, variable;

6.4. Disparadors

Les funcions s’han de cridar explícitament per a la seva execució o per a incloure-les en consultes. Tanmateix, es pot definir que algunes funcions s’executin automàticament quan cert esdeveniment tingui lloc en certa taula. Aquestes funcions es coneixen com a disparadors o *triggers* i s’executen mitjançant les ordres **insert**, **delete** i **update**.

Agreguem la taula *historial* que emmagatzema els productes descontinuats quan s’eliminen de la taula *productes*.

```
create table historial(
data date,
part varchar(20),
tipus varchar(20),
especificacio varchar(20),
preu float(6)
);
```

Per a poder utilitzar una funció com a disparador, no ha de rebre arguments i ha de retornar el tipus especial **trigger**:

```
create function suportar_esborraments() returns trigger as '
begin
insert into historial values (
now(),
old.part,
old.tipus,
old.especificacio,
old.psuggerit
);
return null;
end;
```

La variable old està predefinida per PostgreSQL i es refereix al registre amb els seus antics valors. Per a referir-se als nous valors, es disposa de la variable new.

La funció està preparada per a ser utilitzada com a disparador, només cal definir-lo i associar-lo a la taula i a l’esdeveniment desitjat:

```
create trigger arxivar
before delete
on productes
for each row execute procedure suportar_esborraments();
```

Acabem de crear un disparador de nom *arxivar* que s'activarà quan s'executi l'ordre *delete* en la taula *productes*. L'usuari no necessita saber que s'ha de fer una còpia de seguretat dels registres esborrats, es fa automàticament.

En crear el disparador, hem especificat "before delete" en indicar l'operació. PostgreSQL ens permet llançar el disparador abans o després (*before, after*) que s'efectuïn les operacions. Aquest matís és important, ja que, si aquest mateix disparador l'executem després de l'operació, no veurem cap fila a la taula. És possible definir el mateix disparador per a diverses operacions:

```
create trigger arxivar
before delete or update
on productes
for each row execute procedure suportar_esborraments();
```

7. Administració de PostgreSQL

En les tasques administratives com la instal·lació, la gestió d'usuaris, les còpies de seguretat, les restauracions i l'ús de prestacions internes avançades, és on realment s'aprecien les diferències entre gestors de bases de dades. PostgreSQL té fama de ser més complex d'administrar que els seus competidors de codi obert, cosa que es deu, sobretot, al fet que ofereix més prestacions (o més complexes).

El contingut dels següents apartats tracta de les opcions d'ús comú per a l'administració d'un servidor PostgreSQL. Hi ha tantes alternatives que no és possible incloure-les totes en aquest mòdul, per la qual cosa només es presentaran alguns temes d'importància per a l'administrador, des d'una perspectiva general, que permeti obtenir una visió global de les possibilitats pràctiques de les eines administratives.

Bibliografia

El manual de PostgreSQL és la referència principal que s'ha de tenir sempre a mà per a trobar possibilitats i resoldre dubtes. En especial es recomana llegir els capítols següents:

Capítol III. Server Administration.

Capítol V. Server Programming.

Capítol VII. Internals.

De la mateixa manera, també són molt útils les llistes de correu que es descriuen en el lloc oficial www.postgresql.org.

7.1. Instal·lació

PostgreSQL està disponible per a la majoria de distribucions de GNU/Linux. La seva instal·lació és tan senzilla com executar l'instal·lador de paquets corresponent.

A Debian, el procediment següent instal·la el servidor i el client respectivament:

```
# apt-get install postgresql
# apt-get install postgresql-client
```

En distribucions basades en RPM, els noms dels paquets són una mica diferents:

```
# rpm -Uvh postgresql-server
# rpm -Uvh postgresql
```

Una vegada instal·lat, s'escriurà un *script* d'inici que permet llançar i apagar el servei PostgreSQL; d'aquesta manera, per a iniciar el servei, haurem d'executar l'ordre següent:

```
# /etc/init.d/postgresql start
```

Notació

A més de l'**start** també podrem utilitzar els paràmetres **restart**, **stop**, **reload** que permeten reiniciar, detenir i recarregar el servidor (rellegint-ne la configuració), respectivament.

Si es vol fer una instal·lació a partir del codi font, es pot obtenir del lloc oficial www.postgresql.org. A continuació, es descriu el procés d'instal·lació de manera molt simplificada. A la pràctica es podran trobar algunes diferències; el més recomanable és llegir curosament la documentació inclosa en els arxius INSTALL i README. Qualsevol dubte no resolt per la documentació, es pot consultar en la llista de distribució:

```
# tar xzvf postgresql-7.4.6.tar.gz
# cd postgresql-7.4.6
# ./configure
# make
# make install
```

Amb aquest procés s'instal·la la versió 7.4.6. L'arxiu es descomprimeix utilitzant *tar*. Dins del directori acabat de crear s'executa *configure*, que realitza una comprovació de les dependències de l'aplicació. Abans d'executar *configure,* hem d'instal·lar tots els paquets que necessitarem.

La compilació es realitza amb *make* i, finalment, els binaris produïts es copien en el sistema en els llocs convenients amb *make install.*

Després d'instal·lats els binaris, s'ha de crear l'usuari *postgres* (responsable d'executar el procés *postmaster*) i el directori on s'emmagatzemaran els arxius de les bases de dades.

```
# adduser postgres
# cd /usr/local/pgsql
# mkdir data
# chown postgres data
```

Una vegada creat l'usuari *postgres*, aquest ha d'inicialitzar la base de dades:

```
# su - postgres
# /usr/local/pgsql/initbd -D /usr/local/pgsql/data
```

initdb

L'executable *initdb* realitza el procediment necessari per a inicialitzar la base de dades de postgres, en aquest cas, en el directori /usr/local/pgsql/data.

El *postmaster* ja està llest per a executar-se manualment:

```
# /usr/local/pgsql/postmaster -D /usr/local/pgsql/data
```

Bibliografia

El procés de compilació té múltiples opcions que s'expliquen en la documentació inclosa amb les fonts.

7.1.1. Internacionalització

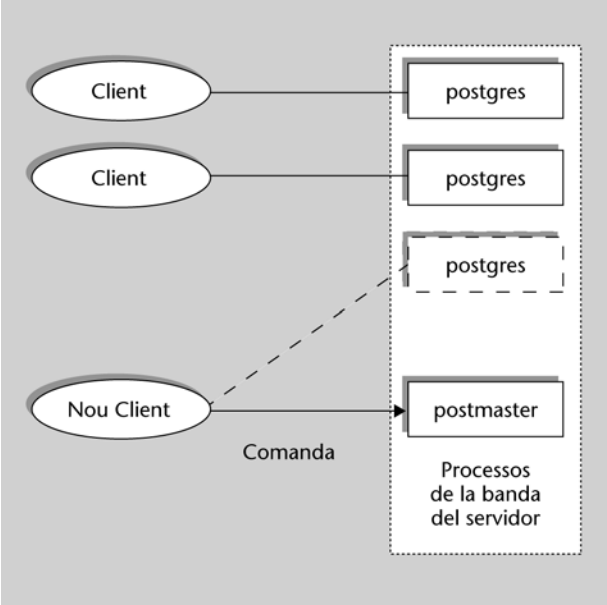
Per omissió, PostgreSQL no està compilat per a suportar missatges en espanyol, per la qual cosa és necessari compilar-lo a partir de les fonts incloent les

següents opcions de configuració, perquè tant el servidor com el client **psql** adoptin la configuració establerta pel programa **setlocales** i les variables d'entorn respectives:

```
# configure --enable-nls -enable-locale
```

7.2. Arquitectura de PostgreSQL

El següent gràfic mostra de manera esquemàtica les entitats involucrades en el funcionament normal del gestor de bases de dades:



PostgreSQL està basat en una arquitectura client-servidor. El programa servidor s'anomena **postgres** i entre els molts programes client tenim, per exemple, **pgaccess** (un client gràfic) i **psql** (un client en mode text).

Un procés servidor *postgres* pot atendre exclusivament un sol client; és a dir, fan falta tants processos servidor *postgres* com clients hi hagi. El procés **postmaster** és l'encarregat d'executar un nou servidor per a cada client que sol·liciti una connexió.

L'equip amfitrió (*host*) que emmagatzema un conjunt de bases de dades PostgreSQL s'anomena **lloc**. En un *lloc* s'executa només un procés *postmaster* i múltiples processos *postgres*. Els clients es poden executar en el mateix lloc o en equips remots connectats per TCP/IP.

És possible restringir l'accés a usuaris o a adreces IP modificant les opcions de l'arxiu pg_hba.conf, que es troba en /etc/postgresql/pg_hba.conf.

Aquest arxiu, juntament amb /etc/postgresql/postgresql.conf són particularment importants, perquè alguns dels seus paràmetres de configuració per de-

fecte provoquen multitud de problemes en connectar inicialment i perquè s'hi especifiquen els mecanismes d'autenticació que usarà PostgreSQL per a verificar les credencials dels usuaris.

Per a habilitar la connexió a PostgreSQL des de clients remots, hem de verificar el paràmetre tcpip_socket = true en el fitxer */etc/postgresql/postgresql.conf*.

A continuació, per a examinar els mètodes d'autenticació i les possibilitats de connexió de clients externs, hem de mirar el fitxer */etc/postgresql/pg_hba.conf*, on s'explicita l'acció que cal emprendre per a cada connexió provinent de cada *host* extern, o grup de *hosts*.

7.3. L'administrador de postgres

En acabar la instal·lació, en el sistema operatiu s'haurà creat l'usuari **postgres**, i en PostgreSQL s'haurà creat un usuari amb el mateix nom.

Ell és l'únic usuari existent en la base de dades i serà l'únic que podrà crear noves bases de dades i nous usuaris.

Normalment, a l'usuari *postgres* del sistema operatiu no se li permetrà l'accés des d'un *shell* ni tindrà contrasenya assignada, per la qual cosa ens haurem de convertir en l'usuari *root*, per a després convertir-nos en l'usuari *postgres* i realitzar tasques en el seu nom:

```
yo@localhost:~$ su
Password:
# su - postgres
postgres@localhost:~$
```

L'usuari **postgres** pot crear noves bases de dades utilitzant l'ordre **createdb**. En aquest cas, li indiquem que l'usuari propietari de la base de dades serà l'usuari postgres:

```
postgres@localhost:~$ createdb demo --owner=postgres
create database
```

L'usuari **postgres** pot crear nous usuaris utilitzant l'ordre **createuser**:

```
postgres@localhost:~$ createuser jo
Shall the new user be allowed to create databases? (y/n) y
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

S'ha creat l'usuari jo amb permisos per a crear bases de dades i sense permisos per a crear usuaris.

Les ordres següents eliminen bases de dades i usuaris, respectivament:

```
postgres@localhost:~$ dropdb demo
postgres@localhost:~$ dropuser jo
```

És recomanable que s'agreguin els usuaris necessaris per a operar la instal·lació de PostgreSQL, i recorri, així, el menys possible a l'ingrés amb *postgres*.

També disposem de sentències SQL per a la creació d'usuaris, grups i privilegis:

```
demo=# create user marc password 'marc21';
CREATE USER
demo=# alter user marc password 'marc22';
ALTER USER
demo=# drop user marc;
DROP USER
```

Els grups permeten assignar privilegis a diversos usuaris i la seva gestió és senzilla:

```
create group elmeugrup;
```

Per a afegir o treure usuaris d'un grup, hem d'usar:

```
alter group elmeugrup add user marc, ... ;
alter group elmeugrup drop user marc, ... ;
```

7.3.1. Privilegis

Quan es crea un objecte en PostgreSQL, se li assigna un amo. Per defecte, serà el mateix usuari que l'ha creat. Per a canviar l'amo d'una taula, índex, seqüència, etc., hem d'usar l'ordre *alter table*. L'amo de l'objecte és l'únic que hi pot fer canvis; si volem canviar aquest comportament, haurem d'assignar privilegis a altres usuaris.

Els privilegis s'assignen i eliminen mitjançant les sentències `grant` i `revoke`. PostgreSQL defineix els següents tipus d'operacions sobre les quals podem donar privilegis:

`select`, `insert`, `update`, `delete`, `rule`, `references`, `trigger`, `create`, `temporary`, `execute`, `usage` i `all privileges`.

Presentem algunes sentències de treball amb privilegis, que segueixen al peu de la lletra l'estàndard SQL:

```
grant all privileges on proveïdors to marc;
grant select on preus to manel;
grant update on preus to group elmeugrup;
revoke all privileges on preus to manel;
grant select on guanys from public;
```

7.4. Creació de tipus de dades

Entre les múltiples opcions per a estendre PostgreSQL, ens queda encara per veure la creació de tipus o dominis (segons la nomenclatura de l'estàndard SQL). PostgreSQL preveu dos tipus de dades definides per l'administrador:

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

A tall d'exemple, vegem un tipus compost i la funció que el torna:

```
create type comptipus as (f1 int, f2 text);
create function gettipus() returns setof comptipus as
  'select id, nom from clients' language sql;
```

Per al tipus de dades simple, la definició és més complexa, ja que s'ha d'indicar a PostgreSQL funcions que tracten amb aquest tipus li permetran usar-lo en operacions, assignacions, etc.

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

Tractar amb el tipus de dades simple

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

```
typedef struct Complex {
   double     x;
   double     y;
} 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
);
```

El procés és una mica confús, però compensa per la gran flexibilitat que aporta a l'SGBD. A continuació, podríem crear alguns operadors per treballar amb aquest tipus (suma, resta, etc.), mitjançant els passos que ja coneixem.

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. La sintaxi és la següent:

```
create domain country_code char(2) not null;
create domain complex_positiu complex not null check
       (complex_major(value,(0,0)))
```

S'ha creat un domini basat en un tipus definit pel sistema en què l'única restricció és la seva longitud.

Evidentment, hauríem d'haver definit l'operador *complex_major* que rebés dos nombres complexos i indiqués si el primer és major que el segon.

7.5. Plantilla de creació de bases de dades

PostgreSQL té definides dues bases de dades de sistema, *template0* i *template1* (que devem haver vist en els exemples en llistar les bases de dades del gestor), amb un conjunt d'objectes com els tipus de dades que suporta o els llenguatges de procediment instal·lats.

La base de dades *template0* es crea en instal·lar el servidor, amb els objectes per defecte d'aquest, i la base de dades *template1* es crea a continuació de l'anterior amb alguns objectes més particulars de l'entorn o sistema operatiu on s'ha instal·lat PostgreSQL. Això fa que sigui molt recomanable heretar sempre de *template1* (que és el comportament per defecte).

Aquestes bases de dades s'utilitzen com a "pares" de la resta de bases de dades que es creen, de manera que, en crear una nova base de dades, es copien tots els objectes de *template1*.

Avantatges d'aquesta situació

Si volem afegir algun objecte (una taula, un tipus de dades, etc.) a totes les bases de dades, només l'hem d'afegir a *template1*, i la resta de bases de dades que s'hagin creat a partir d'aquesta el tindran disponible.

Si s'instal·la un llenguatge sobre la base de dades *template1,* automàticament totes les bases de dades també l'usaran. En les distribucions de Linux és freqüent que s'hagi realitzat d'aquesta manera, per la qual cosa no hi ha necessitat d'instal·lar-lo.

Sens dubte, podem escollir una altra plantilla per a crear bases de dades, especificant-la en la sentència:

```
create database novabd template plantillabd
```

7.6. Copies de seguretat

Fer periòdicament còpies de seguretat de la base de dades és una de les tasques principals de l'administrador de qualsevol base de dades. En PostgreSQL, aquestes còpies de seguretat es poden fer de dues maneres diferents:

- Bolcant a fitxer les sentències SQL necessàries per a recrear les bases de dades.
- Fent còpia a nivell de fitxer de la base de dades.

En el primer cas, disposem de la utilitat pg_dump, que realitza un bolcatge de la base de dades sol·licitada de la manera següent:

```
$ pg_dump demo > fitxer_sortida.sql
```

pg_dump és un programa client de la base de dades (com **psql**), cosa que significa que el podem utilitzar per a fer còpies de bases de dades remotes, sempre que tinguem privilegis per a accedir a totes les seves taules. A la pràctica, això significa que hem de ser l'usuari administrador de la base de dades per a fer-ho.

Si la nostra base de dades usa els OID per a referències entre taules, ho hem d'indicar a **pg_dump** perquè els bolqui també (`pg_dump -o`) en lloc de tornar a crear-los quan insereixi les dades en el procés de recuperació. Així mateix, si tenim BLOB en alguna de les nostres taules, també li ho hem d'indicar amb el paràmetre corresponent (`pg_dump -b`) perquè els inclogui en el bolcatge.

Més informació

Hi ha altres opcions interessants que podem consultar mitjançant el paràmetre `-h`.

Per a restaurar un bolcatge realitzat amb **pg_dump**, podem utilitzar directament el client **psql**:

```
$ psql demo < fitxer_sortida.sql
```

Una vegada recuperada una base de dades d'aquesta manera, es recomana executar la sentència `analyze` perquè l'optimitzador intern de consultes de PostgreSQL torni a calcular els índexs, la densitat de les claus, etc.

pg_dump

pg_dump realitza la còpia a partir de la base de dades de sistema *template0*, per la qual cosa també es bolcaran els tipus definits, funcions, etc. de la base de dades. Quan recuperem aquesta base de dades, l'hem de crear a partir de *template0* si hem personalitzat *template1* (i no de *template1* com ho faria per defecte la sentència `create database`) per a evitar duplicitats.

Les facilitats del sistema operatiu Unix, permeten copiar una base de dades en una altra en un altre servidor de la manera següent:

```
$ pg_dump -h host1 demo | psql -h host2 demo
```

Per a fer la còpia de seguretat a nivell de fitxer, simplement copiem els fitxers binaris on PostgreSQL emmagatzema la base de dades (especificat en temps de

compilació, o en paquets binaris, sol ser */var/lib/postgres/data*), o bé fem un arxiu comprimit amb aquests:

```
$ tar -cvzf copia_bd.tar.gz /var/lib/postgres/data
```

El servei PostgreSQL ha d'estar aturat abans de fer la còpia.

Sovint, en bases de dades grans, aquest tipus de bolcatges origina fitxers que poden superar els límits del sistema operatiu. En aquests casos haurem d'utilitzar tècniques de creació de volums de mida fixa en les ordres **tar** o altres amb què estiguem familiaritzats.

7.7. Manteniment rutinari de la base de dades

Hi ha una sèrie d'activitats que l'administrador d'un sistema gestor de bases de dades ha de tenir presents constantment i que haurà de realitzar periòdicament. En el cas de PostgreSQL, aquestes es limiten a un manteniment i neteja dels identificadors interns i de les estadístiques de planificació de les consultes, a una reindexació periòdica de les taules i al tractament dels fitxers de registre.

7.7.1. *vacuum*

El procés que fa la neteja de la base de dades en PostgreSQL es diu *vacuum*. La necessitat de dur a terme processos de *vacuum* periòdicament es justifica pels motius següents:

- Recuperar l'espai de disc perdut en esborraments i actualitzacions de dades.
- Actualitzar les estadístiques de dades utilitzades pel planificador de consultes SQL.
- Protegir-se davant de la pèrdua de dades per reutilització d'identificadors de transacció.

Per a dur a terme un *vacuum*, haurem d'executar periòdicament les sentències **vacuum** i **analyze**. En cas que hi hagi algun problema o acció addicional a realitzar, el sistema ens ho indicarà:

```
demo=# VACUUM;
WARNING: some databases have not been vacuumed in 1613770184 transactions
HINT: Better vacuum them within 533713463 transactions, or you may have a wraparound failure.
```

```
VACUUM
demo=# VACUUM VERBOSE ANALYZE;
INFO: fent vacuum a "public.guany"
INFO: "guany": es van trobar 0 versions de files eliminables i 2 de no eliminables en 1 pàgines
DETAIL: 0 versions mortes de files no poden ser eliminades encara.
Hi va haver 0 punters d'ítem sense ús.
0 pàgines són completament buides.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO: analitzant "public.guany"
INFO: "guany": 1 pàgines, 2 files mostrejades, s'estimen 2 files en total
VACUUM
```

7.7.2. Reindexació

La reindexació completa de la base de dades no és una tasca gaire habitual, però pot millorar de manera substancial la velocitat de les consultes complexes en taules amb molta activitat.

```
demo=# reindex database demo;
```

7.7.3. Fitxers de registre

És una bona pràctica mantenir arxius de registre de l'activitat del servidor. Almenys, dels errors que origina. Durant el desenvolupament d'aplicacions pot ser molt útil disposar també d'un registre de les consultes efectuades, encara que en bases de dades de molta activitat, disminueix el rendiment del gestor i no és de gaire utilitat.

En qualsevol cas, és convenient disposar de mecanismes de rotació dels fitxers de registre; és a dir, que cada cert temps (12 hores, un dia, una setmana...), es faci una còpia d'aquests fitxers i se'n comencin de nous, cosa que ens permetrà mantenir un historial d'aquests (tants com fitxers puguem emmagatzemar segons la mida que tinguin i les nostres limitacions d'espai en disc).

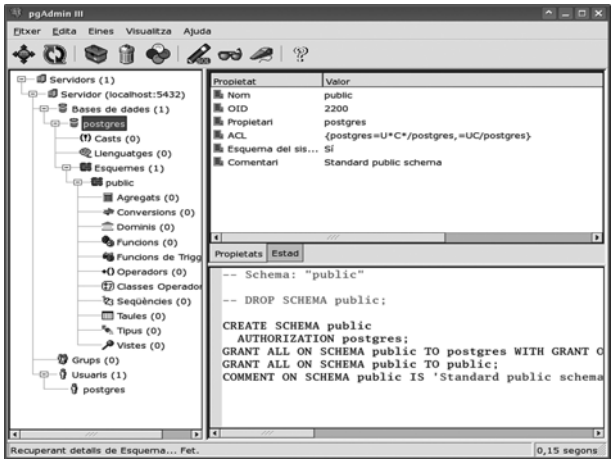
PostgreSQL no proporciona directament utilitats per a realitzar aquesta rotació, però en la majoria de sistemes Unix s'inclouen utilitats com *logrotate* que fan aquesta tasca a partir d'una planificació temporal.

8. Client gràfic: pgAdmin3

El màxim exponent de client gràfic de PostgreSQL és el programari pgAdmin3 que té llicència "Artist License", aprovada per l'FSF.

WEB

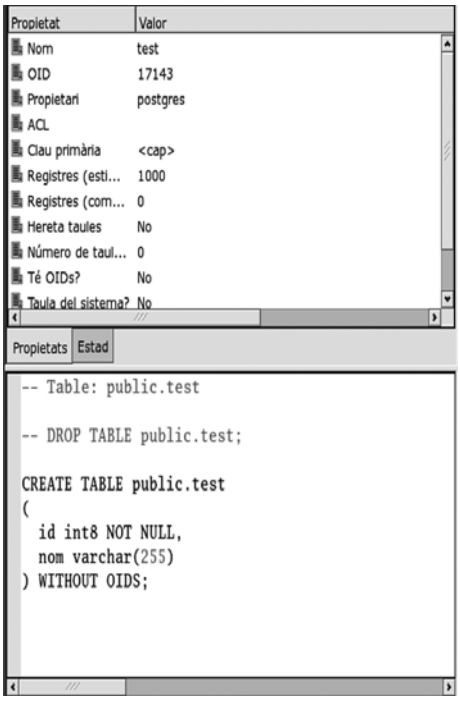
pgAdmin3 està disponible a http://www.pgadmin.org.



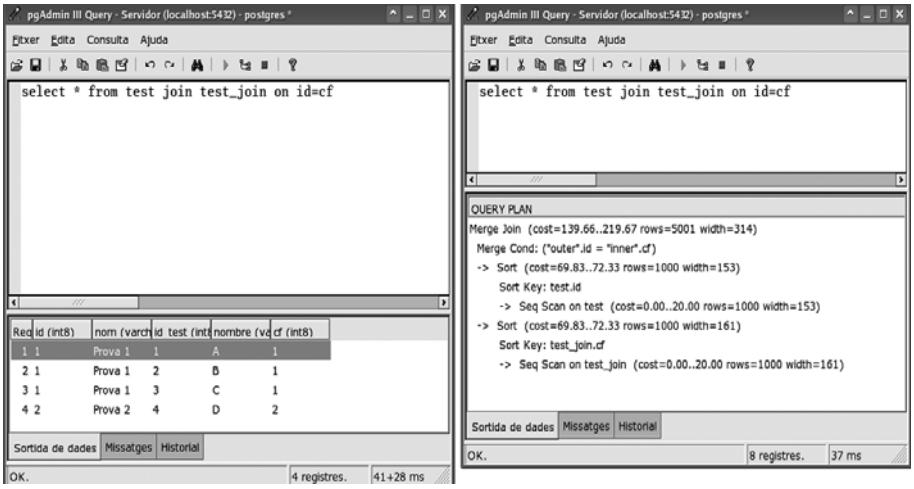
A pgAdmin3 podem veure i treballar amb gairebé tots els objectes de la base de dades, examinar les seves propietats i realitzar tasques administratives.

- Agregats
- Casts
- Columnes
- Constraints
- Conversions
- Bases de dades
- Dominis
- Funcions
- Grups
- Índexs
- Llenguatges (PLpgsql, PLpython, PLperl, etc.)
- Classes d'operadors
- Operadors
- Servidors PostgreSQL
- Regles
- Esquemes
- Seqüències
- Taules
- Triggers
- Tipus de dades
- Usuaris
- Vistes

Una característica interessant de pgAdmin3 és que, cada vegada que realitzem alguna modificació en un objecte, escriu la sentència o sentències SQL corresponents, cosa que fa que, a més d'una eina molt útil, sigui alhora didàctica.



pgAdmin3 també incorpora funcionalitats per a realitzar consultes, examinar-ne l'execució (com l'ordre **explain**) i treballar amb les dades.



Totes aquestes característiques fan de pgAdmin3 l'única eina gràfica que necessitarem per a treballar amb PostgreSQL, tant des del punt de vista de l'usuari com de l'administrador. Evidentment, les accions que podem realitzar en cada moment dependran dels permisos de l'usuari amb qui ens connectem a la base de dades.

Resum

PostgreSQL implementa les característiques necessàries per a competir amb qualsevol altra base de dades comercial, amb l'avantatge de tenir una llicència de lliure distribució BSD.

La migració de bases de dades allotjades en productes comercials a PostgreSQL es facilita gràcies al fet que suporta àmpliament l'estàndard SQL. PostgreSQL té una sèrie de característiques atractives com són l'herència de taules (classes), un ric conjunt de tipus de dades que inclouen arranjaments, BLOB, tipus geomètrics i d'adreces de xarxa. PostgreSQL inclou també el processament de transaccions, integritat referencial i procediments emmagatzemats. En concret, hi ha procediments documentats per a migrar els procediments emmagatzemats desenvolupats en llenguatges propietaris de bases de dades comercials (PL/SQL) a PL/PGSQL.

L'API es distribueix per a diversos llenguatges de programació com C/C++, Perl, PHP, Python, TCL/Tk i ODBC.

Per si això fos poc, PostgreSQL és extensible. És possible agregar nous tipus de dades i funcions al servidor que es comportin com els ja incorporats. També és possible inserir nous llenguatges de programació del costat del servidor per a la creació de procediments emmagatzemats. Tots aquests avantatges fan que molts programadors el triïn per al desenvolupament d'aplicacions en tots els nivells.

Entre les seves principals deficiències podem esmentar els OID. PostgreSQL està encara en evolució, s'espera que en futures versions s'incloguin noves característiques i millores al disseny intern de l'SGBD.

Bibliografia

Documentació de PostgreSQL de la distribució: http://www.postgresql.org/docs/

Silberschatz, A.; Korth, H.; Sudarshan, S. (2002). *Fundamentos de bases de datos* (4a. ed.). Madrid: McGraw-Hill.

Worsley, John C.; Drake, Joshua D. (2002). *Practical PostgreSQL*. O'Reilly.