Bases de datos en PostgreSQL

Marc Gibert Ginestà
Oscar Pérez Mora

P06/M2109/02152

www.uoc.edu

Índice

Introducción

PostgreSQL es un gestor de bases de datos orientadas a objetos (SGBDOO o ORDBMS en sus siglas en inglés) muy conocido y usado en entornos de software libre porque cumple los estándares SQL92 y SQL99, y también por el conjunto de funcionalidades avanzadas que soporta, lo que lo sitúa al mismo o a un mejor nivel que muchos SGBD comerciales.

El origen de PostgreSQL se sitúa en el gestor de bases de datos POSTGRES desarrollado en la Universidad de Berkeley y que se abandonó en favor de PostgreSQL a partir de 1994. Ya entonces, contaba con prestaciones que lo hacían único en el mercado y que otros gestores de bases de datos comerciales han ido añadiendo durante este tiempo.

PostgreSQL se distribuye bajo licencia BSD, lo que permite su uso, redistribución, modificación con la única restricción de mantener el *copyright* del software a sus autores, en concreto el PostgreSQL Global Development Group y la Universidad de California.

PostgreSQL puede funcionar en múltiples plataformas (en general, en todas las modernas basadas en Unix) y, a partir de la próxima versión 8.0 (actualmente en su segunda beta), también en Windows de forma nativa. Para las versiones anteriores existen versiones binarias para este sistema operativo, pero no tienen respaldo oficial.

Para el seguimiento de los ejemplos y la realización de las actividades, es imprescindible disponer de los datos de acceso del usuario administrador del gestor de bases de datos. Aunque en algunos de ellos los privilegios necesarios serán menores, para los capítulos que tratan la administración del SGBDOO será imprescindible disponer de las credenciales de administrador.

Las sentencias o comandos escritos por el usuario estarán en fuente monoespaciada, y las palabras que tienen un significado especial en PostgreSQL estarán en **negrita**. Es importante hacer notar que estas últimas no siempre son palabras reservadas, sino comandos o sentencias de psql (el cliente interactivo de PostgreSQL).

La versión de PostgreSQL que se ha utilizado durante la redacción de este material, y en los ejemplos, es la 7.4, la última versión estable en ese momento, aunque no habrá ningún problema en ejecutarlos en versiones anteriores, hasta la 7.0.

Objetivos

El objetivo principal de esta unidad es conocer el gestor de bases de datos relacionales con soporte para objetos PostgreSQL, y comentar tanto sus características comunes a otros gestores de bases de datos, como las que lo distinguen de sus competidores de código abierto.

Además, se ofrece la oportunidad de aplicar los conocimientos aprendidos en el módulo referido a SQL sobre un sistema gestor de base de datos real y examinar las diferencias entre el estándar y la implementación que hace de él el SGBD.

Por último, se presentan las tareas de administración del SGBD más habituales que un usuario debe llevar a cabo como administrador de Post-greSQL.

1. Características de PostgreSQL

En este apartado comentaremos las características más relevantes de este SGBD con soporte para objetos, tanto sus prestaciones más interesantes o destacadas, como las limitaciones en su diseño o en implementación de los estándares SQL.

También es interesante conocer un poco su historia, ya que tanto por las personas que han participado en su desarrollo como por su relación con otros gestores de bases de datos, nos ayudará a tener una mejor visión de la evolución del mismo.

1.1. Breve historia

La historia de PostgreSQL se inicia en 1986 con un proyecto del profesor Michael Stonebraker y un equipo de desarrolladores de la Universidad Berkeley (California), cuyo nombre original era POSTGRES. En su diseño se incluyeron algunos conceptos avanzados en bases de datos y soporte parcial a la orientación a objetos.

POSTGRES fue comercializado por Illustra, una empresa que posteriormente formó parte de Informix (que comercializaba el conocido SGBD del mismo nombre, recientemente absorbida por IBM y su DB/2). Llegó un momento en que mantener el proyecto absorbía demasiado tiempo a los investigadores y académicos, por lo que en 1993 se liberó la versión 4.5 y oficialmente se dio por terminado el proyecto.

En 1994, Andrew Yu y Jolly Chen incluyeron SQL en Postgres para posteriormente liberar su código en la web con el nombre de Postgres95. El proyecto incluía múltiples cambios al código original que mejoraban su rendimiento y legibilidad.

En 1996 el nombre cambió a PostgreSQL retomando la secuencia original de versiones, por lo que se liberó la versión 6.0. En el año 2004 la última versión estable oficial es la 7.4.6, mientras que la versión 8.0 está ya en fase final de estabilización.

1.2. Prestaciones

PostgreSQL destaca por su amplísima lista de prestaciones que lo hacen capaz de competir con cualquier SGBD comercial:

Los desarrolladores de proyectos basados en software libre tienen muy en cuenta PostgreSQL cuando los requerimientos de un proyecto exigen prestaciones de alto nivel.

- Está desarrollado en C, con herramientas como Yacc y Lex.

- La API de acceso al SGBD se encuentra disponible en C, C++, Java, Perl, PHP, Python y TCL, entre otros.

- Cuenta con un rico conjunto de tipos de datos, permitiendo además su extensión mediante tipos y operadores definidos y programados por el usuario.

- Su administración se basa en usuarios y privilegios.

- Sus opciones de conectividad abarcan TCP/IP, *sockets* Unix y *sockets* NT, además de soportar completamente ODBC.

- Los mensajes de error pueden estar en español y hacer ordenaciones correctas con palabras acentuadas o con la letra 'ñ'.

- Es altamente confiable en cuanto a estabilidad se refiere.

- Puede extenderse con librerías externas para soportar encriptación, búsquedas por similitud fonética (soundex), etc.

- Control de concurrencia multi-versión, lo que mejora sensiblemente las operaciones de bloqueo y transacciones en sistemas multi-usuario.

- Soporte para vistas, claves foráneas, integridad referencial, disparadores, procedimientos almacenados, subconsultas y casi todos los tipos y operadores soportados en SQL92 y SQL99.

- Implementación de algunas extensiones de orientación a objetos. En PostgreSQL es posible definir un nuevo tipo de tabla a partir de otra previamente definida.

1.3. Limitaciones

Las limitaciones de este tipo de gestores de bases de datos suelen identificarse muy fácilmente analizando las prestaciones que tienen previstas para las próximas versiones. Encontramos lo siguiente:

- Puntos de recuperación dentro de transacciones. Actualmente, las transacciones abortan completamente si se encuentra un fallo durante su ejecución. La definición de puntos de recuperación permitirá recuperar mejor transacciones complejas.

- No soporta *tablespaces* para definir dónde almacenar la base de datos, el esquema, los índices, etc.

- El soporte a orientación a objetos es una simple extensión que ofrece prestaciones como la herencia, no un soporte completo.

2. Introducción a la orientación a objetos

Dado que PostgreSQL incluye extensiones de orientación a objetos (aunque no es, como ya hemos comentado, un SGBDOO completo), es interesante repasar algunos de los conceptos relacionados con este paradigma de programación y estructuración de datos.

2.1. El modelo orientado a objetos

En 1967, el lenguaje de programación Simula aplicaba algunas ideas para modelar aspectos de la realidad de forma mucho más directa que los métodos tradicionales. Desde entonces, la orientación a objetos (OO) ha adquirido cada vez mayor popularidad al demostrar sus ventajas, entre las cuales:

- Permite un modelado más "natural" de la realidad.
- Facilita la reutilización de componentes de software.
- Ofrece mecanismos de abstracción para mantener controlable la construcción de sistemas complejos.

En el mercado aparecen constantemente herramientas y lenguajes de programación autodenominados orientados a objetos y los ya existentes evolucionan rápidamente incluyendo nuevas características de OO. De la misma manera, se han desarrollado múltiples métodos y metodologías bajo este enfoque, cada una con aportaciones propias que llegan, en ocasiones, a resultar contradictorias entre sí. Se ha logrado la creación de un lenguaje unificado para el modelado, llamado precisamente UML (*unified modeling language*). La intención de UML es ser independiente de cualquier metodología y es precisamente esta independencia la que lo hace importante para la comunicación entre desarrolladores, ya que las metodologías son muchas y están en constante evolución.

Lamentablemente, a pesar de los muchos esfuerzos y de importantes avances, las ciencias de la computación no han creado aún una definición de **modelo de objetos** como tal. En un panorama como éste, es indispensable, al menos, la existencia de un *modelo informal de objetos* que oriente la evolución de la tecnología y que tenga la aprobación de los expertos en la materia. Un modelo así permitiría su estudio consistente por parte de los profesionales de las tecnologías de la información, facilitaría la creación de mejores lenguajes y herramientas y, lo que es más importante, definiría los estándares para una metodología de desarrollo consistente y aplicable.

Sin embargo, este modelo no existe, lo que provoca inconsistencias incluso en el tratamiento de los principios y conceptos básicos de la OO. Por eso, es frecuente encontrar errores graves en el desarrollo de sistemas OO y, lo que es aún peor, se implementan soluciones de dudosa validez en herramientas de desarrollo que se dicen orientadas a objetos.

Aun sin haber alcanzado la madurez, la orientación a objetos es el paradigma que mejor permite solucionar los muchos y variados problemas que existen en el desarrollo de software. En los próximos apartados analizaremos los conceptos básicos de este modelo para identificar algunos de los problemas que aún debe resolver, lo que facilitará la comprensión y evaluación de métodos y herramientas OO.

2.2. Objetos: clase frente a instancia

Los **objetos** son abstracciones que realizamos del mundo que nos rodea y que identificamos por sus propiedades. Para la OO todo es un objeto.

Cada objeto tiene una existencia un tanto independiente de los demás objetos; es decir, tiene **identidad** propia. Aunque dos objetos tengan exactamente los mismos valores, no por eso serán el mismo objeto, seguirán siendo entidades diferentes. En los modelos OO, la identidad se representa con el identificador de objeto, IDO (OID en inglés, de *object identifier*). Teóricamente, el IDO de un objeto es único e irrepetible en el tiempo y el espacio.

Los IDO son el mecanismo que permite hacer referencia a un objeto desde otro. De esta manera las referencias tejen las relaciones entre objetos.

Función del IDO

El IDO permite que dos objetos idénticos puedan diferenciarse, no es importante que el usuario conozca los IDO, lo importante es que los diferencie el sistema.

Todos los objetos que comparten las mismas propiedades se dice que pertenecen a la misma **clase**. En los modelos OO, las clases le roban el papel central a los objetos, ya que es a través de ellas como se definen las propiedades de éstos y además se utilizan como plantillas para crear objetos.

Elementos fundamentales en OO

objeto	clase

Al crear un objeto utilizando la definición dada por una clase, obtenemos un valor para él, es lo que se llama una **instancia** del objeto. Durante la ejecución de los programas se trabaja con instancias. Como concepto, la instancia es equivalente a una tupla (fila) concreta en una tabla de una base de datos.

2.3. Propiedades: atributo frente a operación

Las propiedades de los objetos pueden ser de dos tipos, dinámicas y estáticas. Un **atributo** representa una propiedad estática de un objeto (color, coste, edad, etc.). Una **operación** representa una propiedad dinámica; es decir, una transformación sobre un atributo o una acción que puede realizar.

El conjunto de valores de los atributos en un momento dado se conoce como **estado** del objeto. Los operadores actúan sobre el objeto cambiando su estado. La secuencia de estados por la que pasa un objeto al ejecutar operaciones definen su **comportamiento**.

La posibilidad de definir comportamientos complejos es lo que hace diferente la OO.

Propiedades de los objetos

objeto (instancia)	clase
atributos (estado)	atributos
operaciones (comportamiento)	operaciones

2.4. Encapsulamiento: implementación frente a interfaz

La estructura interna de los objetos debe estar oculta al usuario de un objeto, no necesita conocerla para interactuar con él. Los objetos se conciben como una cápsula cuyo interior está oculto y no puede ser alterado directamente desde el exterior.

A la estructura interna de un objeto se la denomina **implementación** y a la parte visible, la que se presenta al exterior, **interfaz**. La interfaz se define por sus atributos y operaciones.

La implementación de una operación se conoce como **método**. La implementación de un atributo se realiza generalmente con **variables de instancia**.

Los tipos de datos abstractos

Los tipos de datos abstractos (TDA) obedecen al mismo principio de independencia de la implementación. La diferencia respecto a los objetos es que éstos incluyen los datos y las operaciones en la misma cápsula.

Encapsulamiento

Clase	
implementación	**interfaz**
variables	atributos
métodos	operaciones

El encapsulamiento comporta las siguientes ventajas:

- La modificación interna (de la implementación) de un objeto para corregirlo o mejorarlo no afecta a sus usuarios.

- La dificultad inherente a la modificación de la implementación de un objeto sea independiente del tamaño total del sistema. Esto permite que los sistemas evolucionen con mayor facilidad.

- La simplificación en el uso del objeto al ocultar los detalles de su funcionamiento y presentarlo en términos de sus propiedades. Al elevar el nivel de abstracción se disminuye el nivel de complejidad de un sistema. Es posible modelar sistemas de mayor tamaño con menor esfuerzo.

- Constituye un mecanismo de integridad. La dispersión de un fallo a través de todo el sistema es menor, puesto que al presentar una división entre interfaz e implementación, los fallos internos de un objeto encuentran una barrera en el encapsulamiento antes de propagarse al resto del sistema.

- Permite la sustitución de objetos con la misma interfaz y diferente implementación. Esto permite modelar sistemas de mayor tamaño con menor esfuerzo.

Estas características del encapsulamiento han contribuido en gran medida a la buena reputación de la OO.

Paradójicamente, el encapsulamiento, a pesar de ser uno de los conceptos básicos en la OO, no siempre se interpreta y se aplica correctamente. Especialmente en lo referente a encapsulamiento de atributos.

Diferenciemos operación y método a través de un ejemplo.

Consideremos tres objetos: polígono, círculo y punto.

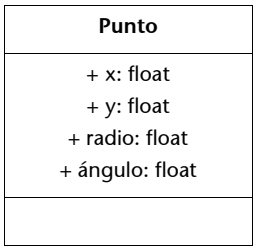
A los tres se les solicita la operación de imprimir. En esta situación, tenemos que:

- La operación solicitada es la misma, porque el significado del resultado es el mismo.
- Cada objeto ejecuta la operación de forma diferente; es decir, con un método diferente.
- Cada objeto, internamente, puede tener más de un método y selecciona el más apropiado según las circunstancias.

Las operaciones no son exclusivas de los tipos de objeto, los métodos sí. Una operación especifica "qué" hacer y un método "cómo" hacerlo. Esta diferencia permite tener múltiples métodos para una misma operación.

Veamos ahora la diferencia entre atributos y variables de instancia, que puede parecer más sutil.

Un atributo es la vista externa de una propiedad estática de un objeto. La representación interna puede variar, los atributos pueden implementarse también con métodos. Tomemos como ejemplo el objeto punto con los atributos que se muestran a continuación:



Los atributos de un punto pueden definirse en coordenadas angulares o rectangulares; en este caso, es posible conocer ambas representaciones. En la implementación de estos atributos, dos pueden ser variables de instancia y los otros dos se implementan como métodos, que se calculan a través de los primeros.

Desde el exterior no debe ser posible conocer la representación elegida internamente. Puede cambiarse la implementación de los atributos sin alterar la interfaz. En algunos casos puede incluso permitirse al sistema la elección de la representación interna de un atributo del mismo modo que una operación elige entre varios métodos disponibles.

2.4.1. Atributo frente a variable de instancia

Un atributo especifica una cualidad de un objeto; una variable de instancia especifica cómo se almacenan los valores para esa cualidad.

Los atributos

Un atributo puede ser almacenado en una variable o calculado por un método.

Consideremos tres objetos, nombre, foto, vídeo, de los que necesitamos conocer el tamaño y prever, así, el espacio necesario para almacenarlos en disco.

En esta situación tenemos que:

- El atributo es el mismo, porque su lectura tiene el mismo significado.
- Cada objeto implementa el atributo de manera diferente. Sin importar la implementación, externamente todos los atributos entregan el mismo tipo de valor. Por ejemplo:
  - El nombre puede utilizar un byte como variable de instancia, porque el tamaño de un nombre no puede ser mayor que 255 caracteres, o se puede implementar un método que calcule el tamaño en tiempo de ejecución.
  - La foto utilizará dos o cuatro bytes.
  - El vídeo puede almacenar el valor de tamaño en múltiplos de K.
- Cada objeto puede tener implementaciones alternativas que se adapten a las circunstancias.

Implementación del encapsulamiento

Clase	
implementación	**interfaz**
variables	operaciones de lectura/escritura
métodos	operaciones

Lamentablemente, los lenguajes de programación comúnmente utilizados no implementan mecanismos adecuados para el encapsulamiento de los atributos, llegando, incluso, a permitir el acceso público a variables de instancia. A continuación, analizaremos las graves consecuencias de este hecho.

Acceder a un atributo es, en realidad, una operación que puede ser de lectura o de escritura. Por este motivo, frecuentemente se define el encapsulamiento como la ocultación de todas las variables permitiendo el acceso del exterior sólo para operaciones. Cuando el lenguaje de programación no ofrece independencia de la implementación en los atributos, se deben definir una variable de instancia y dos métodos por cada atributo: *LeerAtributo* y *EscribirAtributo*.

Las bases de datos relacionales tienen perfectamente diferenciada la interfaz de la implementación en sus tipos de datos: la forma de almacenarlos es completamente independiente de la forma de consultarlos o guardarlos. No se conciben las operaciones como internas a los objetos.

El encapsulamiento ha sido considerado como un principio central de la orientación a objetos y atentar contra él significa para muchos romper con sus reglas fundamentales. Sin embargo, las bases de datos orientadas a objetos tienen entre sus funciones la realización de consultas, que necesita acceder a los atributos de los objetos. Dado que los objetos se implementan con variables, al accederlos se rompe el encapsulamiento.

La mayoría de los lenguajes orientados a objetos permiten romper el encapsulamiento de forma parcial, declarando variables como públicas. El encapsulamiento, en estos casos, se proporciona como un mecanismo opcional, ya que el usuario puede declarar todas las variables públicas y, por lo tanto, accesibles directamente.

Otros lenguajes implementan operaciones de lectura/escritura que permiten acceder a las variables sin romper el encapsulamiento.

2.5. Herencia: jerarquía de clases

La herencia se define como el mecanismo mediante el cual se utiliza la definición de una clase llamada “padre”, para definir una nueva clase llamada “hija” que puede heredar sus atributos y operaciones.

A las clases "hijo" también se les conoce como **subclases**, y a las clases "padre" como **superclases**. La relación de herencia entre clases genera lo que se llama **jerarquía de clases**.

Hablamos de herencia de tipo cuando la subclase hereda la interfaz de una superclase; es decir, los atributos y las operaciones. Hablamos de herencia estructural cuando la subclase hereda la implementación de la superclase; es decir, las variables de instancia y los métodos.

La herencia de tipo

En la herencia de tipo lo que hereda la subclase son los atributos de la superclase, pero no necesariamente su implementación, puesto que puede volver a implementarlos.

La herencia de tipo define relaciones **es-un** entre clases, donde la clase "hijo" tiene todas las propiedades del "padre", pero el "padre" no tiene todas las propiedades del "hijo".

Ejemplo

Un gato es-un animal. Todas las propiedades de la clase "animal" las tiene la clase "gato". Pero un animal no-es necesariamente un gato. Todas las propiedades de gato no las tienen todos los animales.

Consideremos una referencia mascota que es de tipo animal, en algún lenguaje de programación.

```
mimascota: Animal;
```

Puede hacer referencia a objetos de tipo animal, o tipos derivados de éste, como perro, gato o canario, por ejemplo.

```
mimascota = new Canario;
```

Se construye un nuevo canario y se hace referencia a él como mascota.

La propiedad de sustituir objetos que descienden del mismo padre se conoce como polimorfismo, y es un mecanismo muy importante de reutilización en la OO.

La referencia al tipo animal es una referencia polimorfa, ya que puede referirse a tipos derivados de animal. A través de una referencia polimorfa se pueden solicitar operaciones sin conocer el tipo exacto.

```
mimascota.comer();
```

La operación comer tiene el mismo significado para todos los animales. Como ya hemos comentado, cada uno utilizará un método distinto para ejecutar la operación.

Para conocer el tipo exacto del objeto en cuestión, se utiliza el operador de **información de tipo**. De este modo puede accederse a las propiedades específicas de un tipo de objeto que no están en los demás tipos.

En este ejemplo llamamos al operador información de tipo, **instancia-de**.

```
if (mimascota instancia-de Canario)
  mimascota.cantar();
```

Si la mascota es una instancia del tipo Canario entonces se le solicitará cantar, que es una propiedad que no tienen todas las mascotas.

Una clase puede heredar las propiedades de dos superclases mediante lo que se conoce como **herencia múltiple**.

En una herencia múltiple, puede ocurrir que en ambas superclases existan propiedades con los mismos nombres, situación que se denomina **colisión de nombres.** A continuación, se relacionan los posibles casos de colisión de nombres en la herencia de tipo:

- Los nombres son iguales porque se refieren a la misma propiedad (ya hemos visto ejemplos de ello: la operación imprimir y el atributo tamaño). En este caso no hay conflicto porque el significado está claro: es la misma propiedad, sólo hay que definir una implementación adecuada.

- Los nombres son iguales pero tienen significados diferentes. Esta situación es posible porque el modelado es una tarea subjetiva y se soluciona cambiando los nombres de las propiedades heredadas que tengan conflicto.

La herencia múltiple no comporta problemas para la herencia de tipo, puesto que no pretende la reutilización de código, sino el control conceptual de la complejidad de los sistemas mediante esquemas de clasificación.

Por lo que respecta a la herencia estructural, que, recordemos, consiste en que la subclase hereda las variables de instancia y los métodos de la superclase –es decir, la implementación–, la cosa cambia.

Para entender mejor la herencia estructural, diremos informalmente que representa una relación **funciona-como**. Por ejemplo, se puede utilizar para definir un avión tomando como superclase ave, de esta manera la capacidad de volar del ave queda implementada en el avión. Un avión no **es-un** ave, pero podemos decir que **funciona-como** ave.

Al aplicar la herencia de esta manera se dificulta la utilización del polimorfismo: aunque un objeto funcione internamente como otro, no se garantiza que externamente pueda tomar su lugar porque **funciona-como**.

El objetivo de la herencia estructural es la reutilización de código, aunque en algunos casos, como el ejemplo anterior, pueda hacer conceptualmente más complejos los sistemas.

Siempre que es posible aplicar la herencia de tipo, puede aplicarse la herencia estructural, por lo que la mayoría de los lenguajes de programación no hacen distinción entre los dos tipos de herencia.

Ejemplo

Si un canario **es-un** animal, entonces un canario **funciona-como** animal, más otras propiedades especificas de canario.

Los lenguajes de programación comúnmente no hacen distinción entre la herencia estructural y la herencia de tipo.

La **herencia estructural múltiple** permite heredar variables y métodos de varias superclases, pero surgen problemas que no son fáciles de resolver, especialmente con las variables de instancia.

Para resolver el conflicto de una variable de instancia duplicada, se puede optar por las siguientes soluciones:

- Cambiar los nombres, lo que puede provocar conflictos en los métodos que las utilizan.

- Eliminar una de las variables. Pero puede pasar que realicen alguna función independiente, en cuyo caso, sería un error eliminar una.

- No permitir herencia múltiple cuando hay variables duplicadas.

Como se puede observar, no es fácil solucionar conflictos entre variables de instancia, por ello muchos lenguajes optan por diversos mecanismos incluyendo la prohibición de la herencia múltiple.

2.5.1. Tipo y clase

Tenemos que advertir que la mayoría de lenguajes de programación no diferencian los conceptos de tipo y clase y que la diferencia que establecen algunos autores no es demasiado clara. De todas maneras, la tendencia sería definir dichos conceptos como sigue:

- Un tipo es un conjunto de objetos que comparten la misma interfaz.

- Una clase es un conjunto de objetos que comparten la misma implementación.

Una solución que se aplica es incluir en el lenguaje el concepto de **interfaz** que define solamente las operaciones de una clase, pero no ofrece alternativas para los atributos. Sin embargo, con la diferenciación entre clases e interfaces no se logra la diferenciación entre los dos tipos de herencia, pues las clases se utilizan para representar relaciones **es-un**.

2.6. Agregación: jerarquía de objetos

Los objetos son, por naturaleza, complejos; es decir, están compuestos de objetos más pequeños. Un sistema de información debe reflejar esta propiedad de los objetos de forma natural. En una base de datos relacional, un objeto complejo debe ser descompuesto en sus partes más simples para ser almacenado. Al extraerlo, es necesario ensamblar cada una de sus partes.

Ejemplo

Un automóvil está compuesto de carrocería, motor, ruedas, etc.

Por este motivo el modelo relacional comporta problemas cuando se utiliza en aplicaciones como el CAD, donde los objetos que se procesan son muy complejos.

Las bases de datos de objetos deben proporcionar la facilidad de obtener un objeto complejo en una sola consulta de forma transparente. En este caso, los

apuntadores son un mecanismo excelente para representar composición, ya que permiten acceder rápidamente a las partes componentes de un objeto, sin importar su lugar de almacenamiento.

Las bases de datos requieren independencia de la aplicación, lo que provoca un conflicto conceptual cuando se trabaja con objetos compuestos: las bases de datos deben almacenar información independiente de la aplicación para que nuevas aplicaciones puedan hacer diferentes interpretaciones de la información original; pero con los objetos compuestos esto no es tan sencillo, puesto que suelen tener una sola interpretación, o mejor dicho, una sola manera de ser consultado en una base de datos.

2.7. Persistencia

La persistencia se define como la capacidad de un objeto para sobrevivir al tiempo de ejecución de un programa. Para implementarla, se utiliza el almacenamiento secundario.

Se han propuesto varios mecanismos para implementar la persistencia en los lenguajes de programación, entre los que podemos destacar los siguientes:

- Archivos planos. Se crean archivos para almacenar los objetos en el formato deseado por el programador. Los objetos se cargan al abrir el programa y se guardan al finalizar. Ésta es la opción más accesible para todos los lenguajes de programación.

- Bases de datos relacionales. Los objetos son mapeados a tablas, un módulo del programa se encarga de hacer las transformaciones objeto-relacionales. Este enfoque consume mucho tiempo al realizar el mapeo. Existen algunas herramientas que realizan mapeos semiautomáticos.

- Bases de objetos. Los objetos son almacenados de forma natural en una base de objetos, y la consulta y recuperación es administrada por el gestor, de esta forma las aplicaciones no necesitan saber nada sobre los detalles de implementación.

- Persistencia transparente. Los objetos son almacenados y recuperados por el sistema cuando éste lo cree conveniente, sin que el usuario deba hacer ninguna solicitud explícita de consulta, actualización o recuperación de información a una base de objetos. No se requiere, por lo tanto, otro lenguaje para interactuar con las bases de datos.

2.8. PostgreSQL y la orientación a objetos

El argumento a favor de las bases de datos objeto-relacionales sostiene que permite realizar una migración gradual de sistemas relacionales a los orientados

a objetos y, en algunas circunstancias, coexistir ambos tipos de aplicaciones durante algún tiempo.

El problema de este enfoque es que no es fácil lograr la coexistencia de dos modelos de datos diferentes como son la orientación a objetos y el modelo relacional. Es necesario equilibrar de alguna manera los conceptos de uno y otro modelo sin que entren en conflicto.

Uno de los conceptos fundamentales en la orientación a objetos es el concepto de clase. Existen dos enfoques para asociar el concepto de clase con el modelo relacional:

1.er enfoque: las clases definen tipos de tablas
2.o enfoque: las clases definen tipos de columnas

Dado que en el modelo relacional las columnas están definidas por tipos de datos, lo más natural es hacer corresponder las columnas con las clases.

	1.er enfoque	2.o enfoque
Los objetos son	valores	tuplas
Las clases son	dominios	tablas

PostgreSQL implementa los objetos como tuplas y las clases como tablas. Aunque también es posible definir nuevos tipos de datos mediante los mecanismos de extensión.

Los mecanismos de extensión

No es habitual que el usuario utilice los mecanismos de extensión pues se consideran mecanismos avanzados.

Dado que las tablas son clases, pueden definirse como herencia de otras. Las tablas derivadas son polimorfas y heredan todos los atributos (columnas) de la tabla *padre* (incluida su clave primaria). Si no se manejan con precaución, las tablas polimorfas pueden conducir a errores de integridad al duplicar claves primarias. PostgreSQL soporta algunas extensiones del lenguaje SQL para crear y gestionar este tipo de tablas.

Veremos estos conceptos más en detalle en el subapartado 4.2 de esta unidad didáctica.

3. Acceso a un servidor PostgreSQL

3.1. La conexión con el servidor

Antes de intentar conectarse con el servidor, debemos asegurarnos de que está funcionando y que admite conexiones, locales (el SGBD se está ejecutando en la misma máquina que intenta la conexión) o remotas.

Una vez comprobado el correcto funcionamiento del servidor, debemos disponer de las credenciales necesarias para la conexión. Para simplificar, supondremos que disponemos de las credenciales* del administrador de la base de datos (normalmente, usuario PostgreSQL y su contraseña).

*** Distintos tipos de credenciales permiten distintos niveles de acceso.**

En el apartado que concierne a la administración de PostgreSQL se comenta detalladamente los aspectos relacionados con el sistema de usuarios, contraseñas y privilegios del SGBD.

3.2. El cliente psql

Para conectarse con un servidor, se requiere, obviamente, un programa cliente. Con la distribución de PostgreSQL se incluye un cliente, psql, fácil de utilizar, que permite la introducción interactiva de comandos en modo texto.

El siguiente paso es conocer el nombre de una base de datos residente en el servidor. El siguiente comando permite conocer las bases de datos residentes 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)
~$
```

Para realizar una conexión, se requieren los siguientes datos:

- Servidor. Si no se especifica, se utiliza *localhost*.
- Usuario. Si no se especifica, se utiliza el nombre de usuario Unix que ejecuta psql.
- Base de datos.

Ejemplos del uso de psql para conectarse con un servidor de bases de datos

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

Las dos formas anteriores ejecutan psql con la base de datos *demo*.

```
~$ psql -d demo -U yo
~$ psql demo yo
~$ psql -h servidor.org -U usuario -d basedatos
```

A partir del fragmento anterior, el cliente psql mostrará algo similar a lo siguiente:

```
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 símbolo '#', que significa que psql está listo para leer la entrada del usuario.

Notación

Hemos utilizado los dos comandos de ayuda que ofrece el lenguaje:

- **\h** Explica la sintaxis de sentencias SQL.
- **\?** Muestra los comandos internos disponibles.

Para salir de ayuda, se presiona la tecla 'q'.

Las sentencias SQL se envían directamente al servidor para su interpretación, los comandos internos tienen la forma *\comando* y ofrecen opciones que no están incluidas en SQL y son interpretadas internamente por psql.

Para terminar la sesión con psql, utilizamos el comando **\q** o podemos presionar Ctrl-D.

3.3. Introducción de sentencias

Las sentencias SQL que escribamos en el cliente deberán terminar con ';' o bien con '\g':

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

Cuando un comando ocupa más de una línea, el indicador cambia de forma y va señalando el elemento que aún no se ha completado.

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

Indicadores de PostgreSQL	
Indicador	**Significado**
=#	Espera una nueva sentencia
-#	La sentencia aún no se ha terminado con ";" o \g
"#	Una cadena en comillas dobles no se ha cerrado
'#	Una cadena en comillas simples no se ha cerrado
(#	Un paréntesis no se ha cerrado

El cliente psql almacena la sentencia hasta que se le da la orden de enviarla al SGBD. Para visualizar el contenido del *buffer* donde ha almacenado la sentencia, disponemos de la orden '\p':

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

El cliente también dispone de una orden que permite borrar completamente el *buffer* para empezar de nuevo con la sentencia:

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

3.3.1. Expresiones y variables

El cliente psql dispone de multitud de prestaciones avanzadas; entre ellas (como ya hemos comentado), el soporte para sustitución de variables similar al de los *shells* de Unix:

```
demo=>\set var1 demostracion
```

Esta sentencia crea la variable 'var1' y le asigna el valor 'demostración'. Para recuperar el valor de la variable, simplemente deberemos incluirla precedida de ':' en cualquier sentencia o bien ver su valor mediante la orden 'echo':

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

De la misma forma, psql define algunas variables especiales que pueden ser útiles para conocer detalles del servidor al que estamos conectados:

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

El uso de variables puede ayudar en la ejecución de sentencias SQL:

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

Se debe ser muy cuidadoso con el uso de las comillas y también es importante tener en cuenta que dentro de cadenas de caracteres no se sustituyen variables.

3.4. Proceso por lotes y formatos de salida

Además de ser interactivo, psql puede procesar comandos por lotes almacenados en un archivo del sistema operativo mediante la siguiente sintaxis:

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

Aunque el siguiente comando también funciona en el mismo sentido, no es recomendable usarlo porque de este modo, psql no muestra información de depuración importante, como los números de línea donde se localizan los errores, en caso de haberlos:

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

El propio intérprete psql nos proporciona mecanismos para almacenar en fichero el resultado de las sentencias:

- Especificando el fichero destino* directamente al finalizar una sentencia:

*** Hemos almacenado el resultado en el fichero *'/tmp/a.txt'*.**

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

- Mediante una *pipe* enviamos la salida a un comando Unix:

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

Notación

A la orden '\o' se le debe especificar un fichero o bien un comando que irá recibiendo los resultados mediante una *pipe*.

Cuando se desee volver a la salida estándar STDOUT, simplemente se dará la orden '\o' sin ningún parámetro.

- Mediante la orden '\o' se puede indicar dónde debe ir la salida de las sentencias SQL que se ejecuten en adelante:

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

- Se puede solicitar la ejecución de un solo comando y terminar inmediatamente mediante la siguiente forma:

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

- Se puede especificar el formato de salida de los resultados de una sentencia. Por defecto, psql los muestra en forma tabular mediante texto. Para cambiarlo, se debe modificar el valor de la variable interna 'format' mediante la orden '\pset'. Veamos, en primer lugar, la especificación del formato de salida:

La salida del fichero

Al haber especificado que se quiere la salida en html, la podríamos redirigir a un fichero (ya hemos visto cómo hacerlo) y generar un archivo html que permitiese ver el resultado de la consulta mediante 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=#
```

Hay otros formatos de salida, como 'aligned', 'unaligned', 'html' y 'latex'. Por defecto, psql muestra el resultado en formato 'aligned'.

Tenemos también multitud de variables para ajustar los separadores entre columnas, el número de registros por página, el separador entre registros, título de la página html, etc. Veamos un ejemplo:

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

La salida de este fichero

Con esta configuración, y dirigiendo la salida a un fichero, generaríamos un fichero *CSV* listo para ser leído en una hoja de cálculo u otro programa de importación de datos.

Para poder realizar los ejemplos del resto del presente apartado, se debe procesar el contenido del fichero *demo.sql* tal como se transcribe a continuación.

Contenido del fichero *demo.psql*

```
--drop table productos;
--drop table proveedores;
--drop table precios;
--drop table ganancia;

create table productos (
   parte              varchar(20),
   tipo               varchar(20),
   especificación     varchar(20),
   psugerido          float(6),
   clave              serial,

   primary key(clave)
);

insert into productos (parte,tipo,especificación,psugerido) values
   ('Procesador','2 GHz','32 bits',null);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Procesador','2.4 GHz','32 bits',35);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Procesador','1.7 GHz','64 bits',205);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Procesador','3 GHz','64 bits',560);
insert into productos (parte,tipo,especificación,psugerido) values
   ('RAM','128MB','333 MHz',10);
insert into productos (parte,tipo,especificación,psugerido) values
   ('RAM','256MB','400 MHz',35);
```

```
insert into productos (parte,tipo,especificación,psugerido) values
   ('Disco Duro','80 GB','7200 rpm',60);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Disco Duro','120 GB','7200 rpm',78);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Disco Duro','200 GB','7200 rpm',110);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Disco Duro','40 GB','4200 rpm',null);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Monitor','1024x876','75 Hz',80);
insert into productos (parte,tipo,especificación,psugerido) values
   ('Monitor','1024x876','60 Hz',67);

create table proveedores (
   empresa       varchar(20) not null,
   credito       bool,
   efectivo      bool,
   primary key   empresa)
);

insert into proveedores (empresa,efectivo) values ('Tecno-k', true );
insert into proveedores (empresa,credito) values ('Patito', true );
insert into proveedores (empresa,credito,efectivo) values
('Nacio-nal', true, true );

create table ganancia(
   venta      varchar(16),
   factor     decimal (4,2)
);

insert into ganancia values('Al por mayor',1.05);
insert into ganancia values('Al por menor',1.12);

create table precios (
   empresa                 varchar(20) not null,
   clave                   int not null,
   precio  float(6),

   foreign key (empresa)   references proveedores,
   foreign key (clave)     references productos
);

   insert into precios values ('Nacional',001,30.82);
   insert into precios values ('Nacional',002,32.73);
   insert into precios values ('Nacional',003,202.25);
   insert into precios values ('Nacional',005,9.76);
   insert into precios values ('Nacional',006,31.52);
   insert into precios values ('Nacional',007,58.41);
   insert into precios values ('Nacional',010,64.38);
   insert into precios values ('Patito',001,30.40);
   insert into precios values ('Patito',002,33.63);
   insert into precios values ('Patito',003,195.59);
   insert into precios values ('Patito',005,9.78);
```

```
insert into precios values ('Patito',006,32.44);
insert into precios values ('Patito',007,59.99);
insert into precios values ('Patito',010,62.02);
insert into precios values ('Tecno-k',003,198.34);
insert into precios values ('Tecno-k',005,9.27);
insert into precios values ('Tecno-k',006,34.85);
insert into precios values ('Tecno-k',007,59.95);
insert into precios values ('Tecno-k',010,61.22);
insert into precios values ('Tecno-k',012,62.29);
```

3.5. Usar bases de datos

La siguiente orden informa sobre las bases de datos actualmente en el SGBD.

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

La orden '\c' permite conectarse a una base de datos:

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

La consulta de la tabla que contiene la base de datos `demo` se realiza mediante la orden '\d':

```
demo=# \d
                List of relations
 Schema |        Name          |   Type   |  Owner
--------+----------------------+----------+----------
 public | ganancia             | table    | postgres
 public | precios              | table    | postgres
 public | productos            | table    | postgres
 public | productos_clave_seq  | sequence | postgres
 public | proveedores          | table    | postgres
(5 rows)
```

La orden **\d** es útil para mostrar información sobre el SGBD: tablas, índices, objetos, variables, permisos, etc. Podéis obtener todas las variantes de esta sentencia introduciendo \? en el intérprete de comandos.

Consulta de las columnas de cada una de las tablas:

```
demo-# \d proveedores
          Table "public.proveedores"
Column    |Type                   | Modifiers
----------+-----------------------+-----------
 empresa  | character varying(20) | not null
 credito  | boolean               |
 efectivo | boolean               |
Indexes:
    "proveedores_pkey" primary key, btree (empresa)
```

Para crear una nueva base de datos, usaremos la sentencia `create database`:

```
mysql> create database prueba;
```

Para eliminar una base de datos, usaremos la sentencia `drop database`:

```
mysql> drop database prueba;
```

4. Creación y manipulación de tablas

4.1. Creación de tablas

Una vez conectados a una base de datos, la sentencia SQL **create table** permite crear las tablas que necesitemos:

```
demo=#     create table persona (
demo(#     nombre varchar(30),
demo(#     direccion varchar(30)
demo(# );
CREATE
```

El comando **drop table** permite eliminar tablas:

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

La tabla recién creada aparece ahora en la lista de tablas de la base de datos en uso:

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

Podemos consultar su descripción mediante el comando *\d tabla*:

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

La tabla está lista para insertar en ella algunos registros.

```
demo=# insert into persona values ( 'Alejandro Magno' , 'Babilonia' );
INSERT 24756 1
demo=# insert into persona values ( 'Federico García Lorca ' , 'Granada 65' );
INSERT 24757 1
```

El número con el que responde el comando *insert* se refiere al OID del registro insertado.

Este aspecto se explicará en detalle más adelante.

Las consultas se realizan con la sentencia `SQL` **`select`**. En este caso solicitamos que nos muestre todas las columnas de los registros en la tabla *persona*:

```
demo=# select * from persona;
nombre                 |direccion
-----------------------+------------
 Alejandro Magno       | Babilonia
 Federico García Lorca | Granada 65
(2 rows)
demo=#
```

Las tablas creadas en PostgreSQL incluyen, por defecto, varias columnas ocultas que almacenan información acerca del identificador de transacción en que pueden estar implicadas, la localización física del registro dentro de la tabla (para localizarla muy rápidamente) y, los más importantes, el OID y el TABLEOID. Estas últimas columnas están definidas con un tipo de datos especial llamado identificador de objeto (OID) que se implementa como un entero positivo de 32 bits. Cuando se inserta un nuevo registro en una tabla se le asigna un número consecutivo como OID, y el TABLEOID de la tabla que le corresponde.

En la programación orientada a objetos, el concepto de OID es de vital importancia, ya que se refiere a la identidad propia del objeto, lo que lo diferencia de los demás objetos.

Para observar las columnas ocultas, debemos hacer referencia a ellas específicamente en el comando *select*:

```
demo=# select oid, tableoid, * from persona;
oid    |tableoid  |nombre                 |direccion
-------+----------+-----------------------+------------
 17242 |    17240 | Alejandro Magno       | Babilonia
 17243 |    17240 | Federico García Lorca | Granada 65
(2 rows)
demo=#
```

Estas columnas se implementan para servir de identificadores en la realización de enlaces desde otras tablas.

Ejemplo de la utilización de OID para enlazar dos tablas

Retomamos la tabla *persona* y construimos una nueva tabla para almacenar los teléfonos.

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

La tabla *teléfono* incluye la columna *propietario* de tipo OID, que almacenará la referencia a los registros de la tabla *persona*. Agreguemos dos teléfonos a 'Alejandro Magno', para ello utilizamos su OID que es 17242:

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

Las dos tablas están vinculadas por el OID de persona.

```
demo=# select * from telefono;
tipo        |numero       | propietario
------------+-------------+-------------
      móvil |    12345678 |   17242
       casa |      987654 |   17242
(2 rows)
```

La operación que nos permite unir las dos tablas es *join*, que en este caso une *teléfono* y *persona*, utilizando para ello la igualdad de las columnas *telefono.propietario* y *persona.oid*:

```
demo=# select * from telefono join persona on (telefono.propietario = perso-na.oid);
tipo        |numero       | propietario |nombre           | direccion
------------+-------------+-------------+-----------------+-----------
      móvil |    12345678 |       17242 | Alejandro Magno | Babilonia
       casa |      987654 |       17242 | Alejandro Magno | Babilonia
(2rows)
```

Los OID de PostgreSQL presentan algunas deficiencias:

- Todos los OID de una base de datos se generan a partir de una única secuencia centralizada, lo que provoca que en bases de datos con mucha actividad de inserción y eliminación de registros, el contador de 4 bytes se desborde y pueda entregar OID ya entregados. Esto sucede, por supuesto, con bases de datos muy grandes.

- Las tablas enlazadas mediante OID no tienen ninguna ventaja al utilizar operadores de composición en términos de eficiencia respecto a una clave primaria convencional.

- Los OID no mejoran el rendimiento. Son, en realidad, una columna con un número entero como valor.

Los desarrolladores de PostgreSQL proponen la siguiente alternativa para usar OID de forma absolutamente segura:

- Crear una restricción de tabla para que el OID sea único, al menos en cada tabla. El SGBD irá incrementando secuencialmente el OID hasta encontrar uno sin usar.
- Usar la combinación OID - TABLEOID si se necesita un identificador único para un registro válido en toda la base de datos.

Por los motivos anteriores, no es recomendable el uso de OID hasta que nuevas versiones de PostgreSQL los corrijan. En caso de usarlos, conviene seguir las recomendaciones anteriores.

Es posible crear tablas que no incluyan la columna OID mediante la siguiente notación:

```
create table persona (
nombre varchar(30),
direccion varchar(30)
)without oids;
```

4.2. Herencia

PostgreSQL ofrece como característica particular la herencia entre tablas, que permite definir una tabla que herede de otra previamente definida, según la definición de herencia que hemos visto en capítulos anteriores.

Retomemos la tabla *persona* definida como sigue:

```
create table persona (
nombre varchar (30),
direccion varchar (30)
);
```

A partir de esta definición, creamos la tabla *estudiante* como derivada de *persona*:

```
create table estudiante (
demo(# carrera varchar(50),
demo(# grupo char,
demo(# grado int
demo(# ) inherits ( persona );
CREATE
```

En la tabla *estudiante* se definen las columnas *carrera*, *grupo* y *grado*, pero al solicitar información de la estructura de la tabla observamos que también incluye las columnas definidas en *persona*:

```
demo=# \d estudiante
Table "estudiante"
Column     |Type                    | Modifiers
-----------+------------------------+-----------
    nombre | character varying(30) |
 direccion | character varying(30) |
   carrera | character varying(50) |
     grupo |          character(1) |
     grado |               integer |
```

En este caso, a la tabla *persona* la llamamos **padre** y a la tabla *estudiante*, **hija**.

Cada registro de la tabla *estudiante* contiene 5 valores porque tiene 5 columnas:

```
demo=# insert into estudiante values (
demo(# 'Juan' ,
demo(# 'Treboles 21',
demo(# 'Ingenieria en Computacion',
demo(# 'A',
demo(# 3
demo(# );
INSERT 24781 1
```

La herencia no sólo permite que la tabla *hija* contenga las columnas de la tabla *padre*, sino que establece una relación conceptual **es-un**.

La consulta del contenido de la tabla *estudiante* mostrará, por supuesto, un solo registro. Es decir, no se heredan los datos, únicamente los campos (atributos) del objeto:

```
demo=# select * from estudiante;
nombre  |direccion    |carrera                    |grupo  | grado
--------+-------------+---------------------------+-------+-------
   Juan | Treboles 21 | Ingenieria en Computacion |     A | 3
(1 row)
```

Además, la consulta de la tabla *persona* mostrará un nuevo registro:

```
demo=# select * from persona;
nombre                 | direccion
-----------------------+-------------
  Federico Garca Lorca | Granada 65
       Alejandro Magno | Babilonia
                  Juan | Treboles 21
(3 rows)
```

El último registro mostrado es el que fue insertado en tabla *estudiante*, sin embargo la herencia define una relación conceptual en la que un *estudiante* **es-una** *persona*. Por lo tanto, al consultar cuántas personas están registradas en la base de datos, se incluye en el resultado a todos los estudiantes. Para consultar sólo a las personas que no son estudiantes, podemos utilizar el modificador ONLY:

```
demo=# select * from only persona;
nombre                 | direccion
-----------------------+------------
 Alejandro Magno       | Babilonia
 Federico García Lorca | Granada 65
(2 rows)
demo=#
```

No es posible borrar una tabla *padre* si no se borran primero las tablas *hijo*.

```
demo=# drop table persona;
NOTICE: table estudiante depende de table persona
ERROR: no se puede eliminar table persona porque otros objetos dependen de él
HINT: Use DROP ... CASCADE para eliminar además los objetos dependientes.
```

Como es lógico, al borrar la fila del nuevo estudiante que hemos insertado, se borra de las dos tablas. Tanto si lo borramos desde la tabla *persona*, como si lo borramos desde la tabla *estudiante*.

4.3. Herencia y OID

Los OID permiten que se diferencien los registros de todas las tablas, aunque sean heredadas: nuestro estudiante tendrá el mismo OID en las dos tablas, ya que se trata de única instancia de la clase estudiante:

```
demo=# select oid,* from persona ;
oid    |nombre                 | direccion
-------+-----------------------+------------
 17242 | Alejandro Magno       | Babilonia
 17243 | Federico García Lorca | Granada 65
 17247 | Juan                  | Treboles 21
(3 rows)
demo=# select oid,* from estudiante ;
oid    |nombre  |direccion    |carrera                    |grupo  | grado
-------+--------+-------------+---------------------------+-------+-------
 17247 | Juan   | Treboles 21 | Ingenieria en Computación | A     |     3
(1 row)
```

Dado que no se recomienda el uso de OID en bases muy grandes, y debe incluirse explícitamente en las consultas para examinar su valor, es conveniente utilizar una secuencia compartida para padres y todos sus descendientes si se requiere un identificador.

En PostgreSQL, una alternativa para no utilizar los OID es crear una columna de tipo **serial** en la tabla *padre*, así será heredada en la *hija*. El tipo **serial** define una secuencia de valores que se irá incrementando de forma automática, y por lo tanto constituye una buena forma de crear claves primarias, al igual que el tipo AUTO_INCREMENT en MySQL.

```
demo=# create table persona (
demo(# id serial,
demo(# nombre varchar (30),
demo(# direccion 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* se define como un entero y se incrementará utilizando la función *nextval()* tal como nos indica la información de la columna:

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

Al definir un tipo serial, hemos creado implícitamente una secuencia independiente de la tabla. Podemos consultar las secuencias de nuestra base de datos mediante el comando '\ds':

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

Creamos nuevamente la tabla *estudiante* heredando de *persona*:

```
create table estudiante (
demo(# carrera varchar(50),
demo(# grupo char,
demo(# grado int
demo(# ) inherits ( persona );
CREATE
```

El estudiante heredará la columna *id* y se incrementará utilizando la misma secuencia:

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

Insertaremos en la tabla algunos registros de ejemplo, omitiendo el valor para la columna *id*:

```
demo=# insert into persona(nombre,direccion)
values ( 'Federico Garca Lorca' , 'Granada 65' );
demo=# insert into persona(nombre,direccion)
values ( 'Alejandro Magno' , 'Babilonia' );
demo=# insert into estudiante(nombre,direccion,carrera,grupo,grado)
values ( 'Elizabeth' , 'Pino 35', 'Psicologia' , 'B' , 5 );
```

La tabla *estudiante* contendrá un solo registro, pero su identificador es el número 3.

```
demo=# select * from estudiante;
id |nombre     |direccion  |carrera     |grupo  |grado
---+-----------+-----------+------------+-------+-------
3  | Elizabeth | Pino 35   | Psicologia |     B |     5
(1 row)
```

Todos los registros de *persona* siguen una misma secuencia sin importar si son padres o hijos:

```
demo=# select * from persona;
 id |nombre                |direccion
----+----------------------+------------
  1 | Federico Garca Lorca | Granada 65
  2 |      Alejandro Magno | Babilonia
  3 |            Elizabeth | Pino 35
(3 rows)
```

La herencia es útil para definir tablas que conceptualmente mantienen elementos en común, pero también requieren datos que los hacen diferentes. Uno de los elementos que conviene definir como comunes son los identificadores de registro.

4.4. Restricciones

Como ya sabemos, las restricciones permiten especificar condiciones que deberán cumplir tablas o columnas para mantener la integridad de sus datos. Algunas de las restricciones vendrán impuestas por el modelo concreto que se esté implementando, mientras que otras tendrán su origen en las reglas de negocio del cliente, los valores que pueden tomar algunos campos, etc.

Los valores que puede contener una columna están restringidos en primer lugar por el tipo de datos. Ésta no es la única restricción que se puede definir para los valores en una columna, PostgreSQL ofrece las restricciones siguientes:

Ejemplo

Una columna definida como **integer** no puede contener cadenas de caracteres.

- **null** y **not null**. En múltiples ocasiones el valor de una columna es desconocido, no es aplicable o no existe. En estos casos, los valores cero, cadena vacía o falso son inadecuados, por lo que utilizamos **null** para especificar

la ausencia de valor. Al definir una tabla podemos indicar qué columnas podrán contener valores nulos y cuáles no.

```
create table Persona (
nombre varchar(40) not null,
trabajo varchar(40) null,
correo varchar(20),
);
```

El *nombre* de una persona no puede ser nulo, y es posible que la persona no tenga *trabajo*. También es posible que no tenga *correo*, al no especificar una restricción **not null**, se asume que la columna puede contener valores nulos.

- **unique**. Esta restricción se utiliza cuando no queremos que los valores contenidos en una columna puedan duplicarse.

```
create table Persona (
nombre varchar(40) not null,
conyuge varchar(40) unique,
);
```

cónyuge no puede contener valores duplicados, no permitiremos que dos personas tengan simultáneamente el mismo cónyuge.

- **primary key**. Esta restricción especifica la columna o columnas que elegimos como clave primaria. Puede haber múltiples columnas *unique*, pero sólo debe haber una clave primaria. Los valores que son únicos pueden servir para identificar una fila de la tabla de forma unívoca, por lo que se les denomina *claves candidatas*.

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

Al definir una columna como *primary key*, se define implícitamente con *unique*. El nss (número de la seguridad social) no sólo es único, sino que lo utilizamos para identificar a las personas.

- **references y foreign key**. En el modelo relacional, establecemos las relaciones entre entidades mediante la inclusión de claves foráneas en otras relaciones. PostgreSQL y SQL ofrecen mecanismos para expresar y mantener esta integridad referencial. En el siguiente ejemplo, las *Mascotas* tienen como dueño a una *Persona*:

```
create table Mascota(
nombre varchar(20),
dueño varchar(10) references Persona,
);
```

Una referencia por defecto es a una clave primaria, por lo que *dueño* se refiere implícitamente al *nss* de *Persona*. Cuando se capturen los datos de una nueva *mascota*, PostgreSQL verificará que el valor de *dueño* haga referencia a un nss que exista en *Persona*, en caso contrario emitirá un mensaje de error. En otras palabras, no se permite asignar a una mascota un dueño que no exista.

También es posible especificar a qué columna de la tabla hace referencia:

```
create table Mascota(
nombre varchar(20),
dueño varchar(10) references Persona(nss),
);
```

o su equivalente:

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

Podría darse el caso de que la clave primaria de la tabla referenciada tuviera más de una columna, en ese caso, la clave foránea también tendría que estar formada por el mismo número de columnas:

```
create table t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

Si no se especifica otra acción, por omisión la persona que tenga una mascota no puede ser eliminada, porque la mascota se quedaría sin dueño. Para poder eliminar una persona, antes se deben eliminar las mascotas que tenga. Este comportamiento no parece ser el más adecuado para el caso.

Para modificar este comportamiento disponemos de las reglas de integridad referencial del lenguaje SQL, que PostgreSQL también soporta. En el siguiente ejemplo se permite que al eliminar una persona, las mascotas simplemente se queden sin dueño.

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

En cláusula **on delete** se pueden especificar las siguientes acciones:

– **set null**. La referencia toma el valor NULL: si se elimina *Persona* su *Mascota* se quedará sin dueño.

– **set default**. La referencia toma el valor por omisión.

– **cascade**. La acción se efectúa en cascada: si se elimina *Persona* automáticamente se elimina su *Mascota*.

– **restrict**. No permite el borrado del registro: no se puede eliminar una *Persona* que tenga *Mascota*. Ésta es la acción que se toma por omisión.

Si se modifica la clave primaria de la tabla referenciada, se dispone de las mismas acciones que en el caso anterior, que especificaremos con la cláusula ON UPDATE.

- **check**. Esta restricción realiza la evaluación previa de una expresión lógica cuando se intenta realizar una asignación. Si el resultado es verdadero, acepta el valor para la columna, en caso contrario, emitirá un mensaje de error y rechazará el valor.

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

Se han restringido los valores que se aceptarán en la columna de la manera siguiente.

- Edad debe estar entre 11 y 79 años.
- Ciudad no debe una cadena vacía.
- Correo debe tener una arroba.

Comparación de expresiones regulares

El operador ~ realiza comparaciones de cadenas con expresiones regulares.
Las expresiones regulares son patrones de búsqueda muy flexibles desarrollados en el mundo Unix.

Cualquiera de esas restricciones puede tener nombre, de manera que se facilita la referencia a las restricciones específicas para borrarlas, modificarlas, etc. pues puede hacerse por nombres. Para dar nombre a una restricción, utilizamos la sintaxis siguiente:

```
constraint nombre_de_restricción <restricción>
```

4.4.1. Restricciones de tabla

Cuando las restricciones se indican después de las definiciones de las columnas, y pueden afectar a varias de ellas simultáneamente, se dice que son restricciones de tabla:

```
create table Persona (
nss int,
nombre varchar(30),
pareja varchar(30),
jefe int,
correo varchar(20),
primary key (nss),
unique (pareja),
foreign key (jefe) references Persona,
check (correo ~ '@' )
);
```

Esta notación permite que la restricción pueda abarcar varias columnas.

```
create table Curso (
materia varchar(30),
grupo char(4),
dia int,
hora time,
aula int,
primary key (materia, grupo),
unique (dia, hora, aula)
);
```

Un curso se identifica por el grupo y la materia, y dos cursos no pueden estar en la misma aula el mismo día y a la misma hora.

Al igual que la restricción de columna, a las restricciones de tabla puede asignárseles un nombre:

```
create table Persona (
nss int,
nombre varchar(30),
pareja varchar(30),
jefe int,
correo varchar(20),
constraint identificador primary key (nss),
constraint monogamia unique (pareja),
constraint un_jefe foreign key (jefe) references Persona,
check (correo ~ '@' )
);
```

La sentencia `alter table` permite añadir (`add`) o quitar (`drop`) restricciones ya definidas:

```
alter table Persona drop constraint monogamia
alter table add constraint monogamia unique (pareja);
```

4.5. Indexación

PostgreSQL crea índices para las llaves primarias de todas las tablas. Cuando se necesite crear índices adicionales, utilizaremos la expresión del ejemplo siguiente:

```
create index persona_nombre_indice on Persona ( nombre );
```

4.6. Consulta de información de bases de datos y tablas

Como ya sabemos, el cliente **psql** ofrece varias alternativas para obtener información sobre la estructura de nuestra base de datos. En la siguiente tabla se muestran algunos comandos de mucha utilidad.

Comando	Descripción
\l	Lista las bases de datos
\d	Describe las tablas de la base de datos en uso
\ds	Lista las secuencias
\di	Lista los índices
\dv	Lista las vistas
\dp \z	Lista los privilegios sobre las tablas
\da	Lista las funciones de agregados
\df	Lista las funciones
\g archivo	Ejecuta los comandos de archivo
\H	Cambia el modo de salida HTML
\! comando	Ejecuta un comando del sistema operativo

Para obtener la lista de tablas de la base de datos *demo* hacemos lo siguiente:

```
demo=# \d
List of relations
Name                 |Type      | Owner
---------------------+----------+--------
            ganancia |    table | postgres
             precios |    table | postgres
           productos |    table | postgres
 productos_clave_seq | sequence | postgres
         proveedores |    table | postgres
(5 rows)
```

La estructura de la tabla *productos* se solicita de la siguiente manera.

```
demo=# \d productos
Table "productos"
Column        |Type                 | Modifiers
--------------+---------------------+-----------------------------------------
        parte |character varying(20)|
         tipo |character varying(20)|
especificación|character varying(20)|
    psugerido |                real |
       clave |integer              |not null default nextval('"productos_clave_seq"'::text)
Primary key: productos_pkey
Triggers: RI_ConstraintTrigger_17342,
RI_ConstraintTrigger_17344
```

En el ejemplo anterior podemos observar que la columna *clave* contiene dos modificadores:

- El primero especifica que no pueden asignarse valores nulos.
- El segundo especifica el valor por omisión que deberá asignarse a la columna.

En este caso, el valor será automáticamente calculado por la función *nextval()*, que toma como argumento la **secuencia*** *productos_clave_seq*.

*** Una secuencia es un nombre especial que permite la producción de series numéricas.**

El siguiente comando muestra las secuencias creadas en una base de datos:

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

Las secuencias se crean automáticamente cuando se declaran columnas de tipo serial.

En la estructura de la tabla productos encontramos también una clave primaria. PostgreSQL generará siempre un índice para cada tabla utilizando la clave primaria. La lista de los índices de la base de datos se obtiene de la siguiente forma:

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

El conjunto de comandos proporcionados por **psql** que hemos presentado permite obtener información sobre la estructura de nuestra base de datos de una manera directa y sencilla y, también, es útil para explorar bases de datos que no conozcamos.

4.7. Tipos de datos

4.7.1. Tipos lógicos

PostgreSQL incorpora el tipo lógico **boolean**, también llamado **bool**. Ocupa un byte de espacio de almacenamiento y puede almacenar los valores falso y verdadero.

Valor	Nombre
Falso	false, 'f', 'n', 'no', 0
Verdadero	true, 't', 'y', 'yes', 1

PostgreSQL soporta los operadores lógicos siguientes: **and**, **or** y **not**.

Aunque los operadores de comparación se aplican sobre prácticamente todos los tipos de datos proporcionados por PostgreSQL, dado que su resultado es un valor lógico, describiremos su comportamiento en la siguiente tabla:

Operador	Descripción
>	Mayor que
<	Menor que
<=	Menor o igual que
>=	Mayor o igual que
<> !=	Distinto de

4.7.2. Tipos numéricos

PostgreSQL dispone de los tipos enteros **smallint**, **int** y **bigint** que se comportan como lo hacen los enteros en muchos lenguajes de programación.

Los números con punto flotante **real** y **double precisión** almacenan cantidades con decimales. Una característica de los números de punto flotante es que pierden exactitud conforme crecen o decrecen los valores.

Aunque esta pérdida de exactitud no suele tener importancia en la mayoría de las ocasiones, PostgreSQL incluye el tipo **numeric**, que permite almacenar cantidades muy grandes o muy pequeñas sin pérdida de información. Por supuesto, esta ventaja tiene un coste, los valores de tipo **numeric** ocupan un espacio de almacenamiento considerablemente grande y las operaciones se ejecutan sobre ellos muy lentamente. Por lo tanto, no es aconsejable utilizar el tipo **numeric** si no se necesita una alta precisión o se prima la velocidad de procesamiento.

Nombre	Tamaño	Otros nombres	Comentario
smallint	2 bytes	int2	
int	4 bytes	int4, integer	

Nombre	Tamaño	Otros nombres	Comentario
bigint	8 bytes	int8	
numeric(p,e)	11 + (p/2)		'p' es la precisión, 'e' es la escala
real	4 bytes	float, float4	
double precision	8 bytes	float8	
serial			No es un tipo, es un entero auto-incrementable

Serial

La declaración **serial** es un caso especial, ya que no se trata de un nuevo tipo. Cuando se utiliza como nombre de tipo de una columna, ésta tomará automáticamente valores consecutivos en cada nuevo registro.

Ejemplo de una tabla que define la columna *folio* como tipo **serial**.

```
create table Factura(
folio serial,
cliente varchar(30),
monto real
);
```

PostgreSQL respondería esta instrucción con dos mensajes:

- En el primero avisa que se ha creado una secuencia de nombre *factura_folio_seq*:

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

- En el segundo avisa de la creación de un índice único en la tabla utilizando la columna *folio*:

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

Si se declaran varias columnas con **serial** en una tabla, se creará una secuencia y un índice para cada una de ellas.

4.7.3. Operadores numéricos

PostgreSQL ofrece un conjunto predefinido de operadores numéricos, que presentamos en la siguiente tabla:

Símbolo	Operador
+	Adición
-	Substracción
*	Multiplicación
/	División
%	Módulo
^	Exponenciación
|/	Raíz cuadrada

Ejemplo

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

Símbolo	Operador
||/	Raíz cúbica
!	Factorial
!!	Factorial como operador fijo
@	Valor absoluto
&	AND binario
		OR binario
#	XOR binario
~	Negación binaria
<<	Corrimiento binario a la izquierda
>>	Corrimiento binario a la derecha

4.7.4. Tipos de caracteres

Los valores de cadena en PostgreSQL se delimitan por comillas simples.

Recordad

Las comillas dobles delimitan identificadores que contienen caracteres especiales.

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

Se puede incluir una comilla simple dentro de una cadena con \' o '':

```
demo=# select 'Él dijo: ''Hola'' ';
?column?
------------------
Él dijo: 'Hola'
(1 row)
```

Las cadenas pueden contener caracteres especiales utilizando las llamadas secuencias de escape que inician con el caracter '\':

```
\n nueva línea
\r retorno de carro
\t tabulador
\b retroceso
\f cambio de página
\r retorno de carro
\\ el caracter \
```

Las secuencias de escape se sustituyen por el carácter correspondiente:

```
demo=# select 'Esto está en \n dos renglones';
?column?
------------------------------
Esto está en
dos renglones
(1 row)
```

PostgreSQL ofrece los tipos siguientes para cadenas de caracteres:

Tipo	Otros nombres	Descripción
char(n)	character(n)	Reserva *n* espacios para almacenar la cadena
varchar(n)	character var-ying(n)	Utiliza los espacios necesarios para almacenar una cadena menor o igual que *n*
text		Almacena cadenas de cualquier magnitud

4.7.5. Operadores

En la siguiente tabla se describen los operadores para cadenas de caracteres:

Operador	Descripción	¿Distingue mayúsculas y minúsculas?
||	Concatenación	-
~	Correspondencia a expresión regular	Sí
~*	Correspondencia a expresión regular	No
!~	No correspondencia a expresión regular	Sí
!~*	No correspondencia a expresión regular	-

En la siguiente tabla se muestran algunas funciones de uso común sobre cadenas de caracteres:

Bibliografía

Es recomendable consultar el manual para obtener la referencia completa de funciones.

Función	Descripción
Length(cadena)	Devuelve la longitud de la cadena
lower(cadena)	Convierte la cadena a minúsculas
ltrim(cadena,caracteres)	Elimina de la izquierda los caracteres especificados
substring(cadena from patrón)	Extrae la subcadena que cumple el patrón especificado

Sobre las cadenas también podemos utilizar los operadores de comparación que ya conocemos.

Ejemplo

En este caso, el resultado de la comparación *menor que* es VERDADERO:

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

4.7.6. Fechas y horas

En la siguiente tabla se muestran los tipos de datos referentes al tiempo que ofrece PostgreSQL:

Tipo de dato	Unidades	Tamaño	Descripción	Precisión
date	día-mes-año	4 bytes	Fecha	Día
time	hrs:min:seg:micro	4 bytes	Hora	Microsegundo
timestamp	día-mes-año hrs:min:seg:micro	8 bytes	Fecha más hora	Microsegundo
interval	second, minute, hour, day, week, month, year, decade, century, millennium*	12 bytes	Intervalo de tiempo	Microsegundo

*** También admite abreviaturas.**

Existe un tipo de dato **timez** que incluye los datos del tipo **time** y, además, la zona horaria. Su sintaxis es la siguiente:

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

El tipo de datos **date** almacena el día, mes y año de una fecha dada y se muestra por omisión con el formato siguiente: YYYY-MM-DD:

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

Para cambiar el formato de presentación, debemos modificar la variable de entorno *datestyle*:

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

Nombre del formato	Formato	Ejemplo
ISO	Año-mes-día	2004-05-22
GERMAN	Día.mes.año	22.05.2004
POSTGRES	día-mes-año	22-05-2004
SQL	mes/día/año	05/22/2004

4.7.7. *Arrays*

Definición

Los *arrays* no cumplen la primera forma normal de Cood, por lo que muchos los consideran inaceptables en el modelo relacional.

El tipo de datos ***array*** es una de las características especiales de PostgreSQL, permite el almacenamiento de más de un valor del mismo tipo en la misma columna.

```
demo=# create table Estudiante (
demo(# nombre varchar(30),
demo(# parciales int [3]
demo(# );
CREATE
```

La columna *parciales* acepta tres calificaciones de los estudiantes.

También es posible asignar un solo valor del *array*:

```
demo=# insert into Estudiante( nombre, parciales[2]) values ( 'Pedro' , '{90}');
INSERT 17418 1
demo=# select * from Estudiante ;
nombre      | parciales
------------+------------
John Lennon |
Juan        | {90,95,97}
Pedro       | {90}
(3 rows)
```

Los *arrays*, al igual que cualquier columna cuando no se especifica lo contrario, aceptan valores nulos:

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

Los valores del array se escriben siempre entre llaves.

```
demo=# select * from Estudiante;
nombre        | parciales
--------------+------------
John Lennon   |
Juan          | {90,95,97}
(2 rows)
```

Para seleccionar un valor de un *array* en una consulta se especifica entre corchetes la celda que se va a desplegar:

```
demo=# select nombre, parciales[3] from Estudiante;
nombre        | parciales
--------------+-----------
John Lennon   |
Juan          | 97
Pedro         |
(3 rows)
```

Sólo Juan tiene calificación en el tercer parcial.

En muchos lenguajes de programación, los *array* se implementan con longitud fija, PostgreSQL permite aumentar su tamaño dinámicamente:

La columna *parciales* del registro Pablo incluye cuatro celdas y sólo la última tiene valor.

```
demo=# insert into Estudiante( nombre, parciales[4]) values ( 'Pablo' , '{70}');
INSERT 17419 1
demo=# select * from Estudiante;
nombre        | parciales
--------------+------------
John Lennon   |
Juan          | {90,95,97}
Pedro         | {90}
Pablo         | {70}
(4 rows)
```

Mediante la función array_dims() podemos conocer las dimensiones de un *array*:

```
demo=# select nombre, array_dims(parciales) from Estudiante;
nombre        | array_dims
--------------+------------
John Lennon   |
Juan          | [1:3]
Pedro         | [1:1]
Pablo         | [1:1]
(4 rows)
```

4.7.8. BLOB

El tipo de datos BLOB (Binary Large OBject) permite almacenar en una columna un objeto de gran tamaño. PostgreSQL no conoce nada sobre el tipo de información que se almacena en una columna BLOB, simplemente lo considera como una secuencia de bytes. Por este motivo, no se tienen operaciones sobre los tipos BLOB, con la excepción del operador de concatenación, que simplemente une el contenido de dos BLOB en uno.

Veamos cómo almacenar una fotografía en una tabla de personas mediante tipos BLOB.

Una primera manera de hacerlo es importando el contenido del archivo que contiene la imagen mediante la función **lo_import()**:

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

Esta función devuelve como resultado el OID del objeto insertado.¿Dónde se ha almacenado la fotografía si no hemos utilizado el comando *insert*? PostgreSQL mantiene una tabla de nombre **pg_largeobject** con el objetivo de almacenar BLOB. Podemos utilizar el OID para hacer referenciar al objeto en una tabla:

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

Los registros insertados en esta tabla llevan un número entero como OID que, en el siguiente caso, se ha obtenido solicitándolo a la función **lo_import()** anterior.

```
demo=# insert into persona values ( 'Julio' , 'Cedro 54', 17425);
```

La inserción anterior pudo haberse realizado en un solo paso:

```
demo=# insert into persona
values ( 'Julio' , 'Cedro 54', lo_import('/home/quiron/mi-foto.jpg'));
```

Para extraer el contenido de la columna, se utiliza la función **lo_export()**

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

La función **lo_unlink()** permite borrar un BLOB almacenado en *pg_largeobject:*

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

Veamos el formato que utiliza PostgreSQL para visualizar BLOB.

Se ha recortado la salida para hacer más comprensible la tabla.

```
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;\015l\232\033f\\\371\251\0323\003t\307\207~\035GB\271\17
(3 rows)
```

La fotografía se ha dividido en tres registros, que son las páginas 0, 1 y 2 del BLOB. Los tres registros tienen el mismo *loid*, lo que significa que son el mismo objeto. Obsérvese también, que los bytes son desplegados como caracteres de la forma '*ddd*', donde ddd son tres dígitos octales.

Para almacenar un BLOB en una tabla de forma directa; es decir, sin utilizar la tabla del sistema *pg_largeobject* utilizamos el tipo de dato **bytea**:

En el primer caso se está insertando un BLOB de un solo byte, el carácter ASCII cero.

En el segundo caso se están insertando 4 bytes, los tres primeros están representados directamente por los caracteres imprimibles ASCII y el tercero, el caracter ASCII 30 octal, como no es imprimible, se escribe en notación '*ddd*'.

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

La columna fotografía es de tipo **bytea**, lo que permite almacenar objetos de gran tamaño, cerca de 1GB. Para insertar valores en estas columnas, hacemos lo siguiente:

```
demo=# insert into persona values ( 'Jorge' , 'Cerezo 55', '\\000');
INSERT 17436 1
demo=# insert into persona values ( 'Luis' , 'Encino 67', 'abc\\030');
INSERT 17437 1
```

La consulta se visualiza como sigue:

```
demo=# select * from persona;
nombre  | direccion | fotografia
--------+-----------+------------
Jorge   | Cerezo 55 | \000
Luis    | Encino 67 | abc\030
(2 rows)
```

Los caracteres en notación octal se muestran con una barra invertida y con dos tal como se escribieron. Esto es debido a que, en realidad, sólo llevan una barra invertida, pero por cuestiones de diseño PostgreSQL, las literales BLOB deben escribirse con doble barra invertida.

4.8. Modificación de la estructura de una tabla

Para modificar la estructura de una tabla una vez construida, disponemos de la sentencia SQL **alter table**.

Mediante esta sentencia, podemos llevar a cabo las operaciones siguientes:

- Agregar una columna.

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

- Eliminar una columna.

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

- Fijar el valor por omisión de una columna.

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

- Eliminar el valor por omisión de una columna.

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

- Renombrar una columna.

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

- Renombrar una tabla.

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

5. Manipulación de datos

5.1. Consultas

Las consultas a la base de datos se realizan con el comando **select**, que se implementa en PostgreSQL cumpliendo en gran parte con el estándar SQL:

Notación

Omitiremos las referencias comunes a SQL y sólo se mostrarán algunas de las posibilidades de consulta con PostgreSQL. Por lo que respecta a las funciones auxiliares, se han visto algunas en el apartado de tipos de datos y, en todo caso, se recomienda la consulta de la documentación del producto para las operaciones más avanzadas.

```
demo=# select parte, tipo
demo-# from productos
demo-# where psugerido > 30
demo-# order by parte
demo-# limit 5
demo-# offset 3;
parte       | tipo
------------+----------
Monitor     | 1024x876
Monitor     | 1024x876
Procesador  | 2.4 GHz
Procesador  | 1.7 GHz
Procesador  | 3 GHz
(5 rows)
```

Al igual que MySQL, PostgreSQL admite la sentencia **explain** delante de **select** para examinar qué está ocurriendo durante una consulta:

Al igual que en el módulo de MySQL, vemos que no aprovecha los índices (básicamente porque no tenemos ninguno definido).

```
demo=# explain select productos.clave, parte||' '||tipo||' '||especificación as producto,
proveedores.empresa , precio  from productos natural join precios natural join proveedores;
                                    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".clave = "inner".clave)
      -> Seq Scan on precios  (cost=0.00..20.00 rows=1000 width=32)
      -> Hash (cost=20.00..20.00 rows=1000 width=76)
         -> Seq Scan on productos  (cost=0.00..20.00 rows=1000 width=76)
   -> Hash (cost=20.00..20.00 rows=1000 width=24)
      -> Seq Scan on proveedores  (cost=0.00..20.00 rows=1000 width=24)
(9 rows)
demo=#
```

Veamos como mejorar el rendimiento de este **select**:

```
demo=# create index empresa_idx on precios (empresa);
CREATE INDEX
demo=# create index clave_idx on precios (clave);
CREATE INDEX
demo=# explain select productos.clave, parte||' '||tipo||' '||especificación as producto,
proveedores.empresa , precio  from productos natural join precios natural join proveedores;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=29.00..56.90 rows=20 width=104)
   Hash Cond: ("outer".clave = "inner".clave)
   ->  Seq Scan on productos  (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 proveedores  (cost=0.00..20.00 rows=1000 width=24)
         ->  Hash  (cost=1.20..1.20 rows=20 width=32)
            ->  Seq Scan on precios  (cost=0.00..1.20 rows=20 width=32)
(9 rows)
demo=#
```

5.2. Actualizaciones e inserciones

PostgreSQL cumple con el estándar SQL en todos los sentidos para las sentencias de actualización, inserción y borrado. No define modificadores ni otros mecanismos para estas sentencias.

En determinadas cláusulas, y para tablas heredadas, es posible limitar el borrado o la actualización a la tabla *padre* (sin que se propague a los registros de las tablas hijas) con la cláusula **only**:

```
demo=# update only persona set nombre='Sr. '||nombre;
UPDATE 2
demo=# select * from persona;
nombre                     | direccion
---------------------------+-------------
 Sr. Alejandro Magno       | Babilonia
 Sr. Federico García Lorca | Granada 65
 Juan                      | Treboles 21
(3 rows)
demo=#
```

5.3. Transacciones

Definimos *transacción* como un conjunto de operaciones que tienen significado solamente al actuar juntas.

PostgreSQL ofrece soporte a transacciones, garantizando que ambas operaciones se realicen o que no se realice ninguna. Para iniciar una transacción, se utiliza el comando **begin** y para finalizarla, **commit**.

Ejemplo

Una compra puede ser una transacción que conste de dos operaciones:

- Insertar un registro del pago del producto
- Insertar el producto en el inventario.

No se debe insertar un producto que no se haya pagado, ni pagar un producto que no esté en el inventario, por lo tanto, las dos operaciones forman una transacción.

```
demo=# begin;
BEGIN
demo=# insert into productos values ('RAM','512MB','333 MHz',60);
INSERT 17459 1
demo=# select * from productos;
parte      | tipo     | especificación | psugerido | clave
-----------+----------+----------------+-----------+-------
Procesador | 2 Ghz    | 32 bits        |           | 1
Procesador | 2.4 GHz  | 32 bits        | 35        | 2
Procesador | 1.7 GHz  | 64 bits        | 205       | 3
Procesador | 3 GHz    | 64 bits        | 560       | 4
RAM        | 128MB    | 333 MHz        | 10        | 5
RAM        | 256MB    | 400 Mhz        | 35        | 6
Disco Duro | 80 GB    | 7200 rpm       | 60        | 7
Disco Duro | 120 GB   | 7200 rpm       | 78        | 8
Disco Duro | 200 GB   | 7200 rpm       | 110       | 9
Disco Duro | 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 precios values ('Patito',13,67);
INSERT 17460 1
```

El nuevo registro tiene como clave el 13 y, de momento, hasta que finalice la transacción, sólo puede verlo el usuario que lo ha insertado.

Insertamos un registro con el precio del proveedor Patito para el producto con clave 13.

```
demo=# commit;
```

Las transacciones de PostgreSQL

No hay ninguna característica en las transacciones de PostgreSQL que lo diferencien de lo que especifica el estándar. Ya hemos visto en apartados anteriores que las filas implicadas en una transacción mantienen unas columnas ocultas con información acerca de la propia transacción.

Al cerrar la transacción, los registros insertados ya son visibles para todos los usuarios. Si por alguna razón, por ejemplo una caída del sistema, no se ejecuta el **commit**, la transacción se cancela. La forma explícita de cancelar una transacción es con el comando **rollback**.

6. Funciones y disparadores

Como algunos de los gestores de bases de datos relacionales, comerciales líderes en el mercado, PostgreSQL puede incorporar múltiples lenguajes de programación a una base de datos en particular. Este hecho permite, por ejemplo:

- Almacenar procedimientos en la base de datos (***stored procedure***), que podrán lanzarse cuando convenga.
- Definir operadores propios.

PL/pgSQL

PL/pgSQL (*procedual language/postgreSQL*) es una extensión del SQL que permite la creación de procedimientos y funciones al estilo de los lenguajes tradicionales de programación.

PostgreSQL ofrece por defecto soporte para su propio lenguaje procedural, el PL/pgSQL. Para instalarlo, basta con invocar el comando **createlang** desde el sistema operativo, no desde la línea de psql.

```
$ createlang plpgsql demo
```

Mediante este comando, se ha instalado el lenguaje PL/pgSQL en la base de datos *demo*.

PostgreSQL también soporta otros lenguajes directamente, como PL/Tcl, PL/Perl y PL/Python.

6.1. Primer programa

Veamos el programa *HolaMundo* en PL/pgSQL:

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

La función tiene tres partes:

- El encabezado que define el nombre de la función y el tipo de retorno.
- El cuerpo de la función, que es una cadena de texto (por lo tanto, siempre va entre comillas dobles).
- La especificación del lenguaje utilizado.

La función recien creada tiene las mismas características que las integradas.

Puede solicitarse mediante el comando **select**:

```
demo=# select HolaMundo();
holamundo
----------------------
Hola Mundo PostgreSQL
(1 row)
```

Puede eliminarse mediante el comando **drop function**.

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

6.2. Variables

Las funciones pueden recibir parámetros, sólo es necesario especificar los tipos de datos. PostgreSQL asigna los nombres a los parámetros utilizando la secuencia $1, $2, $3...

En este ejemplo veremos todas las posibles maneras de declarar variables en una función.

La sentencia alias crea un nuevo nombre para una variable. La sentencia rename cambia el nombre de una variable.

```
create function mi_funcion(int,char) returns int
as '
declare -- declaración de variables locales
x int; -- x es de tipo entero
y int := 10; -- y tiene valor inicial de 10
z int not null; -- z no puede tomar valores nulos
a constant int := 20; -- a es constante
b alias for $1; -- El primer parámetro tiene dos nombres.
rename $1 to c; -- Cambia de nombre el segundo parámetro
begin
x := y + 30;
end;
' language 'plpgsql';
```

6.3. Sentencias

La estructura básica de una función es el **bloque**, que incluye dos partes, la declaración de variables y la sección de sentencias:

declare
sección de variables
begin
sección de sentencias
end;

Sentencia	Descripción
declare begin end	Bloque
:=	Asignación
select into	Asignación desde un **select**
Sentencias `sql`	Cualquier sentencia `sql`

Sentencia	Descripción
perform	Realiza una llamada a comando sql
execute	Interpreta una cadena como comando sql
exit	Termina la ejecución de un bloque
return	Termina la ejecución de una función
if	Ejecuta sentencias condicionalmente
loop	Repite la ejecución de un conjunto de sentencias
while	Repite un conjunto de sentencias mientras
for	Repite un conjunto de sentencias utilizando una variable de control
raise	Despliega un mensaje de error a advertencia

La sentencia de **asignación** utiliza el operador '**:=**' para almacenar los resultados de expresiones en variables. PostgreSQL proporciona otra sentencia para hacer asignaciones, select. Esta sentencia debe obtener como resultado un solo valor para que pueda ser almacenado en la variable:

```
select into x psugerido from productos where clave = 3;
```

La ejecución de comandos sql como **create**, **drop**, **insert** o **update** pueden hacerse sin ninguna sintaxis especial. La excepción es el comando **select**, que requiere ejecutarse con el comando **perform** a fin de que el resultado de la consulta sea descartado.

```
perform select psugerido from productos;
```

La sentencia **execute** también ejecuta un comando sql pero a partir de una cadena de texto. Esta sentencia comporta el problema de que su sintaxis no se verifica hasta la ejecución. Se puede utilizar, por ejemplo, para procesar parámetros como comandos sql:

```
execute $1
```

El comando **exit** termina la ejecución de un bloque. Se utiliza principalmente para romper ciclos.

La bifurcación, o ejecución condicional, se realiza mediante la sentencia **if**:

```
if ( $1 > 0 ) then
resultado := 'Positivo';
else
resultado := 'Negativo';
end if;
```

También puede utilizarse **if** con más de dos ramas:

```
if ( $1 > 0 ) then
resultado := 'Positivo';
elsif ( $1 < 0 ) then
resultado := 'Negativo';
else
resultado := 'Cero';
end if;
```

Con referencia a los bucles, PL/pgSQL ofrece tres opciones:

- El bucle **loop** es infinito, por lo que tiene una estructura muy simple. Por lo general se utiliza con alguna sentencia **if** para terminarlo:

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

- El bucle **while** incluye la condición al inicio del mismo, por lo que el control de su terminación es más claro:

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

- El bucle **for** permite realizar un número de iteraciones controladas por la variable del ciclo:

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

La sentencia **raise** permite enviar mensajes de tres niveles de severidad:

- **debug**. El mensaje se escribe en la bitácora del sistema (logs).
- **notice**. El mensaje se escribe en la bitácora y en el cliente psql.
- **exception** . El mensaje se escribe en la bitácora y aborta la transacción.

El mensaje puede incluir valores de variables mediante el carácter ' %':

- **raise debug** 'funcion(): ejecutada con éxito;
- **raise notice** 'El valor % se tomo por omisión', variable;
- **raise excepción** 'El valor % está fuera del rango permitido', variable;

6.4. Disparadores

Las funciones deben llamarse explícitamente para su ejecución o para incluirlas en consultas. Sin embargo, se puede definir que algunas funciones se ejecuten automáticamente cuando cierto evento tenga lugar en cierta tabla. Estas funciones se conocen como disparadores o *triggers* y se ejecutan mediante los comandos **insert**, **delete** y **uptade**.

Agregamos la tabla *historial* que almacena los productos descontinuados cuando se eliminan de la tabla *productos*.

```
create table historial(
fecha date,
parte varchar(20),
tipo varchar(20),
especificacion varchar(20),
precio float(6)
);
```

Para poder utilizar una función como disparador, no debe recibir argumentos y debe retornar el tipo especial **trigger**:

```
create function respaldar_borrados() returns trigger as '
begin
insert into historial values (
now(),
old.parte,
old.tipo,
old.especificacion,
old.psugerido
);
return null;
end;
```

La variable old está predefinida por PostgreSQL y se refiere al registro con sus antiguos valores. Para referirse a los nuevos valores, se dispone de la variable new.

La función está lista para ser utilizada como disparador, sólo es necesario definirlo y asociarlo a la tabla y al evento deseado:

```
create trigger archivar
beofre delete
on productos
for each row execute procedure respaldar_borrados();
```

Acabamos de crear un disparador de nombre *archivar* que se activará cuando se ejecute el comando *delete* en la tabla *productos*. El usuario no necesita saber que se debe hacer una copia de seguridad de los registros borrados, se hace automáticamente.

Al crear el disparador, hemos especificado "before delete" al indicar la operación. PostgreSQL nos permite lanzar el disparador antes o después (*before, after*) que se efectúen las operaciones. Este matiz es importante, ya que, si este mismo disparador lo ejecutamos después de la operación, no veremos ninguna fila en la tabla. Es posible definir el mismo disparador para varias operaciones:

```
create trigger archivar
beofre delete or update
on productos
for each row execute procedure respaldar_borrados();
```

7. Administración de PostgreSQL

En las tareas administrativas como la instalación, la gestión de usuarios, las copias de seguridad, restauraciones y el uso de prestaciones internas avanzadas, es donde realmente se aprecian las diferencias entre gestores de bases de datos. PostgreSQL tiene fama de ser más complejo de administrar que sus competidores de código abierto, lo que se debe, sobre todo, a que ofrece más prestaciones (o más complejas).

El contenido de los siguientes apartados contempla las opciones de uso común para la administración de un servidor PostgreSQL. Existen tantas alternativas que no es posible incluirlas todas en este módulo, por lo que sólo se presentarán algunos temas de importancia para el administrador, desde una perspectiva general, que permita obtener una visión global de las posibilidades prácticas de las herramientas administrativas.

Bibliografía

El manual de PostgreSQL es la referencia principal que se debe tener siempre a mano para encontrar posibilidades y resolver dudas. En especial se recomienda leer los siguientes capítulos:

Capítulo III. Server Administration.

Capítulo V. Server Programming.

Capítulo VII. Internals.

De la misma manera, también son muy útiles las listas de correo que se describen en el sitio oficial www.postgresql.org.

7.1. Instalación

PostgreSQL está disponible para la mayoría de distribuciones de GNU/Linux. Su instalación es tan sencilla como ejecutar el instalador de paquetes correspondiente.

En Debian, el siguiente procedimiento instala el servidor y el cliente respectivamente:

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

En distribuciones basadas en RPM, los nombres de los paquetes son un poco diferentes:

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

Una vez instalado, se escribirá un *script* de inicio que permite lanzar y apagar el servicio PostgreSQL; de este modo, para iniciar el servicio, deberemos ejecutar el siguiente comando:

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

Notación

Además del **start** también podremos utilizar los parámetros **restart**, **stop**, **reload** que permiten reiniciar, detener y recargar el servidor (releyendo su configuración), respectivamente.

Si se desea realizar una instalación a partir del código fuente, puede obtenerse del sitio oficial www.postgresql.org. A continuación, se describe el proceso de instalación de forma muy simplificada. En la práctica podrán encontrarse algunas diferencias; lo más recomendable es leer cuidadosamente la documentación incluida en los archivos INSTALL y README. Cualquier duda no resuelta por la documentación, puede consultarse en la lista de distribución.

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

Con este proceso se instala la versión 7.4.6. El archivo se descomprime utilizando *tar*. Dentro del directorio recién creado se ejecuta *configure*, que realiza una comprobación de las dependencias de la aplicación. Antes de ejecutar *configure,* debemos instalar todos los paquetes que vamos a necesitar.

La compilación se realiza con *make* y, finalmente, los binarios producidos se copian en el sistema en los lugares convenientes con *make install*.

Después de instalados los binarios, se debe crear el usuario *postgres* (responsable de ejecutar el proceso *postmaster*) y el directorio donde se almacenarán los archivos de las bases de datos.

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

Una vez creado el usuario *postgres*, éste debe inicializar la base de datos:

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

initdb

El ejecutable *initdb* realiza el procedimiento necesario para inicializar la base de datos de postgres, en este caso, en el directorio /usr/local/pgsql/data.

El *postmaster* ya está listo para ejecutarse manualmente:

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

Bibliografía

El proceso de compilación tiene múltiples opciones que se explican en la documentación incluida con las fuentes.

7.1.1. Internacionalización

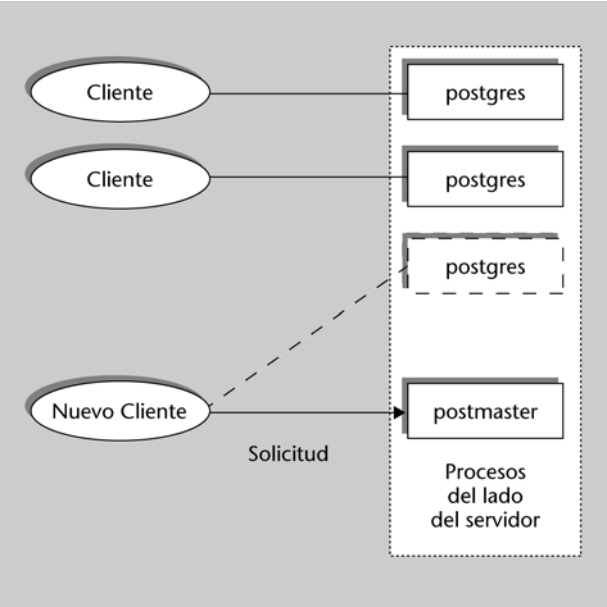
Por omisión, PostgreSQL no está compilado para soportar mensajes en español, por lo que es necesario compilarlo a partir de las fuentes incluyendo las

siguientes opciones de configuración, para que tanto el servidor como el cliente **psql** adopten la configuración establecida por el programa **setlocales** y las variables de entorno respectivas:

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

7.2. Arquitectura de PostgreSQL

El siguiente gráfico muestra de forma esquemática las entidades involucradas en el funcionamiento normal del gestor de bases de datos:



PostgreSQL está basado en una arquitectura cliente-servidor. El programa servidor se llama **postgres** y entre los muchos programas cliente tenemos, por ejemplo, **pgaccess** (un cliente gráfico) y **psql** (un cliente en modo texto).

Un proceso servidor *postgres* puede atender exclusivamente a un solo cliente; es decir, hacen falta tantos procesos servidor *postgres* como clientes haya. El proceso **postmaster** es el encargado de ejecutar un nuevo servidor para cada cliente que solicite una conexión.

Se llama **sitio** al equipo anfitrión (*host*) que almacena un conjunto de bases de datos PostgreSQL. En un *sitio* se ejecuta solamente un proceso *postmaster* y múltiples procesos *postgres*. Los clientes pueden ejecutarse en el mismo sitio o en equipos remotos conectados por TCP/IP.

Es posible restringir el acceso a usuarios o a direcciones IP modificando las opciones del archivo pg_hba.conf, que se encuentra en /etc/postgresql/pg_hba.conf.

Este archivo, junto con /etc/postgresql/postgresql.conf son particularmente importantes, porque algunos de sus parámetros de configuración por defecto

provocan multitud de problemas al conectar inicialmente y porque en ellos se especifican los mecanismos de autenticación que usará PostgreSQL para verificar las credenciales de los usuarios.

Para habilitar la conexión a PostgreSQL desde clientes remotos, debemos verificar el parámetro tcpip_socket = true en el fichero */etc/postgresql/postgresql.conf*.

A continuación, para examinar los métodos de autenticación y las posibilidades de conexión de clientes externos, debemos mirar el fichero */etc/postgresql/pg_hba.conf*, donde se explicita la acción que hay que emprender para cada conexión proveniente de cada *host* externo, o grupo de *hosts*.

7.3. El administrador de postgres

Al terminar la instalación, en el sistema operativo se habrá creado el usuario **postgres**, y en PostgreSQL se habrá creado un usuario con el mismo nombre.

Él es el único usuario existente en la base de datos y será el único que podrá crear nuevas bases de datos y nuevos usuarios.

Normalmente, al usuario *postgres* del sistema operativo no se le permitirá el acceso desde un *shell* ni tendrá contraseña asignada, por lo que deberemos convertirnos en el usuario *root*, para después convertirnos en el usuario *postgres* y realizar tareas en su nombre:

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

El usuario **postgres** puede crear nuevas bases de datos utilizando el comando **createdb**. En este caso, le indicamos que el usuario propietario de la misma será el usuario postgres:

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

El usuario **postgres** puede crear nuevos usuarios utilizando el comando **createuser**:

Se ha creado el usuario yo con permisos para crear bases de datos y sin permisos para crear usuarios.

```
postgres@localhost:~$ createuser yo
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
```

Los siguientes comandos eliminan bases de datos y usuarios, respectivamente:

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

Es recomendable que se agreguen los usuarios necesarios para operar la instalación de PostgreSQL, y recurrir, así, lo menos posible al ingreso con *postgres*.

También disponemos de sentencias SQL para la creación de usuarios, grupos y privilegios:

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

Los grupos permiten asignar privilegios a varios usuarios y su gestión es sencilla:

```
create group migrupo;
```

Para añadir o quitar usuarios de un grupo, debemos usar:

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

7.3.1. Privilegios

Cuando se crea un objeto en PostgreSQL, se le asigna un dueño. Por defecto, será el mismo usuario que lo ha creado. Para cambiar el dueño de una tabla, índice, secuencia, etc., debemos usar el comando *alter table*. El dueño del objeto es el único que puede hacer cambios sobre él, si queremos cambiar este comportamiento, deberemos asignar privilegios a otros usuarios.

Los privilegios se asignan y eliminan mediante las sentencias `grant` y `revoke`. PostgreSQL define los siguientes tipos de operaciones sobre las que podemos dar privilegios:

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

Presentamos algunas sentencias de trabajo con privilegios, que siguen al pie de la letra el estándar SQL:

```
grant all privileges on proveedores to marc;
grant select on precios to manuel;
grant update on precios to group migrupo;
revoke all privileges on precios to manuel;
grant select on ganacias from public;
```

7.4. Creación de tipos de datos

Entre las múltiples opciones para extender PostgreSQL, nos queda aún por ver la creación de tipos o dominios (según la nomenclatura del estándar SQL). PostgreSQL prevé dos tipos de datos definidos por el administrador:

- Un tipo de datos compuesto, para utilizar como tipo de retorno en las funciones definidas por el usuario.

- Un tipo de datos simple, para utilizar en las definiciones de columnas de las tablas.

A modo de ejemplo, veamos un tipo compuesto y la función que lo devuelve:

```
create type comptipo as (f1 int, f2 text);
create function gettipo() returns setof comptipo as
  'select id, nombre from clientes' language sql;
```

Para el tipo de datos simple, la definición es más compleja, pues se debe indicar a PostgreSQL funciones que tratan con este tipo que le permitirán usarlo en operaciones, asignaciones, etc.

A modo de ejemplo, vamos a crear el tipo "numero complejo", tal como lo hace la documentación de PostgreSQL. En primer lugar, debemos definir la estructura donde almacenaremos el tipo:

Tratar con el tipo de datos simple

Habitualmente, las funciones que tratarán con este tipo de datos se escribirán en C.

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

Después, las funciones que lo recibirán o devolverán:

```
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);
}
```

Ahora estamos en condiciones de definir las funciones, y el tipo:

```
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 proceso es un poco farragoso, pero compensa por la gran flexibilidad que aporta al SGBD. A continuación, podríamos crear algunos operadores para trabajar con este tipo (suma, resta, etc.), mediante los pasos que ya conocemos.

La creación de un dominio en PostgreSQL consiste en un tipo (definido por el usuario o incluido en el SGBD), más un conjunto de restricciones. La sintaxis es la siguiente:

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

Se ha creado un dominio basado en un tipo definido por el sistema donde la única restricción es su longitud.

Evidentemente, deberíamos haber definido el operador *complejo_mayor* que recibiera dos números complejos e indicara si el primero es mayor que el segundo.

7.5. Plantilla de creación de bases de datos

PostgreSQL tiene definidas dos bases de datos de sistema, *template0* y *template1* (que habremos visto en los ejemplos al listar las bases de datos del gestor), con un conjunto de objetos tales como los tipos de datos que soporta o los lenguajes de procedimiento instalados.

La base de datos *template0* se crea al instalar el servidor, con los objetos por defecto del mismo, y la base de datos *template1* se crea a continuación de la anterior con algunos objetos más particulares del entorno o sistema operativo donde se ha instalado PostgreSQL. Esto hace que sea muy recomendable heredar siempre de *template1* (que es el comportamiento por defecto).

Estas bases de datos se utilizan como "padres" del resto de bases de datos que se crean, de modo que, al crear una nueva base de datos, se copian todos los objetos de *template1*.

Ventajas de esta situación

Si queremos añadir algún objeto (una tabla, un tipo de datos, etc.) a todas las bases de datos, sólo tenemos que añadirlo a *template1*, y el resto de bases de datos que se hayan creado a partir de ésta lo tendrán disponible.

Si se instala un lenguaje sobre la base de datos *template1*, automáticamente todas las bases de datos también usarán el lenguaje. En las distribuciones de Linux es frecuente que se haya realizado de este modo, por lo que no hay necesidad de instalarlo.

Por supuesto, podemos escoger otra plantilla para crear bases de datos, especificándola en la sentencia:

```
create database nuevbd template plantillabd
```

7.6. Copias de seguridad

Hacer periódicamente copias de seguridad de la base de datos es una de las tareas principales del administrador de cualquier base de datos. En PostgreSQL, estas copias de seguridad se pueden hacer de dos maneras distintas:

- Volcando a fichero las sentencias SQL necesarias para recrear las bases de datos.
- Haciendo copia a nivel de fichero de la base de datos.

En el primer caso, disponemos de la utilidad pg_dump, que realiza un volcado de la base de datos solicitada de la siguiente manera:

```
$ pg_dump demo > fichero_salida.sql
```

pg_dump es un programa cliente de la base de datos (como **psql**), lo que significa que podemos utilizarlo para hacer copias de bases de datos remotas, siempre que tengamos privilegios para acceder a todas sus tablas. En la práctica, esto significa que debemos ser el usuario administrador de la base de datos para hacerlo.

Si nuestra base de datos usa los OID para referencias entre tablas, debemos indicárselo a **pg_dump** para que los vuelque también (`pg_dump -o`) en lugar de volver a crearlos cuando inserte los datos en el proceso de recuperación. Asimismo, si tenemos BLOB en alguna de nuestras tablas, también debemos indicárselo con el parámetro correspondiente (`pg_dump -b`) para que los incluya en el volcado.

Más información

Hay otras opciones interesantes que podemos consultar mediante el parámetro `-h`.

Para restaurar un volcado realizado con **pg_dump**, podemos utilizar directamente el cliente **psql**:

```
$ psql demo < fichero_salida.sql
```

Una vez recuperada una base de datos de este modo, se recomienda ejecutar la sentencia `analyze` para que el optimizador interno de consultas de PostgreSQL vuelva a calcular los índices, la densidad de las claves, etc.

pg_dump

pg_dump realiza la copia a partir de la base de datos de sistema *template0*, por lo que también se volcarán los tipos definidos, funciones, etc. de la base de datos. Cuando recuperemos esta base de datos, debemos crearla a partir de *template0* si hemos personalizado *template1* (y no de *template1* como lo haría por defecto la sentencia `create database`) para evitar duplicidades.

Las facilidades del sistema operativo Unix, permiten copiar una base de datos a otra en otro servidor de la siguiente manera:

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

Para hacer la copia de seguridad a nivel de fichero, simplemente copiamos los ficheros binarios donde PostgreSQL almacena la base de datos (especificado en

tiempo de compilación, o en paquetes binarios, suele ser */var/lib/postgres/data*), o bien hacemos un archivo comprimido con ellos:

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

El servicio PostgreSQL debe estar parado antes de realizar la copia.

A menudo, en bases de datos grandes, este tipo de volcados origina ficheros que pueden exceder los límites del sistema operativo. En estos casos tendremos que utilizar técnicas de creación de volúmenes de tamaño fijo en los comandos **tar** u otros con los que estemos familiarizados.

7.7. Mantenimiento rutinario de la base de datos

Hay una serie de actividades que el administrador de un sistema gestor de bases de datos debe tener presentes constantemente, y que deberá realizar periódicamente. En el caso de PostgreSQL, éstas se limitan a un mantenimiento y limpieza de los identificadores internos y de las estadísticas de planificación de las consultas, a una reindexación periódica de las tablas, y al tratamiento de los ficheros de registro.

7.7.1. *vacuum*

El proceso que realiza la limpieza de la base de datos en PostgreSQL se llama *vacuum*. La necesidad de llevar a cabo procesos de *vacuum* periódicamente se justifica por los siguientes motivos:

- Recuperar el espacio de disco perdido en borrados y actualizaciones de datos.
- Actualizar las estadísticas de datos utilizados por el planificador de consultas SQL.
- Protegerse ante la pérdida de datos por reutilización de identificadores de transacción.

Para llevar a cabo un *vacuum*, deberemos ejecutar periódicamente las sentencias **vacuum** y **analyze**. En caso de que haya algún problema o acción adicional a realizar, el sistema nos lo indicará:

```
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: haciendo vacuum a "public.ganancia"
INFO: "ganancia": se encontraron 0 versiones de filas eliminables y 2 no eliminables en 1 páginas
DETAIL: 0 versiones muertas de filas no pueden ser eliminadas aún.
Hubo 0 punteros de ítem sin uso.
0 páginas están completamente vacías.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO: analizando "public.ganancia"
INFO: "ganancia": 1 páginas, 2 filas muestreadas, se estiman 2 filas en total
VACUUM
```

7.7.2. Reindexación

La reindexación completa de la base de datos no es una tarea muy habitual, pero puede mejorar sustancialmente la velocidad de las consultas complejas en tablas con mucha actividad.

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

7.7.3. Ficheros de registro

Es una buena práctica mantener archivos de registro de la actividad del servidor. Por lo menos, de los errores que origina. Durante el desarrollo de aplicaciones puede ser muy útil disponer también de un registro de las consultas efectuadas, aunque en bases de datos de mucha actividad, disminuye el rendimiento del gestor y no es de mucha utilidad.

En cualquier caso, es conveniente disponer de mecanismos de rotación de los ficheros de registro; es decir, que cada cierto tiempo (12 horas, un día, una semana...), se haga una copia de estos ficheros y se empiecen unos nuevos, lo que nos permitirá mantener un histórico de éstos (tantos como ficheros podamos almacenar según el tamaño que tengan y nuestras limitaciones de espacio en disco).

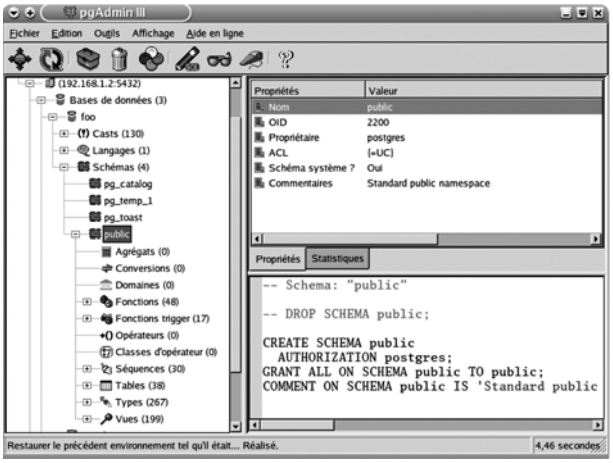
PostgreSQL no proporciona directamente utilidades para realizar esta rotación, pero en la mayoría de sistemas Unix vienen incluidas utilidades como *logrotate* que realizan esta tarea a partir de una planificación temporal.

8. Cliente gráfico: pgAdmin3

El máximo exponente de cliente gráfico de PostgreSQL es el software pgAdmin3 que tiene licencia "Artist License", aprobada por la FSF.

pgAdmin3 está disponible en http://www.pgadmin.org.

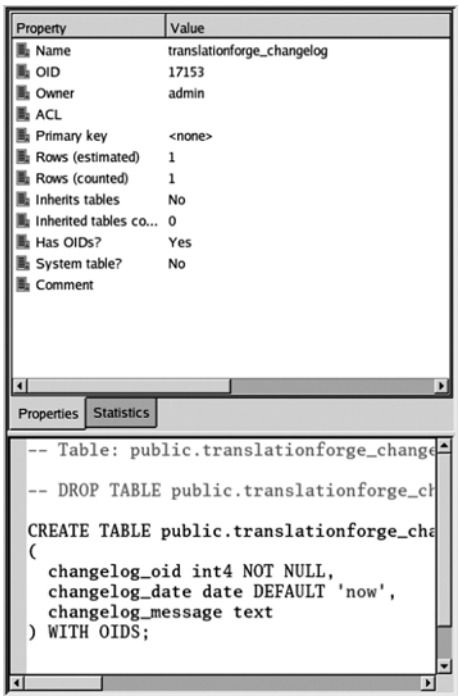
WEB



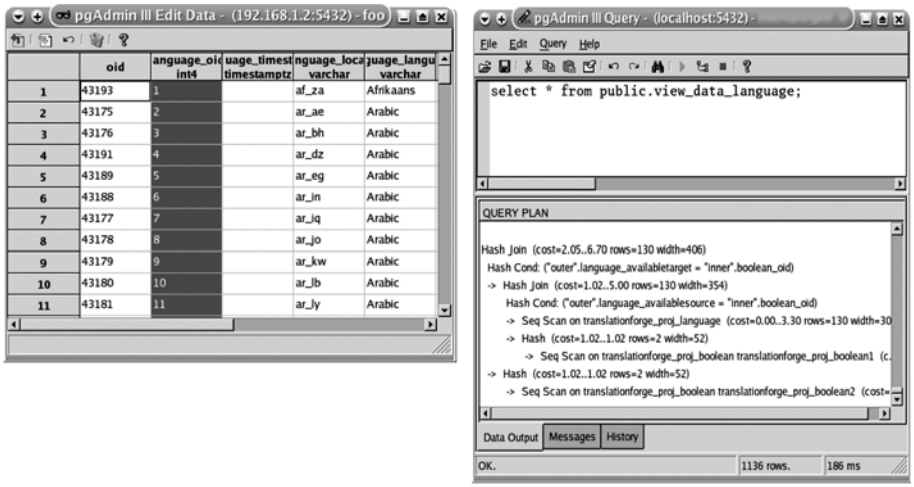
En pgAdmin3 podemos ver y trabajar con casi todos los objetos de la base de datos, examinar sus propiedades y realizar tareas administrativas.

- Agregados
- Casts
- Columnas
- Constraints
- Conversiones
- Bases de datos
- Dominios
- Funciones
- Grupos
- Índices
- Lenguajes (PLpgsql, PLpython, PLperl, etc.)
- Clases de operadores
- Operadores
- Servidores PostgreSQL
- Reglas
- Esquemas
- Secuencias
- Tablas
- Triggers
- Tipos de datos
- Usuarios
- Vistas

Una característica interesante de pgAdmin3 es que, cada vez que realizamos alguna modificación en un objeto, escribe la/s sentencia/s SQL correspondiente/s, lo que hace que, además de una herramienta muy útil, sea a la vez didáctica.



pgAdmin3 también incorpora funcionalidades para realizar consultas, examinar su ejecución (como el comando **explain**) y trabajar con los datos.



Todas estas características hacen de pgAdmin3 la única herramienta gráfica que necesitaremos para trabajar con PostgreSQL, tanto desde el punto de vista del usuario como del administrador. Evidentemente, las acciones que podemos realizar en cada momento vienen condicionadas por los permisos del usuario con el que nos conectemos a la base de datos.

Resumen

PostgreSQL implementa las características necesarias para competir con cualquier otra base de datos comercial, con la ventaja de tener una licencia de libre distribución BSD.

La migración de bases de datos alojadas en productos comerciales a PostgreSQL se facilita gracias a que soporta ampliamente el estándar SQL. PostgreSQL cuenta con una serie de características atractivas como son la herencia de tablas (clases), un rico conjunto de tipos de datos que incluyen arreglos, BLOB, tipos geométricos y de direcciónes de red. PostgreSQL incluye también el procesamiento de transacciones, integridad referencial y procedimientos almacenados. En concreto, hay procedimientos documentados para migrar los procedimientos almacenados desarrollados en lenguajes propietarios de bases de datos comerciales (PL/SQL) a PL/PGSQL.

La API se distribuye para varios lenguajes de programación como C/C++, Perl, PHP, Python, TCL/Tk y ODBC.

Por si esto fuera poco PostgreSQL es extensible. Es posible agregar nuevos tipos de datos y funciones al servidor que se comporten como los ya incorporados. También es posible insertar nuevos lenguajes de programación del lado del servidor para la creación de procedimientos almacenados. Todas estas ventajas hacen que muchos programadores lo elijan para el desarrollo de aplicaciones en todos los niveles.

Entre sus deficiencias principales podemos mencionar los OID. PostgreSQL está aún en evolución, se espera que en futuras versiones se incluyan nuevas características y mejoras al diseño interno del SGBD.

Bibliografía

Documentación de PostgreSQL de la distribución: http://www.postgresql.org/docs/

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

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